Towards a graphical representation for the Æmilia Architecture Description Language

Fabio Calvarese  
Dipartimento di Informatica, Università dell’Aquila, Via Vetoio, L’Aquila, Italy  
fabio_calvarese@tele2.it

Antinisca Di Marco  
Dipartimento di Informatica, Università dell’Aquila, Via Vetoio, L’Aquila, Italy  
antinisca.dimarco@univaq.it

Ivano Malavolta  
Dipartimento di Informatica, Università dell’Aquila, Via Vetoio, L’Aquila, Italy  
vano.malavolta@univaq.it

Abstract—Software verification and validation has always been a hot-topic in software engineering and it is becoming even more important today due to the growing complexity of software systems. Checking functional and non-functional properties on modern systems demands a strong expertise in the use of specific tools, methods and languages. Such expertise is not common inside development teams and this obstacles the regular application of validation techniques.

Æmilia is an Architecture Description Language (ADL) based on a process algebra that allows designers to specify and validate component-based systems with respect to functional, performance and security aspects. The specification of a complex application with Æmilia can be difficult and error-prone due to the carefulness that a language based on process algebra requires.

In this paper we present two initial graphical representations for the Æmilia ADL: the first one is based on UML and the other one is based on DrawNET. Two tools that generate Æmilia textual specifications from the presented graphical descriptions are also presented. Finally, we discuss advantages and disadvantages of the two presented graphical languages.

I. INTRODUCTION

Software verification and validation has always been a hot-topic in software engineering and it is becoming even more important today due to the growing complexity of systems. Nowadays, software systems are checked with respect to functional and non functional requirements through validation models specified at the architecture level. However, most of these models are defined by means of complex textual specifications; so it is difficult to use those languages and their corresponding methodologies in industrial contexts; this is mainly due to the fact that such notations require special skills and knowledge, generally missing in industrial teams.

To simplify the modeling and the integration of the analysis process in the software life-cycle, several researchers have devised approaches that generate ready-to-validate models from the software models used in the development process [1]. From the analysis of such generated models, insights on the quality of the software system under development can be assessed and improved. These approaches can be integrated into a framework for the development of software systems allowing the analysis of functional and non-functional aspects. Such a framework would guide the whole development process of software products that satisfy their functional and extra-functional requirements in a systematic way.

Æmilia [4] is an ADL based on the EMPAₚ process algebra [3]; it allows designers to model the architecture of component-based software systems. Æmilia permits designers to specify stochastic aspects (such as the time needed to execute an action/components interaction) of the software architecture, allowing both functional and non-functional analysis (in particular performance and security). TwoTowers [2] is the tool that automatically performs such analyses. Unfortunately TwoTowers takes as input only textual descriptions. Even if the ideas behind Æmilia are intuitive, the specification of a complex applications can be difficult without a graphical support. Mainly this is due to the carefulness that a language based on a process algebra requires. Hence, the lack of graphical editors prevents the usage of Æmilia and its verification support in industrial development process.

The main purpose of this work is to understand the feasibility of providing intuitive and usable graphical representations for the Æmilia ADL. More specifically, in this paper we (i) define two graphical representations for the Æmilia ADL and (ii) present two tools that generate the Æmilia specifications starting from the just defined graphical notations. The graphical languages we present are realized by (i) profiling UML component and state machine diagrams and (ii) defining a new graphical notation with the DrawNET modelling tool. The final aim is to provide an integrated framework for designers using UML as software modeling language and a more agile environment for those using different modeling solutions. We show both the modeling techniques and the equivalence between the two generated Æmilia descriptions by modeling the well-known dining philosophers problems. Of course, the long-term goal of this work is to provide a unique graphical representation for Æmilia, and to integrate it into TwoTowers (i.e., the Æmilia modeling tool).

