

MOSAICO

D3.2: Orchestrator Framework for AI Communities - Initial Version

April, 2026

Funded by the European Union

Funded by the European Union under the Grant Agreement No 101189664. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency (HADEA). Neither the European Union nor the granting authority can be held responsible for them.

MOSAICO

D3.2 Orchestrator Framework for AI Communities - Initial Version

Work package	WP3 Coordination and collaboration of communities of AIs
Tasks	T3.2 Language and dataset for collaboration patterns T3.4 Scalable dynamic orchestration framework for AI communities
Task Contributors	T3.2 IMT T3.3 IMT, UDA, UY, IMM, COLLINS
Due date	31/03/2026
Submission date	07/05/2026
Type of deliverable	R - Document, report
Dissemination Level	Public
Deliverable lead	IMT
Version	1.0.0
Authors	IMT, UDA
Reviewers	CAI,UDA,UY
Keywords	Workflow, SysML, Orchestration

Document Revision History

Version	Date	Description of change	List of contributor(s)
v0.0.1	12/12/2025	Initial population of template	IMT,UDA
v0.0.2	03/04/2026	First draft sent for internal review	IMT,UDA
v0.0.3	20/04/2026	Integration of Use Cases	UDA,UY,IMM,IMT
v1.0.0	05/05/2026	Integration of reviewers feedbacks	IMT

Partners

























Funding

Funded by
the European Union

Call	Digital and emerging technologies for competitiveness and fit for the Green Deal (HORIZON-CL4-2024-DIGITAL-EMERGING-01)
Topic	HORIZON-CL4-2024-DIGITAL-EMERGING-01-22: Fundamentals of Software Engineering
Type of action	Research and Innovation Action (RIA)

Disclaimer

Funded by the European Union under the Grant Agreement No 101189664. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency (HADEA). Neither the European Union nor the granting authority can be held responsible for them.

Copyright Notice

© MOSAICO Consortium, 2026

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both. Reproduction is authorised provided the source is acknowledged.

Executive Summary

This document presents ongoing work on the orchestration part of the MOSAICO platform. Orchestration refers, for this document, as two primary tasks, 1) decomposing users natural language requests expressed into a workable workflow with a precise sequence of actions to follow and 2) enacting the workflow by giving tasks to agents and monitoring their progress.

In order for results of the first to be enacted by the second, they need to use a common language to express workflows. Workflows can be expressed in a variety of ways, ranging from natural language without structure to precise and detailed languages such as assembly code. Both define a list of instructions to follow and control structure to react depending on result of actions. As one may imagine, these languages offer a tradeoff between preciseness, ease of interpretation and ease of producing by machines / human. In MOSAICO, users are expected to be able to use generated workflows or write their own. Given these constraints we decided to use a subset of SysML v2 to express workflows.

The first step of building the AI component responsible for producing workflow is to build a dataset of natural language examples and their corresponding workflows. However, SysML v2 is a new standard and lacks public datasets and producing a new dataset from scratch is not feasible. Thus, as many workflow languages have similar concepts, we gathered public datasets and developed automated translation scripts to SysML v2. The resulting dataset is composed of about 65,000 pairs of natural language descriptions and corresponding SysML v2 workflows. This dataset covers all the features that are supported by the MOSAICO orchestration agent. It is also this agent that is responsible for querying the agent repository (WP2) and communicating with the consensus agent (WP4).

The second part of this report presents orchestration patterns found in both academic and industrial state of the art. In our context, we refer to agent organization as orchestration patterns. We classify these patterns across several categories based on characteristics such as interaction modality (cooperative vs adversarial), sequencing strategy (sequential, parallel) or execution topology (e.g. hierarchical or peer-to-peer). This classification shows that there is a wide variety of possible patterns that MOSAICO orchestration framework should support. We show our preliminary work on the orchestration framework based on SysML v2. This orchestrator interprets SysML v2 workflows and delegates actions to agents.

In these complex collaboration patterns, agents are expected to exchange messages and artifacts. However, many messages can be ambiguous and lead to errors of interpretation by another agent. Standards from the community of multi agent systems exist to alleviate this issue. They propose a set of *performatives* (or *illocutions*) that are used in messages between agent to qualify the nature of the communication act. These standard are, by design, open, thus they propose a limited set of words and let users add new ones when needed. We gathered a list of such *performatives* and selected the ones that are relevant and should be implemented in MOSAICO and how they can integrate with the A2A protocol (WP1).

Table of Contents

1 Introduction . . . 9

2 Collaboration Pattern Language and Dataset . . . 9
2.1 Language Requirements . . . 9
2.2 Existing Languages . . . 10
2.2.1 UML Activity Diagram . . . 10
2.2.2 BPMN . . . 11
2.2.3 PDDL . . . 13
2.2.4 SysML v2 . . . 14
2.3 The MOSAICO Collaboration Language . . . 16
2.3.1 Structural Modeling . . . 16
2.3.2 Behavioral Modeling . . . 16
2.3.3 Additional MOSAICO Types . . . 18
2.4 Existing Datasets . . . 19
2.5 Transformation of datasets . . . 21
2.5.1 Transforming WorFBench . . . 22
2.5.2 Transforming SAP-SAM . . . 23
2.5.3 Transforming SOP-Bench . . . 24
2.6 Metrics for the MOSAICO Collaboration Language Dataset . . . 26

3 Orchestration Framework . . . 27
3.1 Existing Orchestration Patterns . . . 27
3.1.1 Existing Orchestrators . . . 27
3.1.2 Recurring Orchestration Patterns . . . 28
3.2 MOSAICO Collaboration Language Runtime Interpretation . . . 30
3.2.1 SysML v2 Language Parser . . . 30
3.2.2 MOSAICO Orchestrator Work In Progress . . . 32
3.2.3 Communicative acts . . . 38

4 Use Cases . . . 43
4.1 Agent interactions with A2A + ACL . . . 44
4.1.1 Building an Artifact in a Natural Language . . . 45
4.1.2 Building an Artifact in a Programming Language . . . 45
4.2 The Metagente framework . . . 46
4.3 Extended Reality Use Case . . . 49
4.4 Metamodel Generation Use Case . . . 53

5 Conclusion . . . 56

A AgentSpeak support . . . 60

List of Figures

1 Requirement management workflow in Activity Diagram 11
2 Requirement management workflow in BPMN 12
3 Example of SysML v1 model 14
4 Same example as Listing 1 with the graphical syntax 15
5 Agent interaction modality which lacks the debate modality (X. Li et al., 2024) . . 28
6 Illustration of different agent execution topologies (K. Zhu et al., 2025). 29
7 Zoomed part of the hierarchy of class of SysML v2 30
8 Class diagram of our custom DTOs. 33
9 Object Diagram illustrating the transformation from the SysML v2 program of Listing 4 into our custom DTOs. 37
10 Collaboration diagram for example of generation of requirements. 44
11 Collaboration diagram for example of refactoring. 46
12 Optimization phase of the Metagente framework. 47
13 Evaluation phase of the Metagente framework. 48
14 Full ACL AgentSpeak agent framework 61

List of Tables

1 Datasets across different Modeling Languages 21
2 Transformation example from WorBench workflow to SysML v2 22
3 Mapping between BPMN elements and their corresponding SysML v2 representations 24
4 Transformation example from SOP-Bench workflow to SysML v2 25
5 Summary of source datasets and their transformation into SysML v2 26
6 Frequency of SysML v2 elements in the dataset 26
7 Minimum, mean, and maximum count of SysML v2 elements in the dataset . . . 27
8 Mapping of SysML v2 elements to custom orchestration DTOs 36
9 Refined SysML v2 action descriptions adapted as explicit prompt instructions to guide the LLM agents during the palindrome verification proof-of-concept. 38
10 Performatives in FIPA-ACL (Foundation for Intelligent Physical Agents, 1996–2002) and their potential use in MOSAICO 41

Abbreviations

DTO	Data Transfer Objects
KerML	Kernel Modeling Language
LLM	Large-Language Model
MBSE	Model-Based Systems Engineering
SysML	Systems Modeling Language
UML	Unified Modeling Language

1 Introduction

With the rise of agentic AI, multi-agents systems received a boost in popularity. However, such systems require carefully crafted collaboration patterns between agents to achieve good results. These collaboration patterns are plans that specify which agent is responsible for which task and how they can communicate with each others. The collaboration agent of the MOSAICO platform has two critical roles. First, analyzing users' requests and devising a collaboration pattern between agents. This collaboration pattern can either be written by humans or automatically generated. Then, the orchestration agent uses this collaboration pattern to enact and monitor the collaboration between agents.

This deliverable reports results on tasks 3.2 *Language and dataset for collaboration patterns* and work in progress 3.4 *Scalable dynamic orchestration framework for AI communities*. The report is structured in three main sections. Section 2 presents several candidates for the MOSAICO collaboration pattern language and the generation process used to gather a dataset that will later be used by our orchestration agent (Task 3.4, M30). Then, Section 3 develops on the enaction of the collaboration patterns with an overview of existing orchestration architectures and the architecture we developed in our orchestrator prototype (Task 3.3, M30). Finally, Section 4 illustrates several examples of collaboration patterns.

2 Collaboration Pattern Language and Dataset

This section presents the decision process behind the choice of the collaboration pattern language used in MOSAICO and the gathering of the dataset of examples and natural language descriptions. Section 2.1 presents a brief overview of the core properties that guided our choice of collaboration pattern language. Then, Section 2.2 lists several existing workflow languages with their pros and cons considering the aforementioned requirements. Following the list, Section 2.3 gives a detailed description of the chosen language. Section 2.4 presents a collection of existing datasets in languages presented in Section 2.2. Then, Section 2.5 presents in details the generation processes we develop to aggregate and adapt multiple existing datasets to our needs. Finally, in Section 2.6, we analyze the produced dataset to assess its adequacy to our use case.

2.1 Language Requirements

The collaboration language we aim to select should satisfy a set of requirements that ensures it is suitable for representing collaborative workflows while remaining understandable to use-case providers and suitable for automated processing by tools and LLM-based systems.

- **Expressiveness**: The language should provide sufficient constructs to represent the essential aspects of collaborative workflows. In particular, it should support:
  - **Actions**: The specification of actions, preferably described in natural language, to capture operational steps within a workflow.
  - **Conditionals**: The ability to express decision points and conditional branching.
  - **Cycles**: The possibility to represent loops (e.g., *goto* or *while* constructs) for repeated activities.

  - **Actors**: The representation of actors participating in the workflow, including the possibility to define constraints and properties associated with them.
  - **Data Flow**: The ability to model data exchanges between actions. This aspect is not currently supported but will be in later deliverable.

- **Formality**: The language should have a well-defined formal semantics to ensure precise interpretation and enable automated analysis.
- **Familiarity**: The language should be easily understandable by use case providers and domain experts.
- **Tooling**: The language should have existing editors or parsers available under a permissive open-source license, or it should be straightforward to develop such tooling.
- **LLM Understandability**: The language should be structured such that large language models can generate valid instances with high accuracy.

2.2 Existing Languages

Several existing languages have been proposed to address requirements listed in Section 2.1. These languages differ in their level of expressiveness, the modeling constructs they provide, and their readability for both domain experts and automated systems. In the state of the art, a variety of approaches can be found, including modeling and planning languages such as Activity Diagrams, BPMN, PDDL, SOP-based representations, and SysML v2.

To compare these different approaches, we consider a requirement management workflow as a common use case. By modeling the same workflow across these languages, we aim to evaluate their suitability for representing MOSAICO collaboration workflows.

Requirement management workflow:

We want to model a requirement management workflow that is composed of a sequence of steps. The process starts by receiving a system description from the user. Then, it enters an iterative phase where a new requirement is generated through a decomposition step based on this description. For each generated requirement, three checks are performed in parallel: verifying its correctness with respect to the system description, ensuring it is not redundant with existing requirements, and checking that it is satisfiable. Once these checks are completed, their results are synchronized and a consensus step determines whether the requirement should be accepted. The requirement list is then updated, and the process evaluates whether the set of requirements is complete. If it is not complete, the process repeats to generate additional requirements. Finally, once the requirements are complete, the full list is returned to the user, and the process ends.

2.2.1 UML Activity Diagram

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration, and concurrency. In the Unified Modeling Language (UML), activity diagrams are intended to model both computational and organizational processes (i.e., workflows), as well as the data flows intersecting with the related activities (Object Management Group, 2017).

An activity diagram typically represents actions, control flows, decision nodes, merge nodes, fork and join nodes, as well as initial and final nodes (UML Diagrams, 2026). These constructs

make it possible to describe the sequencing of tasks, conditional branching, and parallel execution in a compact and intuitive way. Because of this expressiveness, activity diagrams are widely used to model procedural knowledge, business processes, and system behaviors at a high level of abstraction.

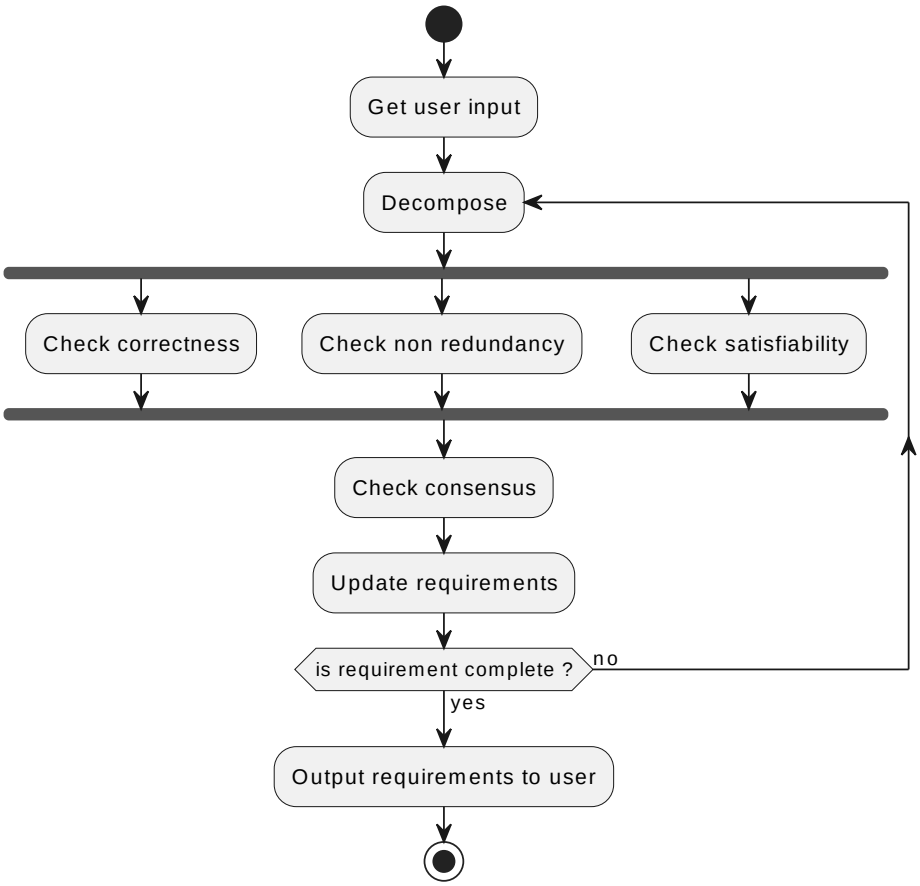


Figure 1: Requirement management workflow in Activity Diagram

Figure 1 shows the requirements management process described in the previous section. The black dot marks the start of the process. Actions are denoted by bubbles. The two black bars mark parallel sections. Condition is represented by the rectangle with chopped angles and the dotted circle represents the end of the process.

2.2.2 BPMN

Business Process Model and Notation (BPMN) is a standardized (Object Management Group, 2014; White, 2004) graphical notation for representing business processes and workflows. It was designed to be understandable by a wide range of stakeholders, including business analysts, process designers, and technical implementers, while remaining precise enough to support the translation of process models into software components. BPMN provides a flowchart-like representation of processes that is independent of any specific implementation environment.

A BPMN model typically includes activities, events, gateways, and flows to represent the behavior of a process. Activities denote units of work, events capture something that hap-

pens during the process, and gateways express branching, synchronization, or merging points in the control flow. In addition, BPMN supports organizational structuring through pools and lanes, which make it possible to distinguish participants and responsibilities within a collaboration. These constructs make BPMN particularly suitable for describing procedural workflows, interactions between actors, and execution dependencies in a visual and intuitive way.

Because of this expressiveness, BPMN is widely used to model business procedures, operational workflows, and collaborative processes at a relatively high level of abstraction. In our context, BPMN is especially relevant because it provides an explicit representation of actions, control-flow dependencies, and participant structure, making it a useful source formalism for deriving more constrained workflow representations.

