An AADL-DEVS Framework for Cyber-Physical Systems Modeling and Simulation Supported with an Integrated OSATE and DEVS-Suite Tools

Ehsan M. Ahmad
*College of Computing and Informatics,
Saudi Electronic University,
Riyadh, Saudi Arabia*

Hessam S. Sarjoughian
*Arizona Center for Integrative Modeling & Simulation
School of Computing, Informatics, and Decision Systems Engineering,
Arizona State University, Tempe, Arizona, USA*

March 9, 2020

Contents

List of Figures

List of Tables

Listings

Abstract

The continuing rise of complexity in mixed computational and physical systems demands dynamical models representing structure and behavior together. The software and, more broadly, system architectures are essential in tackling high-level complexity. System architectures are used to define the structures of components and their relationships. Architecture specifications focus on the requirements and static aspects of systems. The specifications detailing the behavioral complexity of the components and their interactions are needed. Design specifications define how components should react to the external stimuli it may receive over some period of time. Therefore, the architecture and design specifications serve complementary roles in model-based design and virtual experimentation.

The Architecture Analysis & Design Language (AADL) and Discrete Event System Specification (DEVS) are proposed as complementary methods for developing, in a step-wise fashion, architecture and design models. The proposed AADL-DEVS framework is grounded in the foundational modularity and hierarchy principles that are common to the DEVS and AADL modeling methods.

A DEVS Annex (DA) is specified according to the DEVS specification for AADL. This annex supports defining discrete-event component structure and behavior in terms of state transition functions with and without external inputs, output functions, and time advance functions. The DA is defined according to the DEVS-Suite simulator which conforms to the parallel DEVS specification. The Open-Source AADL Tool Environment (OSATE) is extended according to the DEVS-Suite simulator. An implementation of this framework supports translating AADL-DEVS models to models that can be simulated in the DEVS-Suite simulator. The Eclipse-based AADL-DEVS framework is developed to afford as much as possible seamless AADL to DEVS model development and simulation.

The DEVS Annex modeling in OSATE and its simulation in the DEVS-Suite simulator are demonstrated using a model of an infant incubator, a time-critical and safety-critical system. This system exhibits reactive computation, concurrency, and feedback control for Cyber-Physical Systems and, more generally, Systems-of-Systems. The example demonstrates adding basic time-based behaviors to the components of an AADL model of the infant incubator. The simulation model and its execution are described. Selected related works, future work, and conclusion are described.

***Keywords*:** AADL, Behavior Modeling, Cyber-Physical Systems, DEVS, DEVS Annex, DEVS-Suite, OSATE, Safety-Critical Systems

1. Introduction

Building Cyber-Physical Systems (CPS) or more broadly Systems-of-Systems (SoS) is challenging, in part, because of a variety of concepts, methods, frameworks, and tools that are needed to tackle their multifaceted nature [4, 38, 32]. Models that can be used together architectures and designs are challenging to develop in a seamless fashion. There are conceptual gaps in connecting, for example, across multiple specification abstractions.

The Architecture Analysis & Design Language (AADL) [9] and Discrete Event System Specification (DEVS) ([37] provide a foundation for developing specifications that cover and integrate high-level architectural and low-level design abstractions. That is, a key to the development of the Cyber-Physical Systems and Systems-of-Systems is to have specifications that can precisely represent decisions spanning both coarse-grain and fine-grain design needs. The former generally lends itself for characterizing architectures (i.e., components and their relationships) of systems. The latter supports theoretically grounded specifications for hierarchical component behaviors that conform to the defined system architecture specifications. This two-step process belonging to the full CPS and SoS life-cycle development involves combining structures and behaviors at multiple abstraction levels. These observations highlight the importance of addressing system structure and behavior complexity traits using integrated architecture and design models.

Grounded in the AADL and DEVS modeling methods, the rest of this report details the proposed and developed AADL-DEVS framework. Section 2. introduces atomic and coupled DEVS with DEVS-Suite simulator and AADL with its structure and behavior specification mechanism. Section 3. presents a detailed description of the proposed AADL-DEVS framework with descriptions of structure and data modeling using core AADL and behavior modeling using DEVS annex (DA). An AADL to DEVS CoDe generation Engine (ADCoDE) is described for automated simulation code for the DEVS-Suite simulator in this section. In Section 4., a case study on infant incubator (*Isolette*) system is presented to illustrate the use of the DA sub-language and demonstrate the use of the proposed AADL-DEVS framework for time- and safety-critical systems. Section 5. describes the modeling of the *Manage Interface Interface* component of the Isolette system in the AADL-DEVS framework while Section 6. describes the simulation of the component in the DEVS-Suite simulator. Section 7. presents a summary of the related work, and Section 8. summarizes this report.

2. Background

This section presents an overview of the parallel DEVS formalism and the DEVS-Suite simulation framework. Emphasis is on basic atomic and coupled model specification constructs and their realization and execution in the DEVS-Suite simulator. Similarly, the basic concepts of the AADL framework and its implementation environment OSATE are presented. The elemental software and execution aspects of the AADL are highlighted.

2.1. System-Theoretic Discrete-Event Simulation

The Discrete Event System Simulation (DEVS) is generally considered suitable for modeling and simulating systems [37]. Certain classes of software (e.g., [11]), hardware (e.g., [7]), and mixed software/hardware systems (e.g., [23]). As a mathematical formalism, it lends itself for specifying structures and behaviors of Systems of Systems (SoS) including Cyber-Physical Systems (CPS). This modeling formalism is based on the Systems Theory [35] where a system is defined in terms of hierarchical modules that are composed through their inputs and outputs. Models can ~~have~~ communicate arbitrary data types with arbitrary timing and data/event handling. As such, discrete and continuous dynamical systems as discrete event models. Moreover, DEVS can be used to describe any discrete event systems [37]. In DEVS, there are two types of model components: *atomic model* and *coupled model*.

2.1.1. Atomic DEVS Model

A parallel DEVS atomic model is a mathematical structure as defined below

$$DEVS = \langle X^b, Y^b, S, Q, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle \quad (1)$$

where

X^b is a set of input port names, each having a bag of values,

Y^b is a set of output port names, each having a bag of values,

S is a set of sequential states,

Q is a set of *total states* $\{(s, e)|s \in S, 0 \leq e \leq ta(s)$, e is the elapsed time$\}$,

$\delta_{ext} : Q \times X^b \rightarrow S$ is an *external transition function*,

$\delta_{int} : S \rightarrow S$ is the *internal transition function*,

$\delta_{con} : Q \times X^b \rightarrow S$ is an *confluent transition function*,

$\lambda : S \rightarrow Y^b$ is an *output function*, and

$ta : S \rightarrow \Re^{+}_{0,\infty}$ is a *time advance function*.

The input and output ports with their values (i.e., primitive or compound messages) are used to specify the exterior structure of every atomic model. The internal behavior of an atomic model is specified in terms of a set of state variables and a set of functions. A model can have autonomous and reactive behaviors specified in terms of an *internal transition* function and an *external transition* function, respectively. The *output function* is for generating output messages for any number of output ports. The *time advance function* captures the timing of state transitions. The *confluent function* can be used for specifying simultaneous handling of internal and external events. An atomic model can have multiple input and/or output messages. The elapsed time e has the role of allowing external inputs to arrive at arbitrary time instances. The $ta(s) = 0$ allows instantaneous state change which is a basic capability for modeling concurrent and distributed software/hardware systems.

2.1.2. Coupled DEVS Model

A parallel DEVS coupled model is a mathematical structure as defined below

$$CM = \langle X^b, Y^b, D, M_d | d \in D, EIC, EOC, IC \rangle \tag{2}$$

where

X^b is a set of input port names, each having a bag of values,

Y^b is a set of output port names, each having a bag of values,

D is a set of component names,

M_d is a set of basic components for each $d \in D$,

EIC is a set of external input couplings,

EOC is a set of external output couplings, and

IC is a set of internal couplings.

A coupled model is composed of one or more atomic and/or coupled models. The input and output ports and values have the same specification as the atomic model. The structure specification of a coupled model includes input & output ports, a set of components, and component coupling information. DEVS can ensure semantically identical input/output interfaces for atomic and coupled models. The coupling information is categorized as (1) the *external input coupling* (`EIC`) - coupling of coupled model input ports to input ports of some component, (2) the *external output coupling* (`EOC`) - coupling of component output ports to the coupled model output ports, and (3) the *internal coupling* (`IC`) - coupling of component output ports to input ports of components. A coupled model behavior is based on the message exchanges between itself and its components as well as message exchanges among the components (which can be atomic or coupled models), through couplings. The coupling provides interaction between components. DEVS enjoys the property of closed under coupling where any coupled model can be transformed to an atomic model with identical behavior. This property supports modeling larger models in a hierarchical manner. For some purposes, models simulated in real-time, instead of step-wise or logical-time, are needed. For this purpose, the Action-Level Real-Time (ALRT) Discrete-Event System Specification (DEVS) modeling and simulation

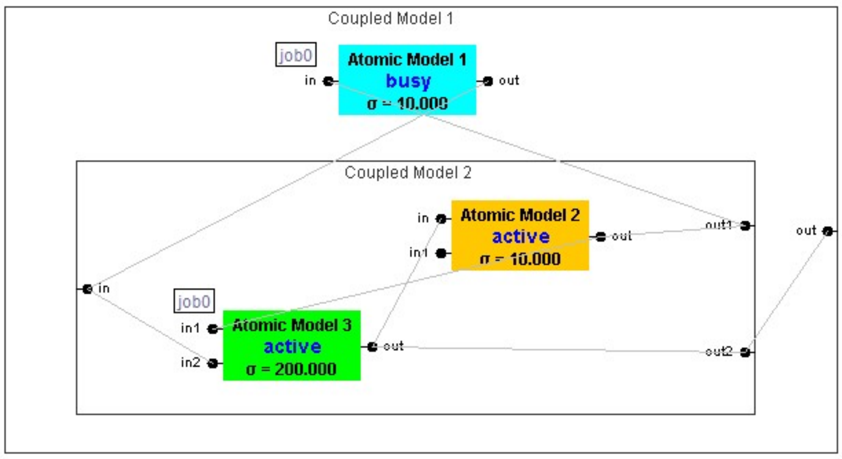

Fig. 1. Graphical notation for Atomic and Coupled Models in DEVS-Suite Simulator

formalism is developed using real-time Statecharts [26]. The internal and external transition functions for ALRT-DEVS are defined at the level of actions with time-windows which in turn support fine-grain execution as well as error handling. Actions are specified to allow execution in real-time under constrained computational resources. General-purpose DEVS models can be specialized to represent software and hardware types with mapping the former to the latter [23].

2.2. DEVS-Suite Simulator

The DEVS-Suite simulator is one of the most commonly used modeling and simulation tools for the parallel DEVS formalism [39, 30, 1]. This simulator has a modeling and simulation engine. The modeling part supports implementing DEVS atomic and coupled models that have hierarchical tree-structures. The *viewableAtomic* and *viewableCoupled* Java classes allows visualization of the atomic and coupled models (see Figure 1). Both atomic and coupled models can receive input and send output entities (e.g., `job0`) only via separate input and output ports. Entities can strictly be transmitted via couplings between any two distinct models that are at the same level belonging to a single node in the hierarchy. The entities do not change during transmissions. The simulation part implements a message-based simulation protocol. It is responsible for executing the internals of atomic models as well as all entity transmissions among atomic and coupled models.

The simulator is developed following the Model-View-Controller (MVC) architectural style [28]. The Model conforms to the atomic and coupled model types and supports their simulation using the parallel abstract protocol [37]. The simulator can execute in logical-time. The internal and external transition functions defined according to the ALRT-DEVS formalism can be executed in near hard real-time execution. The ALRT-DEVS modeling and simulation is supported in the Parallel DEVS-Suite simulator. The FaÇade hides specific design choices and implementations of the atomic and coupled models and the abstract simulator while exposing the data and control to the Control and View. It facilitates loose coupling from Model to View and Controller. The Control provides comprehensive means to configure models and their executions including logical and near hard real-time as well as coordinating run-time simulation animation with timeview trajectories [17, 29, 40]. The View provides unique capabilities including highly flexible configurations for super-dense time-trajectories and tabular input, state, and output data which is key for large-scale model development, simulation, and debugging. The open-source simulator is developed using Java Programming language and the Eclipse BIRT plugin [1].

2.3. Architecture Analysis & Design Language

Architecture Analysis & Design Language (AADL) is an SAE International standard language for the architectural description of embedded systems. It is a Model-based Engineering (MBE) approach which promotes *analyzable* architecture development that enables the dependability prediction of real-time embedded systems. MetaH [34], a prototype of the AADL, was developed by Steve Vestal at Honeywell with the aims of extensive model validation,

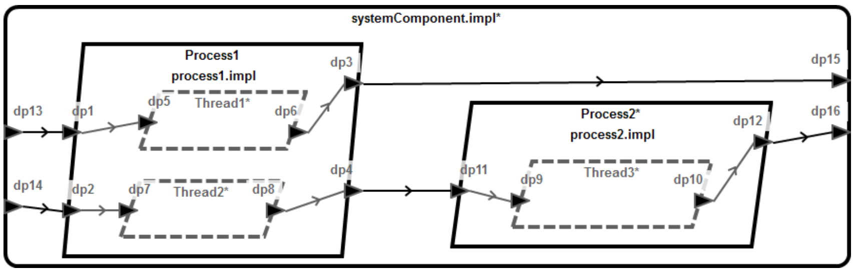

Fig. 2. Graphical notation for hierarchical modeling in AADL

better quality system design, and decreased time-to-market for the aerospace industry. MetaH is a proven concept and have been researched by both academia and industry for more than a decade.

AADL was introduced in 2004 and revised in January 2009. The latest version, AADL V2.0, available as SAE AS5506C, was revised in August 2017 [15]. AADL has been considered by embedded system designers at Honeywell, Rockwell Collins, Lockheed Martin, the European Space Agency, Airbus and other safety-critical industries. An important collaborative System Architecture Virtual Integration (SAVI) project for designing complex distributed aerospace systems has selected AADL as its architecture description language [10]. SAVI emphasizes an "*Integrate, Then Build*" approach—the key concept being to verify virtual integration of architectural components *before* implementing their internal designs. AADL supports virtual integration through an effective mechanism for component contract specification based on interfaces and interactions and through well defined semantics for extensive formal analysis at different architecture levels.

Architectural modeling in AADL is realized through component specification of both the *application software* and the *execution platform*. Component *Type* and *Implementation* classifiers, corresponding to embedded system entities are instantiated and connected together to form the system architecture model. It supports textual, graphical and XML Metadata Interchange (XMI) specification formats. Figure 2 depicts hierarchical system modeling in AADL using graphical notations.

Application software may contain *process*, *data*, *subprogram*, *thread*, and *thread group* components. The process component represents a protected memory space shared among thread subcomponents. A data component represents a type, local data subcomponent, or parameter of a subprogram, *i.e.*, callable code. A thread abstracts sequential control flow. In Figure 2, a composite *systemComponent* is modeled with two process components, *Process1* and *Process2*. Two thread components *Thread1* and *Thread2* are the subcomponents of *Process1* while thread *Thread3* is a subcomponent of *Process2*.

The execution platform is made up of computation and communication resources, consisting of *processor*, *memory*, *bus*, and *device* components. The processor represents the hardware and software responsible for thread scheduling and execution. The memory abstraction is used for describing code and data storage entities. Devices can represent either physical entities in the external environment, or interactive system components like actuators and sensors. Physical connections between execution platform components are accomplished via a bus component.

System components represent compound entities containing software, execution platform or other (sub)system components.

Open Source AADL Tool Environment (OSATE) [31] is an open source platform and toolset built on Eclipse to implement AADL for the modeling and analysis of real-time embedded systems. OSATE not only provides full-features text editor and a set of analysis plug-ins, but also supports domain-specific analysis plug-in development.

In order to furnish AADL for DEVS modeling and DEVS-Suite simulation, the core language must be extended to support continuous time-critical modeling. A DEVS behavioral annex targeted for the DEVS-Suite simulator is developed [2]. AADL data, structure, and behavior modeling is further detailed in the next section in relation with the proposed AADL-DEVS framework.

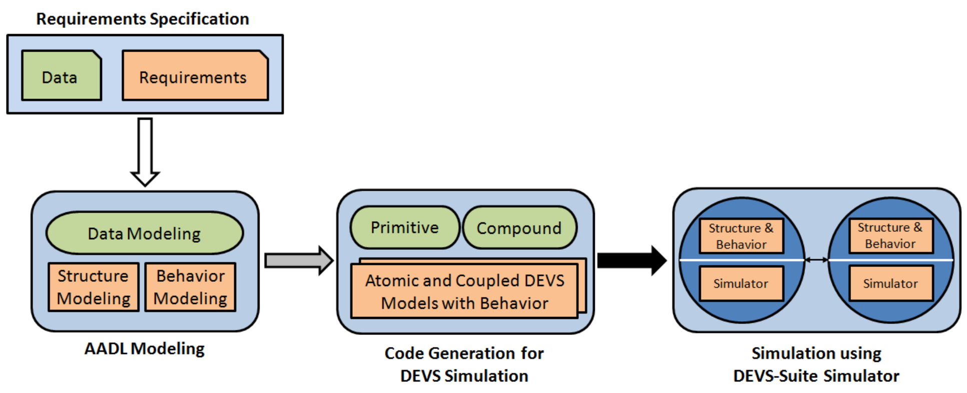

Fig. 3. AADL-DEVS Framework

3. AADL-DEVS Framework

This framework is developed by integrating AADL, a semi-formal architecture description language, and DEVS, a formal discrete-event modeling & simulation language. The result supports both static and dynamic analysis and design of system architectures for safety and time-critical software-intensive system. The AADL and DEVS are specification languages where the former can be used to define possible structural system specifications, and the latter can be used to define combined structural and behavioral system specifications. Each of these languages is modular and hierarchical that makes their integration simpler. The AADL provides explicit types of abstraction for software and hardware components (i.e., nodes) and couplings (i.e., connectors). They can be used to define hybrid system specifications. The DEVS provides general-purpose components (i.e., atomic and coupled models) without the models having designated software, hardware, and hybrid.

The AADL-DEVS framework has two parts – an Annex sublanguage and a code generator for DEVS modeling and simulation. First, the AADL with its OSATE realization is extended with an Annex that conforms to the Parallel DEVS formalism and the DEVS-Suite simulator. This DEVS Annex language is specified in two parts. First, the structural syntax of atomic and coupled DEVS models are added using the OSATE language. In other words, the structural syntax in DEVS-Suite is mapped to that of the AADL syntax in OSATE. Second, the behavioral syntax of the atomic DEVS model is added using the OSATE language. The DEVS Annex structural specification accounts for the differences between the OSATE and DEVS-Suite different primitive and compound data types with different syntax declarations. The data types in DEVS-Suite, unlike their counterparts in OSATE, have behavior. In addition, the DEVS Annex language is extended to support defining the behavior of atomic models (i.e., internal and external functions) as basic state machines. Other functions that are added to DEVS Annex language are for the output and time advance functions.

The second part of the integration is for transforming AADL with DEVS Annex to parallel DEVS models. A code generation engine is developed to transform the AADL-DEVS to atomic and coupled models that can be executed in the DEVS-Suite simulator. As such, first, the AADL-DEVS models are created in OSATE, and second, the models are automatically transformed to simulation code for the DEVS-Suite simulator. In order to generate code for the simulator, it is also necessary to map the input and output primitive and compound data types in OSATE to their counterparts for the DEVS-Suite simulator. This has required two steps. First, looking from the DEVS-Suite language to the OSATE language, a set of generic I/O data types in the mold of the I/O data types are defined in DEVS Annex. Second, looking from the OSATE language to the DEVS-Suite language, the code generation engine is extended to also transform the DEVS Annex I/O data types to their counterparts defined in the DEVS-Suite simulator. Using the developed AADL-DEVS framework, specified parallel DEVS models can be automatically generated, loaded, and executed in the DEVS-Suite simulator. This capability empowers the AADL structural components to be simulatable.

3.1. AADL Modeling

Due to its extensive support for modeling, the AADL-DEVS framework relies on AADL for modeling capability. AADL Modeling in AADL-DEVS framework is focused on Structure, Data, and Behavior modeling based on the data and requirements identified in the requirement specification. Below we detail the structural and data modeling using AADL, while behavior modeling along with the DEVS Annex is described in Section 3.2.