The paper is organized as follow: Section II briefly recalls the Æmilia ADL. Section III describes the common rational of our approach. In Section IV and Section V we describe the proposed graphical languages together with the corresponding tools. Section VI gives a proof of concepts by showing the application of the two approaches on the dining philosophers problem. Section VII presents related work and Section VIII concludes the paper.
II. THE ÂMILIA ADL

Âmilia is an ADL based on the SPA EMPAgr [3], introduced by Bernardo et al. in [4] in order to easy the use of Stochastic Process Algebra [7] as software model notation. Âmilia provides a formal specification language for the compositional and hierarchical modeling of software systems, which is equipped with suitable checks for the detection of possible architectural mismatches. A description in Âmilia represents an Architectural Type, a family of software architectures having the same observable functional behavior and topology, while the internal behavior and the extra functional characteristics can vary. An Architectural Type consists of some Architectural Element Types (AET) and its architectural topology. Figure 1(a) shows the structure of an Âmilia textual description.

![Architectural Element Instance](image)

(a) Structure of an Âmilia textual description.

An AET is defined by (i) its behavior, specified either as a set of EMPAgr sequential terms or through an invocation of a previously defined Architectural Type, and (ii) its interactions, specified as a set of EMPAgr action types occurring in the behavior. EMPAgr is a stochastic process algebra that permits to add temporal information to actions by means of continuous random variables, representing activity durations. This extension permits to build a stochastic process making possible the quantitative analysis of the modeled system.

The architectural topology is specified through the declaration of a set of Architectural Element Instances and a set of Directed Architectural Attachments among the interactions of the instances. Obviously more instances on an AET can be instantiated in the architecture.

Âmilia is equipped with a tool, called TwoTowers [2] that takes as input an Âmilia textual description, translates it in EMPAgr processes, and provides functional verifications and performance evaluation capabilities.

Finally, Âmilia provides a helpful graphical notation for the design of the architecture topology of complex systems. Such a graphical notation is based on flow graphs [8]. In a flow graph representing an Âmilia architectural description, the boxes denote the Architectural Element Instances, the black circles denote the local interactions, the white squares denote the architectural interactions and the directed edges denote the attachments (see Figure 1(b)). However, there is not a graphical editor that can support the whole Âmilia specification.

III. THE PROPOSED SOLUTIONS: A COMMON RATIONAL

The presented results come out from two different experiences conducted separately on purpose. Even if the two project teams worked autonomously, the provided solutions have several points in common in terms of (i) types of graphical models to provide, (ii) generation process of an Âmilia description and, consequently, the high-level structure of the tools realizing the generation.

Both the teams have specified two types of graphical models: (i) one modeling the internal behavior of the component types (AET) in terms of state transitions triggered by actions, that is a state diagram; (ii) one representing the flow graph, that is the topology of the architecture. Hence an Âmilia specification consists of a set of state diagrams describing the AET and a flow graph for the architecture topology.

The steps of the process (please refer to Figure 2) in both cases are: (i) graphical modeling through existing case tools (Poseidon and DrawNet, respectively) possibly extending/making the provided original notation, (ii) XML exporting of those models, (iii) generation of the Âmilia textual description through implemented filters (SEAL and DAX tools, respectively) that take in input the XML exporting, (iv) analysis of the generated Âmilia specification by means of TwoTowers tool.

![Process Diagram](image)

Fig. 2. General Process

As we show later in Section VI the generated Âmilia specifications by the two solutions are equivalent w.r.t functional and performance aspects. In the following we detail the two proposed approaches.

IV. UML-BASED GRAPHICAL SUPPORT TO ÂMILIA

In this section we describe the UML notation for Âmilia and the tool that derives the Âmilia textual specifications. We use extended UML state and component diagrams to model internal component behavior and the architectural topology respectively. For what concerns state diagrams, their basic elements are: (i) simple state modeling the working state of a component, (ii) transitions between states modeling the actions executed by a component, and (iii) initial state indicating the activation point of the component behavior. For the sake of translation, the initial state has to be connected to only one state through a single transition. In order to annotate in the diagram the rate of the actions, we use the Schedulability, Performance and Time UML profile [10]. We stereotype the transitions with \texttt{<<PAdemand>>}, noting the action rate in the \texttt{PAdemand} tag value. Figure 3 summarizes the used state diagram elements.
In Æmilia we can distinguish three types of rates that identify three different action types namely immediate, passive and exponential timed actions.