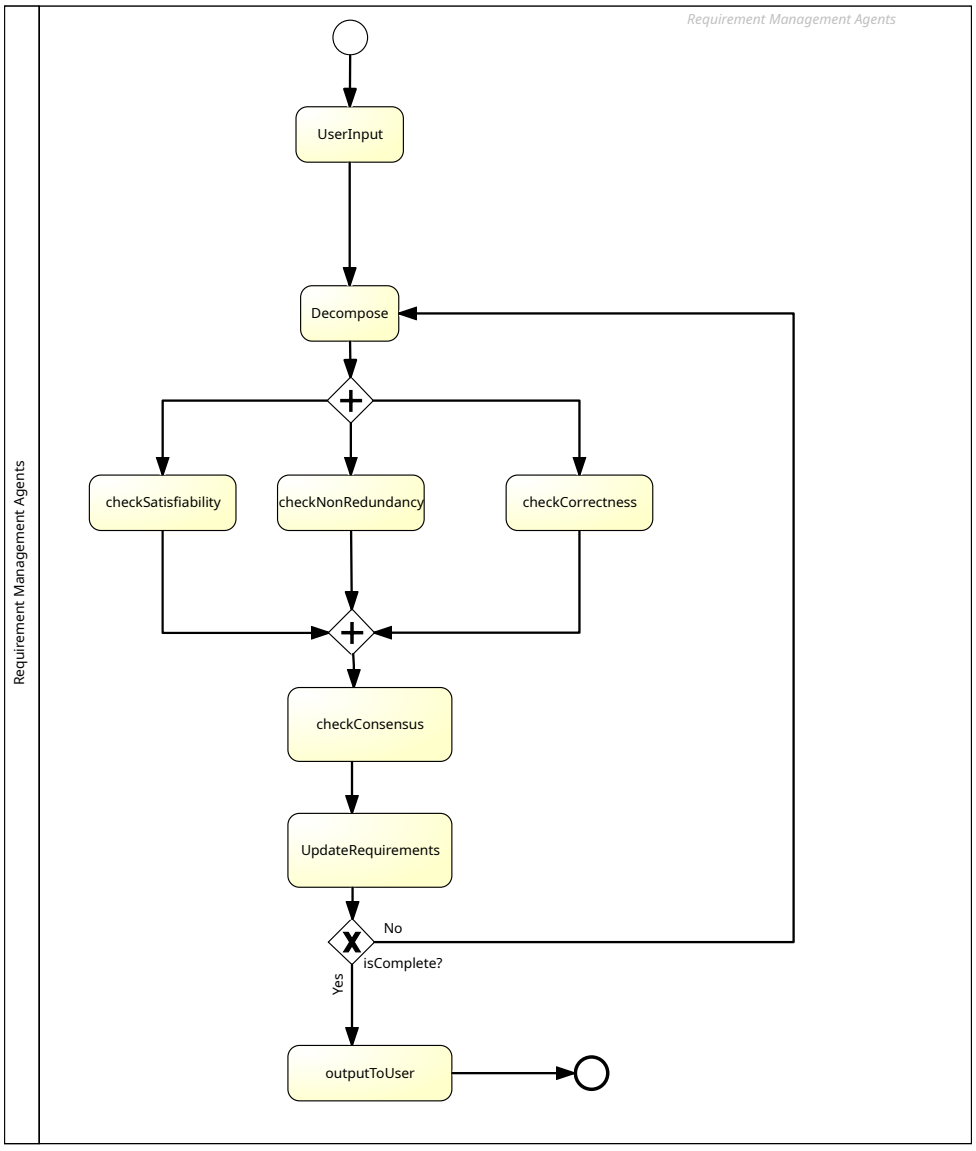


Figure 2: Requirement management workflow in BPMN

Figure 2 depicts the same requirement management process. The graphical syntax is similar to the UML activity diagram. Note that both parallel bar and conditions have been replaced by diamonds with symbols.

2.2.3 PDDL

The Planning Domain Definition Language (PDDL) is a standard formal language for representing automated planning problems through the specification of actions, predicates, objects, initial states, and goal conditions (Fox et al., 2003; McDermott et al., 1998). Since its introduction, PDDL has become the dominant modeling language in classical planning benchmarks and competitions, providing a common representation for expressing planning domains independently of the underlying solver (Fox et al., 2003; Planning.Wiki, 2026).

A PDDL model is typically divided into two parts: a domain file and a problem file. The domain file defines the action schema of the system, including parameters, preconditions, and effects, while the problem file specifies the objects of interest, the initial state, and the goal to be achieved. This separation makes it possible to reuse the same domain model across multiple planning instances while varying the problem configuration. Because of this structured formulation, PDDL is widely used to encode procedural tasks, planning benchmarks, and goal-oriented workflows at a high level of abstraction (Fox et al., 2003; Planning.Wiki, 2026).

In our context, PDDL is relevant because it provides an explicit and formal representation of actions and their logical dependencies. That said, PDDL is primarily suited for tasks where the objective is to capture action sequencing and goal-directed behavior, such as in robotics, which also highlights its main limitation.

Using PDDL elements, it can be difficult, or even impossible, to represent the requirement workflow. To better understand its semantics, let us consider the following simple PDDL example:

Gripper Task: Pick-up Operator

Action/Operator: `pick-up`
Description: The robot can pick up object x in room y using gripper z.
Precondition: `BALL(x)`, `ROOM(y)`, `GRIPPER(z)`, `at-ball(x, y)`, `at-robby(y)`, and `free(z)` must be true.
Effect: `carry(z, x)` becomes true. `at-ball(x, y)` and `free(z)` become false. All other predicates remain unchanged.

In PDDL:

```
(:action pick-up
 :parameters (?x ?y ?z)
 :precondition (and (BALL ?x) (ROOM ?y) (GRIPPER ?z)
                    (at-ball ?x ?y) (at-robby ?y) (free ?z))
 :effect (and (carry ?z ?x)
              (not (at-ball ?x ?y))
              (not (free ?z))))
```

In this example, the robot picks up an object x located in room y using a specific gripper z, provided that all required conditions are satisfied (the object, room, and gripper exist, the robot and object are in the same room, and the gripper is free). Once the action is executed, the robot holds the object, and the object is no longer at its original location while the gripper is no longer available. This illustrates how PDDL models actions through explicit preconditions and effects that define state transitions.

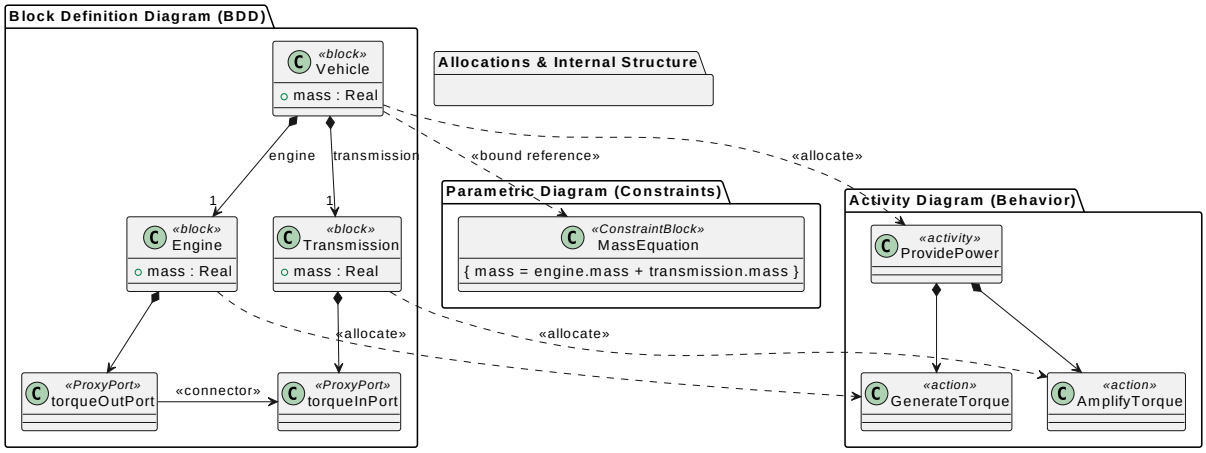


Figure 3: Example of SysML v1 model

2.2.4 SysML v2

Since its adoption in 2006, the Systems Modeling Language version 1 (SysML v1) has served as the foundational standard for Model-Based Systems Engineering (MBSE), providing a widely accepted visual mechanism for defining complex system architectures. However, the initial version of SysML was constructed as a profile of UML (Specification, 2007), making it complex and graphical only. Figure 3[1] shows an example model of a vehicle.

To address these challenges, the Object Management Group (OMG) developed SysML version 2. Unlike its predecessor, SysML v2 is not a UML profile but instead leverages a new, formally defined metamodel known as the Kernel Modeling Language (KerML). Unlike UML, SysML v2 has both a graphical and a textual syntax. Listing 1 and Figure 4 model a basic vehicle powertrain to demonstrate how structural architecture and parametric constraints can be unified.

The primary system block, part `vehicle`, is structurally decomposed into two internal subcomponents: the engine and the transmission. Each component is assigned a mass attribute, which allows the vehicle to define its total mass dynamically as a derived parametric constraint that sums the masses of its internal parts. Beyond structural definition, this example also captures behavioral modeling and allocation. A primary action `providePower` is established and further divided into two specific sub-actions: `generateTorque` and `amplifyTorque`.

Utilizing the `perform` keyword, the model directly maps these behaviors to the physical structure. The power action is assigned to the vehicle as a whole, while the granular torque generation and amplification tasks are delegated to the engine and transmission, respectively. Finally, the model formalizes the physical interaction between these internal components through defined ports. The engine is equipped with a `torqueOutPort`, and the transmission features a `torqueInPort` and a`connect` statement explicitly links them to formally represent the transfer of mechanical power across the defined system architecture.

The resulting syntaxes are significantly more concise and human-readable, allowing users to comprehend the system directly from the text without relying exclusively on generated graphical diagrams.

Our decision to leverage SysML v2 as our collaborative pattern language is driven by its structural and semantic alignment, it gain of popularity in MBSE contexts. Initial experiments utilizing UML demonstrated a lack of formalism. Because UML is built upon a flexible, visually

[1] Example retrieved from `https://www.omg.org/pdf/SysML-v2-Overview.pdf`.

```
package example {
    part vehicle {
        attribute mass = engine.mass + transmission.mass;
        perform providePower;
        part engine {
            attribute mass;
            port torqueOutPort;
            perform providePower.generateTorque;
        }
        part transmission {
            attribute mass;
            port torqueInPort;
            perform providePower.amplifyTorque;
        }
        connect engine.torqueOutPort to transmission.torqueInPort;
    }

    action providePower {
        action generateTorque;
        action amplifyTorque;
    }
}
```

Listing 1: Example of the SysML v2 program used in Figure 3

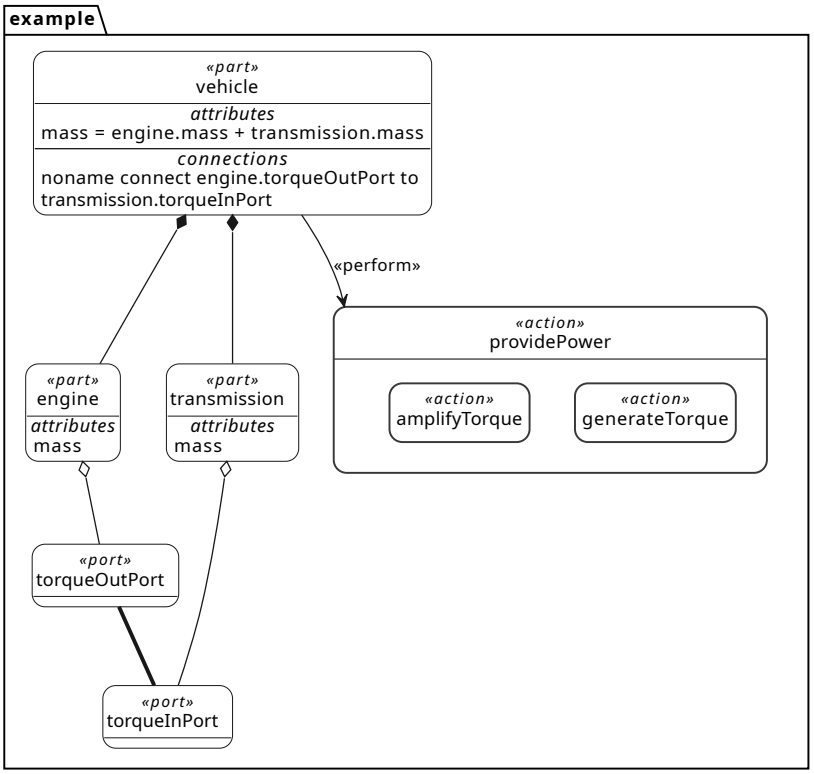


Figure 4: Same example as Listing 1 with the graphical syntax

oriented metamodel, it relies heavily on loose semantic constraints. Consequently, although LLM-generated models are overall syntactically correct, they frequently suffer from "structural hallucinations" i.e., producing architectures that were syntactically permissible in UML, but semantically invalid (De Bari et al., 2024).

Finally, as the newly OMG standard, SysML v2 is rapidly gaining popularity and adoption within the Model-Based Systems Engineering (MBSE) community. Aligning our methodology with this established standard ensures that engineers are not required to learn a novel, proprietary language, thereby lowering the barrier to entry and improving the overall accessibility of the framework. Hereafter, all references to SysML within this document denote SysML v2 unless explicitly stated otherwise.

Listing 2 shows a textual representation of the requirements management process. Unlike Figure 3, this one uses the textual form of the language, which is better suited for LLM processing. Like UML activity diagrams or BPMN, the example has a sequence of `actions` with a parallel section (between the `fork` and `join`) all organized around a `loop` action. Actions are then linked together using the `then` keyword.

2.3 The MOSAICO Collaboration Language

Since we only intend on using SysML v2 to represent workflows and task decomposition, we restrict our usage to a specific subset focused on behavioral workflows. For instance, we specifically utilize the `Action` and `Action Definition` constructs to model process steps, leveraging succession flows to represent temporal ordering rather than control logic alone.

The supported subset is divided into two primary categories: structural definitions (the agents) and behavioral definitions (the execution workflows).

2.3.1 Structural Modeling

We represent the agents that will execute the tasks by utilizing `Part` SysML v2 objects.

- **Agent Templates (`part def`):** we use part definitions to establish reusable templates for specific types of agents. By leveraging the `specializes` keyword, agents inherit potential behaviors and attributes from classes defined in our custom MOSAICO library, cf Section 2.3.3.
- **Property Redefinition (`redefines`):** we support redefinitions to assign specific identities or configurations to these templates, such as identifier strings.
- **Constraints (`constraint`):** Agents can be bounded by logical constraints, which the transformer translates into behavioral rules or system prompt injections.
- **Agent Instantiation (`part`):** concrete agent instances are declared globally within the package to populate the available agent pool for a given workflow, cf Section 3.2.

2.3.2 Behavioral Modeling

The core orchestration logic is captured using Action Definitions and specific control flow nodes. We restrict workflow modeling to the following constructs:

```
private import ScalarValues::*;

package RequirementManagementWorkflowNoParams {
    action def GenerateRequirementsProcess {
        first start;
        then action userInput {
            description = "Get a system description from the user.";
        }
        then loop action constructRequirementList {
            action decompose {
                description = "A new requirement is generated based on the system
                    description.";
            }
            then fork; then checkCorrectness; then checkNonRedundancy; then
                checkSatisfiability;
            action checkCorrectness {
                description = "Checks whether the new Requirement is correct based on
                    the system description.";
            } then joinNode;
            action checkNonRedundancy {
                description = "Checks that the new requirement is not redundant based on
                     the current list of accepted requirements";
            } then joinNode;
            action checkSatisfiability {
                description = "Checks whether the new requirement is satisfiable.";
            } then joinNode;
            then join joinNode;
            then action checkConsensus {
                description = "Tries to reach consensus among all checks.";
            }
            then action updateRequirements {
                description = "Updates the list of accepted requirements and checks
                    whether the list of accepted requirements is complete.";
                out isComplete : Boolean;
            }
        } until constructRequirementList.updateRequirements.isComplete;
        then action outputToUser {
            description = "Returns the list of completed requirements to the user.";
        }
        then done;
    }
}
```

Listing 2: Requirement management workflow in SysML v2

- **Task Definition (`action`):** atomic tasks are modeled as actions typed by `AgentStep` from our custom library. Within these blocks, the framework extracts the description, which serves as the LLM prompt, and the designated agent utilizing the redefinition operator (`:>>`).
- **Data Flows (`in`, `inout`, `out`):** Explicit task outputs are defined using typed parameters, establishing the data flows between sequential tasks.
- **Sequential Execution (`first`, `then`):** linear dependency graphs are established using standard SysML v2 succession flows. The `then` keyword indicates that the preceding action must fully resolve its output before the successor begins, except when specifying otherwise in case of parallel or loop workflow.
- **Parallel Execution (`fork`, `join`):** to model concurrent processing, our framework supports `fork` and `join` control nodes. Branches emanating from a `fork` execute in parallel until synchronized at a `join` node.
- **Iterative Execution (`loop action ... until` *condition*, `while` *condition* `action`):** cyclic refinement processes are supported via `loop` or `while` action constructs. The `loop` block encapsulates a sequence of sub-actions and dictates its termination via an `until` clause which evaluates a boolean expression.
- **Conditional Execution (`if`, `else`):** dynamic decision-making and workflow branching are implemented using `if` and `else` control structures. An `if` block evaluates a specified runtime condition, typically a boolean expression derived from the outputs of preceding actions. If the condition evaluates to true, the execution path is routed to the `then` branch; otherwise, control flow is directed to the optional `else` branch.

2.3.3 Additional MOSAICO Types

While the core SysML v2 metamodel provides the foundational grammar for system definition, it lacks the domain-specific semantics required to natively describe LLM-based agents and their orchestration behaviors. We thus developped the MOSAICO library, as shown in Listing 3, which serves as a domain-specific extension profile and defines a strict taxonomy of agents types and task execution patterns that our transformer and execution engine recognize.

At the core of the structural definition is the abstract `Agent` part. Rather than treating agents as black-box execution nodes, our library allows the definition of important metadata directly into the `Part` via attributes, e.g., *id*, *version*, *skills*, and *licences*. These metadata serve as an interface with the agent repository of Work Package 2 (WP2) and may evolve as the project matures. By embedding `licences` (e.g., "MIT", "GPL", ...) and `skills` (e.g., code generation, specific tool use, ...) into the SysML v2 model, the orchestrator can dynamically query, validate, and leverage appropriate agents from the WP2 repository at runtime.

From this base `Agent`, the library derives a specific taxonomy of specialized agents, each possessing redefined IDs and versions, which can be overridden in the user's SysML v2 program, for precise instantiation.

`ReferenceAgent`: handles user interactions.

`ConsensusAgent`: specializes in resolving conflicts between divergent LLM outputs.

`SupervisionAgent`: handles verification of LLM outputs.

`SolutionAgent`: handles primary task execution.

The behavioral segment of the library defines how these agents interact, or how the task is supposed to be executed. The base `Step` action is specialized into an `AgentStep`, which binds a `description` attribute, that will serve as the LLM prompt, to a potential executing `agent`. To support complex governance policies explored in WP4, we introduce specialized execution steps. These steps will allow the orchestrator to have a specific execution behavior without having to specify every concerned agents.

`ReferenceStep`: this action explicitly binds a `ReferenceAgent` to handle the task.