3.1.1. Structure Modeling

In AADL, a structural model of, an embedded system is a hierarchical composition of software and hardware components. Each component declaration incorporates component *type* and *implementation* classifiers to represent externally visible characteristics and internal realization, respectively. A component type declaration defines the interface elements and may contain *Feature*, *Flows* and *Property*. Feature normally contains communication ports. AADL supports *Data*, *Event* and *Event Data* port to transmit data, control and control and data respectively. Port communication is typed and directional. An *in* port receives data/control and an *out* port sends data/control while an *in out* port can send and receive data. A component implementation declaration defines the internal structure in terms of Subcomponents, subcomponent Connections, Subprogram call sequences, Modes, Flow implementations, and Properties. Ports of a component declared in a type declaration are connected through connections in the AADL implementation declaration.

Embedded systems are the combinations of both software and hardware required to execute the software. Software components are mapped onto execution platform components *i.e.* a thread is mapped to a processor while a data component can be mapped to a memory component. Multiple implementation classifiers can be associated to one type classifier of a specific component.

AADL provides support for both static and dynamic architectural modeling. A static architecture contains hierarchical composition of interconnected subcomponents for each containing component. These interconnected subcomponents form the internal structure of the containing component. Reconfigurable structure specification of AADL facilitates dynamic architectural modeling. Dynamic architectures are realized through modal behavior of the system. Modes contain component and connection configuration for different operational as well as error modes.

In order to support extensive and focused model analysis, AADL core language support extensions through annexes and properties.

3.1.2. Data Modeling

Data modeling in AADL is accomplished through data components. Domain-specific data types are represented using the type classifier of a data component and further be detailed in the implementation classifiers. The declared data types can then be used with ports to represent the kind of data to be transmitted and data sub-components are declared to represent the data items.

AADL provides *Data Modeling Annex* with a property set and pre-defined *Base_Types* to express data modeling in an architectural setting [14]. This annex defines more than fifteen different properties that can be used with user-defined data types (specified in terms of data components) to detail domain data element types. For data representation, the current version of the AADL to DEVS Code Generation Engine (ADCoDE) under the AADL-DEVS framework not only provides transformation support for primitive data types defined in the Base_Types but also incorporates three commonly used compound data types. Primitive data types include *real*, *integer*, *string*, and *Boolean*. The Data Model Annex also supports *Struct* for representing compound data types. A *Struct* has multiple elements, each of which can be either of primitive or compound data types.

The primitive data types defined in AADL map directly to their counterparts in the Java programming language. For the compound data type, however, they must be defined as objects that can be operated on. This is because unlike, AADL *Struct*, simulatable model components communicate with each other using objects. An example of an input called `job0` is shown in Figure 1. For any AADL compound data type that is to be either input or output, a class with data and function has to be constructed. Moreover, for any of the AADL primitive data types that is input and output, classes are also needed. For the AADL-DEVS framework, the existing classes for real, integer, and string data types are provided in the DEVS-Suite simulator. The *Integer_Range* (used to represent a finite range of integer values) and *Real_Range* (used to represent a finite range of real values) are developed given their common utility for Safety-Critical Systems. In the DEVS-Suite simulator, all input and output (primitive or compound) are referred to as entities. For compound data type (e.g., the Isolette data types [24]), their counterparts must be developed (see Section 6.1.). Below is an example of the data component `Speed` representing the speed of a train using the Data Modeling Annex.

```
data Speed
  properties
    Data_Model::Base_Type => (classifier(Base_Types::Float));
    Data_Model::Real_Range => 200.0 .. 250.0;
    Data_Model::Measurement_Unit => "km/h";
end Speed;
```

Here `Speed` data type is of `Float` with possible values between `200.0` and `250.0` (a Real_Range) and measuring unit `km/h` for kilometer per hour. Further details on ADCoDE are presented in Section 3.3.

3.2. Behavior Modeling using DEVS Annex

This section presents the constructs of the DEVS Annex (DA), an extension to AADL for discrete-event modeling [2]. Each DA section is described in detail with its syntax, and grammar with appropriate examples. The annex subclause grammar and semantics is based on the Discrete Event System Specification (DEVS) formalism [37]. This annex brings with it the ability to model detailed behaviors at component level based the concept of the *Atomic DEVS* and at system level based on the concept of the *Coupled DEVS*, where a system is a hierarchical composition of modular components. In use, the DA subclauses can annotate any of the software and execution platform components to model the discrete behavior. The DA is expressive enough to model complex monitored and controlled variables to specify data types for ports and data items communicated through these ports.

The DEVS Annex is implemented as plug-in to an Open Source Architecture Tool Environment (OSATE) [24]. Along with full-features text editor, the OSATE can be extended with analysis plug-ins.

The rest of this section contains the Extended Backus-Naur Form (EBNF) of the DA grammar, in which: literals are printed in **bold**; alternatives are separated by a pipe |; groupings are enclosed with parentheses ( ); square braces [ ] delimit optional elements; and the closures { }+ and { }* are used to signify one-or-more, and zero-or-more of the enclosed element, respectively. Following is the grammar of the DA subclause:

```
devs_annex ::=
    [ variables  { variable_declaration }+ ]
    [ states    state_declaration ]
    [ behavior  atomic_behavior_declaration ]
```

Here, the `variables`, `states`, and `behavior` are the sections of a HA subclause, each of which is dedicated to specify particular aspect of a detailed behavior model.

The DA subclauses are described in the implementation classifiers

```
annex devs
    {**
    ...
    -- DEVS Annex sections
    ...
    **};
```

where `annex` is an AADL keyword and `devs` is the identifier for the DA subclause. Behavior composed of different `devs` sections is specified between `{**` and `**}`.

3.2.1. Variables Section

The local variables in the scope of a DA subclause are declared in the **`variables`** section along with their data types. Data types are assigned to variables by classifier references to the appropriate AADL (user definable) data components. Each variable must have an initial value specified after **`=>`**. Depending on the data type, an initial value can either be simple (*i.e.,* integer, real, boolean, or string literal) or compound consisting of more than one simple values separated by commas and enclosed in parenthesis. Following is the grammar of the **`variables`** section:

```
variable_declaration ::=
   variable_identifier :
   (variable_type_identifier | data_component_classifier_reference) =>
   value_declaration ;

value_declaration ::=
   simple_value | compound_value

simple_value ::=
   INTEGER_LIT | REAL_LIT | (true | false) | STRING

compound_value ::=
   { ( simple_value , simple_value ) }+
```

The referred external data component must either be part of the package containing the component being annotated, or must be declared within the scope of another package that has been imported using the AADL **`with`** clause. Following example shows the use of the **`variables`** section to declare different types of variables.

```
1  ...
2  annex devs {**
3    ...
4    variables
5      speed : Base_Types::Float => 75.0;
6      counter : Base_Type::Integer => 3 ;
7      ct : Iso_Types::Temperature => (98, "Valid");
8    ...
9  **};
```

Variables `speed` on line 5 with initial value `75.0`, and `counter` on line 6 with initial value `3` are of type `Float` and `Integer`, respectively, and are defined in a predefined AADL package `Base_Types`. Data component `ct` on line 7 is of type `Temperature` defined as structure with two elements in another AADL package `Iso_Types`. Sub-statement `(98, "Valid")` specifies initial values for elements of variable `ct`; first element is of type integer initialized with `98` and second element is of type string initialized with `"Valid"`.

Variables `speed` and `counter` are examples of variables with primitive values while `ct` is an example of a variable with compound value. Variables can be input or output in which case they must have their counterparts included in the DEVS-Suite simulator or constructed by the modeler (see Section 3.1.2.).

3.2.2. States Section

All the admissible states of a particular component are defined in the **`states`** section. Each state specification has a name followed by the time advance function stating the time to remain in a particular state before next transition. States with 0 time advance function allow instantaneous transitions to the new state while states with **`INFINITY`** time advance function model the final state. DA only allows one starting state labeled as **`initial`** while the unlabeled states are considered as transient states. A complete behavior starts from an initial state, suspends in transient states (with different time advance functions) until next internal or external transition while performing actions upon each transition, and ends in the final state.

The grammar of the **`states`** section is as follows:

```
state_declaration ::=
   state_identifier :  [ initial ]
   ( REAL_LIT | INFINITY | variable_identifier ) ;
```

Following example shows the use of `states` section to declare different types of constants.

```
 1  ...
 2  annex devs {**
 3    ...
 4    states
 5      Start: initial 0.0;
 6      Chk_Busy: 3.0;
 7      Busy: period;
 8      Passive: INFINITY;
 9    ...
10  **};
```

Starting state `Start` on line 5 is declared as `initial` and with time advance function `0.0`. State `Chk_Busy` on line 6 is a transient state with time advance function `3.0` indicates that the system will suspend in this state for `3.0` time units. The system can stay in `Busy` on line 7 state for `period` time units. Time variable `period` is declared in the `variable` section not shown here. State `Passive` on line 8 with time advance function `INFINITY` specifies a final state.

3.2.3. Behavior Section

The `behavior` section of the DA subclause is used to specify the discrete behavior of annotated AADL component in terms of a state-transition system with three functions and one declaration. The functions `deltext` and `deltint` are used to specify external and internal transitions, respectively. The function `outfn` specifies the system's output based on the change in total state, either due to external or internal state transition. The test inputs for stand-alone testing are specified using the `intest` declaration. The time advance function that can specify the time periods for state transitions (i.e., `deltext` and `deltint` functions) is excluded. This decision is made for simplicity and time period used in the state transition functions are generally as given as values, not computed as functions. All the identifiers used in this must refer to the declarations defined in their respective sections.

The grammar of the `behavior` section is as follows.

```
atomic_behavior_declaration ::=
   deltext external_transition_declaration |
   deltint internal_transition_declaration |
   outfn outfn_declaration |
   intest intest_declaration
```

Below, we explain each of these functions which may constitute a component's behavior individually or in combination with other functions.[1]

3.2.4. External Transition Function

The function `deltext` is used to model the external transition caused by an input message from a source state. This external message interrupts system's behavior moving it to the destination state. The system's response to external messages is specified as *behavior action* and depends on the current state, specific input, and the time elapsed in current state.

An external message in DA sublanguage is composed of the respective port name, the value received, and a variable to store this value for future local use. It is important to note that type of the variable must match with data classifier specified along the port in the type classifier.

[1]A complete DEVS Annex syntax card with detailed grammar productions are described in Appendix A1.

The behavior action in DA sublanguage specifies the activities to be performed upon completion of the external transition. Current version of the sublanguage uses a string to model these actions and should contain syntactically correct Java language statements. Extension of the sublanguage to allow common Java constructs is an important future work in this direction.

Below is the grammar of the `deltext`.

```
external_transition_declaration ::=
   deltext [ source_state_identifier , message ]->
   destination_state_identifier behavior_action

message::=
   port_identifier (? | !)  variable_identifier ;

behavior_action ::=
   { STRING }
```

Following example shows the use of `deltext` function to specify external transitions.

```
1  ...
2  annex devs {**
3    ...
4      behavior
5        deltext [Active, get?iVar]-> Passive {};
6        deltext [Speed, spin?pVar]-> Busy {
7           " if(pVar > 0 && position < EoA)
8               status = "Valid";
9             else
10              status = "Invalid"; "
11          };
12   ...
13 **};
```

**** CHECK PAGE Numbering

The function `deltext` defined on line 5 specifies an external transition with `Active` as source state. On receiving an event on `get` in port, the system behaves as destination state `Passive`. No behavior action is defined along with this external transition as indicated by empty braces. External transition function defined on line 6 specifies that while in the `Speed` state, if a message is received on `spin` in port the received value is stored in variable `pVar`. The behavior action is composed of an if-else statement and `Busy` is the destination. Variables `iVar`, `pVar`, `position`, `EoA`, and `status` are defined in the `variables` section (not shown here) while in ports `get`, and `spin` are defined in type classifier (not shown here) of the respective component.

3.2.5. Internal Transition Function

Function `deltint` is used to model the internal transition caused by progression of the elapsed time (e) to time to make transition to the next state (specified as time advance function) ta, when $e = ta$. Target state identifier follows the source state identifier and is followed by the behavior action specification. As mentioned earlier, current version of the sublanguage uses a string to model these actions and should contain syntactically correct Java language statements.

Below is the grammar of the `deltint` function.

```
internal_transition_declaration ::=
   deltint [ source_state_identifier ]->
   destination_state_identifier behavior_action ;
```

Following example shows the use of `deltint` function to specify internal transitions.

```
1 ...
```

```
2  annex devs {**
3    ...
4     states
5      Ready: initial 0.0;
6      Busy: period;
7      Passive: INFINITY;
8
9     behavior
10     deltint [Ready]-> Busy {"count=count+1;"};
11     deltint [Busy]->  Passive {};
12   ...
13 **};
```

Function `deltint` defined on line 10 specifies an instantaneous (as *ta=0.0* on line 5) internal transition from source state `Ready` to the destination state `Busy` state. Behavior action for this transition specifies increment to a *count* variable declared in the `variables` section (not shown here). Function `deltint` defined on line 11 specifies an internal transition with `Busy` source state and `Passive` as destination state. This transition will occur as soon as the elapsed time *e* equals to `period` (*ta* for `Busy` state). Variable `period` is defined in the `variables` section which is not shown here. State `Passive` on line 7 *ta* equals to `INFINITY` is the end state.

3.2.6. Output Function

The ounction `outfn` is used to model observable outputs. Output messages are only generated on internal transitions before the state change. If an output is required to be generated on an external transitions, the control must be moved to a state with *ta = 0.0* for output triggering followed by an instantaneous internal transition to the actual destination state. An output message contains name of the out port followed by `!` sign and the data to be transmitted. Type of the data must comply with the type of the port defined in the type classifier. An `outfn` specification may also contain conditional expression to further restrict output generation. A conditional expression is made up of boolean terms combined using relational operators while the numeric expressions are defined to specify arithmetic operations (+, -, *, /, *mod*) and the power operation (∧).

Below is the grammar of the `outfn` function.

```
outfn_declaration ::=
   outfn [ source_state_identifier [ , conditional_expression ] ]-> message
   behavior_action ;

conditional_expression ::=
   boolean_term [ and boolean_term [ and boolean_term ] |
   or boolean_term [ or boolean_term ] ]

boolean_term::=
   [ not ] [ variable_identifier |   [ boolean_expression ] | relation ]

relation::=
   ( numeric_expression ( ==| !=| >| <| >=| <= ) numeric_expression )
```

Following example shows the use of `outfn` function to specify output generation.

```
1 ...
2 annex devs {**
3   ...
4    behavior
5     outfn [Running,(isValid=true)]-> op!speed {};
6     outfn [Stop]-> op!0.0 {};
7   ...
8 **};
```

Function `outfn` defined on line 5 specifies an output function generated from from state `Running`. The value of a `speed` variable is transmitted through output port `op` whenever the *e=ta* for state `Running` and boolean variable `isValid` is `true`. Function `outfn` on line 6 models the output generation from state `Stop`. Real value `0.0` is transmitted through output port `op` whenever the *e=ta* for state `Stop` without any further condition.

3.2.7. Test Input Declaration

Declaration **intest** is used to specify the input to be used for stand alone testing during simulation [2].

It is not part of the DEVS formalism rather used by the DEVS-Suite (the target simulator) for injecting test input for model components. This input declaration specification contains input port name followed by the value. Simple values are specified directly and compound values (the one with multiple data items) are parenthesized while separating data items with commas , and organizing them in order.

```
intest_declaration ::=
  intest [ port_identifier , value_declaration ] ;
```

Following example shows the use of **intest** declaration to specify dynamic input usage.

```
1  ...
2  annex devs {**
3    ...
4      behavior
5        intest [status, true];
6        intest [temperature,(98.0, "Valid")];
7
8    ...
9  **};
```

The declaration **intest** on line 5 specifies an input with input port `status` and the value **true**. The test input declaration **intest** defined on line 6 specifies an input for dynamic application on input port `temperature`. The input is expressive enough to have compound inputs. The value to be applied has two data items; the first is of type real with value `98.0` and the second is of type string with value `Valid`.

3.3. Code Generation for Simulation

Code generation for DEVS simulation is the third phase of the proposed DEVS-AADL model. As DEVS-Suite, being one of the most commonly used DEVS modeling and simulation toolset, has been selected for simulation, this phase is focused on transformation from AADL models with DA specification into DEVS-Suite specifications.

DA is implemented as domain specific language (DSL) for behavior specification that is later to be simulated using DEVS-Suite simulator. Although structure of the DA sublanguage is based on DEVS theory but DA specifications are still not executable under DEVS-Suite. For efficient and valid simulation, it is required to generate appropriate DEVS-Suite code, as Java classes, from DA specifications in each section. To ease complex behavior modeling DA seamlessly integrates with AADL core language and does not require re-definition of the interface elements defined in the type classifier of the respective component. Reliance on AADL core language for structural and data modeling further intensifies the challenge of DEVS-Suite code generation from the DA specifications. To cope with this challenge, we have implemented ADCoDE—an AADL to DEVS CoDe generation Engine. ADCoDE is implemented as a plug-in to OSATE and can be activated for an implementation classifier (annotated with DA specifications) of a particular component selected in Eclipse Outline view. An Xtend file *DevsGenerator.xtend* contains the main class with methods responsible for code generation by traversing the type and implementation classifier of the selected component.

As depicted in Figure 4, upon activation for a particular AADL component with detailed data, structure, and behavior modeling using DA sublanguage, code generation through ADCoDE is a three step process. Firstly, in the *data classes generation* step, appropriate Java classes are generated for primitive and compound data types. Secondly, in the *structural code generation* step, structural and interface code for input and output ports is generated based on the type classifier of the AADL component. Thirdly, in the *behavioral code generation*, behavioral code is generated based on the DA specifications in the particular component. All the generated code segments are then structured to form a ViewableAtomic class ready to be simulated in the DEVS-Suite simulator. Code generation for one AADL component results in one ViewableAtomic class. All the ViewableAtomic classes generated for several components in an AADL package are organized in the *Model* folder while the generated data classes are organized

[2]The keyword `infn`, as used in an earlier version of the DEVS Annex introduced in [2], is now replaced with the **intest** to improve readability and understanding

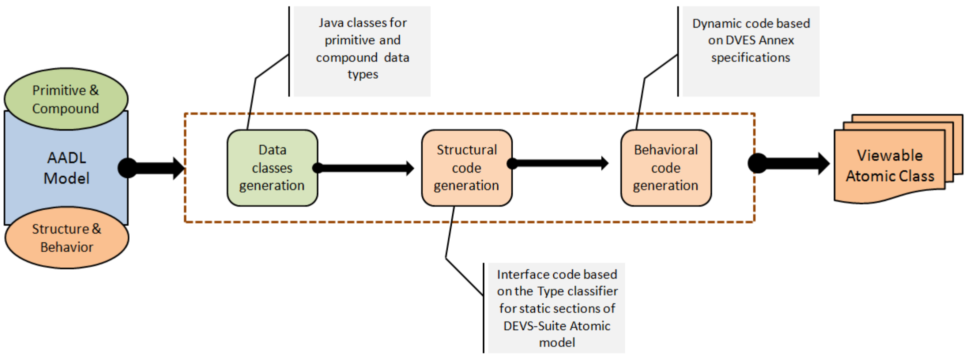
Primitive & Compound
AADL Model
Structure & Behavior
Java classes for primitive and compound data types
Data classes generation
Structural code generation
Interface code based on the Type classifier for static sections of DEVS-Suite Atomic model
Behavioral code generation
Dynamic code based on DVES Annex specifications
Viewable Atomic Class

Fig. 4. AADL to DEVS Code Generation Engine (ADCoDE) Workflow

in a separate folder named same as the AADL package in which the data components are defined. Contents of all these folders are dynamically updated on any new data or ViewableAtomic class generation. This hierarchical organization of the generated code not only improves designer's understanding at the modeling level but also ease the selection and monitoring of atomic and coupled models during simulation.

ADCoDE also realizes code generation for couple DEVS models. Upon activation for a composite AADL component (for example a thread group), a model class extending the ViewableDigraph is generated along with required data and model classes for each AADL sub-component(thread components). Below we explain three step code-generation process of the ADCoDE with appropriate examples while code generation for couple DEVS models in further detailed in Section 3.3.4..