The rate of an exponentially timed action (exp) is given by an expression, whose value must be a positive real, that is interpreted as the rate of the exponentially distributed random variable describing the action duration. The rate of an immediate action (inf) is expressed through a priority, given by an expression whose value must be an integer not less than one, and a weight, given by an expression whose value must be a positive real. The rate of a passive action ($) is again expressed through two expressions denoting a priority and a weight, respectively. If not specified, the values of the priority and the weight of an immediate or passive action are assumed to be one.

To have flexibility on creating an Æmilia textual specification, we have defined different patterns to specify the action rate that has to be assigned to $\text{PAdemand}$ tag value. Those pattern extend the set of value assignable to this tag value and they have been created according to the Æmilia textual notation grammar. Such pattern are summarized in Table I where at the left-hand side the syntax used in the tag value is shown in correspondence to the Æmilia syntax rate reported in the column at the right-hand side. We would just pointing out that the last four patterns are special ones, used when the rate is specified through a variable in a parametric way. In these cases, the variable is opportunely defined and initialized in the header of the EMP $\text{Agr}$ process corresponding to the state diagram.

<table>
<thead>
<tr>
<th>Tag $\text{PAdemand}$ syntax</th>
<th>Æmilia rate</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;exp\',\,valueExpr\'&gt;</code></td>
<td>$\text{exp(valueExpr)}$</td>
</tr>
<tr>
<td><code>&lt;inf\',\,weightExpr\',\,probExpr\'&gt;</code></td>
<td>$\text{inf(weightExpr, probExpr)}$</td>
</tr>
<tr>
<td><code>&lt;name\',\,inf\',\,type\',\,ProbExpr\',\,weightExpr\'&gt;</code></td>
<td>$\text{type name:= ProbExpr inf(weightExpr, name)}$</td>
</tr>
<tr>
<td><code>&lt;name\',\,weightExpr\',\,probExpr\'&gt;</code></td>
<td>$\text{(weightExpr,probExpr)}$</td>
</tr>
<tr>
<td><code>&lt;name\',\,exp\',\,type\',\,valueExpr\'&gt;</code></td>
<td>$\text{type name:= valueExpr exp(name)}$</td>
</tr>
<tr>
<td><code>&lt;name\',\,inf\',\,type\',\,weightExpr\'&gt;</code></td>
<td>$\text{type name:= ProbExpr inf(1, name)}$</td>
</tr>
<tr>
<td><code>&lt;name\',\,type\',\,ProbExpr\',\,weightExpr\'&gt;</code></td>
<td>$\text{type name:= ProbExpr (weightExpr, name)}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usable patterns in the graphic notation.</td>
</tr>
</tbody>
</table>

In Figure 4 we report, as example, the state diagram modeling the internal behavior of a chopstick for the dinning philosophers case study.

The topology of an Æmilia description is instead modeled as a component diagram. To properly model an Æmilia architectural topology we must address the following issues:

- we must clearly introduce in the component diagram the concepts of instance and instance type;
- in the topology, Æmila requires the definition of attachments among architectural element instances that model both the connections among the components and the control flow in the system, whereas in the component diagram the latter is missing;
- we need to explicitly indicate which are the local and architectural interaction and their type, that is which are input/output interactions and when the local interactions are OR, UNI or AND ones.

<table>
<thead>
<tr>
<th>UML element</th>
<th>Notation</th>
<th>Description/constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td><code>&lt;component&gt;</code></td>
<td>It represents one or more instances of a software component.</td>
</tr>
<tr>
<td>Dependence</td>
<td>`</td>
<td>--</td>
</tr>
<tr>
<td>Port</td>
<td><code>[[Name interaction]]</code></td>
<td>It represents local UNI/OR/AND interaction. This is modelled using the element multiplicity: multiplicity $= 1$ for UNI interaction type, multiplicity $= 1 \ast$ for OR interaction type, multiplicity $= 1 \ast \ast$ for AND interaction type.</td>
</tr>
<tr>
<td>Interface</td>
<td><code>[[Provided and Required Interfaces]]</code></td>
<td>They represent input and output architectural interaction, respectively.</td>
</tr>
<tr>
<td>Attribute</td>
<td><code>[[Type]]</code></td>
<td>It allows the designer to assign a previously defined behavioural type.</td>
</tr>
</tbody>
</table>