`ConsensusStep`: this action explicitly binds to a `ConsensusAgent` and introduces a `policy` attribute. The `policy` attribute dictates the programmatic or LLM-driven resolution mechanism (e.g., majority vote, unanimous agreement, or weighted critique) that the orchestrator must enforce to merge divergent branches.

`SupervisionStep`: this action explicitly binds a `SupervisionAgent` to handle the task and suggest the orchestrator to leverage a specific orchestration pattern for the specific subactions of this step.

`ReplicationStep`: to evaluate robustness or generate a diverse set of candidate solutions, the `ReplicationStep` allows the orchestrator to duplicate a task. By defining an array of agents, repetitions, and prompts, this step causes the system to execute parallel instances of the same logical task under varied conditions (e.g., different model temperatures or prompting strategies). This is an important orchestration pattern for WP4, enabling automated multiple generation and mitigating single-path hallucination risks.

2.4 Existing Datasets

To find relevant datasets, we started by looking at previous work on collaboration workflows for AI agents (Qiao et al., 2024). After identifying the main modeling languages, we explored existing datasets and studies that link workflows with natural language descriptions (Sola et al., 2022; M. Zuo et al., 2025). We also performed a broader search using general keywords to find additional relevant datasets and resources.

Table 1 presents the main datasets we identified. For each dataset, we indicate the type of instances (e.g., JSON/XML) and describe how the data was generated in the dataset generation column. We also include a column indicating whether the natural language is accompanied by a workflow representation (NL + Workflow). Finally, we highlight the main features provided by each dataset.

Overall, this process resulted in a large number of workflow models that can potentially be transformed into SysML code. However, we excluded PDDL datasets, as they are less relevant due to their limited expressiveness and their strong focus on robotics. We also excluded hdBPMN, a dataset of BPMN diagrams extracted from handwritten sources, which often results in medium to low quality diagrams due to extraction errors. We opted to include SAP-SAM for which we plan on generating description of the BPMN diagrams using LLMs.



```
package MOSAICO {
    private import ScalarValues::*;
    private import Collections::*;

    // STRUCTURAL DEFINITIONS
    part def Agent {
        attribute id : String[0..1];
        attribute version : String[0..1];
        attribute licences : String[0..*];
        attribute skills : String[0..*];
    }

    part def ReferenceAgent specializes Agent {
        redefines id = "eu.mosaico-project.reference-agent-type";
        redefines version = "1.0";
    }

    part def ConsensusAgent specializes Agent {
        redefines id = "eu.mosaico-project.consensus-agent-type";
        redefines version = "1.0";
    }

    part def SolutionAgent specializes Agent;

    part def SupervisionAgent specializes Agent;

    // BEHAVIORAL DEFINITIONS
    action def Step {
        attribute description : String[0..1];
    }

    action def AgentStep specializes Step {
        ref agent : Agent[0..1];
    }

    action def ReferenceStep specializes AgentStep {
        redefines agent : ReferenceAgent;
    }

    action def ConsensusStep specializes AgentStep {
        redefines agent : ConsensusAgent;
        attribute policy : String[0..1];
    }

    action def SupervisionStep specializes AgentStep {
        redefines agent : SupervisionAgent;
    }

    action def ReplicationStep specializes AgentStep {
        ref agents: Agent[0..*];
        attribute repetitions : Integer[0..*];
        attribute prompts : String[0..*];
    }
}
```

Listing 3: Additional MOSAICO specific data types.

Language	Dataset	Type	Dataset Generation	NL + Workflow	Size
SOP	SOPBench (Jun 2025)	SOP + tool graph (JSON)	GPT4 based generation with domain definitions and manually implemented functions	Yes	903
Graph Workflow	WorfBench (Feb 2025)	Workflow + action graph	**Tasks:** public datasets (ToolBench, WikiHow,) **Nodes:** public datasets + GPT-4 **Graph:** GPT-4 + filtering + human verification	Yes	20k
BPMN	SAP-SAM (Aug 2022)	BPMN (Signavio json format)	collection of models created by academic users (students, teachers, researchers)	No	631k
	hdBPMN (2021)	BPMN (xml)	BPMNs extracted from hand-drawn BPMN models	No	704
PDDL	Planeterium (Jul 2024)	PDDL code	PDDL is programmatically generated + template-based NL descriptions	Yes	145k
	Proc2pddl (jul 2024)	PDDL code	expert-annotated	Yes	Tens

Table 1: Datasets across different Modeling Languages

2.5 Transformation of datasets

Due to the current lack of publicly available, large-scale, and formally structured workflow datasets encoded in SysML v2, we opted to transform existing ones presented in Table 1 into SysML v2. This limitation is primarily explained by the recent emergence of the SysML v2 standard, which remains under active development and gradual adoption within both academic and industrial communities.

To address this gap, we designed fully deterministic pipelines that do not rely on LLMs. The decision to avoid LLM transformations was motivated by the need to guarantee reproducibility, and prevent potential hallucinations or semantic drift during the conversion process. Our approach operates by systematically extracting workflow primitives, such as the list of nodes representing actions, along with the dependency relations defining the execution flows.

During the transformation process, we generate SysML v2 code that includes the following supported elements:

- **Parts**: represent the structural entities of the system and are generally used in our context as agents participating in the workflow.
- **Actions**: represent the execution of a task within the workflow.
- **Forks and joins**: represent parallel execution, where forks split the control flow and joins synchronize them.
- **Decisions**: represent branching points in the workflow based on conditions. We do not translate condition expressions (which, in our source datasets, are typically given in natural language) into formal SysML v2 boolean conditions. Instead, we introduce an action that is intended to be handled by an LLM to evaluate these condition expressions.

The dataset is made available on Zenodo[2], and the transformation code is also available on Mosaico's GitLab[3] .

[2] https://zenodo.org/records/19346990

[3] https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/sysmlv2-workflow-construction

2.5.1 Transforming WorFBench

WorFBench (Qiao et al., 2024) is a benchmark designed to evaluate the ability of Large Language Models (LLMs) to generate graphical workflows from natural language instructions, represented as a list of nodes and a list of edges. The benchmark covers a diverse range of workflow generation tasks, including problem solving, function calling, embodied planning, and open-grounded planning. It comprises 18k training samples, 2146 test samples, and 723 held-out tasks intended to evaluate generalization. In addition, each workflow instance is paired with its corresponding user prompt, which serves as a natural language description of the target workflow.

To adapt this dataset to our modeling setting, we implemented a deterministic transformation pipeline that converts WorFBench workflow graphs into SysML v2 code. More precisely, the nodes of the original graph are interpreted as actions, while the edges are mapped to succession flows capturing the execution order between actions. This transformation converts each workflow graph into SysML v2 code while keeping the original order and dependencies between actions.

To assess the quality of the transformation, we applied two complementary validation steps. First, we manually inspected a subset of transformed instances by visualizing the generated SysML v2 models, in order to verify that the resulting code remained semantically consistent with the original workflows. Second, to validate the correctness of the generated models at scale, we used the SysML v2 pilot implementation[4] parser and PlantUML exporter to generate the graphical version of each SysML file we produced. Using the plantuml exporter has the nice property of validating the full file and allows us to have a second check that the workflow is correct.

Using this transformation process, we generated a dataset of approximately 17.8k SysML v2 workflow models derived from WorFBench. Table 2 illustrates a simple example of this transformation logic.

WorfBench Workflow	**SysML v2 Equivalent**

Prompt:
Given the four digits 2, 4, 6, and 7, how many different positive two-digit integers can be formed using these digits if a digit can be repeated in an integer?
 	
Nodes:
1 - Determine the number of choices for the first digit.
2 - Determine the number of choices for the second digit.
3 - Calculate the total number of different positive two-digit integers that can be formed.
 	
Edges:
(START,1)
(START,2)
(1,3)
(2,3)
(3,END)
	`package PackageName {`
	`  action def ProcessName {`
	`  first start;`
	`  then fork; then action1; then action2;`
	`  action action1 {`
	`  description = "Determine the number of`
	`  choices for the first digit.";`
	`  }`
	`  then joinNode3;`
	`  action action2 {`
	`  description = "Determine the number of`
	`  choices for the second digit.";`
	`  }`
	`  then joinNode3;`
	`  join  joinNode3;`
	`  then action action3 {`
	`  description = "Calculate the total number of`
	`  different positive two-digit integers that`
	`  can be formed.";`
	`  }`
	`  then done;`
	`  }`
	`}`

Table 2: Transformation example from WorBench workflow to SysML v2

[4] github.com/Systems-Modeling/SysML-v2-Pilot-Implementation/

2.5.2 Transforming SAP-SAM

SAP-SAM (Sola et al., 2022) is a large dataset that contains around 1.02M diagrams, including UML and BPMN models. In this work, we focus on transforming BPMN diagrams (about 631.4k in total) into SysML v2 code, under a set of defined conditions.

First, we only consider BPMN diagrams written in English. However, a large portion of the dataset does not have an assigned language. To address this, we use *langdetect*[5], an NLP-based package, to automatically detect the language. We classify a diagram as English if the aggregated descriptions of its tasks have a probability higher than 0.8 of being in English. This filtering step results in approximately 342k English BPMN diagrams.

Next, we restrict our selection to diagrams that use only a subset of BPMN elements that we can map to SysML v2, as previously defined. Applying this constraint reduces the dataset to about 56.6k diagrams, which constitutes the final set we aim to transform.

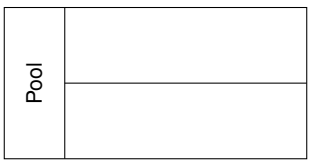
During the transformation process, several design decisions were necessary. BPMN is primarily intended for human interpretation, and therefore many elements, such as conditions, are expressed in natural language. In contrast, SysML v2 expects formal boolean conditions. To bridge this gap, we introduce dedicated actions that encapsulate these conditions and return a boolean result. For example, a BPMN condition such as “check if panel is not empty” is transformed into an action (e.g., `check_panel`) with the same description, which returns a boolean value used in decision nodes.

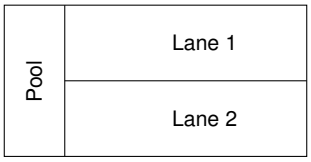
The mapping between BPMN elements and their corresponding SysML v2 representations is summarized in Table 3.

Using this transformation logic, we generated a dataset of approximately 56.2k SysML v2 workflow models derived from SAP-SAM.

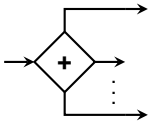
[5] `https://pypi.org/project/langdetect/`

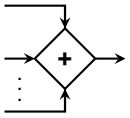
Description in BPMN	Graphical Representation	SysML v2 Equivalent

Pool represents a participant (organization/system) in the process		`package PoolName {`
		`  ...`
		`}`

Lane represents a role or responsibility within a pool		`part def Lane_1 specializes SolutionAgent { }`
		`lane_1 :  Lane_1 ;`
		
		`part def Lane_2 specializes SolutionAgent { }`
		`lane_2 :  Lane_12 ;`

Activity is generic term for work that company performs in process		`action open { ...  }`
Sequence Flow defines execution order between elements		`first A then B;`
Start Event indicates where a process begins		`action startEvent { ...  }` `first start then startEvent;`
End Event indicates where a process terminates		`action endEvent { ...  }` `first endEvent then done;`

Parallel Gateway (AND Split)		`fork forkNode1;`
		`first forkNode1 then A;`
		`first forkNode1 then B;`
		`first forkNode1 then C;`

Synchronization gateway (AND-Join)		`join joinNode1;`
		`first A then joinNode1;`
		`first B then joinNode1;`
		`first C then joinNode1;`

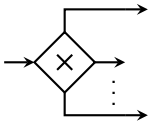
Exclusive Gateway (XOR) represents a decision point		`decide decisionNode1;`
		`first decisionNode1;`
		`if condition1 then A;`
		`if condition2 then B;`
		`else C;`

Table 3: Mapping between BPMN elements and their corresponding SysML v2 representations

2.5.3 Transforming SOP-Bench

SOP-Bench (Nandi et al., 2025) is a dataset designed to evaluate AI agents on realistic, multi-step workflows expressed as Standard Operating Procedures SOPs (Nandi et al., 2025). The transformation of this dataset is similar to what we observe in WorFBench, as both rely on a list of actions and an action graph representation. We successfully transformed 903 SOP-Bench models into SysML v2.

However, SOP-Bench introduces an additional control structure: an OR node. This OR represents alternative execution paths, meaning that action A or action B can be performed.

To handle this in our transformation, we adopt a sequential decision strategy. We first execute action A, then introduce a decision node that checks whether A succeeded. If it succeeds, the workflow continues; otherwise, the alternative path (action B) is executed. This process can be extended to multiple alternatives. Table 4 illustrates a simple example of this mapping logic.

SOP-Bench Workflow	**SysML v2 Equivalent**

Prompt:
Hello! I would like to transfer an amount of $200.00 from my account to another user's account. Here are the details:
- Destination Username: jane_doe
- Amount: 200.0
- Unit: dollars
- Username: alice_smith
- Identification: alicesecurepassword
Additionally, if there are any functionalities that require it, I also have my admin password available: .. Could you assist me with processing this transfer?
 	
Nodes:
0. transfer_funds
1. and
2. or
3. or
4. internal_check_username_exist (dest_username)
5. get_account_balance (username)
6. and
7. login_user
8. or
9. internal_check_username_exist (username)
10. internal_get_database
 	
Edges:
0 -> 1
1 -> 2
1 -> 3
1 -> 7
1 -> 8
2 -> 4
2 -> 10
3 -> 5
3 -> 10
5 -> 6
6 -> 7
6 -> 8
8 -> 9
8 -> 10
	`package bank {`
	
	`action def transfer_funds {`
	
	`part def WorkflowAgent specializes SolutionAgent {`
	`}`
	
	`workflowAgent : WorkflowAgent;`
	
	`\\actions definitions omitted to reduce space`
	`fork forkNode_1;`
	
	`join joinNode_5;`
	
	`join joinNode_7;`
	
	`decide DecisionAfter_6;`
	
	`decide DecisionAfter_7;`
	
	`first start then transfer_funds;`
	`first transfer_funds then forkNode_1;`
	`first forkNode_1 then authenticate_admin_password;`
	`first forkNode_1 then joinNode_5;`
	`first forkNode_1 then internal_check_username_exist;`
	`first forkNode_1 then joinNode_7;`
	`first authenticate_admin_password then joinNode_5;`
	`first joinNode_5 then login_user;`
	`first internal_check_username_exist`
	`then DecisionAfter_6;`
	`first DecisionAfter_6;`
	`if internal_check_username_exist.success`
	`then joinNode_7;`
	`else done;`
	
	`first joinNode_7 then internal_get_database;`
	`first internal_get_database then DecisionAfter_7;`
	`first DecisionAfter_7;`
	`if internal_get_database.success`
	`then internal_check_username_exist_8;`
	`else done;`
	
	`first login_user then done;`
	`first internal_check_username_exist_8 then done;`
	`}`
	`}`

Table 4: Transformation example from SOP-Bench workflow to SysML v2

Table 5 provides a full recap of the transformation of the 3 datasets, including the transformed portion and the method used to obtain the natural language description of each model. In total, we generated approximately 75k SysML v2 workflows.

Language	Source Dataset	Initial Size	Generated Dataset Size (Transformation Rate)	Natural Language Description
Graph	WorfBench (Feb 2025)	18k	17.8k (98.84%)	User prompt and list of actions
BPMN	SAP-SAM (Aug 2022)	Total : 1.02M BPMN : 631.4k English : 367.4k Filtered : 56.6k	56.2k (99.36%)	Generated by providing an LLM with the list of BPMN elements
SOP	SOPBench (Jun 2025)	0.9k	0.9k (100%)	Process description
Total	-	75.1k	74.9k (99.73%)	-

Table 5: Summary of source datasets and their transformation into SysML v2

2.6 Metrics for the MOSAICO Collaboration Language Dataset

To assess the quality and the richness of the generated dataset, we first inspect the frequency of each of the following elements (*action, decision, fork*). Action gives an idea about the minimal usefulness of the workflow. Decision and fork give an idea about the complexity and non-linearity of the workflow.

In the Table 6, the metrics are divided into two sections: per model and global. Per model represents the presence of the element in the model. An action having a 1.0 frequency per model means it is present in all models, which is logical. Global gives the percentage of each element out of all elements; for instance, in SOP-Bench, the action frequency is 0.46, which means that 46% of all elements in all workflows are actions.

	Per Model			Global		
Element	Action	Decision	Fork	Action	Decision	Fork
SOP-Bench	1.0	0.81	0.88	0.46	0.19	0.15
WorfBench	1.0	0.00	0.15	0.92	0.00	0.04
SAP-SAM	1.0	0.6	0.28	0.82	0.11	0.03

Table 6: Frequency of SysML v2 elements in the dataset

In Table 7, we present the minimum, maximum, and mean of each element kind per model. We observe that each workflow contains at least one action, and that datasets have on average between 4 and 9 actions per model. SAP-SAM contains much larger models, with up to 273 actions and 109 decisions at maximum. WorfBench on the other hand contains no decision because its graph notation does not distinguish between action and control flow nodes.

Element	Action			Decision			Fork		
	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
SOP-Bench	1	5.321	10	0	2.268	9	0	1.767	4
WorfBench	1	3.821	15	0	0	0	0	0.168	12
SAP-SAM	1	8.955	273	0	1.248	109	0	0.35	30

Table 7: Minimum, mean, and maximum count of SysML v2 elements in the dataset

3 Orchestration Framework

This section presents, in Section 3.1, several orchestration patterns found in the industry and literature and, in Section 3.2, the overall strategy deployed in MOSAICO to enact SysML collaboration patterns.

3.1 Existing Orchestration Patterns

Deploying multiple independent agents without rigorous coordination inevitably leads to execution problems, e.g., infinite feedback loops, redundant computations, or contradictory outputs. Orchestration refers to the structural topology that dictates how agents are activated, how they communicate, and how their distinct outputs are transformed into a coherent final result. Picture LLM agents as compute nodes of a reasoning engine: orchestration patterns would then function as the underlying operating system, managing task routing, memory allocation, context sharing, and error handling. Recent studies suggest that as underlying LLMs converge in capabilities, the architectural design of this orchestration topology has become the primary driver of system-level performance (G. Yu, 2026).