3.3.1. Data Classes Generation

As explained in Section 3.1.2., AADL allows both primitive and compound data types for data modeling, hence the ADCoDE must also have the capability to map them with respective DEVS-Suite data types. As primitive data types, pre-defined in the Data Modeling Annex and Base_Types, directly map to the basic data types in DEVS-Suite, the core of the DEVS-Suite also needs to be extended. Such an extension must be realized via inheriting from the *Entity* class as DEVS-Suite only allows transmitting objects (messages sent and received) either Entity or any of its subclasses.

As noted in Section 2.2., AADL to DEVS transformation requires the AADL data types representing input and output to be mapped to entity data types. The input and output entities are the events sent and received among atomic and coupled models.

Figure 5 shows the extension of the Entity class for AADL primitive data types. These classes include the primitive **stringEnt**, **booleanEnt**, **intEnt**, and **doubleEnt** data types for their AADL *String*, *Boolean*, *Integer*, *Float* counterparts. Every class has a private variable `v` of the respective primitive type. Each class further includes `getv()`, and `setv()` as getter and setter for the private variable `v`, respectively, and method `print()` to print and method `copy()` to copy current value of variable `v`. Methods `equal()` and `equals()` are to compare an entity and an object of the respective type while method `getName()` outputs the name of the object.

Below is an example of the primitive data type Integer declared in the Base_Types (as on line 3) and the corresponding DEVS-Suite code generated by the ADCoDE for a variable `pd` declared in the **variables** section of a DA specification (as on line 11). It is important to note that no new Java class is generated by the ADCoDE for primitive data types while the corresponding objects of respective subclasses are generated (as on line 15).

```
1  ...
2  -- Integer data type declaration in Base_Types
3  data Integer
4    properties
5      Data_Model::Data_Representation => Integer;
6  end Integer;
7
8  ...
9  -- Variable declaration in DA specification with Integer data type
```

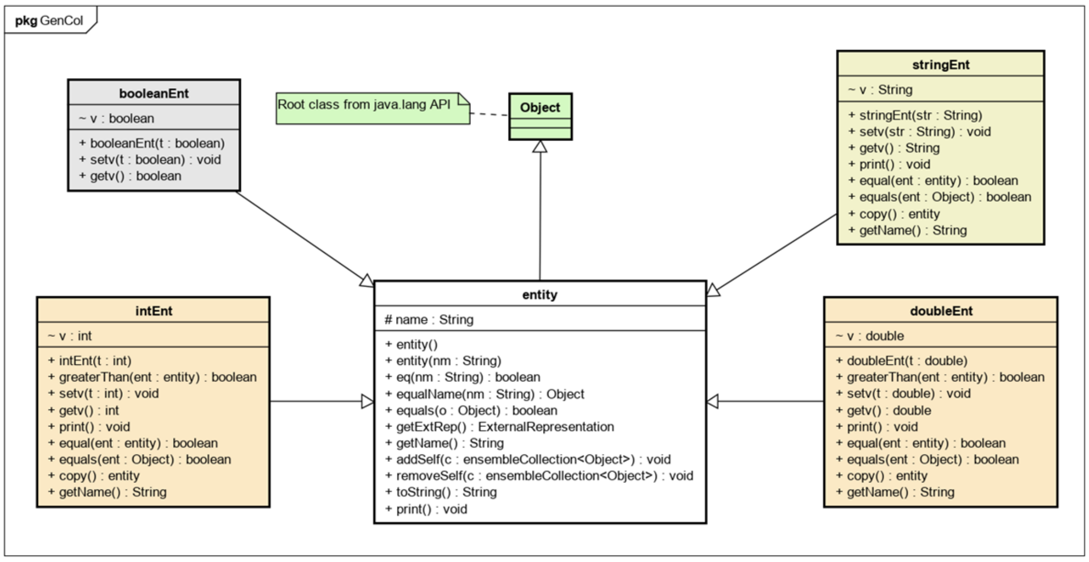
pkg GenCol
booleanEnt
~ v : boolean
+ booleanEnt(t : boolean)
+ setv(t : boolean) : void
+ getv() : boolean
Root class from java.lang API
Object
stringEnt
~ v : String
+ stringEnt(str : String)
+ setv(str : String) : void
+ getv() : String
+ print() : void
+ equal(ent : entity) : boolean
+ equals(ent : Object) : boolean
+ copy() : entity
+ getName() : String
intEnt
~ v : int
+ intEnt(t : int)
+ greaterThan(ent : entity) : boolean
+ setv(t : int) : void
+ getv() : int
+ print() : void
+ equal(ent : entity) : boolean
+ equals(ent : Object) : boolean
+ copy() : entity
+ getName() : String
entity
# name : String
+ entity()
+ entity(nm : String)
+ eq(nm : String) : boolean
+ equalName(nm : String) : Object
+ equals(o : Object) : boolean
+ getExtRep() : ExternalRepresentation
+ getName() : String
+ addSelf(c : ensembleCollection<Object>) : void
+ removeSelf(c : ensembleCollection<Object>) : void
+ toString() : String
+ print() : void
doubleEnt
~ v : double
+ doubleEnt(t : double)
+ greaterThan(ent : entity) : boolean
+ setv(t : double) : void
+ getv() : double
+ print() : void
+ equal(ent : entity) : boolean
+ equals(ent : Object) : boolean
+ copy() : entity
+ getName() : String

Fig. 5. Class diagram for primitive data types

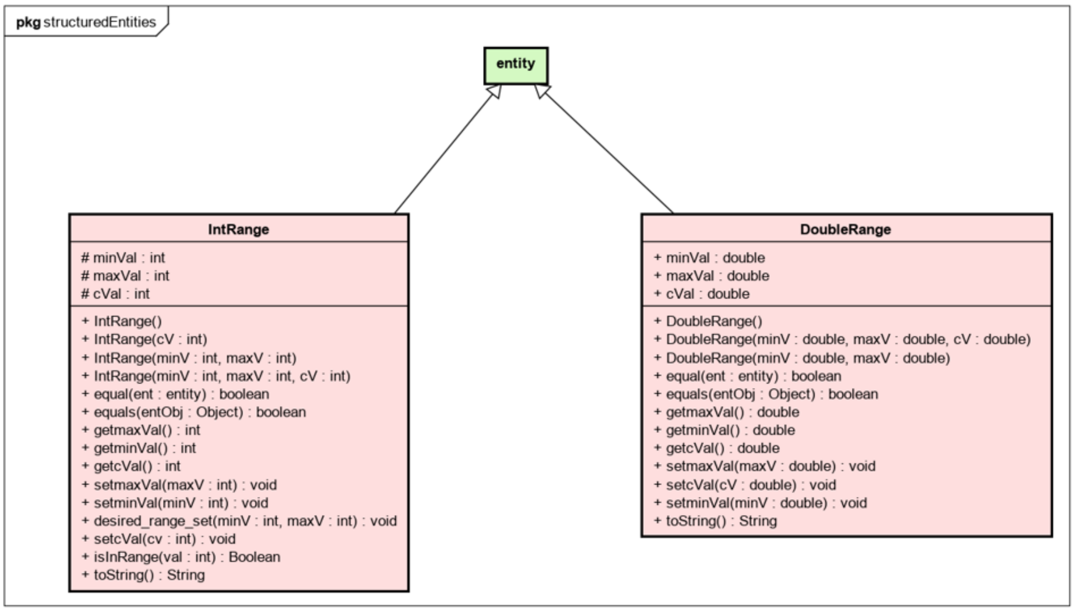
pkg structuredEntities
entity
IntRange
# minVal : int
# maxVal : int
# cVal : int
+ IntRange()
+ IntRange(cV : int)
+ IntRange(minV : int, maxV : int)
+ IntRange(minV : int, maxV : int, cV : int)
+ equal(ent : entity) : boolean
+ equals(entObj : Object) : boolean
+ getmaxVal() : int
+ getminVal() : int
+ getcVal() : int
+ setmaxVal(maxV : int) : void
+ setminVal(minV : int) : void
+ desired_range_set(minV : int, maxV : int) : void
+ setcVal(cv : int) : void
+ isInRange(val : int) : Boolean
+ toString() : String
DoubleRange
+ minVal : double
+ maxVal : double
+ cVal : double
+ DoubleRange()
+ DoubleRange(minV : double, maxV : double, cV : double)
+ DoubleRange(minV : double, maxV : double)
+ equal(ent : entity) : boolean
+ equals(entObj : Object) : boolean
+ getmaxVal() : double
+ getminVal() : double
+ getcVal() : double
+ setmaxVal(maxV : double) : void
+ setcVal(cV : double) : void
+ setminVal(minV : double) : void
+ toString() : String

Fig. 6. Class diagram for compound data types

```
10  variables
11     pd :  Base_Types::Integer =>  100;
12
13  ...
14  -- Code generated by ADCoDE for ViewableAtomic class
15    private intEnt pd;
16
17  ...
```

The core of the DEVS-Suite simulator has also been extended with new classes to incorporate with complex data types for data modeling. Figure 6 depicts the class diagram showing the Entity class extension for complex data types. The classes **IntRange** and **DoubleRange** are defined for the *Integer_Range*, and *Real_Range* datatypes, respectively. Each of these classes has three private variables; *minVal*, *maxVal*, and *cVal* representing minimum, maximum, and current values, respectively. For initialization, objects of these classes along with the values for the variables are automatically extracted from their respective models during code generation. Aside from a default constructor, there are constructors that can have `minVal`, `maxVal`, and `cVal` as input arguments. Each class includes getters and setters such as `getminVal()` for retrieving the `minVal` variable. The method `toString()` returns string representation of the object. The methods `equal()` and `equals()` are to compare an entity with its corresponding object types. For each complex data type used in an AADL component's type or implementation classifier, a separate Java class is generated.

Below is an example of a compound data type defined by an AADL data component `display_temperature` on line 13. As specified on line 6, it is of type *Integer_Range* with minimum value `68` and maximum value `105`. Variable `dt`, specified on line 13, is of type `display_temperature` with initial value `90` and is declared in the **variables** section in a particular DA specification. Line 17 shows the corresponding code generated by ADCoDE with `dt` as an object of the **IntRange** class with minimum value `68`, and maximum value `105` (extracted from declaration on line 6), and current value `90` (extracted from the variable declaration on line 13).

```
1   ...
2   -- Complex data type declaration in package Temp_Types
3   data display_temperature
4     properties
5       Data_Model::Base_Type => (classifier(Base_Types::Integer));
6       Data_Model::Integer_Range => 68 .. 105;
7       Data_Model::Measurement_Unit => "Fahrenheit";
8   end display_temperature;
9
10  ...
11  -- Data type used with a variable in DA specification
12  variables
13     dt :  Temp_Types::display_temperature =>  90;
14
15  ...
16  -- Code generated by ADCoDE for ViewableAtomic class
17    private IntRange dt = new IntRange(68, 105, 90);
18
19  ...
```

It is important to note that no new Java class is generated for the user-defined data types based on primitive and complex data types unless it consists of more than one element e.g., a structure data type.

Below is an example of the structure data type with two elements `s` of `measured_speed_range` and `status` is of `valid_flag` as shown on lines 7, 8, and 9. Data type `measured_speed_range` is a real range (complex data type) with values between `68.0` to `105.0` as shown on line 15 and measure unit kilo meter per hour specified as `km/h`. Data type `valid_flag` is an enumeration with values `Invalid` and `Valid` as shown on line 21. As simulation is only concerned about the quantities so no code is generated for measuring unit.

All three data components to model respective data types are specified in package `Train_Types`;

```
1
2   ...
3   -- Structure data type with two elements declared in package Train_Types
4   data current_speed
5       properties
6        Data_Model::Data_Representation => Struct;
```

```
     Data_Model::Elements_Names => ("s", "status");
     Data_Model::Base_Type => (classifier (Train_Types::measured_speed_range),
                     classifier (Train_Types::valid_flag));
end current_speed;

-- First element measured_speed_range declared in package Train_Types
data measured_speed_range
    properties
     Data_Model::Base_Type => (classifier(Base_Types::Float));
     Data_Model::Real_Range => 68.0 .. 105.0;
     Data_Model::Measurement_Unit => "km/h";
end measured_speed_range;

-- Second element valid_flag declared in package Train_Types
data valid_flag
    properties
     Data_Model::Data_Representation => Enum;
     Data_Model::Enumerators => ("Invalid","Valid");
end valid_flag;

...
```

ADCoDE generates a new Java class by extending the `entity` class. This class imports package `structureEntities` as the class `DoubleRange` used with private variable `s` (on line 8) is defined in it. Values `65.0` and `105.0` are extracted from the definition as minimum and maximum range values. Private variable `Status` (on line 9) is declared as String as AADL enumerations are mapped as Java String for code generation. A default constructor is defined on line 11 while lines 15, 19, 25, and 32 show the rest of the constructors with their respective parameters. Setters for the private variables are shown on lines 37 and 51 while the getters are shown on lines 43 and 55. Method `isInRange`, as shown on line 47, returns `true` if the current speed `s` is in range and `false` otherwise. It uses methods `getminVal` and `getmaxVal` of the parent `DoubleRange` class of the purpose.

```
1  -- New class current_speed generated by ADCoDE
2  package Train_Types;
3
4  import structuredEntities.*;
5
6  public class current_speed extends entity {
7
8    private DoubleRange s = new DoubleRange(68.0, 105.0);
9    private String status;
10
11   public current_speed() {
12
13   }
14
15   public current_speed(double cv) {
16     s.setcVal(cv);
17   }
18
19   public current_speed(double lv, double uv, double cv) {
20     s.setminVal(lv);
21     s.setmaxVal(uv);
22     s.setcVal(cv);
23   }
24
25   public current_speed(double lv, double uv, double cv, String status ) {
26     s.setminVal(lv);
27     s.setmaxVal(uv);
28     s.setcVal(cv);
29     this.status = status;
30   }
31
32   public current_speed(double cv, String status ) {
33     s.setcVal(cv);
34     this.status = status;
35   }
36
37   public void set_s(double lv, double uv, double cv) {
38     s.setminVal(lv);
39     s.setmaxVal(uv);
40     s.setcVal(cv);
41   }
42
43   public DoubleRange get_s() {
```

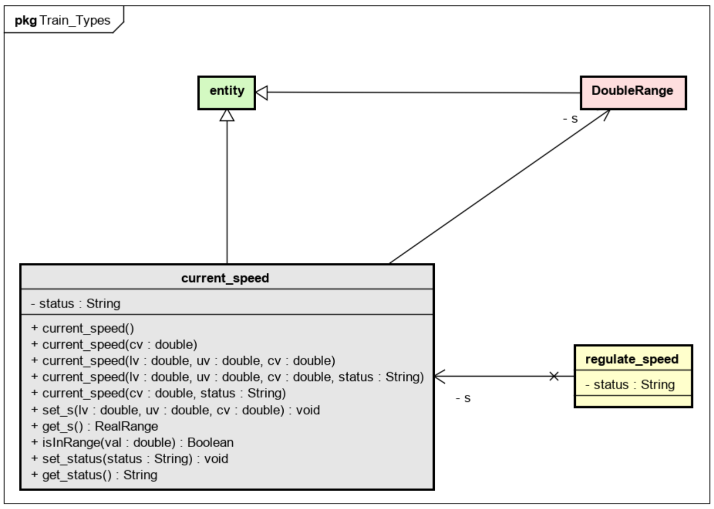
pkg Train_Types
entity
DoubleRange
- s
current_speed
- status : String
+ current_speed()
+ current_speed(cv : double)
+ current_speed(lv : double, uv : double, cv : double)
+ current_speed(lv : double, uv : double, cv : double, status : String)
+ current_speed(cv : double, status : String)
+ set_s(lv : double, uv : double, cv : double) : void
+ get_s() : RealRange
+ isInRange(val : double) : Boolean
+ set_status(status : String) : void
+ get_status() : String
- s
regulate_speed
- status : String

Fig. 7. Newly generated class *current_speed* being used within *regulate_speed* class

```
    return s;
  }

  public Boolean isInRange(double val) {
    return (val >= s.getminVal() && val <= s.getmaxVal());
  }

  public void set_status(String status) {
    this.status = status;
  }

  public String get_status() {
    return this.status;
  }
}
```

Below, variable `cs` with initial values `100.0` for the first element and `"Valid"` for the second element is declared in the **`variables`** section of a DA specification (as on line 7). It is worth mentioning here that only value for the current variable is required for initialization of an integer or real range in a DA specification while the values for the minimum and maximum range variable are automatically extracted from the data type definition.

```
1  -- Data type used with a variable in DA specification
2  thread implementation regulate_speed.impl
3  ...
4  annex devs{**
5   ...
6    variables
7     cs: Train_Types::current_speed => (100.0, "Valid");
8
9    ...
10  **}
11 end regulate_speed.impl
```

This class can be instantiated and used in other classes. Figure 7 depicts one of such examples in which the class is instantiated as `cs` in the `regulate_speed` class with initial values `100` and `"Valid"`. Class `regulate_speed` is generated for the thread component as defined above.

3.3.2. Structural Code Generation

Shown in Figure 4, *structural code generation* is the second phase of code generation through ADCoDE. As explained in Section 3.1.1. structure of an AADL component is composed of its type and implementation classifier. Type classifier contains the interfaces of a component while the implementation classifier models the internal realization. When the ADCoDE is activated for a particular component, firstly a ViewableAtomic class is generated to form the structure of the component in DEVS-Suite domain. Secondly, the interface ports specified in the `features` section of the type classifier are extracted and their respective *in* and *out* ports are added to the ViewableAtomic class. Data types specified with ports in the type classifier and with variables in the DA specification in the implementation classifier are generated as explained in Section 3.3.1. to complete structural code generation.

Below is an example of the structural code generation for an AADL component mapping to an atomic component in DEVS-Suite. Thread `manage_speed` declared on line 4 has two interface ports; `speed` in data port (on line 6), and `status` out data port (on line 7). As shown on line 15, a subclass of the ViewableAtomic class is generated to model the structure of an AADL component (thread `manage_thread` in this case). The code generated for in and out data ports are specified on line 18 and 19.

```
 1
 2  ...
 3  -- Interface declaration in an AADL thread component
 4    thread manage_speed
 5    features
 6       speed : in data port Train_Types::current_speed;
 7       status : out data port Base_Types::Boolean;
 8  ...
 9
10  end manage_speed;
11  ...
12
13  -- Code generated by ADCoDE for ViewableAtomic class
14  ...
15  public class manage_speed extends ViewableAtomic {
16      ...
17
18      addInport("speed");
19      addOutport("status");
20
21  ...
22
23  }
```

In AADL, the structure of a composite component is formed through the port definitions in the `features` section of the type classifier, and sub-component definitions in `subcomponents` section and connections among the ports of the sub-components (and the composite component itself) in the `connections` section of the implementation classifier. Sub-components are instances of the pre-defined implementations to exploit design alternatives.

Upon activation for a composite AADL component (for example a thread group), ADCoDE also generates code to realize couple DEVS models. Firstly, exploring through the `subcomponents` section, required data and atomic model classes are generated for each AADL sub-component(for example thread sub-components). Then a model class extending the ViewableDigraph class is generated along with set of sub-components instantiated based on the information from the `subcomponents` section and already generated atomic model classes. The code for the input and output ports of the composite component are generated based on the input and output ports of all the sub-components. The test inputs for stand-alone testing are generated based on the `intest` declarations (further explained in Section 3.3.3.) of the sub-components specified in their DA specifications. The DA relies on the AADL to ensure semantically identical input/output interfaces for atomic and coupled models.

Code generation for the required coupling categories in any composite AADL component is based on its port connection specifications in the `connections`. Th external input coupling (EIC) is realized based on the connections of the composite component's input ports with the input ports of its sub-components. The external output coupling (EOC) is realized based on the connections of the sub-components' output ports to the composite component's output ports, while the internal coupling (IC) is realized based on the sub-components' output ports to the input ports of other sub-components.

Code generation for composite AADL models, in relation with the case study, is further explained in Section 5.3.

3.3.3. Behavioral Code Generation

In the AADL-DEVS framework, introduced in Section 3.2., the behavior for an AADL component is modeled in the DEVS Annex (DA). The sections of a DA subclause for the implementation classifier of a component are used to specify different aspects of a component's behavior. Dynamic code generation is primarily based on the external, internal, output, and test input functions specified in the `behavior` section of a DA subclause. Hence, no explicit code is generated for the `states` section needed for the `outfn` function as well as the time period required for the `deltext` and `deltint` functions. Below we explain code generation for the `variables` section and the functions used in the `behavior` section of a DA subclause.

