ALF or may be represented using the Parametric Diagram.

The executable viewpoint embodies the realization of both structural and behavioral viewpoints by defining the execution of architectural behavior at runtime. The main objective of execution is to validate the behavioral logic for the fulfillment of requirements and the analysis of architectural functions. From the executable viewpoint, it is possible to specify the specifics of each activity using ALF statements and define and instantiate elements.

SysADL solution provides transformation processes for formal languages that enable the architecture verification, as well as plugins for SysADL Studio tool that execute the verification process (Araújo et al., 2021; Dias et al., 2020).

## 5 EXTENDING SYSADL VIEWPOINTS

The novel viewpoints enhance SysADL to provide modeling of systems and environments, simulating the systems' execution while considering different scenarios to provide architectural execution. Figure 2 illustrates the conceptual model of the novel viewpoints and their diagrams, along with their relationship to the existing viewpoints in SysADL. Due to space restrictions, we've used only code snippets to demonstrate the use of the new grammar elements that support the new viewpoints. The full example code can be found online[2].

### 5.1 Environment Viewpoint

The *Environment Viewpoint* delineates the elements constituting the environment for architectural execution. It represents the entities in the environment, their characteristics, and their connection with other entities. It is described using two diagrams: *BDD* and *IBD*. They represent the definitions of the entities, connections, characteristics, and instances that govern the overall environment. The BDD defines the elements that constitute the environment: *EnvironmentDefinition*, *Entity*, *Connection*, *Role*, and *Property*. The IBD defines the instances of the elements to configure a specific environment through a *EnvironmentConfiguration*.

An *EnvironmentDefinition* specifies each environment and includes the definition of the elements associated with the modeled system that interact with the environment, as well as the elements of the environment itself. These components are referred to as *Entity*. An *Entity* must have at least one *Role*. The *Role* acts as the point of connection, allowing interaction between entities. An entity may optionally contain *Properties*, which describe its characteristics. An *Entity* can additionally be composed of other entities. An example of *EnvironmentDefinition* declaration is depicted in Listing 1, and defines the name `MyFactory` (line 1), and *Entity* with *Properties* and *Roles* (lines 2-9). Lines 10-14 illustrate an *Entity* composed of other entities.

An *EnvironmentDefinition* also contains *Connections* to represent the interaction between *Entities*. It binds with the *Roles* of the connected *Entities*, allowing information to flow between them. An example is in Listing 1, lines 16-18.

The *EnvironmentConfiguration* describes associations, instances, or assigns values to the defined *EnvironmentDefinition* elements. The objective of *EnvironmentConfiguration* is to generate a enough variety of configurations to include all potential environmental possibilities. Consequently, for each *EnvironmentDefinition*, multiple *EnvironmentConfiguration* instances may be defined. To do this, in addition to the name of the *EnvironmentConfiguration* (`MyFactoryConfiguration`), it is necessary to define which *EnviromentDefinition* it refers to (`MyFactory`), as shown in Listing 2, line 1.

Listing 1: Example of EnvironmentDefinition elements declaration

```
1  EnvironmentDefinition MyFactory {
2      Entity def Station {
3          properties {
4              Property def ID
5          }
6          roles {
7              Role def signal
8      }   }
9      Entity def Lane {
10         entities {
11             stations: Station[]
12     }   }
13     ...
14     Connection def Location {
15         from Station.signal to Vehicle
               .sensor
16 }   }
```

The association between the environment and the modeled system is established in the *EnvironmentConfiguration* by associating a *Component* instance from the *Structural viewpoint*. Listing 2, line 2, shows the `agvs`, an instance of a *Component AGVSystem*, associated with the `Vehicle` representing an *Entity* within the environment. An association is also made by relating a *Role* to an instance of a *Port* defined in the *Structural Viewpoint*. An example is shown in Listing 2, line 3, where `Vehicle.outNotification` is a *Role* defined in the *EnvironmentDefinition*, and `agvs.in_outDataAgv.outNotifications` is an instance of a *Port* defined in the *Structural Viewpoint*.

Listing 2: Example of EnvironmentConfiguration elements declarations

```
1  EnvironmentConfiguration
       MyFactoryConfiguration to
       MyFactory {
2    Vehicle: agvs;
3    Vehicle.outNotification: agvs.
         in_outDataAgv.outNotifications;
4    stationA:Station;
5    stationA.ID = "StationA";
6    ...
7    lane1:Lane;
8    lane1.entities.stations = [stationA,
         stationB, stationC];
9  }
```

[2] xxxxxxxxxxxxxx

The instantiation of each environment element uses a colon to separate the instance from the element definition in the format `instance:element`, as in Listing 2, line 4, where `stationA` is the instance of the `Station` entity defined in the *EnvironmentDefinition*. Each element can be instanced several times in a configuration or can be left out entirely from others.