3.1.1 Existing Orchestrators

To manage complexities of task decomposition, inter-agent communication, and state management, various platforms have emerged in recent years, which are presented in the following section. Each orchestrator operates with distinct execution topologies and interaction modalities, these concepts are detailed in Section 3.1.2.

AutoGen (Q. Wu et al., 2024) enables the creation of complex LLM applications by abstracting models, tools, and human inputs into customizable, conversable agents that interact through automated chat. **MetaGPT** (Hong et al., 2023) structures multi-agent workflows like an assembly line, translating human standardized operating procedures into prompt sequences, assigning specialized roles to agents to effectively divide tasks and reduce errors through intermediate verification. **ChatDev** (Qian et al., 2024) functions as a virtual software company where communicative agents follow a chain-structured workflow to actively collaborate and reduce hallucinations across the design, coding, and testing phases. **CrewAI** (Taulli et al., 2025) optimizes for collaborative intelligence by organizing agents with specific goals, backstories, and tools into role-playing "crews" that autonomously execute tasks using the ReAct prompting pattern. **Google ADK** (Google, 2026) provides a modular architecture that structures multi-agent systems into hierarchical pipelines, using specialized sequential, parallel, and loop workflow agents to predictably orchestrate execution flows. **Camunda** (Camunda, 2026) extends

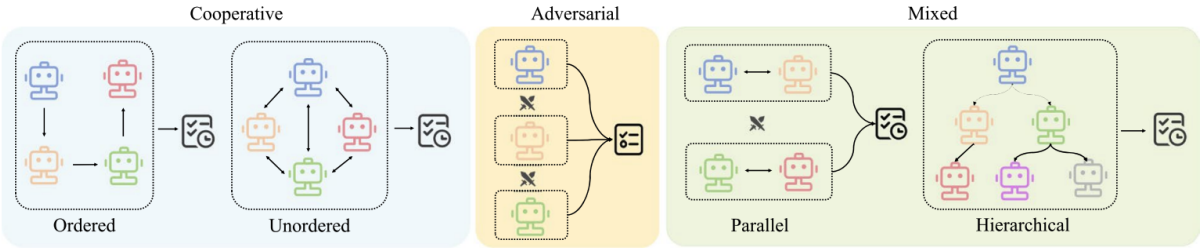


Figure 5: Agent interaction modality which lacks the debate modality (X. Li et al., 2024)

traditional business process management into "agentic orchestration" by embedding AI agents within BPMN ad-hoc sub-processes, allowing LLMs to dynamically select and execute tools while maintaining deterministic enterprise guardrails and human-in-the-loop oversight.

3.1.2 Recurring Orchestration Patterns

The analysis of the current multi-agent systems implementations from the previous section revealed recurring orchestration patterns that align with established architectural paradigms. To formally classify these systems, we delineate three core dimensions: interaction modality, execution workflow, and execution topology. First, the **interaction modality** defines the relational dynamics and goal alignment between agents, partially illustated by Figure 5:

Cooperative: Agents operate as a unified collective with aligned objectives, sharing context and resources to achieve a singular goal (Das et al., 2023; He et al., 2023; Hong et al., 2023; Zhefan Wang et al., 2024).

Adversarial: Agents are assigned conflicting objectives to iteratively pressure-test solutions. For instance, in a cybersecurity scenario designed to expose system vulnerabilities, a *Red Team* agent is tasked with compromising the system, while a *Blue Team* agent actively defends and patches the identified flaws (Zhao et al., 2023).

Debate: Agents act as specialized peers with distinct perspectives. They engage in multi-round argumentation to critique proposed solutions, correct reasoning fallacies, and ultimately converge on a consensus output (Abdelnabi et al., 2024; C.-M. Chan et al., 2023; Davidson et al., 2024; Du et al., 2024; Liang et al., 2024).

Second, the **execution workflow** dictates the temporal sequencing of tasks:

Sequential: Tasks are processed in a strict, linear dependency graph, where the output of one agent serves as the direct input for its successor.

Parallel: Agents or tasks operate concurrently and independently, optimizing for time efficiency and allowing for a broader exploration of the solution space.

Third, the **execution topology** determines the structural hierarchy and control flow of the system, as illustrated by Figure 6 where the star, tree, graph-mesh, and chain structures are respectively classified as centralized, hierarchical, peer-to-peer, and handoff topologies in our proposed taxonomy:

Handoff: An agent gives control of the conversation or the remaining task entirely to a successor agent better suited for the context, terminating its own involvement without further inter-agent communication (B. Lin et al., 2025).

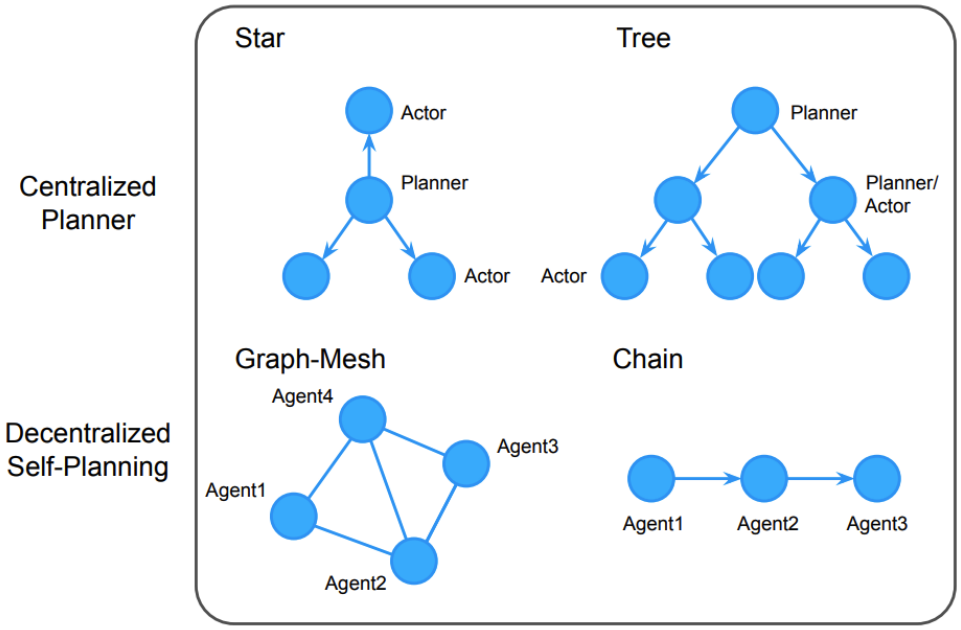


Figure 6: Illustration of different agent execution topologies (K. Zhu et al., 2025).

Peer-To-Peer: Agents operate as equals or specialized partners, dynamically negotiating tasks and sharing results without a strict central manager (G. Chen et al., 2023; W. Chen et al., 2023).

Centralized: A single orchestrator agent acts as the primary controller, monopolizing collaboration decisions, routing sub-tasks, and synthesizing final outputs (Bamouh et al., 2025; Pan et al., 2025).

Hierarchical: Agents are arranged in a layered systems with distinct roles and level of authority (Z. Liu et al., 2024; Qian et al., 2024).

These three dimensions, i.e., modality, workflow, and topology, can be hybridized to instantiate specific orchestration patterns. For example, the *consensus pattern* requires multiple agents to independently generate solutions to the same problem before a secondary mechanism (such as a majority vote or a designated "*judge*" agent) evaluates and selects the optimal answer. This specific pattern synthesizes a debate interaction modality, a sequential workflow, and a peer-to-peer execution topology.

At runtime, the MOSAICO orchestrator serves as the central manager, responsible for both the interaction modalities and execution workflows of the multi-agent system. While this architecture inherently defaults to a centralized execution topology, ongoing development aims to support the emulation of alternative orchestration patterns, such as peer-to-peer or hierarchical structures. In these advanced configurations, the orchestrator will function as a transparent middleware layer, dynamically routing context and managing state transitions without acting as a monolithic bottleneck, while simultaneously providing adaptive fault tolerance to detect and mitigate execution errors at runtime.

3.2 MOSAICO Collaboration Language Runtime Interpretation

As explained in Section 2.3, the MOSAICO orchestrator uses a subset of SysML v2 to express the collaboration patterns.

3.2.1 SysML v2 Language Parser

In order to build SysML v2 compliant orchestrator we opted to leverage the open source SysML v2 Pilot Implementation[6] which includes a parser for the SysML v2 textual syntax. Being an official project from the Systems Modeling Community which is involved in the standardization process of SysML, it is kept up to date with incremental changes to the standard and has a permissive license compatible with our project. This parser is written in Xtext[7] and uses EMF[8] which are both well known tools from the Java modeling community.

Because SysML v2 is a broad language which can model many facets of software and system modeling, it possesses many concepts. In its latest version[9], the SysML v2 metamodel is composed of 175 classes and 7 enumerations. Figures 7 gives a partial overview of the class hierarchy, this view hint at the complexity of the SysML v2 hierarchy and the need for a simpler, smaller and specialized internal representation of the concepts our orchestrator uses.

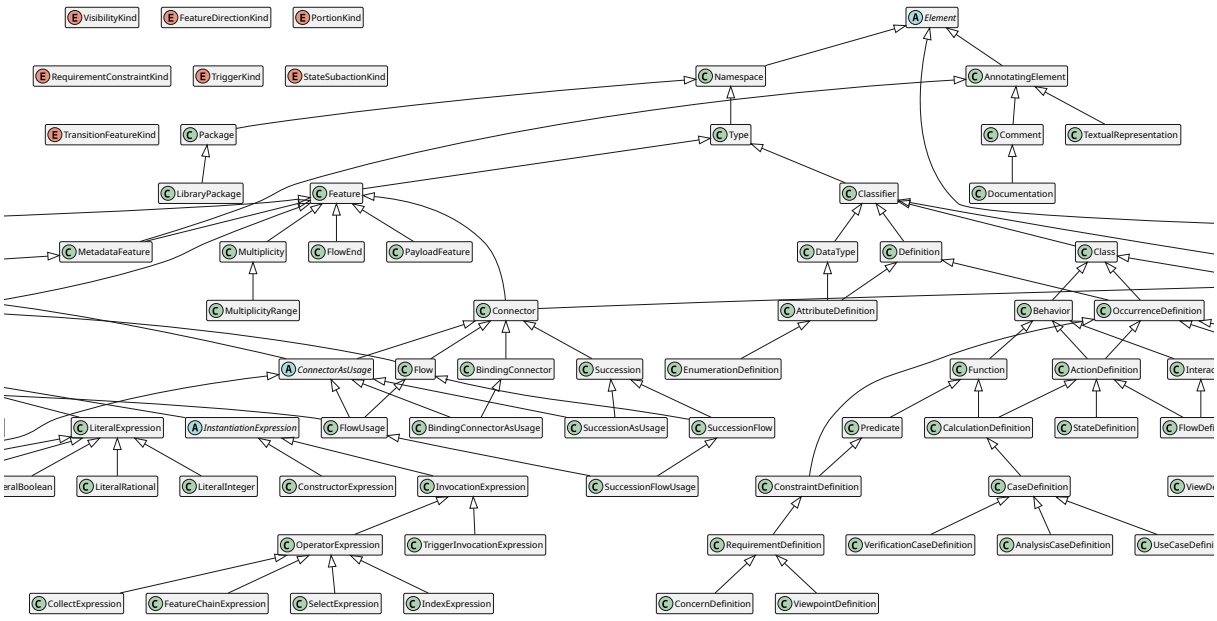


Figure 7: Zoomed part of the hierarchy of class of SysML v2

As explained in Section 2.3, collaborations patterns in MOSAICO use a subset of SysML v2 corresponding mostly to the *Action* concept. A more detailed presentation of the subset of SysML supported by the orchestrator is presented in paragraph 3.2.2.2. Listing 4 refines the Listing 2 example by using the MOSAICO orchestration concepts presented in Section 2.3.3 to represent the different tasks to perform while specifying the performing agents when desired, and using action data parameters to model the data flows between them.

```
package RequirementManagementAgentsNoParams {
  private import ScalarValues::*;
```

[6] https://github.com/Systems-Modeling/SysML-v2-Pilot-Implementation/
[7] https://eclipse.dev/Xtext/
[8] https://eclipse.dev/emf/
[9] Release 2026-02

```
private import SequenceFunctions::*;
private import MOSAICO::*;
  // Agents Specification
part def RequirementManagerAgent specializes SolutionAgent {
  :>> id = "eu.mosaico-project.requirement-manager-agent-type.v1";
  constraint { licences->includes("MIT") }
}
part def CheckCorrectNessAgent specializes SupervisionAgent {
  :>> id = "eu.mosaico-project.check-correctness-agent-type.v1";
}
part def CheckNonRedundancyAgent specializes SupervisionAgent {
  :>> id = "eu.mosaico-project.check-non-redundancy-agent-type.v1";
}
part def CheckSatisfiabilityAgent specializes SupervisionAgent {
  :>> id = "eu.mosaico-project.check-satisfiability-agent-type.v1";
}
  // Agent Instanciation
referenceAgent : ReferenceAgent;
requirementManagerAgent : RequirementManagerAgent;
checkCorrectnessAgent : CheckCorrectNessAgent;
checkNonRedundancyAgent : CheckNonRedundancyAgent;
checkSatisfiabilityAgent : CheckSatisfiabilityAgent;
consensusAgent : ConsensusAgent;
action def GenerateRequirementsProcess {
  first start;
  then action userInput : AgentStep {
    :>> description = "Get a system description from the user.";
    :>> agent = referenceAgent; // Agent assignment
          out systemDescription : String;
          out requirements : String;
  }
  then loop action constructRequirementList {
          in systemDescription : String;
          inout requirements : String;
    action decompose : AgentStep {
      :>> description = "A new requirement is generated based on the system
          description.";
      :>> agent = requirementManagerAgent;
             in systemDescription : String;
             out newRequirement : String;
    }
    then fork;
      then checkCorrectness;
      then checkNonRedundancy;
      then checkSatisfiability;
    action checkCorrectness : AgentStep {
      :>> description = "Checks whether the new Requirement is correct based on the
          system description.";
             in newRequirement : String;
             in systemDescription : String;
             out isCorrect : Boolean;
    }
    then joinNode;
    action checkNonRedundancy : AgentStep {
      :>> description = "Checks that the new requirement is not redundant based on
          the current list of accepted requirements";
```

```
            in newRequirement : String;
            in systemDescription : String;
            in requirements : String;
            out isNonRedundant : Boolean;
      }
      then joinNode;
      action checkSatisfiability : AgentStep {
        :>> description = "Checks whether the new requirement is satisfiable.";
            in newRequirement : String;
            in systemDescription : String;
            in requirements : String;
            out isSatisfiable : Boolean;
      }
      then joinNode;
      join joinNode;
      then action checkConsensus : ConsensusStep {
        :>> agent = consensusAgent;
            in isCorrect : Boolean;
            in isNonRedundant : Boolean;
            in isSatisfiable : Boolean;
            out allValid : Boolean;
      }
      then action updateRequirements : AgentStep {
        :>> description = "Updates the list of accepted requirements and checks whether
             the list of accepted requirements is complete.";
        :>> agent = requirementManagerAgent;
            inout requirements : String;
            in newRequirement : String;
            in systemDescription : String;
        out isComplete : Boolean;
      }
    } until constructRequirementList.updateRequirements.isComplete;
    then action outputToUser : AgentStep {
      :>> description = "Returns the list of completed requirements to the user.";
      :>> agent = referenceAgent;
          in requirements : String;
    }
    then done;
  }
}
```

Listing 4: Refined version of Listing 2 with agents assignment and data parameters.

3.2.2 MOSAICO Orchestrator Work In Progress

The following subsections detail the core components of this orchestration architecture, which remains under active development. First, we introduce a custom intermediate representation utilizing Data Transfer Objects (DTOs) that effectively act as an execution-optimized Domain-Specific Language (DSL). Second, we detail the Transformer engine, which recursively traverses the SysML AST to map structural and behavioral nodes into these executable DTOs. Finally, we present the workflow execution environment, outlining our current deterministic transformation process, and our current simulation strategy used to validate graph traversal, dependency resolution, and control flow logic with minimal integration with live LLMs.

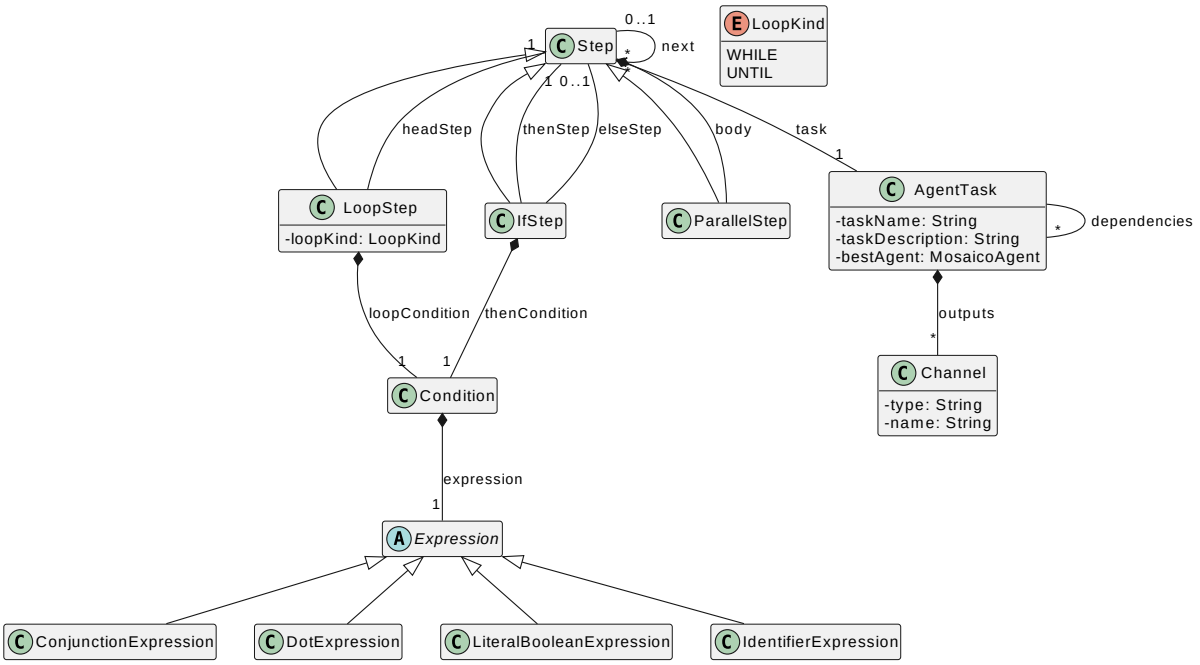


Figure 8: Class diagram of our custom DTOs.