In Figure 5 we report the elements used to model an Æmilia topology allowing us to fix the issues above. In particular, we use the Component element to model an Architectural Element Instance, the UML Dependence for the attachments, the port feature for the local interactions and the provided and required interfaces for architectural input and output interactions respectively. To bind the internal behavioral definition with an instance, we introduce an attribute that allows the user to specify the state diagram of the behavior previously specified. Even if in UML it is possible to define a state
machine within a component, we do not use this UML feature since this imposes the definition of the same behavior many times, one for every instance of an architectural element type introduced in the topology, even if it is the same for all instances. Finally, we use different multiplicity associated to a port to model UNI/OR/AND interaction. More precisely we use 1 multiplicity for UNI interactions, 1..* for OR interactions and * for AND interactions.

In Figure 6 we report the topology of Dinning Philosopher case study with three philosophers and three chopsticks modeled by means of a component diagram.

![UML component diagram for the 3 Dining Philosophers example.](image)

Software architecture uml modeling of AemiLia Specification (SEAL) tool: We have developed the Software architecture uml modeling of AemiLia Specification (SEAL) tool to transform the UML-based Aemilia graphical notation to the textual description. SEAL tool takes as input the XMI exporting of Poseidon 4.01, that is the UML case tool considering for the graphical modeling step. The tool uses a XMI Parser that is able to find the information and to structure it in an intermediate representation java class based. This representation allows a simple use of the data in the transformation phase. After this step, the tool operates the transformation towards the Aemilia syntax. All the elements are mapped in the corresponding textual notation that, at the end of this process, can be saved.

The architecture of the SEAL tool has been shown by means of the component diagram in Figure 7. The tool is composed of four components: the GUI, the Parser, the Data Structures and the Transformer. The tool operations are available in the GUI component. The user can invoke from the GUI the load of the XMI exporting of the model. This operation, provided from the Parser, starts the process of transformation. The Parser parses the XMI exporting of Poseidon UML case tool and all the elements parsed are loaded in the Data Structures. When the parsing finishes, the Data Structures is populated from a class instance representation of the UML diagram, so the Parser can leave the XMI representation and start the transformation process offered by the Transformer component.

This component communicates with the Data Structures component through the getElements interface to find the elements needed for the transformation. At the end of this step, the result is returned to the GUI component that can show the result in the user interface.

The graphic user interface is very simple and intuitive. It consists of a simple editor that permits to visualize the textual notation resulting from the transformation process. From this interface we can open a XMI file resulting from the Poseidon exporting and start the transformation. The transformation result is showed in the editor that is able to identify the keywords of the Aemilia textual notation and to use different colors to have a better representation. It is also able to recognize some semantic errors (e.g., actions with the same name that are specified with different rates), that are signaled in red color. Finally the editor signals default specifications, such as for example instances multiplicity values, through the yellow color to help the designer to quickly find out aspects not modeled from himself.

SEAL tool also performs some check to make sure that the graphical notation in input is without semantic error (syntax errors can prevent the transformation).

V. INTRODUCING A GRAPHICAL SUPPORT TO AEMILIA: AN ORIGINAL SOLUTION

In this section we present the second solution for Aemilia graphical notation based on DrawNET [6] modeling system that is a framework supporting the design of models expressed in any graph-based formalism. We have used this tool to define the formalism (associated with the graphical notation) representing the basic constructs of Aemilia. The user can define an Aemilia specification through this graphical notation and then generate the Aemilia textual description.

The project has been realized into two steps: (i) DrawNET formalism definition for flow graph and the state-charts used to model architectural topology and the behavior of architectural element types, respectively; (ii) development of DAX (from Drawnet to Aemilia teXtual description) tool that makes the translation towards Aemilia.

DrawNET modeling System (DMS) is an open environment that includes an XML based language family used to define formalisms and multi-formalism models. Definition and developing of formalisms are supported by a generic Graphical User Interface (GUI), an abstract Data Definition Language (DDL), and a set of access libraries, written in Java language, and based on XML technology. One of the most important
aspects of the DrawNET is its openness, allowing the user to add components to the system.