Variables section: A local variable defined in the `variables` section has a data type and initial value. Data types are the classifier references to the appropriate AADL (user definable) data components. Code generation for a variable in the DA subclause is based on mapping it to an object of the class generated for the data type. Primitive data types *String*, *Boolean*, *Integer*, *Float* are mapped with `stringEnt`, `booleanEnt`, `intEnt`, and `doubleEnt`. Variables with compound data types are mapped to the objects of the newly generated classes based on their data types (as explained in Section 3.3.1.).

Below is an example of code generation for `variables` section. Variable `temp` defined on line 6 is of type `current_temperature` specified in another package `User_Types`. The initial value of the `temp` variable indicates that the associated data type has two elements; a real to be initialized to `96.0` and a string to be initialized to `"Valid"`. The required classes (not specified here) for both of these elements are generated as explained for data classes generation. The variable `counter` defined on line 7 is of primitive type Integer initialized to `10`.

Code generated by the ADCoDE for the variable `temp` is specified on line 14. It is an object of a newly generated class `current_temperature` which itself extends the DoubleRange class defined to extend DEVS-Suite for real ranges. Class `current_temperature` (not shown here) is generated based on the definition of the `current_temperature` data component specified in the package `User_Types`.

Code generated by the ADCoDE for variable `counter` is specified on line 15 as an object of the class `intEnt` with the initial value `10`. No need to generate any new class as the variable is of primitive type.

```
 1 ...
 2  annex devs {**
 3 ...
 4 -- variable declaration in a DA subclause
 5   variables
 6     temp : User_Types::current_temperature => (96.0, "Valid");
 7     counter : Base_Types::Integer => 10;
 8     ...
 9
10  **};
11
12 -- Code generated by ADCoDE for variables
13 ...
14     private current_temperature temp =  new current_temperature(96.0, "Valid");
15     private intEnt counter = new intEnt(10);
16
17 ...
```

Function deltext: Function `deltext` models the external transition and is composed of a port name, the value received, and a variable to store this value. Currently, the behavior action to be performed upon completion of the external transition is modeled as a String. External transitions in DEVS-Suite are realized through a public method `deltext` which accepts an object of the *message* class defined in the DEVS-Suite. As messages are implemented as bags so method *messageOnPort* is used to explore the bag to find a message received on a particular port at a particular time and then the required behavior actions are executed accordingly.

Code generation for `deltext` function specified in a DA subclause is based on combining all its occurrences using a control structure supported by the Java language. For a particular occurrence, the source state is identified and is used in a method *phaseIs* to structure the control. Method *messageOnPort* is then used to explore the message bag on the particular in port specified in the function. The String containing the behavior action is then added as it is. The destination state and its time advance function extracted from the `states` section are used in the *holdIn* function to set the next state and time advance function for this state.

Below is an example of code generation for a `deltext` function. Function defined on line 6 has `Start` as source state `Stop` as destination state. It receives a message on `iPort` and stores it in variable `ct` which is declared in the

variables section (not shown here). Behavior action for this function is specified on line 7. It sets the value of a variable `valid` based on the value of the variable acquired using `ct.getv()` method.

```
1  ...
2   annex devs {**
3  ...
4  -- behavior section in a DA subclause
5    behavior
6      deltext [ Start, iPort?ct ]-> Stop {
7       " if(ct.getv() >= 96 &&  ct.getv() <= 99 )
8           valid = true;
9         else
10          valid = false; "
11     };
12
13     ...
14
15  **};
16
17 -- Code generated by ADCoDE for deltext function
18 ...
19     public void deltext (double e, message x){
20       Continue(e);
21
22       if (phaseIs("Start")) {
23         for(int i=0; i<x.getLength(); i++) {
24            if(messageOnPort(x, "iPort", i)) {
25            ct = (current_temperature) x.getValOnPort ("iPort",i);
26            if(ct.getv() >= 96 &&  ct.getv() <= 99 )
27              valid = true;
28             else
29              valid = false;
30              holdIn("Stop",pd);
31            }
32         }
33       }
34     }
```

Corresponding DEVS-Suite **deltext** method, generated by the ADCoDE, is specified on line 19. The `Continue(e)` statement on line 20 is used to reflect the passage of elapsed time (*e*). Line 22 contains the control structure defined for the function. As there is only one occurrence of the deltext function in DA subclause, so the control structure only contains an if statement with `phaseIs("Start")` marking the source state. For loop specified on line 23 explores the bag of messages and uses `messageOnPort` method, specified on line 24, to confirm if the message is received on `iPort` in port. Code generated for behavior action is specified on line 26 and is same as specified in the model on line 7. Method `holdIn` on line 30 specifies the next system state with destination state `Stop` and its time advance function `pd`. States and variables are defined in their respective sections.

Function deltint: Function **deltint** models the internal transition with source and destination states followed by the behavior action. Internal transition happens when elapsed time (*e*) progresses and reaches to the time advance function (*ta*), when $e = ta$). Internal transitions in DEVS-Suite are realized through a public method **deltint**.

Code generation for a **deltint** function is based on combining all its occurrences using a control structure supported the Java language. For each occurrence, the source state is then identified and used in the method *phaseIs* for controlling the change in the state of the model. The string containing the behavior action is then added as it is. The destination state and its time to next event (i.e., sigma) are extracted from the **states** section and used in the *holdIn* function.

Below is an example of a generated code for the **deltint** section. The function defined on line 6 has `Start` as source state and `Set_Values` as destination state. No other actions are specified for this internal transition function.

```
1  ...
2   annex devs {**
3  ...
4  -- behavior section in a DA subclause
5    behavior
6      deltint  [ Start ]-> Set_Values {} ;
7
8       ...
9
10   **};
11
12  -- Code generated by ADCoDE for deltint function
13  ...
14      public void deltint (){
15        if (phaseIs("Start"))
16            holdIn("Set_Values", p);
17        }
```

The DEVS-Suite `deltint` method, generated by the ADCoDE, is specified on line 14. Line 15 contains the `if` control statement. As there is only one occurrence of the `deltint` function in DA subclause, so the control structure is quite simple with only if statement having `phaseIs("Start")`. The method `holdIn` on line 16 specifies the next system state with destination state `Set_Values` and its time advance function `p`.

Function outfn: The function `outfn` is used to model observable outputs generated prior to the internal transition execution. The output messages are transmitted through named out ports. This output generation can be further restricted through conditional expressions. Output functions in DEVS-Suite are realized through a public method `out` which returns an object of the *message* class. DEVS-Suite only allows objects of the *entity* class or its subclasses to be transmitted, method *makeContent* is used to setup the contents to be communicated through a specific port. [] Code generation for an `outfn` function is based on combining all its occurrences using any control structure supported in the Java language. For a particular occurrence, the source state is identified and used in the method *phaseIs*. The conditional statement, if present, is then used to further constraint the output generation. The method *add* of the *message* class is then used to extend the message bag with this new message set by the method *makeContent*.

Below is an example of code generation for `outfn` section. Function defined on line 6 has `Set_Values` as source state. It models the behavior that if condition expression `valid == true` holds then value of the variable `ct` will be transmitted through `oPort` out port.

```
1   ...
2    annex devs {**
3   ...
4   -- behavior section in a DA subclause
5     behavior
6       outfn [Set_Values,  (valid == true)]-> oPort!ct {};
7
8      ...
9
10   **};
11
12  -- Code generated by ADCoDE for outfn function
13  ...
14      public message out(){
15       message m = new message();
16        if (phaseIs("Set_Values"))
17          if(valid == true)
18            m.add(makeContent("oPort", ct.getv()));
19        return m;
20            }
```

The DEVS-Suite `out` method, generated by the ADCoDE, is specified on line 14. A new object of the *message* is created as specified on line 15. As there is only one occurrence of the `outfn` function in DA subclause, so the control structure only contains an if statement with `phaseIs("Set_Values")` marking the source state. Line 17 contains the conditional expression extracted from the model while the content of the message set for `oPort` with `ct.getv()` (assuming that variable `ct` is of type `intEnt`) are added through `add` method, as specified on line 18.

Declaration intest: Although, not being part of the standard DEVS, declaration `intest` is used by the DEVS-Suite (the target simulator) for dynamic input provision for model components with input port name and the value. Input functions in DEVS-Suite are realized through a public method `addTestInput` which gets name of the port and the value to be to used as input.

Code generation for an `intest` declaration is based on generating an `addTestInput` method with respective in ports and the data value. Simple input values are specified as it is but the complex values are instantiated and the objects are included in the function.

Below is an example of code generation for `intest` section. Declaration defined on line 6 has `iPort` as input port and the value to be injected is `"Valid"`.

```
1  ...
2   annex devs {**
3  ...
4  -- behavior section in a DA subclause
5    behavior
6      intest [iPort, "Valid"];
7
8      ...
9
10  **};
11
12 -- Code generated by ADCoDE for intest function
13 ...
14     addTestInput("iPort", new stringEnt("Valid"));
15
16 ...
```

Method `addTestInput`, generated by the ADCoDE, is specified on line 14. Name of the input port `iPort` is extracted and added. For value, a new object of the `stringEnt` class is created with the given value `"Valid"`. During simulation, this value will now be available for stand alone testing.

3.3.4. Code Generation for Composite Components

In AADL, the structure of a composite model is formed through the port definition in the `features` section of the type classifier, and sub-component definitions in `subcomponents` section and connections among the ports of the sub-components (and the composite component itself) in the `connections` section of the implementation classifier. Sub-components are instances of the pre-defined implementations to exploit design alternatives.

Upon activation for a composite AADL component (for example a thread group), ADCoDE also generates code to realize coupled DEVS models. Firstly, exploring through the `subcomponents` section, required data and atomic model classes are generated for each AADL sub-component(for example thread sub-components). Then a model class extending the ViewableDigraph class is generated along with set of sub-components instantiated based on the information from the `subcomponents` section and already generated atomic model classes. Input and output ports of the composite component are generated based on the input and output ports of all the sub-components, respectively. Test inputs for stand-alone testing are generated based on the `intest` declarations of the sub-components specified in the their DA specifications. DA relies on AADL to ensure semantically identical input/output interfaces for atomic and coupled models.

Code generation to realize required coupling categories is based on the port connection specifications in the `connections` section of composite components. The external input coupling (EIC) is realized based on the connections of the composite component in ports with in ports of some sub-component. The external output coupling (EOC) is realized based on the connections of sub-component output ports to the composite model output ports, while the internal coupling (IC) is realized based on the sub-component output ports to the in ports of other sub-components.

Code generation for composite AADL models, in relation with the case study, is further explained in Section 5.3.

4. A Case Study of Isolette Thermostat System

In order to demonstrate application of the AADL-DEVS framework for complex and highly integrated time-critical and safety-critical systems, this section presents a case study of the *Isolette* system, an infant incubator described in the Requirement Engineering Management Handbook (REMH) published by Federal Aviation Administration (FAA) in 2009 [21]. This specification is simple enough to grasp, yet rich enough to highlight the need for our proposed framework. We use it to show all four phases of the proposed AADL-DEVS framework; *Requirements*

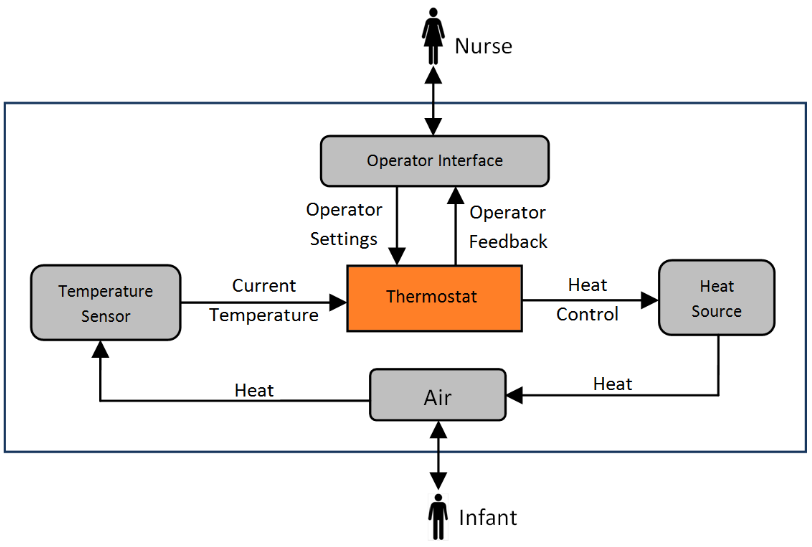

Fig. 8. Isolette Context Diagram with Controller (Thermostat) and Physical Environment (Air)

Specification, *AADL Modeling*, *Code Generation for DEVS Simulation*, and *Simulation using DEVS-Suite Simulator*. The Isolette example has previously been used to introduce important research efforts, and to advocate for AADL-based development as well as new annexes. We have previously used it to introduce Hybrid Annex for continuous behavior and cyber-physical interaction modeling [3]. Blouin has used it to illustrate the Requirements Annex [5], and Larson to explain detailed behavior modeling with the BLESS Annex, and to demonstrate hazard analysis techniques using the Error Model Annex(v.2) [19].

Figure 8 depicts operational context of the Isolette system to maintain temperature of the *Air* within the desired range set by the Nurse using *Operator Interface*. The *Thermostat* monitors the *Air* temperature through the *Temperature Sensor*, and attempts to manipulate it with the *Heat Source* actuator. The control strategy followed by the Thermostat, is derived from a process model that is implicit in the interpretations it gives to the current Air information coming from the sensor, and the commands it has issued to the actuator. In this study, we focus on modeling and simulation of the behavior of the Thermostat, and its interactions with the Heat Source and Temperature Sensor units. The internals of the Operator Interface and continuous behavior of the Air are not considered.

To keep current temperature within the desired range, Thermostat performs two major functions; *i) Monitor Temperature*, and *ii) Regulator Temperature*. Monitor Temperature function receives Current Temperature from the Temperature Sensor, and Desired Alarm Range from the Nurse through Operator Interface. It then turns on or off the Alarm Control if the Current Temperature ascends or descends beyond the Desired Alarm Range.

The Regulate Temperature function receives current temperature from the Temperature Sensor, and Desired Temperature Range from the Nurse through Operator Interface. It then turns on or off the Heat Source to maintain the Current Temperature within the Desired Temperature Range. The key requirements are to set value for the Heat Control, Regulator Status, and Display Temperature controlled variables. As depicted in Figure 9 Regulate Temperature function consists of following four subfunctions;

- **Manage Regulator Interface:** Obtains the Desired Temperature Range from the Operator Interface and reports back the Regulator Status and the Display Temperature.
- **Manage Regulator Mode:** gets status of Regulator Interface Failure from Manage Regulator Interface to determine Mod of the Regulate Temperature function
- **Manage Heat Source:** turns on and off the Heat Source
- **Detect Regulator Failure:** sets value of the Regulate Temperature Failure variable to reveal the internal failure

What follows are details of the data and functional requirements of the Manage Regulator Interface function as

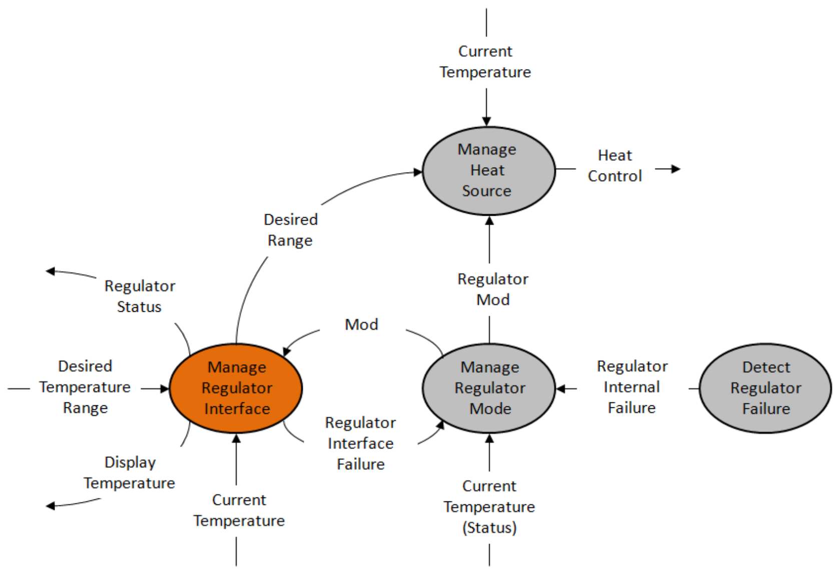

Fig. 9. Regulate Temperature Function

Table 1. Complex data type variables used in the Manage Regulator Interface

Name	Type	Range	Description
Current Temperature	Real	[68.0 .. 105.0]	Current air temperature inside Isolette
	Status	Invalid, Valid	
Regulator Mode	Enum	Init	Initializing following power-up
		NORMAL	Normal mode of operation
		FAILED	Internal failure detected
Regulator Status	Enum	Init	Status of the Thermostat Regulator Function
		On	
		Failed	
Lower Desired Temperature	Integer	[97.. 99]	Lower value of Desired Temperature
	Status	Invalid, Valid	
Upper Desired Temperature	Integer	[98.. 100]	Upper value of Desired Temperature
	Status	Invalid, Valid	
Lower Desired Temp	Integer	[96 .. 1021	Lower value of desired range
Upper Desired Temp	Integer	[97 .. 102]	Upper value of desired range
Display Temperature	Integer	[68 .. 105]	Displayed temperature of Isolette

specified in Section A.5.1.1 of the REMH while architectural modeling & simulation of the same subfunctions is described in Section 5. to demonstrate the practicality of AADL-DEVS framework [3].

4.1. Manage Regulator Interface Function

4.1.1. Data Specification

The definitions for primitive and complex variable types in AADL and Java (and thus DEVS-Suite) share some concepts, but also have differences.

Regulator Interface Failure, indicating operator failure, is of type Boolean and is the only primitive data type variable used in Manage Regulator Interface function.

Table 1 specifies compound data type variables used in Manage Regulator Interface function.

[3]Complete AADL models with detailed behavior specified using DEVS Annex, corresponding java classes automatically generated using ADCoDE tool, and the DEVS-Suite simulation results are available at https://github.com/ehah/AADL-DEVS-Framework

4.1.2. Functional Requirements Specification

The Manage Regulator Interface function requires Desired Range in terms of Lower Desired Temperature and Upper Desired Temperature from Operator Interface and manipulates following controlled and internal variables;

Regulator Status: Controlled variable Regulator Status is dependant on Regulator Mode received from Manage Regulator Mode subfunction and is set based on the following requirements;

REQ-MRI-1: If the Regulator Mode is INIT, the Regulator Status shall be set to Init.

REQ-MRI-2: If the Regulator Mode is NORMAL, the Regulator Status shall be set to On.

REQ-MRI-3: If the Regulator Mode is FAILED, the Regulator Status shall be set to Failed.

Display Temperature: Controlled variable Display Temperature also depends on Regulator Mode. It is the rounded value of the Current Temperature within the accuracy of $0.6°$F based on the following requirements;

REQ-MRI-4: If the Regulator Mode is NORMAL, the Display Temperature shall be set to the value of the Current Temperature rounded to the nearest integer.

REQ-MRI-5: If the Regulator Mode is not NORMAL, the value of the Display Temperature is UNSPECIFIED.

Regulator Interface Failure: Variable Regulator Failure indicates internal error depending on the status of the Lower and Upper Desired Temperature as follows;

REQ-MRI-6: If the Status attribute of the Lower Desired Temperature or the Upper Desired Temperature is Invalid, the Regulator Interface Failure shall be set to True.

REQ-MRI-7: If the Status attribute of the Lower Desired Temperature and the Upper Desired Temperature is Valid, the Regulator Interface Failure shall be set to False.

Desired Range: Depending on the Regulator Interface Failure, internal variable Desired Range is set as follows;

REQ-MRI-8: If the Regulator Interface Failure is False, the Desired Range shall be set to the Desired Temperature Range.

REQ-MRI-9: If the Regulator Interface Failure is True, the Desired Range is UNSPECIFIED.

5. Manage Regulator Interface Modeling & Simulation under the AADL-DEVS Framework

This section is dedicated for modeling and simulation of the Manage Regulator Interface function with the data and requirements specification described in Section 4.1.. An AADL model for the Manage Regulator Interface is used to demonstrate the use of the proposed AADL-DEVS framework where AADL models of a safety-critical system are developed using the DEVS Annex and then transformed to executable code for the DEVS-Suite simulator. The AADL described below uses snippets from the BLESS [18] annex model.