3.2.2.1 Custom DTOs

While SysML v2 provides a robust formal syntax for defining system architectures, directly executing the parsed object structures proved highly inefficient. The objects generated by the SysML parser are the product of multiple, interlocking grammar rules. Thus, the resulting AST yields data structures that are deeply nested, verbose, and tightly coupled to the SysML specification rather than to runtime execution logic. Relying directly on these raw parser products would require the orchestration engine to constantly navigate complex object graphs, increasing overhead and the risk of execution errors.

To decouple the execution engine from the SysML metamodel complexity, we introduced a custom intermediate representation using DTOs. This abstraction layer simplifies the dense SysML models into streamlined execution plans, effectively acting as an embedded DSL tailored specifically for multi-agent orchestration. Furthermore, this decoupled architecture ensures framework extensibility, facilitating the future integration of alternative collaboration pattern languages, such as BPMN.

The custom DTO structure is designed around a composite pattern that supports both atomic tasks and nested workflows, mirroring the execution topologies discussed in Section 3.1.2. The data model consists of the following constructs and is illustrated by Figure 8:

- `Step`**:** rather than relying on a monolithic execution plan, workflows are constructed as a directed graph of `Step` objects. A standard `Step` encapsulates a single `AgentTask` and maintains a `next` reference to the subsequent step, forming a sequential execution chain.
- `AgentTask`**:** the atomic unit of work assigned to a specific LLM agent. By decoupling the task logic from the structural `Step`, the framework isolates the semantic execution payload. An `AgentTask` contains:
  - `taskName` **and** `taskDescription`**:** the semantic payload and user prompt instruction directed at the agent.

  - `bestAgent`: the specific, specialized LLM agent assigned to resolve the task.
  - `dependencies`: a collection of preceding `AgentTask` nodes. This ensures the agent receives the necessary contextual output from prior steps before its own execution directly through the user prompt.
  - `outputs`: a collection of `Channel` objects (defined by a name and a type) that formalize the expected data schema generated by the agent.

- `ParallelStep`: enables simultaneous execution by containing a `body` comprised of multiple independent `Step` instances.
- `LoopStep`: enables iterative workflows. It contains a `headStep` to execute, is governed by a `Condition`, and utilizes a `LoopKind` enumeration (`WHILE` or `UNTIL`) to control the cyclic reasoning and prevent infinite execution loops. The `headStep` may have a `next` step which is also executed and repeated by the loop, until the final step which has no next.
- `IfStep`: enables dynamic decision nodes into the workflow. It evaluates a `thenCondition` to route the execution path to a `thenStep` based on runtime context.
- `Condition` **and** `Expression`: a `Condition` aggregates one `Expression` object, which functions as an embedded AST for runtime predicates. The `Expression` hierarchy supports composite logic (`ConjunctionExpression`), object property traversal (`DotExpression`), literal values (`LiteralBooleanExpression`), and variable references (`IdentifierExpression`). This guarantees that the orchestrator can safely and deterministically evaluate complex criteria without executing arbitrary code. We also plan to support disjunction expressions and other features identified through user needs, all of which will be detailed in Deliverable 3.3.

3.2.2.2 SysML v2 Transformer

To convert the strict, grammatical constructs of the SysML v2 AST into out executable DTOs, we developed a custom transformation engine which is illustrated by Table 8. Since SysML v2 allows for deep nesting and redefinition of properties, a single-pass extraction may be insufficient. The transformer uses a recursive, multi-pass traversal strategy to resolve dependencies before building the execution plan. The transformation process operates in two primary phases:

1. **Resource Discovery and Agent Instantiation:** the engine first scans the SysML program contents (`Package`) to identify structural agent templates (`PartDefinition`) and their referential usages within the workflow (`ReferenceUsage`). During this phase, the `PartMapper` component evaluates subclassifications to instantiate the correct domain-specific agent (e.g., *ConsensusAgent*, *SupervisionAgent*, ...) and recursively extracts redefined properties such as the agent's ID, description, and behavioral constraints.
2. **Control Flow Resolution:** once the agents are established, the engine identifies the root `SuccessionAsUsage` elements. The `FlowMapper` then traverses the directed graph to construct the different `Step` objects.

The `FlowMapper` component dynamically determines the required execution topology based on the SysML control nodes encountered during traversal:

- **Sequential Flows:** standard `SuccessionAsUsage` links between distinct actions generate a sequential execution. The parser links the preceding node to its successor via the `next` attribute of the base `Step` class.

- **Parallel Forks:** when a `ForkNode` is encountered, the mapper spawns a `ParallelStep`. The engine traces all outgoing branches concurrently until a corresponding `JoinNode` is reached, encapsulating the different actions within the `body` collection of the `ParallelStep`.
- **Iterative Loops:** encounters with `LoopActionUsage` or `WhileLoopActionUsage` generate a `LoopStep`. The encapsulated actions are mapped into the `body` of the loop, while the termination logic is explicitly parsed into a `LoopCondition` (defined by `WHILE` or `UNTIL LoopKind`).
- **Conditional Branches:** SysML decision nodes or guarded transitions are transformed into an `IfStep`. The parser translate the guard logic to generate a `thenCondition`, mapping the divergent execution paths to the `thenStep` and the optional `elseStep` references.

The `ActionMapper` component translates SysML v2's `ActionUsage` nodes into our `AgentTask` DTOs. This involves a deep traversal of the element's relationships to extract specific metadata, such as string literals representing the task description and feature reference expressions indicating the required agent. The mapper then binds the newly formed `AgentTask` to the pre-instantiated `MosaicoAgent` pool based on matching names. If no agent was specified to perform the task, then the engine falls back to the instantiation of a generic `SolutionAgent`.

To support the dynamic routing required by the `LoopStep` and `IfStep`, the transformer translates SysML constraints and Object Constraint Language (OCL) snippets into a safe, executable Expression tree. Instead of relying on arbitrary string evaluation at runtime, the parser decomposes SysML logic into discrete AST nodes—such as `ConjunctionExpression` (for logical AND), `DotExpression` (for object property traversal), and `IdentifierExpression` (for variable references). These expressions are aggregated into a `Condition` object, guaranteeing deterministic and secure runtime evaluation by the orchestrator.

It is important to note that this framework is under active development and that the architecture described in this section may evolve. The finalized architecture, alongside any necessary adaptations, will be thoroughly presented in Deliverable 3.3.

Specifically, future work aims to broaden the supported SysML v2 subset by incorporating assignment actions, enabling external data initialization for global variables, and handling multi-valued data types like arrays and collections. As the framework expands to support a broader subset of SysML v2, the custom DTOs may be extended with new `Step` specializations to capture and execute these new concepts. We also plan on adapting execution based on the MOSAICO library types from Listing 3, for instance a `ReferenceStep` implies that there will be a direct interaction with a human, and thus they may be a back-and-forth conversation that may change runtime context before being able to pass to the next step.

SysML v2 AST Element	Orchestration DTO	Architectural Purpose
`PartDefinition`	`MosaicoAgentType`	Defines the structural template, capabilities, skills, and constraints of an LLM agent.
`ReferenceUsage`	`MosaicoAgent`	Instantiates a specific agent to participate in the collaboration.
`SuccessionAsUsage`	`Step.next`	Defines a sequential execution of a set of tasks.
`ForkNode/JoinNode`	`ParallelStep`	Defines a parallel execution of a set of tasks.

`WhileLoopActionUsage`	`LoopStep`	Defines an iterative execution ended by a condition.
`TransitionUsage`	`IfStep`	Defines the conditional execution of a set of tasks depending on runtime context.
`ActionUsage`	`AgentTask`	Represents an atomic unit of work assigned to a specific agent.
`Redefinition`	`Task/MosaicoAgent` properties	Specifies properties defined in a SysML v2 program.
`ConstraintUsage`	`Condition/MosaicoAgent` constraints	Represents constraints via OCL.
`Feature` from `ActionUsage.getOutput()`	`Channel`	Defines the expected outputs, meaning names and types, of each `AgentTask`.

Table 8: Mapping of SysML v2 elements to custom orchestration DTOs

3.2.2.3 Workflow Execution

The execution first begins by transforming the SysML v2 program describing the workflow into our DTO architecture. Let's take the example from Listing 4, Figure 9 illustrates the resulting in-memory graph of the translation process. As shown, the sequential `first` and `then` clauses are successfully resolved into a linked chain of `Step` objects. The sequence begins directly with the initial `userInput` task, as the parser intentionally omits the default `start` and `done` control nodes native to SysML v2.

Furthermore, the parser correctly translates complex execution topologies: the SysML v2 loop action is instantiated as a `LoopStep` governed by a deeply parsed dot-expression `Condition`, while the `fork` and `join` nodes successfully encapsulate the three concurrent tasks within a unified `ParallelStep`. This resulting graph confirms that both the semantic payloads, e.g., agent assignments and task descriptions, and the structural execution flow are correctly preserved and ready for the orchestrator's processing.

To evaluate the framework's viability with live LLMs, the SysML v2 action descriptions were refined to better serve as prompt instructions, as detailed in Table 9. This proof-of-concept validates SysML v2 as an effective agentic orchestration language. For this demonstration, we utilized a straightforward palindrome verification system since the primary objective was to confirm the orchestrator's ability to interface successfully with an LLM, maintain execution context, and accurately parse responses.

At runtime, the Collaboration Agent acts as the central orchestrator, consuming the initial `Step` and dynamically routing control flow based on the node type. For a standard `Step`, the underlying task is executed sequentially before advancing to the `next` node. For a `ParallelStep`, the constituent tasks within its body are dispatched concurrently. The orchestrator suspends the parent execution thread until all nested branches, including their subsequent `next` steps, have fully resolved. When encountering an `IfStep`, the engine evaluates the runtime condition to determine whether control flow routes to the `then` branch or the optional `else` branch. Finally, for a `LoopStep`, the orchestrator traverses the encapsulated `headStep` and traverses its potential `next` node. This recursive graph traversal continues until a leaf node with no `next`

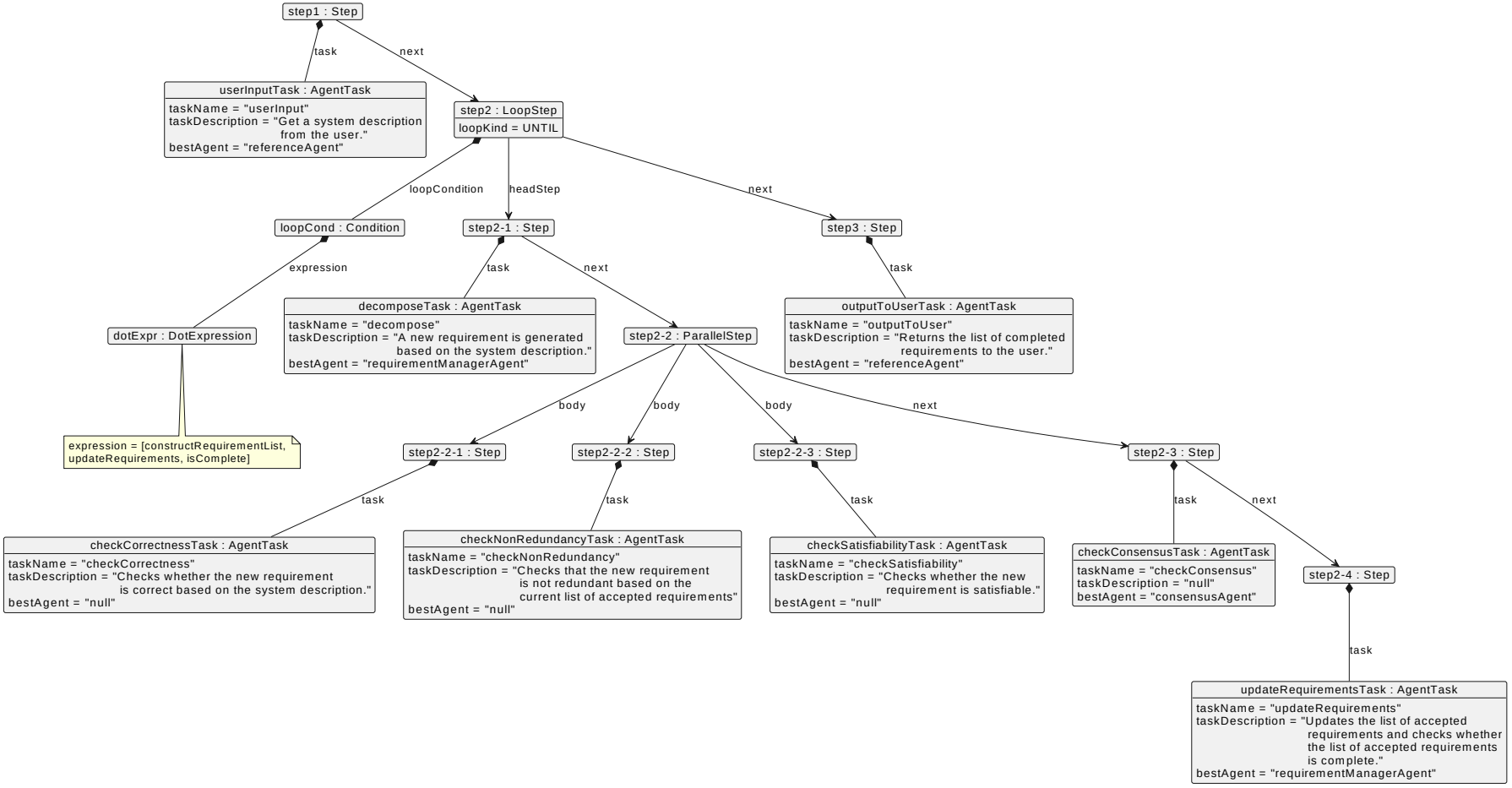


Figure 9: Object Diagram illustrating the transformation from the SysML v2 program of Listing 4 into our custom DTOs.

reference is reached, the termination condition is then evaluated to dictate whether the iteration repeats or exits.

During atomic task execution, the designated LLM receives a prompt comprising the task description combined with the resolved outputs of its prerequisite dependency channels. Upon completion, the system generates a `TaskOutput` object that strictly binds the LLM's response to the expected data type. While the current implementation restricts these types to `Boolean` and `String`, this ensures deterministic data contracts. A trace of the resulting channel execution is presented in Listing 5, which provides a clear, chronological record of the data flowing between your LLM agents during a single, successful iteration of the requirement generation loop.

Action Name	**New Description**
`userInput`	For the `systemDescription` channel, output a description for a system that verifies whether a given word is a palindrome. For the `requirements` channel, respond with an empty string.
`decompose`	Generate a single, atomic requirement based on the provided system description. Ensure the requirement is narrow in scope and covers only one specific aspect of the system, as this prompt will be executed iteratively to build the complete requirement set over multiple passes.
`checkConsensus`	Evaluate the results of the preceding validation steps. Return `true` if all incoming validation checks are true; otherwise, return `false`.
`updateRequirements`	If the `allValid` channel evaluates to true, append the `newRequirement` value to the existing `requirements` string and output the fully rewritten, updated list. Next, evaluate this updated list against the `systemDescription`. Output `true` to the `isComplete` channel if the requirements now comprehensively cover all aspects of the system description; otherwise, output `false`.

Table 9: Refined SysML v2 action descriptions adapted as explicit prompt instructions to guide the LLM agents during the palindrome verification proof-of-concept.

3.2.3 Communicative acts

In D3.1 we described how *performatives* (or *illocutions*) are used in messages between agents to describe the nature of the communicative act. In particular we used two performatives: `tell` and `achieve` to distinguish messages that convey information from those which specify a request to do some task. For instance, if we consider the *content* of a message to be *"The program is correct."*, the performative attached to the message indicates if the meaning is *"I think that the program is correct"* (`tell`) or *"I want you to make something to make the program correct"* (`achieve`).

```
userInput.channel:systemDescription (String) = StringValue{value='This system
verifies whether a given word is a palindrome by comparing the original word
to its reverse. It processes the input by stripping any spaces and ignoring
case sensitivity, ensuring accurate comparisons. The system returns a boolean
value indicating whether the input word reads the same forwards and backwards.'},

userInput.channel:requirements (String) = StringValue{value='""'},

decompose.channel:newRequirement (String) = StringValue{value='The system shall
return a boolean value indicating whether the input word is a palindrome by
comparing the processed version of the input (with spaces stripped and case
ignored) to its reversed version.'},

checkNonRedundancy.channel:isNonRedundant (Boolean) = BooleanValue{value=true},

checkSatisfiability.channel:isSatisfiable (Boolean) = BooleanValue{value=true},

checkCorrectness.channel:isCorrect (Boolean) = BooleanValue{value=true},

checkConsensus.channel:allValid (Boolean) = BooleanValue{value=true},

updateRequirements.channel:requirements (String) = StringValue{value='The system
shall return a boolean value indicating whether the input word is a palindrome
by comparing the processed version of the input (with spaces stripped and case
ignored) to its reversed version.'},

updateRequirements.channel:isComplete (Boolean) = BooleanValue{value=true}
```

Listing 5: Stack trace of all generated channels of a single loop iteration execution.

Attaching performatives to messages helps to specify or understand the role of the message in the orchestration without knowing the content of the message. Therefore, it is useful when we generate a collaboration pattern (to specify the structure of the workflow), to disambiguate the content of some messages (when two agents communicate), or to check that an agent has a correct behavior (supervision).