In DrawNet, it is possible to define a new formalism by means of the DDL and then create a model. To this aim DDL is composed of two levels: the first level is used to describe the formalism by the list of possible basic elements (formalisms form a hierarchy, enabling a rapid derivation of new ones); the second level, that is the element level, is used to define hierarchical graph-structured models from a given formalism. Models consistency checks are permitted throughout constraints associated to the edge-types of the formalism.

The above levels are composed of a set of XML-based languages. In particular,

- the formalisms are represented by: (a) Formalism Definition Layer (FDL) that defines the set of primitives of the formalism; (b) Result Definition Layer (RDL) that defines which results may be computed; (c) Formalism Representation Layer (FRL) that defines the graphical notation of the models conforming to the formalism;
- the models are represented by: (a) Model Definition Layer (MDL) that contains the definition of the model; (b) Model Query Layer (MQL) that defines which results must be computed; (c) Model Representation Layer (MRL) that contains the graphical aspects of the model.

Our work consists of defining a new formalism for Æmilia, and developing a transformation tool that creates the Æmilia specification by analyzing only the Model Definition Layer.

The new formalism we defined in DMS provides the following elements to model the behavior of an architectural type whose graphical symbols are reported in Figure 8. We recall that we use state-charts to model such behaviors. The elements and their properties are:

- initial node of the state-chart diagram;
- generic node of the state-chart diagram; a mnemonic name can be associated to every node;
- formal parameters used in any transition of the state-chart diagram. Its properties are: (i) name that is a unique name identifying the parameter; (ii) type that defines the type of parameter. It can represent every Æmilia type (integer, real, boolean, list, array, record, prio, rate, weight). (iii) value used to assign an initial value to the parameter. If the user wants to define a default initial value, this property can be set to default value; in this case the DAX tool will automatically set the parameter with the default value, according to its type.
- transition of the state-chart diagram: this element is mapped to an Æmilia action. Its properties are: (i) name that identifies the action type in Æmilia; (ii) action rate indicating if the action is exponential timed, immediate or passive; (iii) Condition representing the optional condition related to the action; (iv) from representing the source node of the transition; (v) to representing the target node of the transition; (vi) Input-Interaction that is a boolean property, if its value is true the action is translated as input-interaction of the architectural; (vii) Output-Interaction that is a boolean property, if its value is true the action is translated as output-interaction of the architectural; (viii) Interaction-Type that contains the type of the action (UNI, OR, AND).

Table II contains the relations between the combined values of Input-Interaction and Output-Interaction properties and the generated Æmilia specification.

The Æmilia architectural topology is instead modeled through a flow graph. The DAX notation for flow graph is composed by the following elements, also summarized in Figure 9:

- Instance: it models an instance of architectural type and contains the type property defining the architectural type

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Description</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td></td>
<td>Instance of an architectural type.</td>
<td>- Name</td>
</tr>
<tr>
<td>Archi interaction</td>
<td></td>
<td>Point of interaction with an external model.</td>
<td>- Selector, Archi-Interaction</td>
</tr>
<tr>
<td>Archi Interaction Connector</td>
<td></td>
<td>Link between an Archi Interaction and its Instance</td>
<td>- Name, From, To</td>
</tr>
<tr>
<td>Attachment</td>
<td></td>
<td>Interaction between two instances.</td>
<td>- Name, Input-Interaction, Output-Interaction, From, To</td>
</tr>
</tbody>
</table>

**TABLE II**

**RELATIONS BETWEEN INPUT-INTERACTION AND OUTPUT-INTERACTION.**

![Fig. 9. Topology graphical elements.](image-url)
of the instance. The values of the type property are dynamically generated by DAX.

- Archi Interaction: it represents an optional point of interaction with an external model; it is composed of the following parameters: (i) Archi-Interaction defines the interaction of the type of instance to which it is associated; and (ii) Selector (optional) that is an integer expression. The values of the Archi-Interaction property are dynamically generated by DAX.