We can observe the assignment of values in Listing 2, line 5, where `"StationA"` is assigned to the property `ID` of the instance `stationA`. In line 7, another kind of assignment, where the instance `lane1` is composed of three `Station` instances: `stationA`, `stationB`, and `stationC`.

We used the SDFormat (Foundation, 2024) format as inspiration to create these elements. It was designed for modeling environments in robot simulation. Nevertheless, we realized that the environmental descriptions may be modified to satisfy our requirements for modeling software architectures, beyond their original application for robots. Consequently, SDFormat served as our inspiration for the design of the environments with the components proposed.

## 5.2 Scenario Viewpoint

The *Scenario Viewpoint* enables the modeling of event occurrences within an environment to simulate its execution and interactions. A scenario defines the behavior of entities by specifying events that include actions performed by entities, interactions among entities, and entity states that trigger or condition other events. The viewpoint is described using two diagrams: *BDD* and *Sequence Diagrams (SD)*. They represent the definition of scenarios, scenes, and events that govern the overall environment's behavior. The BDD defines the use of the *Entity* instances from the environment to build the *Scenes* and the *Scenarios*. A *Scene* is composed of *Events* that trigger other events that the entities can react to. A Scenario is a composition of scenes. Different scenarios provide different ways to simulate the architecture in the environment. The *SD* defines the execution sequence for scenarios and models *Events* within a *ScenarioExecution*.

The *EventsDefinitions* defines the events that may occur related to an *EnvironmentConfiguration* (Listing 3, line 1). *Events* can represent either actions or states of entities, as well as interactions between them. Thus, actions and states are specializations of an event. Events are defined in association with specific entities, meaning that for each *Entity instance*, the corresponding associated events must be specified (Listing 3, line 2 and line 9), where `supervisor` (line 2) `stationA` (line 9) are instances of entities defined in the *EnvironmentConfiguration* named `MyFactoryConfiguration`.

Each event can be initiated by a trigger (`cmdSupervisor`, in line 4) or by a condition (`agv1.sensor == stationA`, in line 10), defined in the event's `ON` clause, which executes the block defined in the `THEN` clause. Each `THEN` block specifies a name (`cmdAGV2toC` on line 4, and `AGV1locationStationA` on line 11), so that it can be triggered by other events. In each `THEN` block, you may allocate values to the entities' *Roles* and utilize *Connections* to transmit the values among the *Roles*. An event can have multiple `ON` clauses, and it can execute multiple `THEN` clauses.

Listing 3: Example of EventsDefinitions elements declarations

```
1  EventsDefinitions MyEvents to
       MyFactoryConfiguration {
2    Event def SupervisoryEvents for
         supervisor {
3      ON cmdSupervisor
4        THEN cmdAGV2toC {
5          supervisor.outCommand.
               destination=stationC;
6          supervisor.outCommand.
               armCommand=idle;
7          :Command(supervisor, agv2);
8    }   }
9    Event def StationAEvents for
         stationA {
10     ON agv1.sensor == stationA
11       THEN AGV1locationStationA {
12         agv1.location = stationA.
               signal;
13 } }   }
```

Following on from describing the events, it is necessary to define *Scenes*, which consist of a set of events surrounded by the initial and terminal events. The definitions of scenes are described in *SceneDefinitions* (Listing 4, line 1), and must define their name (`MyScenes`) and the corresponding set of events (`MyEvents`).

A *Scene* is constructed, defining pre-conditions and post-conditions blocks. `Pre-condition` clause describes the initial states under which events will be executed in the *Scene* (Listing 4, lines 3-5), and reference elements instantiated in *EnvironmentConfiguration*. `Post-condition` clause specifies the expected states or actions following the execution of the *Scene* (Listing 4, lines 8-10). *Post-condition* clause will be used in the verification and validation of a scene execution and for a scenario composition. For instance, in a scenario, the pre-conditions of the following scene must match its predecessor.

The `start` clause define the initial *Event* for the *Scene* (Listing 4, line 6). The `finish` clause define the terminal *Event* for the *Scene* (Listing 4, line 7). All events triggered between them are defined in the event chain under *EventDefinition*.

Listing 4: Example of SceneDefinitions elements declarations

```
1  SceneDefinitions MyScenes to MyEvents
       {
2    Scene def SCN_MoveAGV1toA on {
3      pre-condition {
4        agv1.location == stationC.ID;
5        part.location == stationA.ID; }
6      start cmdSupervisor;
7      finish AGV1NotifArriveA;
8      post-condition {
9        agv1.location == stationA.ID;
10       part.location == stationA.ID; }
11 } }
```

A *Scenario* is composed of a set of *Scenes*. Upon defining the scenes, the scenario is described in the *ScenarioDefinitions* and referring to the set of *SceneDefinitions*, as depicted in Listing 5, line 1, where `MyScenarios` is the name of the set of *Scenarios* using the *Scenes* defined in `MyScenes`.