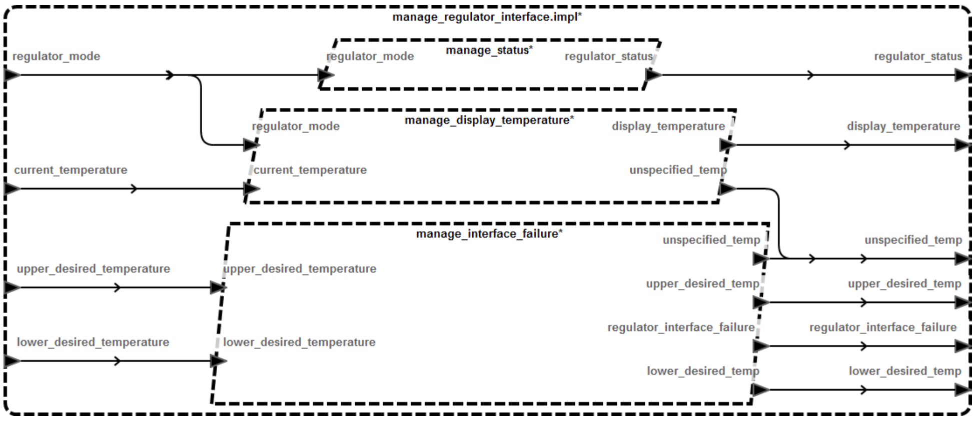

Fig. 10. Manage Regulate Interface AADL model

5.1. Structure and Data Modeling

Figure 10 depicts architectural model of the composite component Manage Regulator Interface function using AADL graphical notations. Based on the functional requirements to be considered to set controlled and internal variables, the architectural model consists of the following three `thread` and one `thread group` components;

5.1.1. Manage Interface Failure & Desired Range

As internal variable Desired Range is directly related to the internal variable Regulator Interface Failure, they are modeled using a single AADL thread `manage_interfaceFailure_desiredRange`. The type classifier of Listing 1 declares the interface of this thread component.

Listing 1: Manage Interface Failure and Desired Range component type

```
thread manage_interfaceFailure_desiredRange
  features
    lower_desired_temperature : in data port Iso_Types::lower_desired_temperature;
    upper_desired_temperature : in data port Iso_Types::upper_desired_temperature;
    regulator_interface_failure : out data port Base_Types::Boolean;
    lower_desired_temp :  out data port Iso_Types::lower_desired_temp;
    upper_desired_temp :  out data port Iso_Types::upper_desired_temp;
    unspecified_temp : out data port  Iso_Types::unspecified_value;

  properties
    Dispatch_Protocol => Periodic;
    Period =>100 ms;
end manage_interfaceFailure_desiredRange;
```

The `lower_desired_temperature` and `upper_desired_temperature` in data ports are used to get the lower and upper values of the desired temperature, respectively. Data types for these in data ports are defined within the scope of another package `Iso_Types` that has been imported using the AADL `with` clause. As specified in Table 1, Lower and Upper Desired Temperature are compound variables with two data items; one of type *Integer* range to represent the current temperature while the other is of type *Enumeration* to represent the status of the respective temperature value. Within the scope of `Iso_Types`, these data types are modeled as follows;

```

--Lower Desired Temperature, "t" is temperature range, "status" is the valid/invalid flag
data lower_desired_temperature
  properties
    Data_Model::Data_Representation => Struct;
    Data_Model::Element_Names => ("t", "status");
    Data_Model::Base_Type => (classifier (Iso_Types::lower_desired_range),
                classifier(Iso_Types::valid_flag));
end lower_desired_temperature;

```

```
11 data lower_desired_range
12   properties
13     Data_Model::Base_Type => (classifier(Base_Types::Integer));
14     Data_Model::Integer_Range => 97 .. 99;
15     Data_Model::Measurement_Unit => "Fahrenheit";
16 end lower_desired_range;
17
18 --Upper Desired Temperature, "t" is temperature range, "status" is the valid/invalid flag
19 data upper_desired_temperature
20   properties
21     Data_Model::Data_Representation => Struct;
22     Data_Model::Element_Names => ("t", "status");
23     Data_Model::Base_Type => (classifier (Iso_Types::upper_desired_range),
24                 classifier(Iso_Types::valid_flag));
25 end upper_desired_temperature;
26
27 data upper_desired_range
28   properties
29     Data_Model::Base_Type => (classifier(Base_Types::Integer));
30     Data_Model::Integer_Range => 98 .. 100;
31     Data_Model::Measurement_Unit => "Fahrenheit";
32 end upper_desired_range;
33
34 data valid_flag
35   properties
36     Data_Model::Data_Representation => Enum;
37     Data_Model::Enumerators => ("Invalid","Valid");
38 end valid_flag;
```

Data component `lower_desired_temperature` on line 3 models the Lower Desired Temperature as a structure with two elements; `t` to represent the temperature value, and `status` to represent the status value. The first element `t` is of type `lower_desired_range` defined on line 11 which is an integer range from `97..99` with measuring unit `Fahrenheit`. The second element `status` is of type `valid_flag` defined on line 34 as `Enum` with values `Invalid` and `Valid`.

Data component `upper_desired_temperature` on line 19 models the Upper Desired Temperature as a structure with two elements; `t` to represent the temperature value, and `status` to represent the status value. The first element `t` is of type `upper_desired_range` defined on line 27 which is an integer range from `98..100` with measuring unit `Fahrenheit`. The second element `status` is of type `valid_flag` defined on line 34 as `Enum` with values `Invalid` and `Valid`. All the properties for both these data types are defined using Data Model annex which is an AADL extension to facilitate detailed data modeling.

Out data port `regulator_interface_failure`, specified in Listing 1, is of type Boolean which is specified in the scope of another package `Base_Types`. It is defined to communicate *true* or *false* based the status of the Upper and Lower Desired Temperature as specified in requirements *REQ-MRI-6* and *REQ-MRI-7*. In the `properties` section, `manage_interfaceFailure_desiredRange` is declared as a periodic thread with period of `100 ms`.

Listing 2: Manage Display Temperature component type

```
thread manage_display_temperature
  features
    regulator_mode: in data port Iso_Types::regulator_mode;
    current_temperature: in data port Iso_Types::current_temperature;
    display_temperature: out data port Iso_Types::display_temperature;
    unspecified_temp : out data port Iso_Types::unspecified_value;

  properties
    Dispatch_protocol => Periodic;
    Period => 100 ms;-- Iso_Types::Thread_Period;

end manage_display_temperature;
```

5.1.2. Manage Display Temperature

The type classifier of Listing 2 declares the interface of the `manage_display_temperature` thread component.

The `regulator_mode` in port is used to get regulator mode from the Manage Regulator Mode function, modeling of which is not part of this study. It receives data of type `regulator_mode` defined in the scope of `Iso_Types` as follows:

```
1
2  data regulator_mode
3    properties
4      Data_Model::Data_Representation => Enum;
5      Data_Model::Enumerators => ("Init", "NORMAL", "FAILED");
6  end regulator_mode;
```

Data component `regulator_mode` models Regulator Mode, as specified in Table 1, as `Enum` with values `Init`, `NORMAL`, `FAILED`.

In Listings 2 in data port `current_temperature` is used to receive the value of type temperature which modeled as below;

```
1  data current_temperature
2    properties
3      Data_Model::Data_Representation => Struct;
4      Data_Model::Element_Names => ("t", "status");
5      Data_Model::Base_Type => (classifier (Iso_Types::measured_temperature_range),
6                  classifier(Iso_Types::valid_flag));
7  end current_temperature;
8
9  data measured_temperature_range
10   properties
11     Data_Model::Base_Type => (classifier(Base_Types::Float));
12     Data_Model::Real_Range => 68.0 .. 105.0;
13     Data_Model::Measurement_Unit => "Fahrenheit";
14 end measured_temperature_range;
```

Data component `current_temperature` on line 1 models the Current Temperature as a structure with two elements; `t` to represent the temperature value, and `status` to represent the status value. The first element `t` is of type `measured_range` defined on line 9 which is a real range from `68.0..105.0` with measuring unit `Fahrenheit`. The second element `status` is of type `valid_flag` which is same as previously modeled for Upper and Lower Desired Temperature variables.

Out data port `display_temperature` in Listing 2 is used to transmit the value of the temperature to be displayed. As temperature is displayed as rounded value of the Current Temperature, `display_temperature` data type is modeled as below;

```
1  data display_temperature
2    properties
3      Data_Model::Base_Type => (classifier(Base_Types::Integer));
4      Data_Model::Integer_Range => 68 .. 105;
5      Data_Model::Measurement_Unit => "Fahrenheit";
6  end display_temperature;
```

Data component `display_temperature` models the Display Temperature as an integer range with values between `68` and `105` as specified on line 4. The measuring unit is `Fahrenheit`. As `display_temperature` data type is only used for displaying the current temperature and there is no validity associated with it so element valid flag is not part to this data type. In order to realize *REQ-MRI-5*, `unspecified_temp` is defined as an out data port in Listing 2 to transmit a string "UNSPECIFIED" if Regulator Mode is Normal and nothing otherwise.

5.1.3. Manage Status

The type classifier of Listing 3 declares the interface of the `manage_status` thread component.

Listing 3: Manage Status component type

```
thread manage_status
    features
        regulator_mode: in data port Iso_Types::regulator_mode;
        regulator_status: out data port Iso_Types::regulator_status;

    properties
        Dispatch_protocol => Periodic;
        Period => 100 ms;

end manage_status;
```

The `regulator_mode` in data port is used to receive Regulator Mode. The data type for this port is the same as specifies in Section 5.1.2.. The `regular_status` out data port transmits newly computed Regulator Status bases on *REQ-MRI-1*, *REQ-MRI-2*, and *REQ-MRI-3*. Based on the specification in Table 1, data type `regulator_status` is modeled as follows:

```
data status
  properties
    Data_Model::Data_Representation => Enum;
    Data_Model::Enumerators => ("Init", "On", "Failed");
end status;
```

All input connections external to these components are modeled as *EIC*, and all external output connections are modeled as *EOC*.

5.1.4. Manage Regulator Interface

With reference to Figure 10, the type classifier of Listing 4 declares the interface of the `manage_regulator_interface` thread group component. Interfaces are defined to establish *EIC* and *EOC* for the coupled component.

Listing 4: Manage Regulator Interface component type

```
thread group manage_regulator_interface
    features
        lower_desired_temperature : in data port Iso_Types::lower_desired_temperature;
        upper_desired_temperature : in data port Iso_Types::upper_desired_temperature;
        current_temperature :  in  data port Iso_Types::current_temperature;
        regulator_mode : in data port Iso_Types::regulator_mode;

        regulator_status : out data port Iso_Types::regulator_status;
        display_temperature : out data port Iso_Types::display_temperature;
        unspecified_temp : out data port Iso_Types::unspecified_value;
        regulator_interface_failure : out data port Base_Types::Boolean;
        lower_desired_temp : out data port Iso_Types::lower_desired_temp;
        upper_desired_temp : out data port Iso_Types::upper_desired_temp;

end manage_regulator_interface;
```

With reference to Figure 9, `lower_desired_temperature` and `upper_desired_temperature` in data ports are used to the get the lower and upper values of desired temperature, respectively. In ports `current_temperature`, and `regulator_mode` are specified to accept current temperature and regulator mode.

Out data ports, `regulator_status`, `display_temperature`, `unspecified_temp`, and `regulator_interface_failure` are specified to transmit data of regulator status, display temperature, unspecified value, and Boolean type. Out ports `lower_desired_temp` and `upper_desired_temp` are specified to transmit desired temperature range.

Data types for these data ports are defined within the scope of another package `Iso_Types` that has been imported using the AADL `with` clause.

5.2. Behavior Modeling

This section describes component behavior modeling with the proposed DEVS Annex (DA). The implementation classifier of each component annotated with DA sub-clause is explained in detail. The external data components referred to as data types are declared within the scope of a package `Iso_Types` and are detailed in Section 5.1. along with respective thread components. The AADL data types (see Section 3.1.2.) are mapped to primitive and compound data types in Java programming language (see Section 3.3.). All data types that are to be communicated between any two DEVS models implemented for the DEVS-Suite simulator are compound and are inherited from the entity class. Variables that are not input events nor output events need to be defined as Java classes. In other words, given the entities, variables, and states in the DEVS Annex, they fall into either input/output or state data categories for the generated code in the DEVS-Suite simulator.

5.2.1. Manage Interface Failure & Desired Range

The **`variables`** section in Listing 5 shows the variable declarations of the DA specification for the example Isolette system. They have been identified through consideration of the system requirements REQ-MRI-6 ,.., REQ-MRI-9. Variable `dt` is of type `display_temperature`, representing the temperature value to be displayed initialized with `90`. Variable `ldt` represents the variable *Lower Desired Temperature* and variable `udt` represents the variable *Upper Desired Temperature*. Data types `lower_desired_temperature` and `upper_desired_temperature` specified for `ldt` and `udt` are same as explained in Section 5.1.1.

Listing 5: Manage Interface Failure and Desired Range component implementation

```
thread implementation manage_interfaceFailure_desiredRange.impl
 annex devs {**

  variables
   dt: Iso_Types::display_temperature => 90;
   ldt: Iso_Types::lower_desired_temperature => (98,"Valid");
   udt: Iso_Types::upper_desired_temperature => (99, "Valid");
   rif: Iso_Types::Bool => true;
   pd : Base_Types::Float => 100.0;
   unspecified_value: Iso_Types::unspecified_value => "unspecified_value";

  states
   Start: initial 0.0;
   Chk_Status: pd;
   Set_Vars: 0.0;  -- for output, as no direct output from external events

  behavior
   deltint [ Start ]-> Chk_Status {} ;
   deltint [Set_Vars]-> Chk_Status {};

   deltext  [Chk_Status, lower_desired_temperature?ldt]-> Set_Vars {
         "if(ldt.get_status() == \"Invalid\" || udt.get_status() == \"Invalid\")
              {rif.setv(true);}
           else
              {rif.setv(false);}"
        };

   deltext  [Chk_Status, upper_desired_temperature?udt]-> Set_Vars {
         "if(udt.get_status() == \"Invalid\" || ldt.get_status() == \"Invalid\")
              {rif.setv(true);}
            else
              {rif.setv(false);}"
          };

    outfn [Set_Vars]-> regulator_interface_failure!rif {};
    outfn [Set_Vars,   (rif != true)]-> lower_desired_temp!ldt.t {};
    outfn [Set_Vars,   (rif != true)]-> upper_desired_temp!udt.t {};
    outfn [Set_Vars,   (rif == true)]-> unspecified_temp!unspecified_value {};

    intest [lower_desired_temperature, (97, "Valid")];
    intest [lower_desired_temperature, (100, "Invalid")];
    intest [upper_desired_temperature, (101, "Invalid")];
    intest [upper_desired_temperature, (97, "Valid")];

    intest [lower_desired_temp, 98];
    intest [regulator_interface_failure, true];


**};

end manage_interfaceFailure_desiredRange.impl;
```

As explained in Section 3.3., for initialization an IntRange object only requires value for *cVal*, thus both `ldt` and `udt` are initialized with value `(98, "Invalid")`, and `(99, "Invalid")`, respectively, where the `98`, and `99` are the initial values for the first elements that are to be modeled as an *IntRange*, and `"Invalid"` is the value is for second elements to be modeled as enumeration. Variable `rif` is to represent either **true** or **false** to indicate validity of the Lower and Upper Desired Temperature. Variable `pd` is of type `Float` with value `100.0`, representing the period of the thread.

The `states` section of Listing 5 contains declarations for admissible states of our running example. The `Start` state marked as `initial` with $ta =$ `0.0` represents an instantaneous starting state. State `Chk_Status` with $ta =$ `pd` is a transient state. Validity of the status element of variables `ldt` and `udt` is checked in this state and variable `rif` is set accordingly. State `Set_Vars` is also an instantaneous state with $ta =$ `0.0`. It has been defined to facilitate output generation as these are only allowed on internal transitions.

In Listing 5, the external transition functions starting with `deltext` have `Chk_Status` as source state and `Set_Vars` as destination state. The external message in the first function specifies that the message received on port `lower_desired_temperature` is stored in variable `ldt` while in the second function the message received on port `upper_desired_temperature` is stored in variable `udt`. The behavior actions contain an *if-else* statement for setting value for `rif` variable[4]. If the status of `ldt` or `udt` is `Invalid` variable `rif` is assigned `true` otherwise it is assigned `false` as per requirements REQ-MRI-6 and REQ-MRI-7.

For our running example, two internal transition functions starting with `deltint` are defined in Listing 5. One internal transition function has `Start` as source state and `Chk_Status` as destination state while the other internal transition function has `Set_Vars` as source state and `Chk_Status` as destination state. Empty braces indicate that no behavior action is required to be specified.

In Listing 5, four output functions are specified starting with `outfn` have `Set_Vars` as source state. No conditional statement is specified for first input function as it transmits current value of variable `rif` through `regulator_interface_failure` out port. For second and third out put functions, values of the first element `t`, temperature range, of the `ldt` and `udt` are transmitted through out ports `lower_desired_temp` and `upper_desired_temp`, respectively, if the condition `(rif != true)` holds as per requirement REQ-MRI-8. Fourth output function is specified for requirement REQ-MRI-9 and transmits `unspecified_value` through out port `unspecified_temp` if condition `(rif == true)` holds.

For our running example, in Listing 5, two input functions are specified to provide test inputs for `lower_desired_temperature` in port with values `(97, "Valid")` and `(100, "Invalid")`. Two Input functions are also specified to provide test inputs for `upper_desired_temperature` in port with values `(98, "Valid")` and `(101, "Invalid")`. The first element in these compound variables represent the current temperature while the second element represents the status of the current temperature. One input function is defined for `upper_desired_temp` with value `98` and one input function is defined for `regulator_interface_failure` with value `true`.

5.2.2. Manage Display Temperature

The DA subclause of the implementation classifier of Listing 6 models the detailed behavior of `manage_display_temperature` thread component. Variable `rgm` with initial value `INIT` is of type `regulator_mode` and represents the regulator mode. In the `variables` section, variable `crt` is of type `current_temperature` with initial value `68.0` for the first element and `Valid` for the second element. Variable `ust` is of type Boolean with `false` initial value and is used as flag to indicate the unspecified temperature according to *REQ-MRI-5*. Variable `pd` is of type `Float` and holds the value `100.0` representing the period of the thread.

The `states` section of Listing 6 contains declarations for admissible states of the `manage_display_temperature` thread. The `Start` state with $ta =$ `0.0` is an `initial` state and represents an instantaneous starting state. State

[4]As double quotation marks delimit strings in Xtext grammar so `\"` is used to specify double quotation marks with in a string in behavior actions.

`Chk_Mode` with ta = `pd` is a transient state declared to update the variable `ust` based on the regulator mode. State `Set_Vars` is also an instantaneous state with ta = `0.0`. It has been defined to facilitate output generation as these are only allowed on internal transitions.

`behavior` section of the DA subclause in Listing 6 includes three internal transitions functions. First internal transition function has `Start` as source state and `Chk_Mode` as destination state. The second internal transition function is defined for `Chk_Mode` and has `Set_Vars` as destination state while the third has `Chk_Mode` as source state and `Set_Vars` as destination state. Empty braces indicate that no behavior action is required to be specified.

In Listing 6, two external transition functions starting with **`deltext`** and having `Chk_Mode` as source states are defined. One has `Chk_Mode` as destination state while other has `Set_Vars` as destination state. The external message in the first function specifies that the message received on port `regulator_mode` is stored in variable `rgm` while in the second function the message received on port `current_temperature` is stored in variable `crt`. No behavior action is define for the first function while the behavior action for the second function contains an *if-else* statement for setting value for `ust` variable. If the Regulator Mode is `Normal`, variable `ust` is assigned **`true`** otherwise it is assigned **`false`** as per requirements REQ-MRI-4 and REQ-MRI-5.

In Listing 6, two output functions are specified for `Set_Vars` state. For the first output function, value of the first element `t`, temperature range, of the `crt` is transmitted through `display_temperature` if the condition `(ust != true)` holds as per requirement REQ-MRI-4. Second output function is specified for requirement REQ-MRI-5 and transmits `unspecified_value` through out port `unspecified_temp` if condition `(ust == true)` holds.