- Archi-interaction Connector is the link between an Archi-Interaction and its associated instance. It contains two parameters, from and to, that represent the linked Instance and Archi-Interaction;

- Attachment: it models an interaction between two instances contained in the topology; it has the following parameters: (i) Input-Interaction representing the local target interaction of the attachment; (ii) Output-Interaction representing the local source interaction of the attachment; (iii) from representing the source instance of the attachment; and (iv) to representing the source instance of the attachment. The values of Input-Interaction and Output-Interaction are dynamically generated.

In Figure 10 and Figure 11 we report the DAX modeling for a philosopher and the topology of 3 Dining Philosophers example.

from Drawnet to Aemilia textual description (DAX) Tool: DAX generates from the defined formalism the corresponding Aemilia textual description taking in input the XML exporting of DrawNet. Its architecture is showed in Figure 12.

In the proposed implementation, we assume that the designer models first the architectural element type behavior and then the topology. To easy the DAX modeling and to reduce the consistency errors in the modeling we implemented an external script, FormalismRefresher, that dynamically updates the XML file of the formalism of the topology by introducing the previously defined architectural element type behavior. In other words, the script adds in the formalism new types usable to define architectural element instances in the topology. In this way, the designer can use, during the topology modeling, a predefined set of architectural element type, architectural interactions and attachments that prevents the consistency errors. Examples of errors that can be prevented are: (i) the use of a non-defined architectural element type, or (ii) the specification of an action as architectural interaction that does not belong to any architectural element type. Obviously this feature speeds up the topology modeling.

The components that implements the DAX tool are: Core, Manager, Input-handler and Output-creator.

Core directly models architectural entities (e.g. element types, connections between element instances) and contains information about each architectural type of the Draw-Net models. It interacts with the Manager component to keep track of input and output interactions, and of the parameters associated with the element type (e.g. the value of service rate). This component models the behavior of architectural element type as a Labeled Transition System described as a set of states and transitions.

Manager contains auxiliary classes that provide services to other modules. In particular, it contains all the interactions declared into the Draw-Net project. Architectural interactions, local attachments and their relationships with the element types are managed by this component. Moreover, it stores and manages the parameters declared into architectural types. These parameters will be translated into the formal parameters of each Aemilia element type.
Input-handler and Output-creator have the responsibility to coordinate input and output phases of the tool, respectively. Input-handler contains classes that check the input and initialize the data structures of the tool. It stores the parameters set launching the tool (i.e., the name of the files to transform) and checks that the provided Draw-Net model contains well-formed XML files. Moreover, it parses the files of the Draw-Net project, gaining information about the architectural types. Output-creator produces the Æmilia specification corresponding to the Draw-Net model. It performs a set of predefined semantic checks on the Draw-Net modeling and manages the creation of the whole Æmilia specification. It interacts with the Manager component that provides the required parameters (and the corresponding values) to instantiate architectural types.

VI. A PROOF OF CONCEPT

As a proof of concept the problem of the dining philosophers has been modeled using both UML and the defined DrawNet formalism. Then each tool has been executed on such models. The tools generate syntactically different Æmilia specifications. Such differences are due to distinct implementations and different starting graphical notations.

The main differences reside in (i) the description of the behaviour of architectural types, (ii) the topology of the architecture. In case (i) the tools perform a different analysis of the state-chart associated to each architectural type.

SEAL tool creates the Æmilia specification of each element type in two steps; it performs depth-first visits of the state-chart associated to each architectural type. DAX does not perform the analysis of the state-chart. It generates Æmilia specification in one pass, associating each state to a unique EMPAgr equation. In this way there is a one-to-one mapping between graphical and textual notations. Listings 1 and 2 report the chopstick specification.