A scenario defines the flow of control of scenes in sequences Listing 5, lines 2-7), loops, conditional branches, or other programming elements Listing 5, lines 8-14). It is also possible for a scenario to execute other scenarios.

Listing 5: Example of ScenarioDefinitions elements declarations

```
1 ScenarioDefinitions MyScenarios to
     MyScenes {
2   Scenario def Scenario1 {
3     SCN_MoveAGV1toA;
4     SCN_MoveAGV2toC;
5     SCN_AGV1movePartToC;
6     SCN_AGV2movePartToE; }
7   Scenario def Scenario3 {
8      let i: Integer = 1;
9      while (i < 5) {
10        SCN_MoveAGV1toA;
11        SCN_AGV1movePartToC;
12        i++;
13 }  }   }
```

Finally, the *ScenarioExecution* specifies the instances of the defined *ScenarioDefinition* that will be included in the simulation (Listing 6). The execution of a *Scenario* governs the execution of its constituent *Scenes*. These executions can also be organized into sequences, loops, or conditional branches, and can use variables. To facilitate repetitive execution, we use the `repeat <n>` command, where ¡n¿ is the number of times to repeat the execution, as shown in line 9. The *ScenarioExecution* can refer to all elements instanced in *EnvironmentConfiguration*, *EventsDefinitions*, *SceneDefinitions*, and *ScenarioDefinitions*.

Listing 6: Example of ScenarioExecution elements declarations

```
1 ScenarioExecution to MyScenarios {
2   agv1.location = stationC.ID;
3   agv2.location = stationD.ID;
4   part.location = stationA.ID;
5   Scenario1;
6   Scenario2;
7   Scenario3;
8   Scenario4;
9   repeat 5 Scenario1;
10 }
```

These language elements were initially inspired by OpenSCENARIO (ASAM e.V., 2022), specialized in describing scenarios and events for driving and traffic simulation. We draw inspiration from the model of many events that may take place during a simulation. However, we have adapted it for applications beyond driving and traffic, proposing its use in broad situations. We also took concepts from JavaScript event listeners (ECMA International, 2024), but we changed the syntax to be more expressive in our context.

## 6 TRANSFORMATION TO OPERATIONAL SEMANTICS

The execution of a software architecture is enabled by incorporating executable elements into its architecture descriptions (Oquendo et al., 2016). It may be necessary to undergo a transformation process to execute an architecture. Transformations involve generating a target model from a source model using a set of transformation rules that describe how to transform a model in the source language into a model in the target language. These models can be converted into code, documentation, configurations, and tests for the software system. These transformations help minimize the effort required by developers by producing executable artifacts, resulting in enhanced software quality, reduced complexity, and shorter development time and effort (Naveed et al., 2024).

To enable the execution of SysADL, the architectural descriptions must be transformed to assign operational semantics and subsequently executed by the execution tool that we created. As mentioned in Section 5.2, our environmental and scenario modeling is event-driven; hence, we developed the execution tool in JavaScript, which is an event-driven programming language, meaning that the flow of a program is determined by a series of events. Consequently, transformation mechanisms, including those for operational semantics, are being devised in JavaScript.

Figure 3 depicts the components of our SysADL Execution Tool. Basically, the **SysADL Model** (model.sysadl) file with all architectural views of the model is applied to the transformation process, resulting in a **JavaScript Model** (model.js), corresponding to the same elements of the SysADL model but defined in JavaScript, with the capacity for execution. Next, we will explain the other elements of the execution tool.

### 6.1 Language Definitions

Peggy[3] is a Parsing Expression Grammar (PEG-based) parser generator written in JavaScript that allows you to define grammars declaratively. Based on the grammar described in its syntax, it automatically generates a parser capable of recognizing and interpreting input strings according to the defined rules. The generated parser produces an Abstract Syntax Tree (AST), which can be used directly in applications such as compilers, interpreters, or model transformers. As depicted in Figure 3A, we defined the SysADL grammar using Peggy (sysadl.peg), and through a Peggy procedure, we generated the language parser (sysadl-parser.js) based on the grammar. This parser is used by the SysADL Execution Tool to extract the elements of a SysADL model.

### 6.2 Operational Semantics Base

To serve as a semantic operational foundation, we developed the base library (SysADLBase.js), a base class for the SysADL runtime, providing the generic infrastructure needed to represent and execute SysADL models in

[3] peggyjs.org