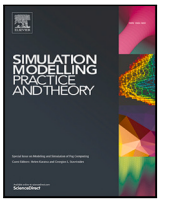
Simulation Modelling Practice and Theory 123 (2023) 102690

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat



An Environment for Developing Simulatable AADL-DEVS Models

Ehsan Ahmad [a,*], Hessam S. Sarjoughian [b]

[a] College of Computing and Informatics, Saudi Electronic University, Riyadh, 11673, Saudi Arabia
[b] Arizona Center for Integrative Modeling and Simulation, School of Computing and Augmented Intelligence, Arizona State University, Tempe, 85281, AZ, United States

ARTICLE INFO

Keywords:
AADL
Behavior modeling
Code generation
DEVS annex
DEVS-Suite
OSATE
Simulation
Structure modeling

ABSTRACT

Reducing complexity in system architecture and design specifications, and more specifically from the software aspect, is essential. The architecture specifications focus on what the requirements and static aspects of systems are. The design specifications define dynamic aspects of computational components that sense and react to external stimuli. As the architecture and design specifications serve complementary roles in developing systems, the use of Architecture Analysis & Design Language (AADL) and Discrete Event System Specification (DEVS) is proposed for developing, in a step-wise fashion, combined architecture and design models. The proposed AADL-DEVS framework is grounded in the foundational modularity and hierarchy principles common to the AADL and DEVS modeling approaches. A realization of this framework capable of transforming and simulating the AADL-DEVS specifications is developed using the Open Source AADL Tool Environment (OSATE) and DEVS-Suite simulator. The scope of this paper is on the computational aspect of systems. The proposed AADL-DEVS framework is demonstrated using a model for the software part of an infant incubator, a time-sensitive and safety–critical system.

1. Introduction

Building Cyber–Physical Systems (CPS) or, more broadly, Systems-of-Systems (SoS) remains challenging. Many kinds of modeling approaches have been proposed for the development of integrated computational and physical systems [1–5]. For these systems, the distinct models for architecture and design serve complementary purposes. Therefore, a key challenge is to develop well-formed architecture and design artifacts individually and collectively. Models can be developed and simulated for mixed actual-simulated CPS (*e.g.,* [6]). Simulation can also assist in architecture evaluation for alternative selections. Employing models using different modeling languages and supported with methodologies, frameworks, and standards becomes more useful as the complexity of systems increases.

The development of simulation models can be undertaken by specifying architectures and extending them with design specifications. Separate and integrated modeling is supported at the architecture and design abstraction levels, a key benefit for developing simulation models as close as possible to those needed for building actual systems. The use of modeling and simulation aids in reducing design complexity using the architecture purposed for higher-level specifications with an emphasis on structures and relationships while the design is purposed for lower-level specifications with an emphasis on behaviors and executions. The relationship between the architecture and design abstraction levels should be concisely and explicitly formulated. These observations highlight the importance of addressing system structure and behavior complexity and scalability traits using integrated architecture and design modeling [7].

* Corresponding author.
E-mail addresses: e.ahmad@seu.edu.sa (E. Ahmad), sarjoughian@asu.edu (H.S. Sarjoughian).

https://doi.org/10.1016/j.simpat.2022.102690
Received 22 June 2022; Received in revised form 15 November 2022; Accepted 18 November 2022
Available online 23 November 2022
1569-190X/© 2022 Published by Elsevier B.V.

The Architecture Analysis & Design Language (AADL) [8] supports software/hardware architecture specifications for building actual systems. For design, the Discrete Event System Specification (DEVS) [9] supports specifying modular, hierarchical simulation models. These approaches serve as the means for a methodology to rigorously model combined architecture and design specifications and generation of simulatable models. The above considerations enable the development of the proposed AADL-DEVS framework where the AADL and DEVS serve to specify the structure and behavior of systems, respectively.

Frameworks supported with robust tools are needed to develop architecture and design specifications. Examples are the Open Source AADL Tool Environment (OSATE) [10] and the DEVS-Suite simulator [11]. The OSATE tool supports AADL requirements, latency, and safety analyses. The DEVS-Suite supports modeling, simulation, experimentation, and evaluation of discrete-event dynamical systems. AADL-DEVS framework is realized through OSATE-DEVS-Suite tool. An infant incubator [12], a time-sensitive and safety–critical system, as an exemplar, shows an implementation of the novel algorithms for generating parallel, hierarchical DEVS simulatable models from AADL-DEVS hierarchical model specifications under the OSATE-DEVS-Suite tool.

1.1. Contributions

This study presents a methodology for developing simulatable code using the AADL and DEVS modeling languages. A DEVS Annex (DA) adds behavior to the AADL's core static structure. Following a Model-Based Engineering (MBE) approach, the systematic combination of the AADL and DA supports generating code for the DEVS-Suite simulator. The contributions of this paper can be summarized as follows.