Listing 1. Chopstick element type specification generated by SEAL

```c
ELEM_TYPE chopstick_Type(void) {
    BEHAVIOR
    Chopstick_type(void,void)=
        choice
        <pick_up_then, _>Node122(), 
        <pick_up_first, _>Node121();
    }
    Node122(void,void) =
        <put_down, _>InitialNode20();
    Node121(void,void) =
        <put_down, _>InitialNode20();
    INPUT_INTERACTIONS
    OR put_down;
    pick_up_then;
    pick_up_first
    OUTPUT_INTERACTIONS
    void
}
```

Listing 2. Chopstick element type specification generated by DAX

```c
<pick_up_then, _>Node122(),
<pick_up_first, _>Node121();

Model122(void,void) =
    <put_down, _>InitialNode20();
Model121(void,void) =
    <put_down, _>InitialNode20();

INPUT_INTERACTIONS
OR put_down;
pick_up_then;
pick_up_first

OUTPUT_INTERACTIONS
void

Listing 3. Topology generated by SEAL

```c
ELEM_TYPE Chopstick_type(void)

BEHAVIOR
Chopstick_type(void)=
    choice
    
    <pick_up_first, _>Chopstick_type(
    
    }

Listing 4. Topology generated by SEAL

```c
ELEM_TYPE Chopstick_type(void)

BEHAVIOR
Chopstick_type(void)=
    choice
    
END
```

The generated Æmilia topologies differ in the use of multiplicity. It is caused by the fact that SEAL generates the specification starting from a model in UML (in which multiplicity can be used); DAX instead generates the specification starting from an ad-hoc formalism in which multiplicity has been not defined. Listings 3 and 4 point out the differences between the topologies.
In spite of the above mentioned differences, the resulting specifications are semantically equivalent. We adopted the following method to demonstrate it:

1. definition of a reward file (see Listing 5) that can be used to make performance analysis on the architecture of the dining philosophers.

   Listing 5. Reward file to analyze the problem of dining philosophers

   ```plaintext
   MEASURE mean_number_eating_philosophers IS
   FOR ALL i IN I0
   ENABLED(P[i].eat) -> STATE_REWARD(1)
   END
   
   Stationary value of the performance measures for 
   -UNAMED_TYPE:
     -Value of measure "mean_number_eating_philosophers": 0.999347
   
   Listing 6. Result of the analysis performed on the model generated by SEAL (see Listing 6).

2. we use the reward file to make performance analysis on the specification generated by SEAL (see Listing 6).

3. we adapt the reward file to fit the specification created by DAX. This step is unavoidable because DAX does not generate specifications with multiplicity. Listing 7 shows the modified reward file. Syntactic differences between Æmilia specifications make the reward files different. However, they defines the same performance index.

   Listing 7. Reward file modified to fit the specification generated by DAX

   ```plaintext
   MEASURE mean_number_eating_philosophers IS
   ENABLED(10.eat) -> STATE_REWARD(1)
   END
   
   MEASURE mean_number_eating_philosophers1 IS
   ENABLED(11.eat) -> STATE_REWARD(1)
   END
   
   MEASURE mean_number_eating_philosophers2 IS
   ENABLED(12.eat) -> STATE_REWARD(1)
   END
   
   
   Listing 4. Topology generated by DAX

   ```plaintext
   FROM 12.pick_up_left_first TO 13.pick_up_first:
   FROM 11.pick_up_left_first TO 15.pick_up_first:
   FROM 10.pick_up_left_first TO 14.pick_up_first:
   FROM 12.put_down_right TO 15.put_down:
   FROM 12.pick_up_right_first TO 15.pick_up_then:
   FROM 11.put_down_right TO 14.put_down:
   FROM 11.pick_up_right_then TO 14.pick_up_then:
   FROM 10.put_down_right TO 13.put_down:
   FROM 10.pick_up_right_then TO 13.pick_up_then:
   FROM 12.pick_up_right_first TO 15.pick_up_first:
   FROM 11.pick_up_right_first TO 14.pick_up_first:
   FROM 10.pick_up_right_first TO 13.pick_up_first:
   END
   ```

4. we use the modified reward file to make performance analysis on the specification generated by DAX. Since the resulting values of the performed analysis are equal, the generated Æmilia specifications are equivalent from the performance and behavioral points of view.

VII. RELATED WORK

In literature, there are many experiences that defines graphical languages for ADLs. Mainly, such graphical languages are based on UML. In such direction, [9] aims at assessing of UML’s expressive power for modeling software architectures, considering both UML “as is” and (lightweight and heavy-weight) UML extensions that incorporate useful features of existing ADLs. Hence, there are three ways to use UML: (i) by trying to identifying UML elements that could map the ADL characteristics [5]; (ii) by defining new UML profiles that capture the characteristics of the ADLs they refer to and possibly defining OCL constraints to guaranties model properties [11]; and (iii) by defining changes on the UML meta-model to allow the modeling of ADL aspects [12].