Listing 6: Manage Display Temperature component implementation

```
thread implementation manage_display_temperature.impl
 annex devs {**

  variables
   rgm : Iso_Types::regulator_mode => "INIT" ;
   crt : Iso_Types::current_temperature => (68.0, "Valid");
   ust : Iso_Type::Bool => false;
   pd : Base_Types::Float => 100;
   unspecified_value : Iso_Types::unspecified_value => "unspecified_value";


  states
   Start: initial  0.0;
   Chk_Mode: pd;
   Set_Vars: 0.0;

  behavior
   deltint [ Start ]-> Chk_Mode {} ;
   deltint [Chk_Mode]-> Set_Vars {};
   deltint [Set_Vars]-> Chk_Mode {};

   deltext  [Chk_Mode, regulator_mode?rgm]-> Chk_Mode {};
   deltext  [Chk_Mode, current_temperature?crt]-> Set_Vars {
          "if(rgm.getv() == \"NORMAL\")
              {ust.setv(false);}
          else if(rgm.getv() == \"INIT\" || rgm.getv() == \"FAILED\")
              {ust.setv(true)}"
          };

   outfn [Set_Vars,  (ust != true)]-> display_temperature!crt.t  {};
   outfn [Set_Vars,  (ust == true)]-> unspecified_temp!unspecified_value {};

   intest [regulator_mode,  "NORMAL"];
   intest [regulator_mode,  "INIT"];
   intest [regulator_mode,  "FAILED"];
   intest [current_temperature, (102, "Valid")];
   intest [current_temperature, (106, "Invalid")];

**};

end manage_display_temperature.impl;
```

In Listing 6, two output functions are specified for `Set_Vars` state. For the first output function, value of the first element `t`, temperature range, of the `crt` is transmitted through `display_temperature` if the condition `(ust != true)` holds as per requirement REQ-MRI-4. Second output function is specified for requirement REQ-MRI-5 and trans-

mits unspecified_value through out port unspecified_temp if condition (ust == true) holds.

In Listing 6, three input functions are specified to provide test inputs for regulator_mode in port with all possible values "NORMAL", "INIT", and "FAILED". Two input functions are defined for current_temperature in data port with values (102, "Valid") and (106, "Invalid"). The first element in these composite values represent the current temperature while the second element represents the status of the current temperature.

5.2.3. Manage Status

DA subclause of the implementation classifier of Listing 7 specifies the behavior of the manage_status thread component. Variable rgm with initial value INIT is of type regulator_mode and represents the Regulator Mode. Variable rgs represents Regulator Status and is of type regulator_status with initial value Init. Variable pd is of type Float and holds the value 100.0 representing the period of the thread.

In Listing 7, the **states** section contains declarations for admissible states of the manage_status thread. The Start state with $ta = 0.0$ is an **initial** state and represents an instantaneous starting state. State Chk_Mode with $ta =$ pd is a transient state declared to update the variable ust based on the regulator mode. State Set_Vars is also an instantaneous state with $ta = 0.0$. It has been defined to facilitate output generation as these are only allowed on internal transitions.

Listing 7: Manage Status component implementation

```
thread implementation manage_status.impl
 annex devs{**

  variables
   rgm : Iso_Types::regulator_mode => "INIT";
   pd : Base_Types::Float =>  100.0;
   rgs : Iso_Types::regulator_status => "Init";

  states
   Start: initial  0.0;
   Chk_Mode: pd;
   Set_Vars: 0.0;

  behavior
   deltint [ Start ]-> Chk_Mode {} ;
   deltint [Chk_Mode]-> Set_Vars{};
   deltint [Set_Vars]-> Chk_Mode{};

   deltext [Chk_Mode, regulator_mode?rgm]-> Set_Vars {
          "if(rgm.getv() == \"INIT\") {rgs.setv(\"Init\");}
           else if(rgm.getv() == \"NORMAL\" )
             {rgs.setv(\"On\");}
           else if(rgm.getv() == \"FAILED\")
             {rgs.setv(\"Failed\");} "
        };

   outfn [Set_Vars]-> regulator_status!rgs {};

   intest [regulator_mode,  "NORMAL"];
   intest [regulator_mode,  "INIT"];
   intest [regulator_mode,  "FAILED"];

**};

end manage_status.impl;
```

In Listing 7, the **behavior** section of the DA subclause includes three internal transitions functions. One internal transition function has Start as source state and Chk_Mode as destination state. The second internal transition function is defined for Chk_Mode and has Set_Vars as destination state while the third has Chk_Mode as source state and Set_Vars as destination state. Empty braces indicate that no behavior action is required to be specified.

The external transition function in Listing 7 is defined with Chk_Mode as source and destination state. The external message in this function specifies that the current value received on port regulator_mode is stored in variable rgm. The behavior action contains an *if-else* statement for setting value for rgs variable. If the Regulator Mode is Normal, variable rgs is assigned Init, On, or FAILED as per requirements REQ-MRI-1,..., REQ-MRI-3.

In Listing 7, an output functions is specified for `Set_Vars` state to transmit value of the `rgs` variable.

For stand-alone testing, three test input functions are specified to provide inputs for `regulator_mode` in port with all possible values `NORMAL`, `INIT`, and `FAILED`.

5.2.4. Manage Regulator Interface

With reference to Figure 10, implementation classifier in Listing 8 specifies the subcomponents and connections between them. Section `subcomponents` has references to the respective implementation classifiers of the thread components explained above.

Section `connections` in Listing 8 specifies all the port connections (EICs and EOCs) among the subcomponents. Each connection specification starts with a connection identifier followed by the key word `port` (to mark the port connection), name of the source(component and) out data port, and the destination (component and) in data port.

Listing 8: Manage Regulate Interface component implementation

```
thread group implementation manage_regulator_interface.impl

  subcomponents
    manage_status : thread manage_status.impl;
    manage_display_temperature: thread manage_display_temperature.impl;
    manage_interface_failure: thread manage_interfaceFailure_desiredRange.impl;

  connections
    EIC1 : port regulator_mode -> manage_status.regulator_mode;
    EOC2 : port manage_status.regulator_status -> regulator_status;

    EIC3 : port regulator_mode -> manage_display_temperature.regulator_mode;
    EIC4 : port current_temperature -> manage_display_temperature.current_temperature;
    EOC5 : port manage_display_temperature.display_temperature -> display_temperature;
    EOC6 : port manage_display_temperature.unspecified_temp -> unspecified_temp;

    EIC5 : port lower_desired_temperature -> manage_interface_failure.lower_desired_temperature;
    EIC6 : port upper_desired_temperature -> manage_interface_failure.upper_desired_temperature;
    EOC7 : port manage_interface_failure.lower_desired_temp -> lower_desired_temp;
    EOC8 : port manage_interface_failure.upper_desired_temp -> upper_desired_temp;
    EOC9 : port manage_interface_failure.regulator_interface_failure ->
                                                      regulator_interface_failure;
    EOC10: port manage_interface_failure.unspecified_temp -> unspecified_temp;

end manage_regulator_interface.impl;
```

Listing 8 has four EICs and six EOCs.

5.3. Code Generation for DEVS Simulation

This section specifies code generation using the AADL to DEVS CoDE generation Engine (ADCoDE), as explained in Section 3.3., for all three components of the Isolette example. Data classes, structural, and behavioral code generated for each component is explained in detail. Data classes are organized in a package *Iso_Types* created by the ADCoDE with the name same as the AADL file containing the data components modeling the data types while the model classes are organized in package *Model*. Name of the model is extended with "_sim" to mark the simulation class.

5.3.1. Manage Interface Failure & Desired Range

Data Classes: ADCoDE generates two data classes required to be used for correct modeling of the `manage_interfaceFailure_desiredRange` thread component. One for the `lower_desire_temperature` and one is for the `upper_desired_temperature`. Listing 9 contains the class generated for `lower_desired_temperature` while the class generated for `upper_desired_temperature` is same except for the values, hence is not discussed here. Complete AADL component models along with generated classes are avaiable at `https://github.com/ehah/AADL-DEVS-Framework`.

In Listing 9, class `lower_desired_temperature` is generated based on the data modeling explained in Section 5.1. It extends Java class *entity* and has two private variables. Variable `t` is of type **`IntRange`** with `97` and 99 as `minVal` and `maxVal`, respectively. As AADL enumeration type is mapped to String, variable `status` is generated as of type `String`.

Along with a default constructor, three other constructors are also generated with for this class to set the values for variable `cVal`, variable `t` (with `maxVal`, `minVal`, and `cVal`), both variables `t` and `status`, and the variable `cVal` and `status`, respectively. Getters and Setters are generated for both the private variables.

Method `isInRange` gets an integer value and returns if this value is within the temperature range `t` defined by the `minVal` and `maxVal`.

Structural Code and Behavioral Code: Listing 10 contains the model class generated by the ADCoDE for the respective thread `manage_interfaceFailure_desiredRang`.

Listing 9: Data class generated for *lower_desired_temperature*

```
package Iso_Types;

import GenCol.*;
import structuredEntities.*;

public class lower_desired_temperature extends entity {

  private IntRange t = new IntRange(97, 99);
  private String status;

  public lower_desired_temperature() {

  }

  public lower_desired_temperature(int cv) {
    t.setcVal(cv);
  }

  public lower_desired_temperature(int lv, int uv, int cv) {
    t.setminVal(lv);
    t.setmaxVal(uv);
    t.setcVal(cv);
  }

  public lower_desired_temperature(int lv, int uv, int cv, String status ) {
    t.setminVal(lv);
    t.setmaxVal(uv);
    t.setcVal(cv);
    this.status = status;
  }

  public lower_desired_temperature(int cv, String status ) {
    t.setcVal(cv);
    this.status = status;
  }

  public void set_t(int lv, int uv, int cv) {
    t.setminVal(lv);
    t.setmaxVal(uv);
    t.setcVal(cv);
  }

  public IntRange get_t() {
    return t;
  }

  public Boolean isInRange(int val){
    return (val >= t.getminVal() && val <= t.getmaxVal());
  }

  public void set_status(String status) {
    this.status = status;
  }

  public String get_status() {
    return this.status;
  }
}
```

Defined in package `RegulateTemperature` (which will contain all the data and model classes), Class `manage_interfaceFailure_desiredRange_impl_sim` class imports packages `structuredEntities` and `Iso_Types` that contain DEVS-Suite extension and ADCoDE generated data classes, respectively. Rest of the imports are required for DEVS-Suite simulation.

This class extends the `ViewableAtomic` with required variables and interface ports as specified in Listing 1 and Listing 5. For variable `dt` which is of type **`IntRange`**, the current value `90` is extracted from the specification in DA subclause while the values `68` and `105`, for minimum and maximum limits, are extracted from its definition in package `Iso_Types`. Variables `ldt` and `udt` are of type `lower_desired_temperature` and `upper_desired_temperature`, respectively, with particular initial values. Variable `rif` is generated as of type **`booleanEnt`** while `pd` is of type **`doubleEnt`**. Variable `unspecified_value` is generated as of type **`stringEnt`**. Integer ranges `lower_desired_temp` and `upper_desired_temp` are generated with specific values extracted from specifications in DA subclause and definitions in the package `Iso_Types`. All the required classes have already been generated as explained previously.

Listing 10: Model class generated for thread *manage_interfaceFailure_desiredRange* with thread implementation *manage_interfaceFailure_desiredRange_impl*

```
-- generated by ADCoDE @ 2019-11-11 00:32:03
-- This class represents DEVS atomic model for thread manage_interfaceFailure_desiredRange
-- with manage_interfaceFailure_desiredRange.impl

package RegulateTemperature;

import java.lang.*;
import GenCol.*;
import model.modeling.*;
import model.simulation.*;
import view.modeling.ViewableAtomic;
import view.simView.*;
import Component.structuredEntities.*;
import Component.Iso_Types.*;

public class manage_interfaceFailure_desiredRange_impl_sim extends ViewableAtomic {

  -- variables
  private IntRange dt = new IntRange(68, 105, 90);
  private lower_desired_temperature ldt = new lower_desired_temperature(98, "Valid");
  private upper_desired_temperature udt = new upper_desired_temperature(99, "Valid");
  private booleanEnt rif = new booleanEnt(true);
  private doubleEnt pd = new doubleEnt(100.0);
  private stringEnt unspecified_value =  new stringEnt("unspecified_value");

  -- for port datatypes
  private IntRange lower_desired_temp = new IntRange(96, 101, 98);
  private IntRange upper_desired_temp = new IntRange(97, 102, 0);

  public manage_interfaceFailure_desiredRange_impl_sim() {
    this("manage_interfaceFailure_desiredRange.impl");
  }

  public manage_interfaceFailure_desiredRange_impl_sim(String name) {
    super(name);

    -- input and output ports from and for other atomic/coupled models
    -- it is recommended to use short names

    addInport("lower_desired_temperature");
    addInport("upper_desired_temperature");
    addOutport("regulator_interface_failure");
    addOutport("lower_desired_temp");
    addOutport("upper_desired_temp");
    addOutport("unspecified_temp");

    -- test input for standalone testing
    addTestInput("lower_desired_temperature", new lower_desired_temperature(97, "Valid"));
    addTestInput("lower_desired_temperature", new lower_desired_temperature(100, "Invalid"));
    addTestInput("upper_desired_temperature", new upper_desired_temperature(101, "Invalid"));
    addTestInput("upper_desired_temperature", new upper_desired_temperature(98, "Valid"));
    addTestInput("lower_desired_temp", new intEnt(98));
    addTestInput("regulator_interface_failure", new booleanEnt(true));
  }

  public void initialize() {
    -- Can be updated by the modeler
    phase = "Start";
    sigma = 0.0;

    super.initialize();
  }

  public void deltint() {
    if (phaseIs("Start")) {
      holdIn("Chk_Status", pd.getv());
    }
    if (phaseIs("Set_Vars")) {
      holdIn("Chk_Status", pd.getv());
    }
  }

  public void deltext(double e, message x) {
    Continue(e);

    if (phaseIs("Chk_Status")) {
      for(int i=0; i<x.getLength(); i++) {
        if(messageOnPort(x, "lower_desired_temperature", i)) {
        ldt = (lower_desired_temperature) x.getValOnPort("lower_desired_temperature", i);
        if(ldt.get_status() == "Invalid" || udt.get_status() == "Invalid")
                      {rif.setv(true);}
                  else {rif.setv(false);}
        holdIn("Set_Vars", 0.0);
        }
      }
    }
    if (phaseIs("Chk_Status")) {
      for(int i=0; i<x.getLength(); i++) {
        if(messageOnPort(x, "upper_desired_temperature", i)) {
        udt = (upper_desired_temperature) x.getValOnPort("upper_desired_temperature", i);
        if(udt.get_status() == "Invalid" || ldt.get_status() == "Invalid")
                      {rif.setv(true);}
                  else { rif.setv(false); }
        holdIn("Set_Vars", 0.0);
        }
      }
    }
  }

  public message out() {
    message m = new message();

    if (phaseIs("Set_Vars"))
    m.add(makeContent("regulator_interface_failure", rif));
    if (phaseIs("Set_Vars"))
     {
     if(  rif.getv()!= true)
     m.add(makeContent("lower_desired_temp",  ldt.get_t()));
     }
    if (phaseIs("Set_Vars"))
     {
     if(  rif.getv()!= true)
     m.add(makeContent("upper_desired_temp",  udt.get_t()));
     }
    if (phaseIs("Set_Vars"))
     {
     if(  rif.getv()== true)
     m.add(makeContent("unspecified_temp",  unspecified_value));
     }

    return m;
  }
}
```

In Listing 10, the interface ports are added using the `addInport` and `addOutport` methods. The `lower_desired_temperature` and the `upper_desired_temperature` are used to map to the `lower_desired_temperature` and the `upper_desired_temperature` in their corresponding AADL type classifier. The `regulator_interface_failure`, the `lower_desired_temp`, the upper_desired_temp, and the unspecified_temp out data ports are mapped to their counterparts as specified in the **features** section of their corresponding AADL type classifier.

In Listing 10, the addTestInput methods are generated to map the **intest** declarations with one-to-one mapping. Two of these methods provide the test inputs for lower_desired_temperature in port with values (97, "Valid") and (100, "Invalid"). Two methods are also specified to provide test inputs for upper_desired_temperature in port with values (98, "Valid") and (101, "Invalid") while one input method is defined for upper_desired_temp with value 98. One input function is defined for regulator_interface_failure with a **booleanEnt** object using value **true**, and one input function is defined for lower_desired_temp with an **intEnt** object using value 98.

Method initialize in Listing 10 contains the explicit initialization of the variables required by the DEVS-Suite. Variable phase="Start" is to set the initial state with time advance function Sigma=0.0. Statement super.initialize() calls the initialize method of the ViewableAtomic class.

In Listing 10, internal transition function **deltint** has a control structure with methods phaseIs to map the internal transition functions defined in Listing 5. If phaseIs("Start") meaning form the Start state, when the elapsed time $e = ta$, the control moves the Chk_Status with $ta = pd$. The second if statement states that if phaseIs("Set_Vars") and when the elapsed time $e = ta$, the control moves to the Chk_Status with $ta = pd$.

External transition function in Listing 10, is generated for the external transition functions defined in Listing 5 by combining both the occurrences in a control structure. If the phaseIs("Chk_Status") and a message is received on either the lower_desired_temperature or upper_desired_temperature in data port, the contents of the message are explored using method messageOnPort and the value is stored in object ldt or udt, accordingly. Variable rif is then updated based on the current value of the second elements of both ldt and udt. The control is then passed to the Set_Vars state with $ta=0.0$ which is an instantaneous state and is introduced to support output generation.

Listing 11: Data class generated for *current_temperature*

```
package Iso_Types;

import GenCol.*;
import structuredEntities.*;

public class current_temperature extends entity {

  private DoubleRange t = new DoubleRange(68.0, 105.0);
  private String status;

  public current_temperature() {

  }

  public current_temperature(double cv) {
    t.setcVal(cv);
  }

  public current_temperature(double lv, double uv, double cv) {
    t.setminVal(lv);
    t.setmaxVal(uv);
    t.setcVal(cv);
  }

  public current_temperature(double lv, double uv, double cv, String status ) {
    t.setminVal(lv);
    t.setmaxVal(uv);
    t.setcVal(cv);
    this.status = status;
  }

  public current_temperature(double cv, String status ) {
    t.setcVal(cv);
    this.status = status;
  }

  public void set_t(double lv, double uv, double cv) {
    t.setminVal(lv);
    t.setmaxVal(uv);
    t.setcVal(cv);
  }
  public DoubleRange get_t() {
    return t;
  }

  public Boolean isInRange(double val) {
    return (val >= t.getminVal() && val <= t.getmaxVal());
  }

  public void set_status(String status) {
    this.status = status;
  }

  public String get_status() {
    return this.status;
  }
}
```

In Listing 10, method `out` is generated for the output functions defined in Listing 5. The output is only generated on `Set_Vars` state by adding the message contents to the message object created using the `makeContent` method. If condition `(rif!=true)` holds then values of first element of both the objects `ldt` and `udt` are transmitted on `lower_desired_temp` and `upper_desired_temp` output ports, respectively. If condition `(rif==true)` holds then `unspecified_value` is transmitted on output port `unspecified_temp`.

5.3.2. Manage Display Temperature

Data Classes: Listing 9 contains the class generated for `lower_desired_temperature` required to be used for correct modeling of the `manage_display_temperature` thread component.

In Listing 11, class `current_temperature` is generated based on the data modeling explained in Section 5.1. It extends Java class *entity* and has two private variables. Variable `t` is of type `DoubleRange` with `68.0` and `105.0` as `minVal` and `maxVal`, respectively. As AADL enumeration is mapped to String in Java, the variable `status` has type `String`.

Along with a default constructor, three other constructors are also generated with for this class to set the values for variable `cVal`, variable `t` (with `maxVal`, `minVal`, and `cVal`), both variables `t` and `status`, and the variable `cVal` and `status`, respectively. Getters and Setters are generated for both the private variables.

Method `isInRange` gets an integer value and returns if this value is within the temperature range `t` defined by the `minVal` and `maxVal`.

Structural Code and Behavioral Code: Listing 12 contains the model class generated by the ADCoDE for the respective thread `manage_display_temperature`.

Defined in package `RegulateTemperature` (which will contain all the data and model classes), Class `manage_display_temperature_impl_sim` class imports packages `structuredEntities` and `Iso_Types` that contain DEVS-Suite extension and ADCoDE generated data classes, respectively. Rest of the imports are required for DEVS-Suite simulation.