Standardized Communicative Acts : KQML

Our use of those two performatives, `tell` and `achieve`, was inspired by their use in AgentSpeak, described also in D3.1. The set of performatives which are available in AgentSpeak is a subset of those of KQML Agent Communication Language (Finin et al., 1993). KQML contains 40 different performatives, listed in (Vidal, 2009), among which some are direclty relevant to MOSAICO and some are less useful.

FIPA-ACL is another set of performatives dedicated to communication between agents. We found that it suits better the need of MOSAICO. Next section describes FIPA-ACL.

Standardized Communicative Acts : FIPA-ACL

The Foundation for Intelligent Physical Agents (FIPA) has defined more recently an Agent Communication Language (ACL) together with a library of performatives called Communicative Act Library (CAL) (Foundation for Intelligent Physical Agents, 1996–2002). Table 10 gives that set of standardized performatives along with their potential use in MOSAICO. Note that from the

CAL specification : *"FIPA- compliant agents are not required to implement any of the CAL languages, except the* `not-understood` *composite act."* (Foundation for Intelligent Physical Agents, 1996–2002).

Performative	**Description in CAL**
agree	The action of agreeing to perform some action, possibly in the future.
refuse	The action of refusing to perform a given action, and explaining the reason for the refusal.
failure	The action of telling another agent that an action was attempted but the attempt failed.
cancel	The action of one agent informing another agent that the first agent no longer has the intention that the second agent perform some action.
request	The sender requests the receiver to perform some action.
request-when	The sender wants the receiver to perform some action when some given proposition becomes true.
query-if	The action of asking another agent whether or not a given proposition is true.
query-ref	The action of asking another agent for the object referred to by a referential expression.
subscribe	The act of requesting a persistent intention to notify the sender of the value of a reference, and to notify again whenever the object identified by the reference changes.
inform	The sender informs the receiver that a given proposition is true.
inform-if	A macro action for the agent of the action to inform the recipient whether or not a proposition is true.
inform-ref	A macro action for sender to inform the receiver the object which corresponds to a descriptor, for example, a name.
confirm	The sender informs the receiver that a given proposition is true, where the receiver is known to be uncertain about the proposition.
disconfirm	The sender informs the receiver that a given proposition is false, where the receiver is known to believe, or believe it likely that, the proposition is true.
propagate	The sender intends that the receiver treat the embedded message as sent directly to the receiver, and wants the receiver to identify the agents denoted by the given descriptor and send the received propagate message to them.
proxy	The sender wants the receiver to select target agents denoted by a given description and to send an embedded message to them.
not-understood	The sender of the act (for example, i) informs the receiver (for example, j) that it perceived that j performed some action, but that i did not understand what j just did. A particular common case is that i tells j that i did not understand the message that j has just sent to i.
cfp	The action of calling for proposals to perform a given action.



propose	The action of submitting a proposal to perform a certain action, given certain preconditions.
accept-proposal	The action of accepting a previously submitted proposal to perform an action.
reject-proposal	The action of rejecting a proposal to perform some action during a negotiation.
request-whenever	The sender wants the receiver to perform some action as soon as some proposition becomes true and thereafter each time the proposition becomes true again.

Table 10: Performatives in FIPA-ACL (Foundation for Intelligent Physical Agents, 1996–2002) and their potential use in MOSAICO

An A2A Extension for Performatives

Following MOSAICO D1.1, communications between agents in MOSAICO rely on the Agent2Agent (A2A) protocol.[10] That protocol does not provide any support for performatives in messages but it can be extended to do so. A2A specifies precisely how to extend it while preserving interoperability with agents that do not support a given extension, and with agents that implement several protocol extensions.[11] An extension is defined by giving:

1. A unique name (URI).
2. An encoding of additional data as a mapping of given keys to values (a dictionary) together with a set of operations to add those meta-data in outgoing messages and to read them from incoming messages.

We provide such an extension to support performatives in messages (directory `specification` of the A2A-ACL repository[12]). That extension is identified by the following URI :

`https://mosaico-project.eu/extensions/a2a-acl/`

We also require that a performative attached to a message is a sequence of `a-z` or `-` characters, starting with a `a-z` character.

Implementation of the A2A Extension for Performatives

Together with the specification of our extension, an implementation for Python, is provided (directory `python/` of the A2A-ACL repository). In this library, we define a type `Performative` that extends strings (`str`) and enforces the form required by the specification (Listing 6).

We provide primitive functions to add or extract a performative to/from a message (Listings 7, 8).

Agent cards must declare that they implement the extension (Listing 9).

We also provide a Java library with similar services (directory `java/` of the A2A-ACL repository).

[10] `https://a2a-protocol.org/latest/specification/`
[11] `https://a2a-protocol.org/latest/specification/#46-extensions`
[12] `https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/a2a-acl`

```python
def check_performative_regexpr(p: str) -> bool:
    """A performative starts with a lowercase letter,
       and contains only lowercase letters and dash '-' character."""
    return bool(re.fullmatch(r"[a-z][a-z-]*", p))

class Performative(str):
    def __init__(self, s):
        super().__init__()
        if not check_performative_regexpr(s):
            raise ValueError("Invalid performative: " + s)
```

Listing 6: Performatives as a subtype of string.

```python
def add_performative(m: Message, p: Performative, codec: str) -> Message:
    """Add a performative to a message with the ACL extension.
        The message is modified."""
    if m.extensions is None:
        m.extensions = []
    if m.metadata is None:
        m.metadata = {}
    md: dict[str, Any] = {
        "performative": p,
        "codec": codec,
        "mediaType": MIME_TAG,
    }
    m.extensions.append(A2A_ACL_PROTOCOL_URI)
    m.metadata[A2A_ACL_PROTOCOL_URI] = md
    return m
```

Listing 7: Primitive to set a performative in a message.

```python
def extract_performative(context: RequestContext):
   p = context.message.metadata[A2A_ACL_PROTOCOL_URI]["performative"]
   return Performative(p)

def extract_codec(context: RequestContext):
   return context.message.metadata[A2A_ACL_PROTOCOL_URI]["codec"]
```

Listing 8: Primitive to read the performative in a message.

```
AgentExtension(
    uri=A2A_ACL_PROTOCOL_URI, description="MOSAICO A2A ACL", required=True
)
```

Listing 9: Extension to insert in the A2A AgentCard of agents

Semantics of the performatives

As explained above, we rely in MOSAICO on the set of performatives defined by FIPA-ACL (Table 10). The semantics of those semantics is defined in the FIPA reference document (Foundation for Intelligent Physical Agents, 1996–2002).

In practice each agent is responsible for supporting or not an illocution, but in our experiments, we have found that the two performatives `inform` (formerly `tell`) and `request` (formerly `achieve`) are often used and for this reason we adopted the following convention with respect to the A2A layer:

- When an `inform` message is received, the communication can be closed immediately, possibly after an immediate reply.
- When a `request` message is received, the receiver should create a *task*, with a relevant state (*submitted, rejected*), then set it to the *working* state if relevant (to keep the reference agent informed of the progress), and, eventually, set it to a *completed* or *failed* state.

Those two mechanisms are supported by a library we provide. The description of that library is out of scope of this report, but it can found on the same repository as our A2A extension. See also Figure 14.

We also provide a tight integration of those performatives with agents developed in AgentSpeak (see App. A).

Interaction with A2A task status updates

In multi-agent systems, agents usually communicate by sending messages and, in our implementation, performatives are attached to messages. In A2A, agents are also expected to communicate by updating the status of tasks or by adding artifacts to tasks.

Performatives cannot be directly attached to status updates because of the way extension points work in A2A. However, to add a performative to a status update, one can add a message to a status update, and then add a performative to that message.

On the other hand, some task states seems to overlap with some performatives. For instance, setting an A2A task to the state `failed` seems to be redundant with sending a message with the `failure` performative. However, those two information can take place at different levels. For instance, one step of a task may have failed, and a supervision agent can receive a message with that information, while the task itself could be eventually successful.

On the opposite, the `input-required` task state in A2A is not directly covered by any FIPA-ACL performative, while it is clearly interesting to structure some agent interactions.

Further experiments with the orchestrator under development and with use cases will allow us to refine the MOSAICO protocol to fix the set of possible performatives in messages, the minimum set of performatives an agent must support given its category, and to arbitrate between performatives and A2A task states when there is an potential overlap.

4 Use Cases

This section presents several uses cases of agents that interact using the ACL extension of the A2A protocol and workflow models of use case providers.

4.1 Agent interactions with A2A + ACL

To assess some architecture choices, we have developped some examples of agents that interact on top of A2A with our ACL extension.

Note that because we work in parallel we other groups, we use here some *ad-hoc* agents rather than the one provided concurrently in MOSAICO.

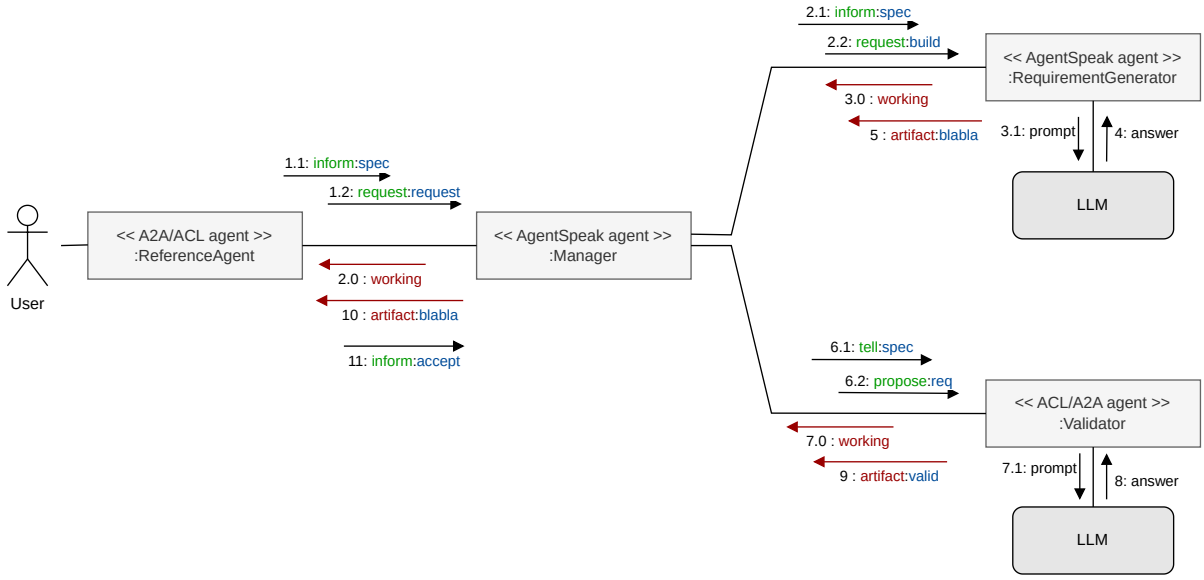


Figure 10: Collaboration diagram for example of generation of requirements.

In the two examples below, we have several kinds of agents:

Reference Agent: The reference agent is the interface between the user and the MOSAICO platform. In the A2A vocabulary, it is a *client* : it can initiate requests to other A2A agents, but it has no *server* to receive new requests.[13] New requests are performed by user inputs.

The reference agent uses our ACL library to embed performatives in messages that it sends.

Collaboration agent: The collaboration agent (labeled *Manager* in the diagrams) is a plain A2A agent,[14] with an A2A *server* to receive requests. It keeps a connection to its client, the reference agent, by the means of an A2A task. The responses to the client are made by task updates and artifact updates.

The collaboration agent uses other agents (solution agents) to make the work and evaluate it.

Solution Agents: The solution agents (labeled *RequirementGenerator* and *RefactoringAdvisor* in the diagrams) are A2A agents that can handle atomic tasks, often by using a LLM. Like the collaboration agent, a solution agent receives requests by its A2A server and updates its client by the means of A2A task updates and artifact updates.

[13] In D1.2, this is refined into a *reference client* and a *reference agent* which has an A2A server.

[14] In this experiment we use an *ad-hoc* orchestration specified in AgentSpeak instead of the orchestrator described in Sec. 3.2.2.

Supervision Agents: Supervision agents (labeled *Validator* and *CodeValidator* in the diagrams) are similar to solution agents, but instead of generating artifacts, they evaluate the work of other agents, often by using an LLM.

4.1.1 Building an Artifact in a Natural Language

We provide an example where several agents work together to build a set of requirements for a given specification.[15] The collaboration diagram for this example is given in Figure 10. That diagram illustrates a typical scenario:

Initialization. To initiate the work, the Reference agent sends two ACL messages: first an *inform*-message to give the relevant information about the problem, in particular the specification given by the user (1.1), then a *request*-message to request to build some requirements (1.2).[16]

The collaboration agent creates a new A2A task when receiving the *request*-message and updates its status to *working* (2.0).

Generation. The collaboration agent sends the specification to the solution agent in an *inform*-message (2.1), followed by a *request*-message (2.2). The solution agent acknowledges the request by creating a new task, in *working* state (3.0). Then the solution agents initiates a prompt request with it LLM (3.1) which gives its answer (4). The solution agent can answer to it client by attaching an artifact to the A2A task (5) and setting the task state to *completed* (not shown in the diagram).

Agent-based validation. Before submitting this result to the user, the collaboration agent submits it to a supervision agent. The collaboration agent sends the convenient messages to the supervision agents similarly to the interaction with the solution agent (6.1, 6.2) and the supervision agent uses and LLM also in a similar way (7.1, 8) and gives its result also in a similar way (9).

User validation. After successful validation by an agent, the collaboration agent replies to the user by attaching the generated requirement to an artifact submitted to the reference agent (10). The reference agent can request a validation to the user, and then tell the collaboration agent to add that requirement to the set of considered requirements (11). After that the session could continue with further generations (not implemented in this example).

4.1.2 Building an Artifact in a Programming Language

In this example, we illustrate a situation where we have some code that is working, and we want to rewrite it to improve its quality. We use a collaboration agent that will ask a proposal of refactoring to a specialized solution agent, and then submits the proposal to a supervision agent.

The workflow is very similar to the one in the previous example. One noteworthy difference is that the code to refactor, which is sent in messages, must not be parsed by the agent

[15] https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/mosaico-extra/a2a-agentspeak/-/tree/main/samples/llm_req_manager_with_reference_agent

[16] In typical cases, one message is sufficient to send all the parameters of a request. However, some *inform*-messages can still be relevant to update the *beliefs* of an agent or to give it some general configuration parameters.

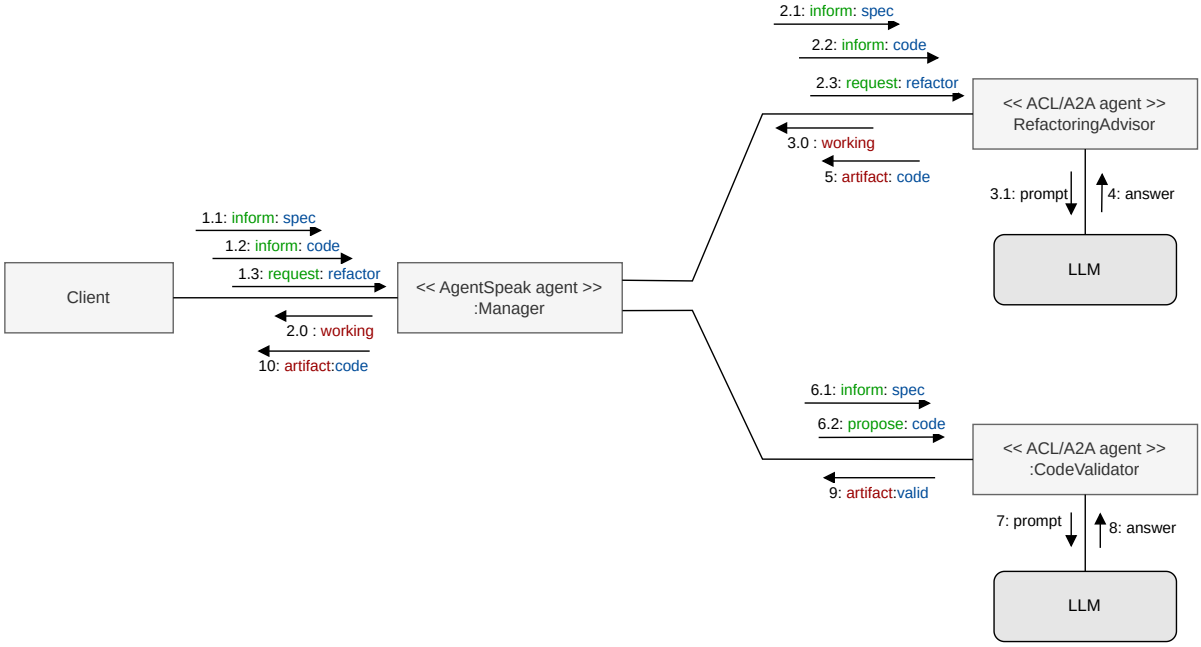


Figure 11: Collaboration diagram for example of refactoring.

as directives because it contains raw data. This difference is implemented in different ways, depending on the technology used to build the agent. For messages sent by the client (plain Python) to an AgentSpeak agent (message 1.2), the code is attached as a file to the message (`FilePart` in A2A). For messages sent by the AgentSpeak agent to the solution agent, the code is sent in a bytes string (message 2.2). Indeed, we have two primitives actions in our AgentSpeak framework to send messages : `.send` and `.send_bytes`.

For upstream messages (answers by agents to incoming requests), the code is encoded in an A2A `Artifact` (messages 5, 9, and 10).

4.2 The Metagente framework

Modern software engineering environments are characterized by an overwhelming volume of technical documentation, ranging from raw source code descriptions to complex user manuals and requirement specifications. Maintaining, navigating, and extracting meaningful insights from these documents presents a significant challenge for developers and stakeholders alike. While manual summarization is time-consuming and prone to human error, traditional automated approaches often fail to capture the nuanced semantic context of software artifacts.

The emergence of Large Language Models (LLMs) offers a potential solution; however, a single-model approach often struggles to maintain consistency, avoid hallucinations, and adhere to specific structural constraints. There is a critical need for a systematic, automated methodology to transform raw, often noisy software documentation into concise, accurate summaries. The Metagente framework (Nguyen et al., 2026) addresses this challenge by employing a Multi-Agent System (MAS) architecture, which decomposes the summarization task into specialized roles—extraction, refinement, and evaluation—to ensure the production of high-quality, reliable software documentation.