Our work differs from all of them since we implement two tools that from the graphical models conforming to the defined languages, automatically derive the corresponding Æmilia textual description.

VIII. DISCUSSION AND CONCLUDING REMARKS

In this paper we presented two graphical notations for the Æmilia ADL and the corresponding generative tools towards its textual description. Generative tools allow designers to use such a language for software specification and (functional and extra-functional) analysis. The first graphical representation (SEAL) is based on a UML profile that tailores UML component and state machine diagrams with concepts from the Æmilia ADL. The other graphical representation (DAX) is based on the DrawNet modeling system. We decided to provide two different graphical languages in order to support both the designers using UML as software modeling notation and the designers that use software models different from UML. The first solution, in fact, integrates with UML modeling environments in those cases where the UML is used as modeling notation; the second one provides a more nimble environment for teams using modeling languages different from UML.

Supposing that designers use UML as modeling notation, the UML based language requires little modeling efforts compared to the DAX one. This is due to the fact that the additional work for the UML designers is to stereotype the component and the state diagrams elements accordingly with the defined profile. More effort is required while adopting the DAX approach: designers have to model again the software system in the DAX formalism. However, DAX notation and the relative tool is easy to use by both beginners and experts.

Differently to the SEAL approach, DAX approach does not allow: (i) the parametric modeling of process behavioral (more precisely it is not possible to specify EMPA_gr process terms having parametric equation headers); (ii) the specification of instance multiplicities preventing the usage of compact Æmilia topology specification. These limitations depend on the defined modeling language that does not consider such aspects. However, their treatment may be easily introduced even if they are not fundamental.

For what concern the transformation algorithms, both DAX and SEAL, show a complexity equal $\Theta(I + A)$ (where I is the number of Architectural Element instances and A the number of attachments in the modelled topology) for the transformation of the architectural topology. Instead, for the translation of an architectural element type behavior, DAX has a complexity equal to $\Theta(S)$ whereas SEAL shows a complexity of $\Theta(S + T)$ (where S and T are the number of states and transitions in the state diagram, respectively).
This difference is due to the different transformation algorithm applied: (i) SEAL performs depth-first visits of the state diagram, then information gained from the visits are used to generate the specification making the generation specification readable from humans; (ii) DAX does not traverses the state diagram going along the transitions, but it accesses to the data structure modeling the state diagrams and directly translates the states and all the out-going transitions.

Both the devised tools provide modeling helps. SEAL is equipped by an editor that highlights the specification errors over the Æmilia textual description resulting form the transformation step. DAX checks that an action has not been declared as input and output interaction at the same time; and it verifies the correctness of the attachments in terms of type of the interactions and instances involved in (for example an attachment must be defined between two different instances). Moreover, DAX imposes that the architectural element types must be specified before the design of the architectural topology. The new defined AETs are embedded in the DAX formalism for the topology. In this way, when the designer models the topology, DAX suggests her the defined AETs as types to use in the specification of architectural element instances.

As future work, several directions can be taken into account. First of all, we are working on the definition of a metamodel representing all the concepts of the Æmilia ADL, and then we will utilize it as the main driver of an ad-hoc graphical tool based on Eclipse technologies (like EMF, GMF, XText). This metamodel will fit best the Æmilia language and we will exploit all the well-known facilities provided by the Eclipse platform. Secondly, the aspects that we did not considered (e.g., hierarchical modeling) could be addressed in future. Moreover, since to execute performance analysis, TwoTowers (the tool supporting the verification and analysis of software architecture) requires the definition of performance indices through the reward technique, another future work would be the definition of a (graphical) language for the specification of performance indices that hides the reward definition. From our experience, on one end, the definition of performance indices through rewards it is not an easy task. On the other end, in general, such a definition has a common pattern hence can be standardized. Finally, one can be study how to enrich the proposed graphical languages to support also the modeling of security aspects already dealt by Æmilia and TwoTowers.

REFERENCES