This class extends the `ViewableAtomic` with required variables and interface ports specified in Listing 2 and Listing 6. For variable `rgm` which is of type `stringEnt` with initial value `INIT`. Variable `crt` is of type `current_temperature` (a newly generated class in the previous step) with initial values `68` and `Valid`.

Variable `ust` is generated as of type `booleanEnt` while `pd` is of type `doubleEnt` with values `false` and `100.0` extracted from the specification in the DA subclause. Variable `unspecified_value` is generated as of type `stringEnt`.

All the required classes have already been generated as explained previously.

Listing 12: Model class generated for thread *manage_display_temperature* with thread implementation *manage_display_temperature_impl*

```
-- generated by ADCoDE @ 2019-11-11 03:00:02
-- This class represents DEVS atomic model for thread manage_display_temperature with
-- manage_display_temperature.impl

package RegulateTemperature;

import java.lang.*;
import GenCol.*;
import model.modeling.*;
import model.simulation.*;
import view.modeling.ViewableAtomic;
import view.simView.*;
import Component.structuredEntities.*;
import Component.Iso_Types.*;

public class manage_display_temperature_impl_sim extends ViewableAtomic {
  -- variables
  private stringEnt rgm =  new stringEnt("INIT");
  private current_temperature crt = new current_temperature(68.0, "Valid");
  private booleanEnt ust = new booleanEnt(false);
  private doubleEnt pd = new doubleEnt(100.0);
  private stringEnt unspecified_value =  new stringEnt("unspecified_value");

  -- for port datatypes
  private IntRange display_temperature = new IntRange(68, 105, 0);

  public manage_display_temperature_impl_sim() {
    this("manage_display_temperature.impl");
  }

  public manage_display_temperature_impl_sim(String name) {
    super(name);

    -- input and output ports from and for other atomic/coupled models
    -- it is recommended to use short names

    addInport("regulator_mode");
    addInport("current_temperature");
    addOutport("display_temperature");
    addOutport("unspecified_temp");

    -- test input for standalone testing
    addTestInput("regulator_mode", new stringEnt("NORMAL"));
    addTestInput("regulator_mode", new stringEnt("INIT"));
    addTestInput("regulator_mode", new stringEnt("FAILED"));
    addTestInput("current_temperature", new current_temperature(102, "Valid"));
  }

  public void initialize() {
    -- Can be updated by the modeler
    phase = "Start";
    sigma = 0.0;

    super.initialize();
  }

  public void deltint() {
    if (phaseIs("Start")) {
      holdIn("Chk_Mode", pd.getv());
    }
    if (phaseIs("Chk_Mode")) {
      holdIn("Set_Vars", 0.0);
    }
    if (phaseIs("Set_Vars")) {
      holdIn("Chk_Mode", pd.getv());
    }
  }

  public void deltext(double e, message x) {
    Continue(e);

    if (phaseIs("Chk_Mode")) {
      for(int i=0; i<x.getLength(); i++) {
        if(messageOnPort(x, "regulator_mode", i)) {
        rgm = (stringEnt) x.getValOnPort("regulator_mode", i);
        holdIn("Chk_Mode", pd.getv());
        }
      }
    }
    if (phaseIs("Chk_Mode")) {
      for(int i=0; i<x.getLength(); i++) {
        if(messageOnPort(x, "current_temperature", i)) {
        crt = (current_temperature) x.getValOnPort("current_temperature", i);
        if(rgm.getv() == "NORMAL")
                  {ust.setv(false);}
               else if(rgm.getv() == "INIT" || rgm.getv() == "FAILED")
                 {ust.setv(true);}
        holdIn("Set_Vars", 0.0);
        }
      }
    }
  }

  public message out() {
    message m = new message();

    if (phaseIs("Set_Vars"))
     {
     if(  ust.getv()!= true)
     m.add(makeContent("display_temperature",  crt.get_t()));
     }
    if (phaseIs("Set_Vars"))
     {
     if(  ust.getv()== true)
     m.add(makeContent("unspecified_temp",  unspecified_value));
     }

    return m;
  }
}
```

In Listing 12, interface ports are added using `addInport` and `addOutport` methods. In port `regulator_mode`, `current_temperature` in data ports and `display_temperature`, `unspecified_temp` out data ports are generated to map in and out data ports as specified in the **`features`** section of the respective AADL type classifier.

In Listing 12, Three `addTestInput` methods are generated for `regulator_mode` in port with values `NORMAL`, `INIT` , `FAILED`. One input method is generated for `current_temperature` in port with value `102` for the first element and value `Valid` for the second element.

Method `initialize` in Listing 12 contains the explicit initialization of the variables required by the DEVS-Suite. Variable `phase="Start"` is to set the initial state with time advance function `Sigma=0.0`. Statement `super.initialize()` calls the initialize method of the `ViewableAtomic` class.

In Listing 12, internal transition function **`deltint`** has a control structure with methods `phaseIs` to map the internal transition functions defined in Listing 5. If `phaseIs("Start")` meaning form the `Start` state, when the elapsed time $e = ta$, the control moves the `Chk_Mode` with $ta = pd$. The second if statement states that if `phaseIs("Chk_Mode")` and when the elapsed time $e = ta$, the control moves to the `Set_Vars` with $ta = 0.0$. The third if statement states that if `phaseIs("Set_Vars")` and when the elapsed time $e = ta$, the control moves to the `Chk_Mode` with $ta = pd$.

External transition function in Listing 12, is generated for the external transition functions defined in Listing 6 by combining both the occurrences in a control structure. If the `phaseIs("Chk_Mode")` and a message is received on the `regulator_mode` in data port, the contents of the message are explored using method `messageOnPort` and the value is stored in object `rgm`. The control stays in `Chk_Mode` for the next `pd` time units. If an input is received at `current_temperature` in data port, variable `crt` is updated with this newly value. Then value of the `ust` is updated based on the current value of the `rgm` and the control is then passes to `Set_Vars` state with `ta=0.0` which is an instantaneous state and is introduced to support output generation.

In Listing 12, method **`out`** is generated for the output functions defined in Listing 6. The output is only generated on `Set_Vars` state by adding the message contents to the message object created using the `makeContent` method. If condition `(ust.getv()!=true)` holds then value of first element `crt` is transmitted on `display_temperature`. If condition `(ust.getv()==true)` holds then `unspecified_value` is transmitted on output port `unspecified_temp`.

5.3.3. Manage Status

Data Classes: As specified in Listing 3 and Listing 7, all data types used with this thread component are primitive data types so no new data class is generated.

Structural Code and Behavioral Code: Listing 13 contains the model class generated by the ADCoDE for the respective thread `manage_status`.

Defined in package `RegulateTemperature` (which will contain all the data and model classes), the class `manage_status_impl_sim` imports packages `structuredEntities` and `Iso_Types` that contain the DEVS-Suite extension and ADCoDE generated data classes, respectively. Rest of the imports are required for DEVS-Suite simulation.

This class extends the `ViewableAtomic` with required variables and interface ports specified in Listing 3 and Listing 7. For variable `rgm` which is of type **`stringEnt`** with initial value `INIT`. Variable `crt` is of type `current_temperature` (a newly generated class in the previous step) with initial values `68` and `Valid`.

Variable `rgm` is generated as of type **`stringEnt`** while `pd` is of type **`doubleEnt`** with values **`false`** and `100.0` extracted from the specification in the DA subclause. Variable `rgs` is generated as of type **`stringEnt`**.

All the required classes have already been generated as explained previously.

Listing 13: Model class generated for thread *manage_status* with thread implementation *manage_status_impl*

```
-- generated by ADCoDE @ 2019-11-11 16:08:06
-- This class represents DEVS atomic model for thread manage_status with manage_status.impl

package RegulateTemperature;

import java.lang.*;
import GenCol.*;
import model.modeling.*;
import model.simulation.*;
import view.modeling.ViewableAtomic;
import view.simView.*;
import Component.structuredEntities.*;
import Component.Iso_Types.*;

public class manage_status_impl_sim extends ViewableAtomic {
  -- variables
  private stringEnt rgm =  new stringEnt("INIT");
  private doubleEnt pd = new doubleEnt(100.0);
  private stringEnt rgs =  new stringEnt("Init");


  public manage_status_impl_sim() {
    this("manage_status.impl");
  }

  public manage_status_impl_sim(String name) {
    super(name);

    -- input and output ports from and for other atomic/coupled models
    -- it is recommended to use short names

    addInport("regulator_mode");
    addOutport("regulator_status");

    -- test input for standalone testing
    addTestInput("regulator_mode", new stringEnt("NORMAL"));
    addTestInput("regulator_mode", new stringEnt("INIT"));
    addTestInput("regulator_mode", new stringEnt("FAILED"));
  }

  public void initialize() {
    -- Can be updated by the modeler
    phase = "Start";
    sigma = 0.0;
    super.initialize();
  }

  public void deltint() {
    if (phaseIs("Start")) {
      holdIn("Chk_Mode", pd.getv());
    }
    if (phaseIs("Chk_Mode")) {
      holdIn("Set_Vars", 0.0);
    }
    if (phaseIs("Set_Vars")) {
      holdIn("Chk_Mode", pd.getv());
    }
  }

  public void deltext(double e, message x) {
    Continue(e);

    if (phaseIs("Chk_Mode")) {
      for(int i=0; i<x.getLength(); i++) {
        if(messageOnPort(x, "regulator_mode", i)) {
        rgm = (stringEnt) x.getValOnPort("regulator_mode", i);
        if(rgm.getv() == "INIT") {rgs.setv(Init);}
                else if(rgm.getv() == "NORMAL")
                {rgs.setv("On");}
                else if(rgm.getv() == "FAILED")
                {rgs.setv("Failed");}
        holdIn("Set_Vars", 0.0);
        }
      }
    }
  }

  public message out() {
    message m = new message();

    if (phaseIs("Set_Vars"))
    m.add(makeContent("regulator_status", rgs));

    return m;
  }
}
```

In Listing 13, interface ports are added using `addInport` and `addOutport` methods. In port `regulator_mode` in data port and `regulator_status` out data port is generated to map in and out data ports as specified in the **features** section of the respective AADL type classifier.

In Listing 13, three `addTestInput` methods are generated for `regulator_mode` in port with values `NORMAL`, `INIT` , `FAILED`.

Method `initialize` in Listing 13 contains the explicit initialization of the variables required by the DEVS-Suite. Variable `phase="Start"` is to set the initial state with time advance function `Sigma=0.0`. Statement `super.initialize()` calls the initialize method of the `ViewableAtomic` class.

In Listing 13, internal transition function **`deltint`** has a control structure with methods `phaseIs` to map the internal transition functions defined in Listing 7. If `phaseIs("Start")` meaning form the `Start` state, when the elapsed time $e = ta$, the control moves the `Chk_Mode` with $ta = pd$. The second if statement states that if `phaseIs("Chk_Mode")` and when the elapsed time $e = ta$, the control moves to the `Set_Vars` with $ta = 0.0$. The third if statement states that if `phaseIs("Set_Vars")` and when the elapsed time $e = ta$, the control moves to the `Chk_Mode` with $ta = pd$.

External transition function in Listing 13, is generated for the external transition functions defined in Listing 7 in a control structure. If the `phaseIs("Chk_Mode")` and a message is received on the `regulator_mode` in data port, the contents of the message are explored using method `messageOnPort` and the value is stored in object `rgm`. Then value of the `rgs` is updated based on the current value of the `rgm` and the control is then passes to `Set_Vars` state with `ta=0.0` which is an instantaneous state and is introduced to support output generation.

In Listing 13, method **`out`** is generated for the output function. The output is only generated on `Set_Vars` state by adding the message contents to the message object created using the `makeContent` method. Upon activation, current value of the `rgs` is transmitted through `regulator_status` out data port.

5.3.4. Manage Regulator Interface

Listing 14 contains the model class generated by the ADCoDE for the respective thread group `manage_regulator_interface`.

Defined in package `RegulateTemperature` (which will contain all the data and model classes), the class `manage_status_impl_sim` imports packages `structuredEntities` and `Iso_Types` that contain DEVS-Suite extension and ADCoDE generated data classes, respectively. Rest of the imports are required for DEVS-Suite simulation.

This class extends the `ViewableDiagraph` with required interface ports specified in Listing 4 and Listing 8. Three objects `manage_status`, `manage_display_temperature`, and `manage_interface_failure` are generated and added to instantiate ViewableAtomic class for already generated respective atomic models.

Interface ports extracted from respective atomic models are added using `addInport` and `addOutport` methods. All the test input ports are also extracted from respective atomic models and are added to using `addTestInput` method. Multiple occurrences of method `addCoupling` are then generated for the connections (EICs and EOCs) specified in Listing 4. Specification of each occurrence starts with a connection identifier followed by the key word **`port`** (to mark the port connection), name of the source component followed by the out data port, which is then followed by the destination component and in data port. For an EIC key word `this` is used for the source component while for an EOC it is used for destination component.

Listing 14: Model class generated for thread *manage_regulator_interface* with thread implementation *manage_regulator_interface_impl*

```
-- generated by ADCoDE @ 2019-11-12 01:25:19
-- This class represents DEVS coupled model for thread group manage_regulator_interface.impl

package RegulateTemperature;

import java.awt.*;
import GenCol.*;
import model.modeling.*;
import model.simulation.*;
import view.modeling.ViewableAtomic;
import view.modeling.ViewableComponent;
import view.modeling.ViewableDigraph;
import view.simView.*;
import Component.structuredEntities.*;
import Component.Iso_Types.*;

public class manage_regulator_interface_impl_sim extends ViewableDigraph() {

  public manage_regulator_interface_impl_sim() {
    super("manage_regulator_interface");  --represents manage_regulator_interface.impl

    ViewableAtomic manage_status = new manage_status_impl_sim("manage_status");
    ViewableAtomic manage_display_temperature = new manage_display_temperature_impl_sim(
                                                    "manage_display_temperature");
    ViewableAtomic manage_interface_failure = new manage_interfaceFailure_desiredRange_impl_sim(
                                                    "manage_interface_failure");

    add(manage_status);
    add(manage_display_temperature);
    add(manage_interface_failure);

    addInport("lower_desired_temperature");
    addInport("upper_desired_temperature");
    addInport("current_temperature");
    addInport("regulator_mode");
    addOutport("regulator_status");
    addOutport("display_temperature");
    addOutport("unspecified_temp");
    addOutport("regulator_interface_failure");
    addOutport("lower_desired_temp");
    addOutport("upper_desired_temp");
    addTestInput("regulator_mode", new stringEnt("NORMAL"));
    addTestInput("regulator_mode", new stringEnt("INIT"));
    addTestInput("regulator_mode", new stringEnt("FAILED"));
    addTestInput("current_temperature", new current_temperature(102, "Valid"));
    addTestInput("lower_desired_temperature", new lower_desired_temperature(97, "Valid"));
    addTestInput("lower_desired_temperature", new lower_desired_temperature(100, "Invalid"));
    addTestInput("upper_desired_temperature", new upper_desired_temperature(101, "Invalid"));
    addTestInput("upper_desired_temperature", new upper_desired_temperature(98, "Valid"));
    addTestInput("lower_desired_temp", new intEnt(98));
    addTestInput("regulator_interface_failure", new booleanEnt(true));

    addCoupling(this, "regulator_mode", manage_status, "regulator_mode");
    addCoupling(manage_status, "regulator_status", this, "regulator_status");
    addCoupling(this, "regulator_mode", manage_display_temperature, "regulator_mode");
    addCoupling(this, "current_temperature", manage_display_temperature, "current_temperature");
    addCoupling(manage_display_temperature, "display_temperature", this, "display_temperature");
    addCoupling(manage_display_temperature, "unspecified_temp", this, "unspecified_temp");
    addCoupling(this, "lower_desired_temperature", manage_interface_failure,
                                                  "lower_desired_temperature");
    addCoupling(this, "upper_desired_temperature", manage_interface_failure,
                                                  "upper_desired_temperature");
    addCoupling(manage_interface_failure, "lower_desired_temp", this, "lower_desired_temp");
    addCoupling(manage_interface_failure, "upper_desired_temp", this, "upper_desired_temp");
    addCoupling(manage_interface_failure, "regulator_interface_failure", this,
                                                  "regulator_interface_failure");
    addCoupling(manage_interface_failure, "unspecified_temp", this, "unspecified_temp");

  }

}
```

6. Simulation using DEVS-Suite

The Regulator Interface model developed in the AADL-DEVS framework can be simulated in the DEVS-Suite simulator. The UML class diagram for this simulation model is shown in Figure 11. The manage_status_impl_sim, manage_display_temperate_impl_sim, and manage_intefaceFailure_desiredRange_impl_sim clases are the generated atomic DEVS models. The componentized visualization of the manage regulator interface is depicted in Figure 12. This is a hierarchical model that has three atomic models. This model has input and output ports with external input and external output couplings. The three atomic models produce four output messages; these are compound DEVS-Suite data types. The output events shown in Figure 12 are the result of two simulation steps. The manage_regulator_interface_impl_sim is the generated coupled DEVS model.

6.1. Atomic Model Simulation

The DEVS-Suite simulator has two simulation protocols supporting the execution of atomic and coupled models. The atomic simulator protocol defines the order of executions of the external, internal, confluent, and output functions of any atomic model. For example, considering the manage_status atomic model, once an event is received on the regulator_mode input port, the external transition function retrieves and evaluates its value to set the mode of the regulator. For example, when the value of the received input event is "NORMAL", the regulator status is set to "Set_Vars" (see Listing 13). Since sigma for processing the external events is 0.0, the output event "on" is dispatched immediately on the "regulator_status" output port. After the dispatching of the output, the phase is set to "Chk_Mode" using the internal transition function. Each atomic model can be independently simulated using test input ports such as "addTestInput("regulator_model", new stringEnt("Normal")).

6.2. Coupled Model Simulation

The coupled simulator protocol is responsible for execution of all atomic and coupled models as well as all input/output communications (i.e., transmitting events amongst to and from every eligible atomic and coupled models). The coupled simulator delegates the execution of the atomic models to their respective independent simulators. In the first step of the simulation depicted in Figure 12, the test input ports are used to inject input events to the atomic models via the input ports of the manage_regulator_interface coupled model. All atomic model receive their input events concurrently and execute their external transition functions in parallel. In this step, each model's

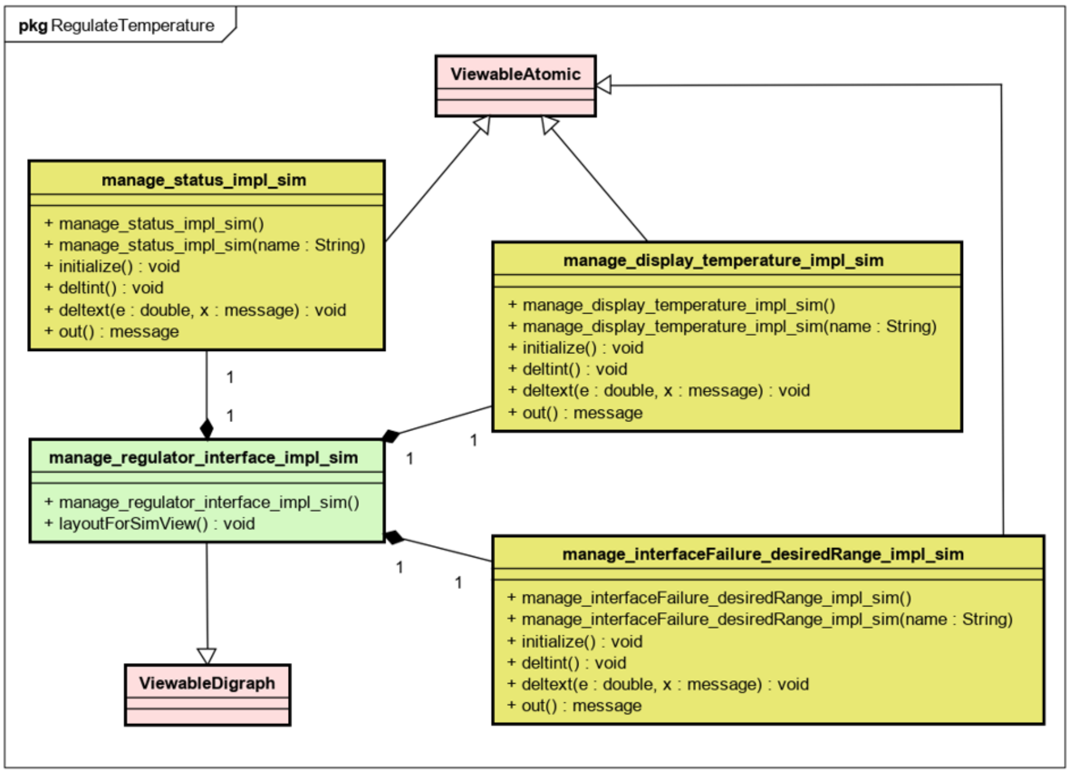
pkg RegulateTemperature
ViewableAtomic
manage_status_impl_sim
+ manage_status_impl_sim()
+ manage_status_impl_sim(name : String)
+ initialize() : void
+ deltint() : void
+ deltext(e : double, x : message) : void
+ out() : message
manage_display_temperature_impl_sim
+ manage_display_temperature_impl_sim()
+ manage_display_temperature_impl_sim(name : String)
+ initialize() : void
+ deltint() : void
+ deltext(e : double, x : message) : void
+ out() : message
manage_regulator_interface_impl_sim
+ manage_regulator_interface_impl_sim()
+ layoutForSimView() : void
manage_interfaceFailure_desiredRange_impl_sim
+ manage_interfaceFailure_desiredRange_impl_sim()
+ manage_interfaceFailure_desiredRange_impl_sim(name : String)
+ initialize() : void
+ deltint() : void
+ deltext(e : double, x : message) : void
+ out() : message
ViewableDigraph
1
1
1
1
1
1

Fig. 11. Manage Regulate Interface UML class diagram

input events are evaluated and processed (i.e., independently their states are updated and the time for their next internal events are set). In the second step, each model's output function followed by its internal transition function are executed in the order given. The output events are produced and then transmitted concurrently to the output ports of the manage_regulator_interface coupled model. For each model's internal transition function, the state and its time to next internal event are updated. Although the atomic models in the coupled manage_regulator_interface model are not coupled to one another, they can have feedback relationships to one another.