The Metagente framework operates through a two-stage architectural pipeline that first establishes an optimized communication protocol and subsequently applies it to operational data.

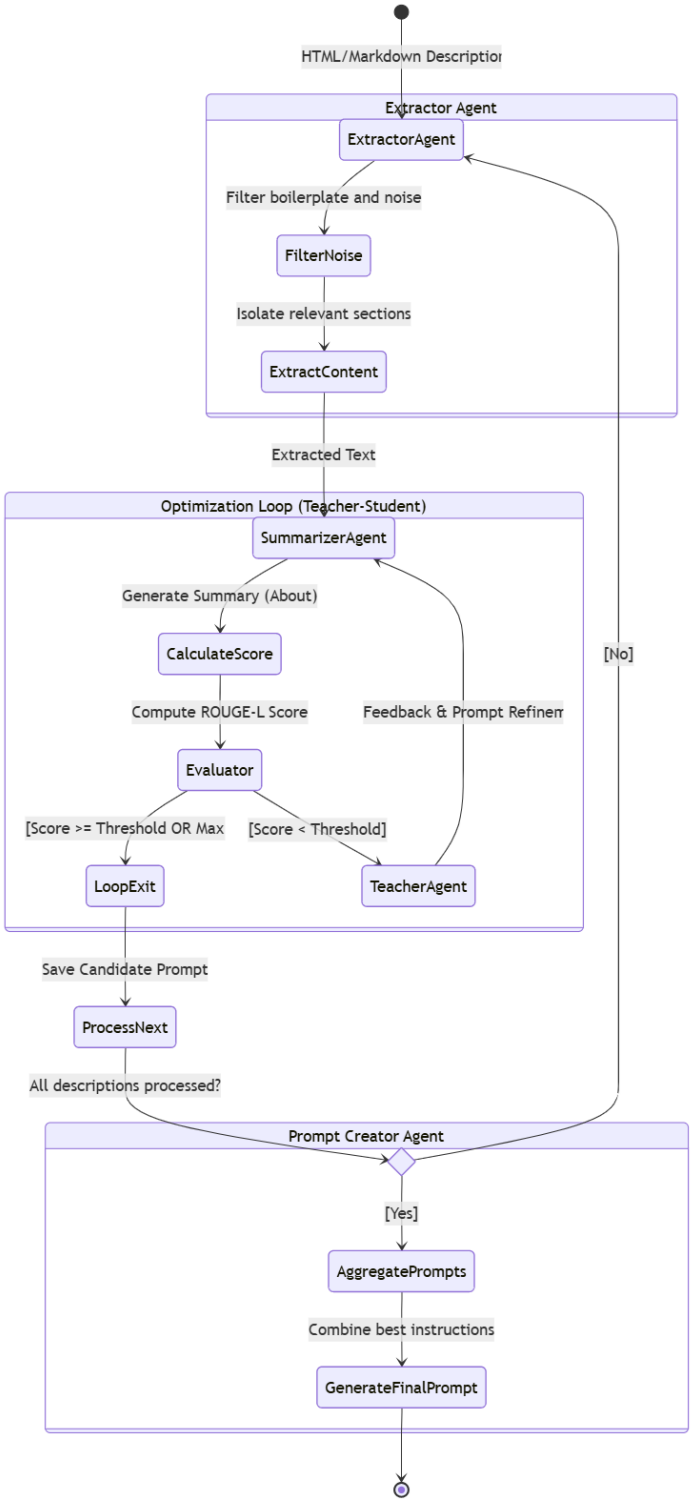


Figure 12: Optimization phase of the Metagente framework.

During the *Optimization Phase* (see Fig. 13, the system utilizes a multi-agent *Teacher-Student* configuration to refine the summarization logic. The process begins with the `Extractor Agent` cleaning raw HTML or Markdown input to isolate core descriptive content, which is then processed by the `Summarizer Agent` to produce a candidate summary. This output is quantitatively assessed against a ground truth using the ROUGE-L metric; if the quality is insufficient,

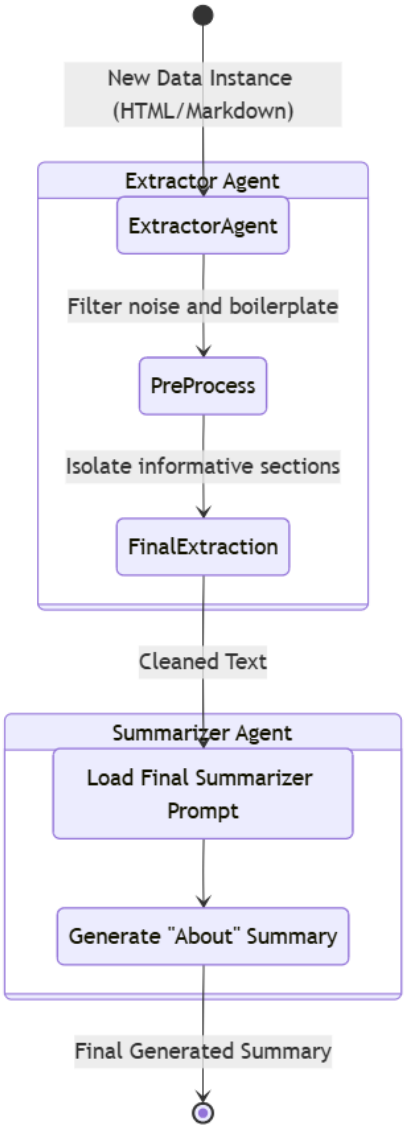


Figure 13: Evaluation phase of the Metagente framework.

the `Teacher Agent` performs a critical analysis of the discrepancies and updates the prompt instructions. This iterative loop continues until the performance converges or reaches a predefined threshold, at which point the `Prompt Creator Agent` aggregates the most successful strategies to synthesize a `Final Summarizer Prompt`.

Once this optimized prompt is established, the system transitions to the *Evaluation Phase*, which serves as the streamlined inference workflow for new data instances. In this stage, the `Extractor Agent` performs the same noise reduction and content isolation on incoming software descriptions, ensuring the input is focused on essential features. This cleaned text is then processed by the `Summarizer Agent` using the previously consolidated `Final Summarizer Prompt`, bypassing the need for further iterative refinement or teacher intervention. By decoupling the resource-intensive prompt engineering from the final execution, the system ensures that the resulting "About" descriptions maintain high semantic accuracy and conciseness..

4.3 Extended Reality Use Case

The Listing 10 defines an automated, agent-driven workflow designed to generate and optimize a 3D Extended Reality (XR) scene starting from 2D reference images. It begins by creating data types, such as `Image`, `Model3D`, and hardware metrics like `GPULoad`, and instantiates a `Specialized3DAgent`. This agent acts as a configured proxy that interfaces with an external endpoint to handle the complex generation and manipulation of 3D assets.

The core generation pipeline is encapsulated within the `generate3DAssets` action. This workflow begins by retrieving a 2D moodboard from the user and utilizes a parallel execution block to independently isolate and generate 3D models for both persons and machines. After the validation of a human through the action `userValidation`, the models are processed through a sequential optimization pipeline. Here, the specialized AI agent iteratively refines the topology, spacing, textures, and polygon density to create suitable levels of detail for XR headsets. Finally, the models are rigged, scaled, and assembled into a complete `UnityScene`, fully populated with main subjects, background elements, and a skybox.

To ensure the generated scene performs well on the target hardware, the `optimizeGPULoad` action introduces a dynamic performance feedback loop. It relies on two consecutive until loops that actively monitor system metrics. The first loop iteratively deletes background models from the generated scene until the GPU memory usage drops below an 80% threshold. Once memory is stabilized, the second loop retrieves and optimizes the underlying source code iteratively until the active GPU processing load also falls below 80%, guaranteeing a stable runtime environment for the XR application.

```
private import MOSAICO::*;
private import ScalarValues::*;
package XRUseCase {
    part def Specialized3DAgent specializes SolutionAgent {
        :>> id = "agent.immersion.v1";
        attribute url = "https://your-A2A-agent-local/agent.json/";
    }
    item def Image {
        attribute file : String;
    }
    item def Model3D {
        attribute file : String;
    }
    item def SkyBox {
        attribute file : String;
    }
    item def UnityScene {
        attribute file : String;
    }
    item def GPULoad {
        attribute usagePercentage : Real;
        attribute usageMemory : Real;
    }
    specialized3DAgent : Specialized3DAgent;
    action def generate3DAssets {
        out animatedScene : Model3D;
        first start;
        then action userInput : ReferenceStep {
            out moodBoard : Image [0..*];
        }
        then action associate3DLayout : Step {
            :>> agent = specialized3DAgent;
            :>> description =
```

```
        "Identify and isolate machines and persons of the moodboard's images.";
    in moodBoard : Image [0..*];
    out person3DModels : Model3D [0..*];
    out machine3DModels : Model3D [0..*];
    then fork;
        then filterPersons;
        then filterMachines;
    action filterPersons : Step {
        :>> description =
            "Create images with each person identified by cropping the moodboard
                's images.";
        in moodBoard : Image [0..*];
        out imagesWithPersons : Image [0..*];
    }
    then action associatePerson3DShape : Step {
        :>> description = "Generate the 3D layout for each person on the images.
            ";
        in imagesWithPersons : Image [0..*];
        out person3DModels : Model3D [0..*];
    }
    then joinNode;
    action filterMachines : Step {
        :>> description =
            "Create images with each machine identified by cropping the
                moodboard's images.";
        in moodBoard : Image [0..*];
        out imagesWithMachines : Image [0..*];
    }
    then action associateMachine3DShape : Step {
        :>> description = "Generate the 3D layout for each machine on the images
            .";
        in imagesWithMachines : Image [0..*];
        out machine3DModels : Model3D [0..*];
    }
    then joinNode;
    join joinNode;
}
then action userValidation : ReferenceStep {
    in person3DModels : Model3D [0..*];
    in machine3DModels : Model3D [0..*];
    out validatedPerson3DModels : Model3D [0..*];
    out validatedMachine3DModels : Model3D [0..*];
}
then action optimize3DLayouts : Step {
    :>> agent = specialized3DAgent;
    :>> description =
        "Choose the level of optimization adapted to targeted device (XR headset
            ) and then trying to reach best quality.";
        "Create at least 3 degrated models (LOD) : low medium and high poly
            models.";
    in validatedPerson3DModels : Model3D [0..*];
    in validatedMachine3DModels : Model3D [0..*];
    out optimizedPerson3DModels : Model3D [0..*];
    out optimizedMachine3DModels : Model3D [0..*];
    action improveTopology : Step {
        in validatedPerson3DModels : Model3D [0..*];
```

```
        in validatedMachine3DModels : Model3D [0..*];
        out optimizedTopologyPerson3DModels : Model3D [0..*];
        out optimizedTopologyMachine3DModels : Model3D [0..*];
    }
    then action improveSpace : Step {
        in optimizedTopologyPerson3DModels : Model3D [0..*];
        in optimizedTopologyMachine3DModels : Model3D [0..*];
        out optimizedSpacePerson3DModels : Model3D [0..*];
        out optimizedSpaceMachine3DModels : Model3D [0..*];
    }
    then action improveTexture : Step {
        :>> description :
            "Target low texel density for background models and high texel
                density for non background models.";
        in optimizedSpacePerson3DModels : Model3D [0..*];
        in optimizedSpaceMachine3DModels : Model3D [0..*];
        out optimizedTexturePerson3DModels : Model3D [0..*];
        out optimizedTextureMachine3DModels : Model3D [0..*];
    }
    then action improvePolygoneDensity : Step {
        in optimizedTexturePerson3DModels : Model3D [0..*];
        in optimizedTextureMachine3DModels : Model3D [0..*];
        out optimizedPerson3DModels : Model3D [0..*];
        out optimizedMachine3DModels : Model3D [0..*];
    }
    then action filterOut3DModels : Step {
        :>> description = "Decrire la contrainte du filtre.";
        inout optimizedPerson3DModels : Model3D [0..*];
        inout optimizedMachine3DModels : Model3D [0..*];
    }
}
then action animate3DModels : Step {
    in optimizedPerson3DModels : Model3D [0..*];
    out riggedPerson3DModels : Model3D [0..*];
    then fork;
        then animatePerson3DModels;
    action animatePerson3DModels : Step {
        in optimizedPerson3DModels : Model3D [0..*];
        out animatedPerson3DModels : Model3D [0..*];
        action updateRig : Step {
            in optimizedPerson3DModels : Model3D [0..*];
            out riggedPerson3DModels : Model3D [0..*];
        }
        then action scaleModel : Step {
            in riggedPerson3DModels : Model3D [0..*];
            out scaledRiggedPerson3DModels : Model3D [0..*];
        }
        then action pivotModel : Step {
            in scaledRiggedPerson3DModels : Model3D [0..*];
            out animatedPerson3DModels : Model3D [0..*];
        }
    }
    then joinNode;
    action animateMachine3Models : Step {
        in optimizedMachine3DModels : Model3D [0..*];
        out animatedMachine3DModels : Model3D [0..*];
```

```
            action updateRig : Step {
                in optimizedMachine3DModels : Model3D [0..*];
                out riggedMachine3DModels : Model3D [0..*];
            }
            then action scaleModel : Step {
                in riggedMachine3DModels : Model3D [0..*];
                out scaledRiggedMachine3DModels : Model3D [0..*];
            }
            then action pivotModel : Step {
                in scaledRiggedMachine3DModels : Model3D [0..*];
                out animatedMachine3DModels : Model3D [0..*];
            }
        } then joinNode;
        join joinNode;
    }
    then action create3DScene : Step {
        :>> description :
            "Save created 3D models into a folder which will be connected to Unity
                to import automatically.";
        in moodBoard : Image [0..*];
        in animatedPerson3DModels : Model3D [0..*];
        in animatedMachine3DModels : Model3D [0..*];
        out animatedScene : Model3D;
        action initializeSkyBox : Step {
            out skyBox : SkyBox;
        }
        then action createScene : Step {
            in moodBoard : Image [0..*];
            in skyBox : SkyBox;
            out animatedScene : UnityScene;
            action userChoose3DModelsPlacement {
                in animatedPerson3DModels : Model3D [0..*];
                out mainPerson3DModels : Model3D [0..*];
                out mainMachine3DModels : Model3D [0..*];
                out backgroundPerson3DModels : Model3D [0..*];
                out backgroundMachine3DModels : Model3D [0..*];
            }
            then action addBackground3DModels : Step {
                :>> description = "Decrire la contrainte directement dans la
                    description.";
                in skyBox : SkyBox;
                in backgroundPerson3DModels : Model3D [0..*];
                in backgroundMachine3DModels : Model3D [0..*];
                out backgroundAnimatedScene : UnityScene;
            }
            then action addMain3DModels : Step {
                in backgroundAnimatedScene : Model3D;
                in mainPerson3DModels : Model3D [0..*];
                in mainMachine3DModels : Model3D [0..*];
                out animatedScene : UnityScene;
            }
        }
    }
    then done;
}
action def optimizeGPULoad {
```

```
        in animatedScene : UnityScene;
        action retrieveGPULoad : ReferenceStep {
            out _GPULoad : GPULoad;
        }
        first start;
        then loop action saveGPUMemory {
            inout animatedScene : UnityScene;
            in _GPULoad : GPULoad;
            action optimize3DScene : Step {
                :>> description = "Delete some background models.";
                in animatedScene : UnityScene;
                out optimizedAnimatedScene : UnityScene;
            }
        } until retrieveGPULoad._GPULoad.usageMemory < 80;
        then loop action saveGPULoad {
            in _GPULoad : GPULoad;
            out optimizedSourceCode : String [0..*];
            then action retrieveSourceCode : Step {
                out sourceCode : String [0..*];
            }
            then action optimizeSourceCode : Step {
                in sourceCode : String [0..*];
                out optimizedSourceCode : String [0..*];
            }
        } until retrieveGPULoad._GPULoad.usagePercentage < 80;
        then done;
    }
}
```

Listing 10: SysML v2 program to describe a 3D extended reality scene starting from 2D reference images.

4.4 Metamodel Generation Use Case

This use case, from University of York, illustrates a metamodel generation workflow involving several agent and showcasing how errors and retry mechanisms can be specified using SysML v2. Like previous examples, this one starts by instantiating several agents (`solutionAgent`, `syntaxAgent` and `validatorAgent`), then the definition of the process starts with attributes that correspond to inputs from the user (`domainDescription`) and outputs, the generated metamodel (`emfaticSource`) and a boolean (`success`) to indicate success or failure of the task. Along with theses inputs and outputs we have two attributes that save internal state of the collaboration, a retry counter (`retryCounter`) and a feedback to pass between agents (`feedback`).

This workflows starts by asking the `solutionAgent` to generate the metamodel from user prompts. Note that in this example data flows are given explicitly (using `bind` and `succession flow`) unlike in previous previous example. Current implementation of the orchestrator does not support this yet.

Then, the `syntaxAgent` checks the syntax of the generated metamodel. If the syntax is correct then it passes it to the `validatorAgent` for a semantic check. Otherwise it increment the retry counter and if the retry counter is bellow 3 it goes back to the `solutionAgent` with the feedback from the syntax check. Otherwise the task fails.