It is important to note that the couple model manage_regulate_interface in Figure 12 corresponds to the implementation classifier *manage_regulator_interface.impl* specified in Listing 8. The atomic models correspond the sub components, specified in the `subcomponents` section Listing 8, declared as the specialization of the respective implementation classifiers.

7. Related Work

The existing research that relate to this work can be viewed in three perspectives. First, extending the AADL architectural core language with the DEVS behavioral modeling language. Second, the code generation from AADL models for simulation and execution. Third, the development of an integrated framework where structural and behavioral designs are supported in as much as possible under a unified specification and execution framework.

In view of the first perspective, the annex mechanism in AADL is used, for example, to develop the BLESS and Hybrid (HA) annexes [18, 3]. The BLESS annex is created to specify behavior for component interfaces, define formal semantics for component implementations, and provide tool support for reasoning about the compliance of behaviors to component's specifications. It uses states and state transitions for specification and proving discrete behavior correctness of control systems. Hybrid annex was introduced for the continuous physical behavior and communication with discrete cyber systems. The BLESS annex, as compared with the Hybrid annex, is closely related to the proposed AADL-DEVS framework considering BLESS is for discretized behaviors of physical systems. The DEVS Annex (DA), in contrast to BLESS, is based on DEVS, a universal modeling language for discrete-event and discrete-time systems.

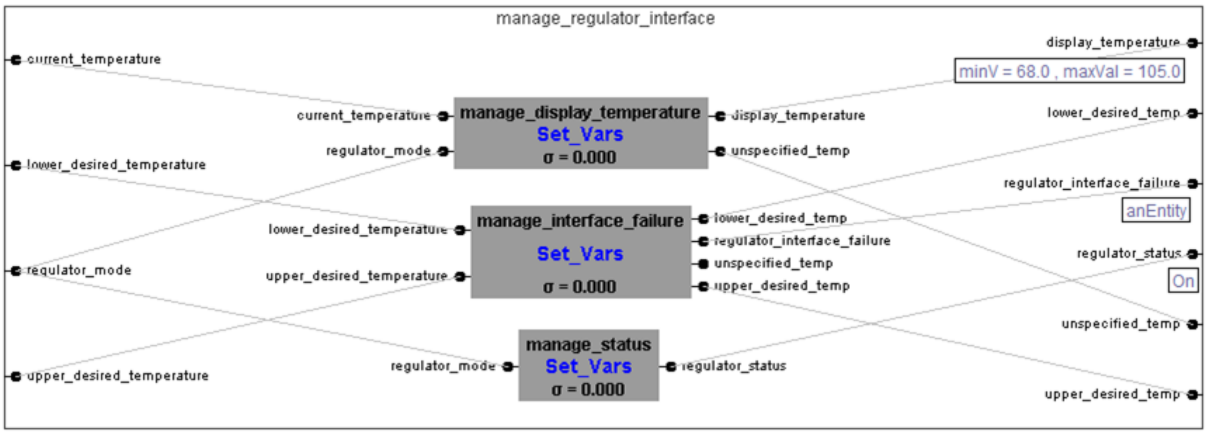

Fig. 12. Manage Regulate Interface Coupled DEVS model

The AADL facilitates code generation which is explored and used in different research projects. Code generation to support Revenscar Profile restriction on architecture models with OCARINA tool-suite is explored in [20]. Generated C and Ada code is then used for schedualabiliy and safety analyses. In [36], system-level co-simulation of integrated systems with Polychrony is discussed. The synchronous sub-set of AADL with Simulink is exploited for functional behavior modeling while the system-level architecture is modeled using AADL. Automatic C language code generation without human intervention is discussed in [41]. Respective code is automatically generated for the AADL process, thread, subprogram, and data components. To facilitate multi-platform execution, template-based code generation is exploited [13]. Rules are defined for code generation for different object platforms based on predefined templates. The code can be generated from AADL models based on model transformation with Model-driven Architecture(MDA) techniques for RTOS [6].

In contrast to these approaches, code generation in the AADL-DEVS framework is targeted for the DEVS-Suite simulator. Java code is generated from the combined AADL and DA specifications. Required data and model classes are generated for data types (specified using data components) and software components, respectively. Structural code is generated using the interface specification in the type classifier, while the behavioral code is generated for different sections of the DA subclause.

From the third perspective, early works have utilized meta-programmable and model-integrated computing for virtual systems-of-systems evaluations [33]. DEVS simulation is also used with AADL for verification of TT-Ethernet modeled using AADL [25]. Considering the DEVS modeling approach, it is advocated to be used for aviation system simulation by combining it AADL and the Aviation Scenario Definition Language (ASDL) [16]. The DEVSML [22], EMF-DEVS [27], and DEVS Natural Language (DNL) [38], among others, have been developed for the Parallel DEVS formalism. The DEVSML and DNL use Xtext which supports the Extended BNF grammar within the Eclipse Modeling Framework (EMF). The EMF-DEVS uses the Ecore language also supported by EMF. Other research include automation for translating Conceptual DEVS Models to code for target simulators [8]. Compared to these approaches, the proposed DA grammar is grounded in both the AADL and DEVS modeling languages. Furthermore, the time constraints defined for the AADL-compliant model can be used to extend the DA grammar to support Action-Level, Real-Time DEVS (ALRT-DEVS) [26, 12] modeling formalism. Such extension to the DA can support simulation under real-time constraints defined in the model and enforced by the host the DEVS-Suite simulator's host computing platform. A first prototype of the DA supporting logical-time simulation using the DEVS-Suite simulator is developed as a plug-in for OSATE. The DA provides a basis for combined static and dynamic design, verification, and validation.

8. Conclusion and Future Work

Cyber-Physical Systems (CPS) are pervasive in our daily life. Due to the complexity and high-integration, the development of such systems is challenging. To cope with this challenge, we have presented an AADL-DEVS framework for modeling and simulation of time-critical systems. The data, structure, and behavior of the computational part of embedded or CPS can be specified with the Architecture Analysis & Design Language (AADL) and DA. The core AADL is extended with DEVS Annex for detailed behavior modeling. Such models can be executed using the DEVS-Suite simulator. AADL to DEVS Code Generation Engine (ADCoDE) is developed to transform composite models with DA specifications to DEVS-Suite code.

Short-term future work includes using the AADL-DEVS framework for modeling and simulation of more complex systems like Transactive energy. Long-term future work includes extending the DA to support ALRT-DEVS modeling and simulation. The integration of DEVS and Hybrid annexes for continuous modeling is also an interesting future work. Another closely related future research is to support DA behavior modeling with the UML Statecharts and Activity specification.

A DEVS Annex Syntax Card

Grammar rules follow AS5506A[5] with literal symbols written in

A1. Lexical Elements

```
base ::= digit [ digit ]

based_integer_literal ::= base # based_numeral #


based_numeral ::= extended_digit [underline] extended_digit

character ::= graphic_character | format_effector
   | other_control_character

comment ::= - {non_end_of_line_character}*

decimal_integer_literal ::= numeral

decimal_real_literal ::= numeral .  numeral [ exponent ]

exponent ::= E [+] numeral | E - numeral

extended_digit ::=
digit | A | B | C | D | E | F | a | b | c | d | e | f

graphic_character ::= identifier_letter | digit | space_character
    | special_character

integer_literal ::=
   decimal_integer_literal | based_integer_literal

identifier ::= identifier_letter {[underline] letter_or_digit}*

letter_or_digit ::= identifier_letter | digit

numeral ::= digit {[underline] digit}*

numeric_literal ::= integer_literal | real_literal

real_literal ::= decimal_real_literal

string_element ::= \` | \' | non_string_bracket_graphic_character

string_literal ::= ` {string_element}* '
```

[5]SAE Standard AS5506A, Section 1.5 *Method of Description and Syntax Notation*

A2. Grammar Productions

```
devs_annex ::=
   [ variables { variable_declaration }+ ]
   [ states   state_declaration ]
   [ behavior  atomic_behavior_declaration ]


variable_declaration ::=
  variable_identifier :
  (variable_type_identifier | data_component_classifier_reference) =>
  value_declaration ;

value_declaration ::=
  simple_value | compound_value

simple_value ::=
  INTEGER_LIT | REAL_LIT | (true | false) | STRING

compound_value ::=
  { ( simple_value , simple_value ) }+


state_declaration ::=
  state_identifier :  [ initial ]
  ( REAL_LIT | INFINITY | variable_identifier ) ;


atomic_behavior_declaration ::=
  deltext external_transition_declaration |
  deltint internal_transition_declaration |
  outfn outfn_declaration |
  intest intest_declaration


external_transition_declaration ::=
  deltext [ source_state_identifier , message ]->
  destination_state_identifier behavior_action

message::=
  port_identifier (?  | !)  variable_identifier ;

behavior_action ::=
  { STRING }


internal_transition_declaration ::=
  deltint [ source_state_identifier ]->
  destination_state_identifier behavior_action ;


outfn_declaration ::=
  outfn [ source_state_identifier [ , conditional_expression ] ]-> message
  behavior_action ;

conditional_expression ::=
  boolean_term [ and boolean_term [ and boolean_term ] |
  or boolean_term [ or boolean_term ] ]

boolean_term::=
  [ not ] [ variable_identifier |   [ boolean_expression ] | relation ]
```

```
boolean_expression ::=
    boolean_term
   | boolean_term { and boolean_term }*
   | boolean_term { or boolean_term }*
   | boolean_term { xor boolean_term }*
   | relation

boolean_term ::=
   [ not ] ( true | false | ( boolean_expression )
   | relation )

relation::=
   ( numeric_expression ( relational_symbol ) numeric_expression )


intest_declaration ::=
  intest [ port_identifier , value_declaration ] ;


numeric_expression ::=
    numeric_term | numeric_term - numeric_term
   | numeric_term / numeric_term
   | numeric_term mod numeric_term
   | numeric_term { + numeric_term }+
   | numeric_term { * numeric_term }+
   | numeric_term ^ numeric_literal


numeric_term::=
    [ - ] ( numeric_literal | variable_identifier |
    (numeric_expression) )

numeric_literal ::= integer_literal | real_literal

relation_symbol ::= = | <> | > | < | <= | >=
```

References

[1] ACIMS, *DEVS-Suite simulator, version 5.0.0*, https://acims.asu.edu/software/devs-suite, 2017.

[2] E. M. Ahmad and H. S. Sarjoughian, *A behavior annex for AADL using the DEVS formalism*, 2019 Spring Simulation Conference (SpringSim), April 2019, pp. 1–12.

[3] Ehsan Ahmad, Brian R. Larson, Stephen C. Barrett, Naijun Zhan, and Yunwei Dong, *Hybrid annex: An AADL extension for continuous behavior and cyber-physical interaction modeling*, Ada Lett. **34** (2014), no. 3, 29–38.

[4] Rajeev Alur, *Principles of cyber-physical systems*, MIT Press, 2015.

[5] Dominique Blouin and Skander Turki, *AADL requirements annex (draft, progress update)*, Tech. report, 2009.

[6] M. Brun, J. Delatour, and Y. Trinquet, *Code generation from AADL to a real-time operating system: An experimentation feedback on the use of model transformation*, 13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008), March 2008, pp. 257–262.

[7] Yu Chen and Hessam S. Sarjoughian, *A component-based simulator for MIPS32 processors*, Simulation **86** (2010), no. 5-6, 271–290.

[8] Maximiliano Cristiá, Diego A. Hollmann, and Claudia S. Frydman, *A multi-target compiler for CML-DEVS*, Simulation **95** (2019), no. 1.

[9] Peter Feiler and David Gluch, *Model-based engineering with AADL: An introduction to the SAE architecture analysis & design language*, Addison-Wesley, 2012.

[10] Peter Feiler, Jörgen Hansson, Dionisio de Niz, and Lutz Wrage, *System architecture virtual integration: An industrial case study*, Tech. Report CMU/SEI-2009-TR-017, SEI, CMU, 2009.

[11] Andrew E. Ferayorni and Hessam S. Sarjoughian, *Domain driven simulation modeling for software design*, Proceedings of the 2007 Summer Computer Simulation Conference (San Diego, CA, USA), SCSC '07, Society for Computer Simulation International, 2007, pp. 297–304.

[12] Soroosh Gholami and Hessam S. Sarjoughian, *Action-level real-time network-on-chip modeling*, Simulation Modelling Practice and Theory **77** (2017), 272–291.

[13] Kai Hu, Zhangbo Duan, Jiye Wang, Lingchao Gao, and Lihong Shang, *Template-based AADL automatic code generation*, Front. Comput. Sci. **13** (2019), no. 4, 698–714.

[14] SAE International, *Architecture analysis & design language (AADL) annex volume 2: Annex d:behavior model annex*, (2011).

[15] ———, *SAE AS5506C, Architecture Analysis & Design Language (AADL)*, (2012).

[16] Shafagh Jafer, Bernard Zeigler, and Doohwan D. H. Kim, *A framework for rapid configuration of collaborative aviation system-of-systems simulations*, Modelling and Simulation for Autonomous Systems (Jan Mazal, ed.), 2018, pp. 92–105.

[17] Sungung Kim, Hessam S. Sarjoughian, and Vignesh Elamvazhuthi, *DEVS-suite: A simulator supporting visual experimentation design and behavior monitoring*, Proceedings of the 2009 Spring Simulation Multiconference (San Diego, CA, USA), SpringSim '09, Society for Computer Simulation International, 2009, pp. 161:1–161:7.

[18] B. Larson, P. Chalin, and J. Hatcliff, *BLESS: Formal specification and verification of behaviors for embedded systems with software*, NASA Formal Methods, LNCS, vol. 7871, Springer, 2013, pp. 276–290.

[19] Brian R. Larson, John Hatcliff, Kim Fowler, and Julian Delange, *Illustrating the AADL error modeling annex (v.2) using a simple safety-critical medical device*, Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology, HILT '13, ACM, 2013, pp. 65–84.

[20] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérome Hugues, *Ocarina : An environment for AADL models analysis and automatic code generation for high integrity applications*, Reliable Software Technologies – Ada-Europe 2009 (Berlin, Heidelberg) (Fabrice Kordon and Yvon Kermarrec, eds.), Springer Berlin Heidelberg, 2009, pp. 237–250.

[21] David L. Lempia and Steven P. Miller, *Requirement engineering management handbook*, Tech. Report DOT/FAA/AR-08/32, Federal Aviation Administration, 2009.

[22] Saurabh Mittal and Scott A. Douglass, *DEVSML 2.0: the language and the stack*, 2012 Spring Simulation Multiconference, SpringSim '12, Orlando, FL, USA, March 26-29, 2012, Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium, 2012, p. 17.

[23] Mohammed A. Muqsith, Hessam S. Sarjoughian, Dazhi Huang, and Stephen S. Yau, *Simulating adaptive service-oriented software systems*, Simulation **87** (2011), no. 11, 915–931.

[24] OSATE, *Open Source AADL Tool Environment, version 2.3.4*, http://osate.org/, 2018.

[25] Tiyam Robati, Amine El Kouhen, Abdelouahed Gherbi, and John Mullins, *Simulation-Based Verification of Avionic Systems Deployed on IMA Architectures*, MoDELS'15, 2015.

[26] Hessam S. Sarjoughian and Soroosh Gholami, *Action-level real-time DEVS modeling and simulation*, Simulation **91** (2015), no. 10, 869–887.

[27] Hessam S. Sarjoughian and Abbas Mahmoodi Markid, *EMF-DEVS modeling*, 2012 Spring Simulation Multiconference, SpringSim '12, Orlando, FL, USA, March 26-29, 2012, Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium, 2012, p. 19.

[28] Hessam S. Sarjoughian and Randy K. Singh, *Building simulation modeling environments using systems theory and software architecture principles*, Proceedings of the 2004 Advanced Simulation Technology Symposium, SCSC '07, 2007, pp. 297–304.

[29] Hessam S. Sarjoughian and Savitha Sundaramoorthi, *Superdense time trajectories for DEVS simulation models*, Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, part of the 2015 Spring Simulation Multiconference, SpringSim '15, Alexandria, VA, USA, April 12-15, 2015, 2015, pp. 249–256.

[30] Hessam S. Sarjoughian and Bernard P. Zeigler, *Devsjava: Basis for a DEVS-based collaborative M&S environment*, SCS International Conference on Web-Based Modeling and Simulation, 1998, pp. 29–36.

[31] Software Engineering Institute, Carnegie Mellon University, *Osate2*, `https://wiki.sei.cmu.edu/aadl/index.php/Osate_2`, Jan 2012.

[32] Janos Sztipanovits, Ted Bapty, Xenofon D. Koutsoukos, Zsolt Lattmann, Sandeep Neema, and Ethan K. Jackson, *Model and tool integration platforms for cyber-physical system design*, Proceedings of the IEEE **106** (2018), no. 9, 1501–1526.

[33] Janos Sztipanovits and Gabor Karsai, *Model-integrated computing*, IEEE Computer **30** (1997), no. 4, 110–111.

[34] Steve Vestal, *MetaH*, SIGSOFT Softw. Eng. Notes **25** (2000), no. 1, 105–.

[35] A. Wayne Wymore, *Model-based systems engineering*, CRC Press, 1993.

[36] Huafeng Yu, Yue Ma, Yann Glouche, Jean-Pierre Talpin, Loïc Besnard, Thierry Gautier, Paul Le Guernic, Andres Toom, and Odile Laurent, *System-level co-simulation of integrated avionics using polychrony*, Proceedings of the 2011 ACM Symposium on Applied Computing (New York, NY, USA), SAC '11, Association for Computing Machinery, 2011, p. 354–359.

[37] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim, *Theory of modeling and simulation*, 2nd ed., Academic Press, Inc., Orlando, FL, USA, 2000.

[38] Bernard P. Zeigler and Hessam S. Sarjoughian, *Guide to modeling and simulation of systems of systems*, 2nd ed., Simulation Foundations, Methods and Applications, Springer, 2017.

[39] Bernard P. Zeigler, Hessam S. Sarjoughian, and Vincent Au, *Object-Oriented DEVS*, Proceedings of AeriSense, vol. 3083, SPIE, 1997, pp. 100–111.

[40] Chao Zhang and Hessam S. Sarjoughian, *Cellular automata DEVS: A modeling, simulation, and visualization environment*, Proceedings of the 10th EAI International Conference on Simulation Tools and Techniques, SIMUTOOLS 2017, Hong Kong, China, September 11-13, 2017, 2017, pp. 11–19.

[41] Chen Zhang, Xinyi Niu, and Bin Yu, *A method of automatic code generation based on AADL model*, Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence (New York, NY, USA), CSAI '18, ACM, 2018, pp. 180–184.