The same check and retry mechanism happen for the `validatorAgent` to check that the produced metamodel is semantically correct.

```
private import ScalarValues::*;
private import SequenceFunctions::*;
private import MOSAICO::*;

package MetamodelCollaborationAgents {

    part def MetamodelSolutionAgent specializes SolutionAgent {
        constraint { skills->includes("emfatic-generation") }
    }

    part def SyntacticSupervisionAgent specializes SupervisionAgent {
        constraint { skills->includes("emfatic-syntax-validation") }
    }

    part def SemanticSupervisionAgent specializes SupervisionAgent {
        constraint { skills->includes("emfatic-requirement-validation") }
    }

    part solutionAgent : MetamodelSolutionAgent;
    part syntaxAgent : SyntacticSupervisionAgent;
    part validatorAgent : SemanticSupervisionAgent;

    action MetamodelGenerationProcess {
        // Initial prompt from the user
        in domainDescription : String;
        // Outputs from the agent: Emfatic source for metamodel + success/fail flag
        out emfaticSource : String;
        out success : Boolean;

        // Counter for retries (we get 3)
        attribute retryCounter : Integer = 0;
        // Feedback from previous checks to the solution agent, initially empty
        attribute feedback : String = "";

        first start;
        then merge retry;

        then action generateMetamodel : AgentStep {
            ::> agent = solutionAgent;
            in domainDescription : String;
            in feedback : String;
            out emfaticSource : String;
        }
        bind generateMetamodel.domainDescription = domainDescription;
        bind generateMetamodel.feedback = feedback;
        bind emfaticSource = generateMetamodel.emfaticSource;

        then action checkSyntax : SupervisionStep {
            ::> agent = syntaxAgent;
            in emfaticSource : String;
            out feedback : String;
            out isValidSyntax : Boolean;
        }
        succession flow generateMetamodel.emfaticSource to checkSyntax.emfaticSource;
```

```
        then decide;
            if checkSyntax.isValidSyntax then checkRequirements;
            if not checkSyntax.isValidSyntax then retryIfInvalidSyntax;

        action retryIfInvalidSyntax {
            retryCounter = retryCounter + 1;
            feedback = checkSyntax.feedback;
            success = false;
        }
        if retryCounter < 3 then retry;

        action checkRequirements : SupervisionStep {
            ::> agent = validatorAgent;
            in emfaticSource : String;
            in domainDescription : String;
            out feedback : String;
            out meetsRequirements : Boolean;
        }
        succession flow generateMetamodel.emfaticSource to checkRequirements.
            emfaticSource;
        bind checkRequirements.domainDescription = domainDescription;

        then decide;
            if checkRequirements.meetsRequirements then succeeded;
            if not checkRequirements.meetsRequirements then retryIfUnmetRequirements;

         action retryIfUnmetRequirements {
            retryCounter = retryCounter + 1;
            feedback = checkRequirements.feedback;
            success = false;
         }
         if retryCounter < 3 then retry;

         action succeeded : Step {
            success = true;
         }
    }
}
```

Listing 11: SysML v2 workflow representation of the metamodeling agent



5 Conclusion

In this deliverable we presented a new dataset of SysML v2 workflows. It consist of about 74,900 SysML v2 files. Part of it still requires a description / user prompt to be generated but we already started using the dataset for our SysML v2 generation effort. Currently, these SysML v2 files are used as examples to help LLM generate syntactically and semantically correct workflows.

We also presented the architecture of our orchestrator. As of the writing of the deliverable it is still in an early state of prototype that supports a subset of SysML v2 on mock agents. Later versions of the orchestration agent will focus on, data flows between user and agents and between agents, better representation of external resources agents can leverage (such as databases, MCP tools or external memory).



References

Abdelnabi, Sahar, Amr Gomaa, Sarath Sivaprasad, Lea Schönherr, and Mario Fritz (2024), "Cooperation, competition, and maliciousness: Llm-stakeholders interactive negotiation, 2024", *in*: *URL https://arxiv. org/abs/2309.17234*. [Cited on page 28]

Bamouh, Léna, Thibault Béziers La Fosse, and Massimo Tisi (2025), "Towards Diagram-Based Data Model Generation with LLMs", *in*: *International Conference on Conceptual Modeling*, Springer, pp. 157–175. [Cited on page 29]

Camunda (2026), *The Process Orchestration Handbook - Building a foundation for agentic orchestration at enterprise scale*, `https://camunda.com/process-orchestration/` [Accessed: 2026-03-12]. [Cited on page 27]

Chan, Chi-Min, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu (2023), "Chateval: Towards better llm-based evaluators through multi-agent debate", *in*: *arXiv preprint arXiv:2308.07201*. [Cited on page 28]

Chen, Guangyao, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F Karlsson, Jie Fu, and Yemin Shi (2023), "Autoagents: A framework for automatic agent generation", *in*: *arXiv preprint arXiv:2309.17288*. [Cited on page 29]

Chen, Weize, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. (2023), "Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors", *in*: *The Twelfth International Conference on Learning Representations*. [Cited on page 29]

Das, Ayushman, Shu-Ching Chen, Mei-Ling Shyu, and Saad Sadiq (2023), "Enabling synergistic knowledge sharing and reasoning in large language models with collaborative multiagents", *in*: *2023 IEEE 9th International Conference on Collaboration and Internet Computing (CIC)*, IEEE, pp. 92–98. [Cited on page 28]

Davidson, Tim R, Veniamin Veselovsky, Martin Josifoski, Maxime Peyrard, Antoine Bosselut, Michal Kosinski, and Robert West (2024), "Evaluating language model agency through negotiations", *in*: *arXiv preprint arXiv:2401.04536*. [Cited on page 28]

De Bari, Daniele, Giacomo Garaccione, Riccardo Coppola, Marco Torchiano, and Luca Ardito (2024), "Evaluating large language models in exercises of uml class diagram modeling", *in*: *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 393–399. [Cited on page 16]

DeHart, John K (2024), "Leveraging large language models for direct interaction with sysml v2", *in*: *INCOSE International Symposium*, vol. 34, 1, Wiley Online Library, pp. 2168–2185.

Du, Yilun, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch (2024), "Improving factuality and reasoning in language models through multiagent debate", *in*: *Forty-first international conference on machine learning*. [Cited on page 28]

Finin, Tim, Jay Weber, Gio Wiederhold, Michael Genesereth, Richard Fritzson, Donald P McKay, James McGuire, Richard Pelavin, Stuart C Shapiro, Christopher Beck, et al. (1993), "Specification of the KQML agent-communication language", *in*: ARPA Knowledge Sharing Initiative External Interfaces Working Group. [Cited on page 39]

Foundation for Intelligent Physical Agents (1996–2002), *FIPA Communicative Act Library Specification*, Accessed on `https://web.archive.org/web/20250323175301/http://fipa.org/specs/fipa00037/index.html`, URL: `http://www.fipa.org/`. [Cited on pages 39–41, 43]

Fox, Maria and Derek Long (2003), "PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains", *in*: *Journal of Artificial Intelligence Research* 20, pp. 61–124. [Cited on page 13]

Google (2026), *Agent Development Kit*, https://google.github.io/adk-docs/ [Accessed: 2026-03-12]. [Cited on page 27]

He, Zhitao, Pengfei Cao, Yubo Chen, Kang Liu, Ruopeng Li, Mengshu Sun, and Jun Zhao (2023), "LEGO: A multi-agent collaborative framework with role-playing and iterative feedback for causality explanation generation", *in*: *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 9142–9163. [Cited on page 28]

Hong, Sirui, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. (2023), "MetaGPT: Meta programming for a multi-agent collaborative framework", *in*: *The twelfth international conference on learning representations*. [Cited on pages 27, 28]

Li, Xinyi, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang (2024), "A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges", *in*: *Vicinagearth* 1.1, p. 9. [Cited on page 28]

Liang, Tian, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu (2024), "Encouraging divergent thinking in large language models through multi-agent debate", *in*: *Proceedings of the 2024 conference on empirical methods in natural language processing*, pp. 17889–17904. [Cited on page 28]

Lin, Bohan, Kuo Yang, Zelin Tan, Yingchuan Lai, Chen Zhang, Guibin Zhang, Xinlei Yu, Miao Yu, Xu Wang, Yudong Zhang, et al. (2025), "AgentAsk: Multi-Agent Systems Need to Ask", *in*: *arXiv preprint arXiv:2510.07593*. [Cited on page 28]

Liu, Zijun, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang (2024), "A dynamic LLM-powered agent network for task-oriented agent collaboration", *in*: *First Conference on Language Modeling*. [Cited on page 29]

McDermott, Drew, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins (1998), *PDDL — The Planning Domain Definition Language*, AIPS-98 Planning Competition Committee. [Cited on page 13]

MOSAICO Consortium, IMT Atlantique, LIST, Uni. of York, Uni. dell'Aquila, Intrasoft, Immersion, NBG, Collins Aerospace, Unparallel, Qodo, Eclipse Foundation, and F6S (2025), *MOSAICO – Management, Orchestration and Supervision of AI-agent Communities*, [Online; accessed 12. Jun. 2025], URL: https://mosaico-project.eu.

Nandi, Subhrangshu, Arghya Datta, Nikhil Vichare, Indranil Bhattacharya, Huzefa Raja, Jing Xu, Shayan Ray, Giuseppe Carenini, Abhi Srivastava, Aaron Chan, et al. (2025), "Sop-bench: Complex industrial sops for evaluating llm agents", *in*: *arXiv preprint arXiv:2506.08119*. [Cited on page 24]

Nguyen, Duc S. H., Minh T. Nguyen, Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio (2026), "Automated summarization of software documents: an LLM-based multi-agent approach", *in*: *Autom. Softw. Eng.* 33.2, p. 43, DOI: 10.1007/S10515-025-00588-4, URL: https://doi.org/10.1007/s10515-025-00588-4. [Cited on page 46]

Object Management Group (2014), *BPMN(TM) – Business Process Model and Notation, Version 2.0.2*, https://www.omg.org/spec/BPMN/2.0.2/About-BPMN, Accessed: 2026-03-16. [Cited on page 11]

— (2017), *Unified Modeling Language (UML), Version 2.5.1*, https://www.omg.org/spec/UML/2.5.1/About-UML, Accessed: 2026-03-16. [Cited on page 10]

Pan, Bo, Jiaying Lu, Ke Wang, Li Zheng, Zhen Wen, Yingchaojie Feng, Minfeng Zhu, and Wei Chen (2025), "Agentcoord: Visually exploring coordination strategy for llm-based multi-agent collaboration", *in*: *Computers & graphics*, p. 104338. [Cited on page 29]

Planning.Wiki (2026), *PDDL — Planning Domain Definition Language*, https://planning.wiki/, Accessed: 2026-03-16. [Cited on page 13]

Qian, Chen, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. (2024), "Chatdev: Communicative agents for software development", *in*: *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 15174–15186. [Cited on pages 27, 29]

Qiao, Shuofei, Runnan Fang, Zhisong Qiu, Xiaobin Wang, Ningyu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen (2024), "Benchmarking agentic workflow generation", *in*: *arXiv preprint arXiv:2410.07869*. [Cited on pages 19, 22]

Sola, Diana, Christian Warmuth, Bernhard Schäfer, Peyman Badakhshan, Jana-Rebecca Rehse, and Timotheus Kampik (2022), "SAP Signavio academic models: A large process model dataset", *in*: *International Conference on Process Mining*, Springer, pp. 453–465. [Cited on pages 19, 23]

Specification, OMG Available (2007), "Omg systems modeling language (omg sysml™), v1. 0", *in*: *Object Management Group*. [Cited on page 14]

Taulli, Tom and Gaurav Deshmukh (2025), "CrewAI", *in*: *Building Generative AI Agents: Using LangGraph, AutoGen, and CrewAI*, Springer, pp. 103–145. [Cited on page 27]

UML Diagrams (2026), *UML Activity Diagrams – Control Nodes*, `https://www.uml-diagrams.org/activity-diagrams-controls.html`, Accessed: 2026-03-16. [Cited on page 10]

Vidal, José M. (2009), *KQML Performatives*, URL: `https://jmvidal.cse.sc.edu/talks/agentcommunication/kqmlperformatives.html`. [Cited on page 39]

Wang, Zhefan, Yuanqing Yu, Wendi Zheng, Weizhi Ma, and Min Zhang (2024), "Macrec: A multi-agent collaboration framework for recommendation", *in*: *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2760–2764. [Cited on page 28]

White, Stephen A. (2004), *Introduction to BPMN*, `https://www.omg.org/bpmn/Documents/Introduction_to_BPMN.pdf`, Accessed: 2026-03-16. [Cited on page 11]

Wu, Qingyun, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. (2024), "Autogen: Enabling next-gen LLM applications via multi-agent conversations", *in*: *First conference on language modeling*. [Cited on page 27]

Yu, Geunbin (2026), "AdaptOrch: Task-Adaptive Multi-Agent Orchestration in the Era of LLM Performance Convergence", *in*: *arXiv preprint arXiv:2602.16873*. [Cited on page 27]

Zhang, Yifan, Jingqin Yang, Yang Yuan, and Andrew Chi-Chih Yao (2023), "Cumulative reasoning with large language models", *in*: *arXiv preprint arXiv:2308.04371*.

Zhao, Qinlin, Jindong Wang, Yixuan Zhang, Yiqiao Jin, Kaijie Zhu, Hao Chen, and Xing Xie (2023), "Competeai: Understanding the competition dynamics in large language model-based agents", *in*: *arXiv preprint arXiv:2310.17512*. [Cited on page 28]

Zhu, Kunlun, Hongyi Du, Zhaochen Hong, Xiaocheng Yang, Shuyi Guo, Zhe Wang, Zhenhailong Wang, Cheng Qian, Xiangru Tang, Heng Ji, and Jiaxuan You (2025), *MultiAgentBench: Evaluating the Collaboration and Competition of LLM agents*, arXiv: `2503.01935`, URL: `https://arxiv.org/abs/2503.01935`. [Cited on page 29]

Zuo, Max, Francisco Piedrahita Velez, Xiaochen Li, Michael Littman, and Stephen Bach (2025), "Planetarium: A rigorous benchmark for translating text to structured planning languages", *in*: *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 11223–11240. [Cited on page 19]

A AgentSpeak support

To separate the definition of agent behaviors from boilerplate code, we use AgentSpeak as a programming language for our manager agents in several examples (Sec. 4.1.1 and 4.1.2).

A first experiment with AgentSpeak on top of AutoGen is described in D3.1, and we have adapted that work to A2A instead of AutoGen.[17]

AgentSpeak is tightly coupled with performatives. Specifically:

- Messages received with the `inform` performative are directly reflected into the *beliefs* of agents.
- Messages received with the `request` performative trigger the creation of a new A2A task. A status update to the `submitted` state is also triggered for the sender.
- The status of a task can be updated by *custom actions* : `.working_on_task`, `.reject_task`, `.close_task`, `.add_artifact_on_task`, `.input_required`, `.report_failure_for_task`.

See Fig. 14 for the class diagram of our A2A/AgentSpeak framework.

[17] `https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/mosaico-extra/a2a-agentspeak`



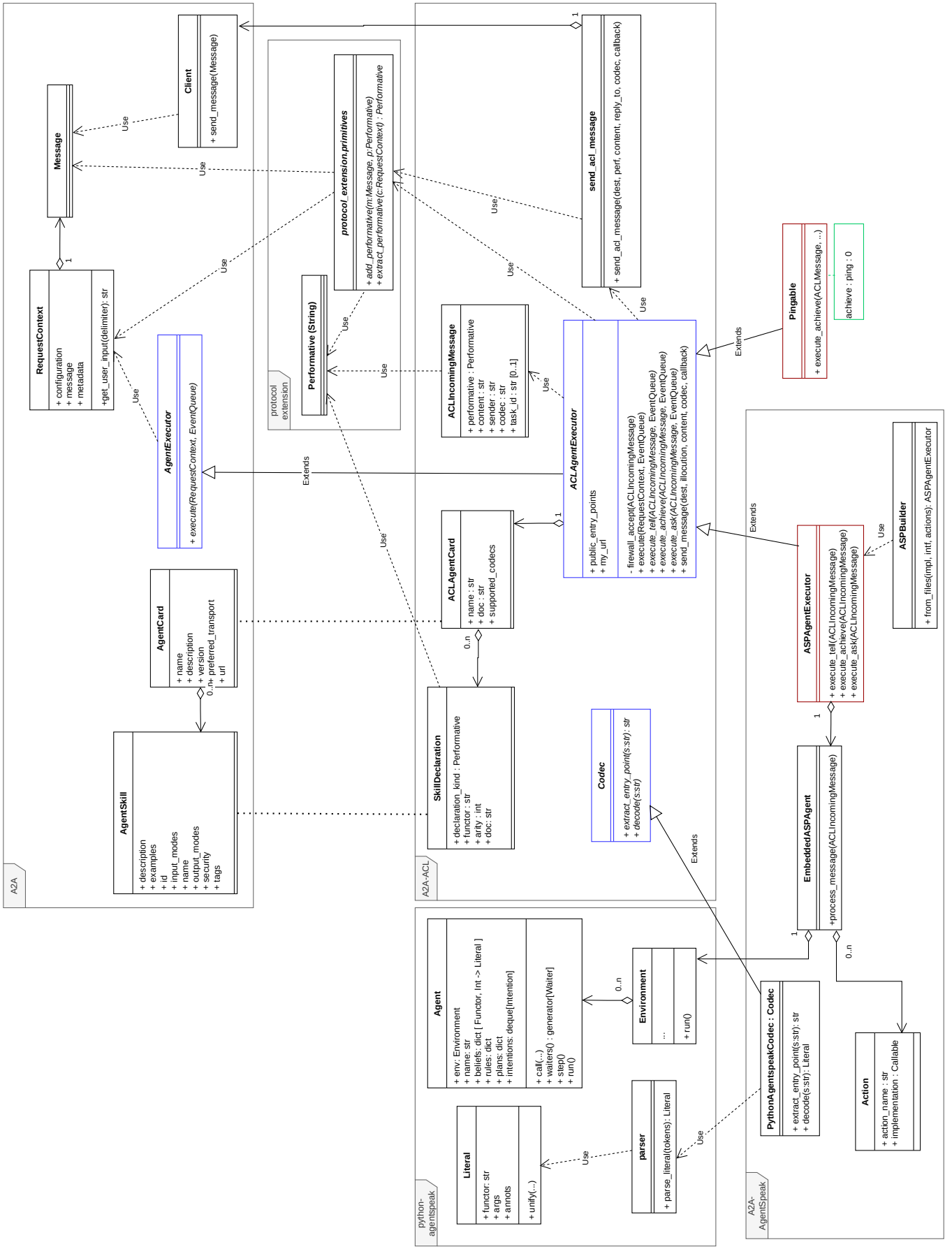


Figure 14: Full ACL AgentSpeak agent framework