First, the proposed AADL-DEVS framework is developed for combined structure and behavior modeling of time-constrained safety–critical systems. The realization of this framework using OSATE and DEVS-Suite enables disciplined combined modeling of architecture and design specifications to achieve simulations as close as desired to actual implementation. To our knowledge, the DEVS modeling formalism for *behavior specification* has not been explored and included in AADL. In a related work, the DEVS meta-modeling is used to extend AADL to generate code for *model structure* but without behavior and code generation for model behavior [13]. Compared to this work, the AADL-DEVS framework promotes detailed behavior modeling with code generation through extending the AADL core language. The generated code for the DEVS-Suite simulator "imports" the AADL hierarchical models, I/O ports with their assigned data, and state information.

Second, the AADL to DEVS CoDe generation Engine (ADCoDE) is developed as a plugin for OSATE. Three novel algorithms are introduced and used to develop a translator to generate partial code for the DEVS-Suite simulator. The ADCoDE follows a constructive approach for generating simulatable AADL-DEVS model. Thus, the resultant executable code by the simulator is traceable to the AADL-DEVS model. These algorithms are capable of code generation for hierarchical AADL-DEVS models.

Third, a demonstrative example for the AADL-DEVS modeling and code generation is developed for an infant incubator. The specification is simple yet rich enough to show the need and usefulness of AADL-DEVS framework for modeling and simulation of complex time-sensitive and safety–critical systems.

1.2. Outline

Grounded in the AADL and DEVS modeling approaches, the rest of this paper details the proposed and developed AADL-DEVS framework. Section 2 introduces DEVS and the DEVS-Suite simulator as well as AADL with OSATE. Section 3 presents a detailed description of the proposed AADL-DEVS framework with descriptions of structure and data modeling using the core AADL and behavior modeling and DEVS Annex (DA). In Section 4, an AADL to DEVS CoDe generation Engine (ADCoDE) is described for automated simulation code for the DEVS-Suite simulator. In Section 5, *Isolette* system (*i.e.,* an infant incubator) is presented as an exemplar to demonstrate the use of the proposed AADL-DEVS framework for time-critical and safety–critical systems. Section 6 highlights the use of the DEVS-Suite for simulating the generated code for a sub-system of the Isolette system. Section 7 presents some key related works. Section 8 presents a summary of this research.

2. Background

This section presents an overview of the parallel DEVS formalism and the DEVS-Suite simulator. The emphasis is on the atomic and coupled model specification constructs and their realization and execution in the simulator. Similarly, the basics of the AADL and OSATE for modeling software component structures are presented as well.

2.1. System-theoretic discrete-event simulation

A variety of systems can be modeled using the parallel Discrete Event System Simulation (DEVS) formalism [9]. As a mathematical language, it lends itself for specifying structures and behaviors of Systems-of-Systems (SoS) including Cyber–Physical Systems (CPS). This modeling approach is based on Systems Theory [14] where a system is defined in terms of hierarchical parts that are composed through their inputs and outputs. Models can send/receive arbitrary data objects to/from each other at any arbitrary time instances. DEVS can be used to describe any discrete time and discrete event systems [9]. In DEVS, there are two types of models: *atomic* and *coupled* components. The abstract simulator protocols (*i.e.,* operational semantics) for atomic and coupled models are excluded for brevity [9].

2.1.1. Atomic DEVS model

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

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

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 input/output 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. The $ta(s)$ presents time allowed to be in a particular state s. The $ta(s) = 0$ specifies instantaneous state change. A model can have autonomous and reactive behaviors in response to any number of input messages 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* specifies timings of state transitions. The *confluent function* specifies the handling of simultaneous internal and external events. An atomic model can have multiple input and/or output ports and messages (objects). The elapsed time e allows external inputs to arrive at arbitrary future time instance. The atomic models can be simulated according to an abstract simulator protocol [9].

2.1.2. Coupled DEVS model

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

$$CM = \langle X^b, Y^b, D, M_d | d \in D, EIC, EOC, IC \rangle \text{ where} \quad (2)$$

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 specifications as the atomic model. The structure specification of a coupled model includes input and output ports, a set of named components, and couplings. DEVS can ensure semantically identical input/output interfaces for atomic and coupled models. The couplings amongst a coupled model and its atomic or coupled models are (i) the *external input couplings* (`EIC`) - the couplings of the coupled component input ports to the input ports of one or more of its components, (ii) the *external output couplings* (`EOC`) - the couplings of the output ports of the components to the output ports of their coupled component, and (iii) the *internal couplings* (`IC`) - the couplings of the output ports to the input ports of the components of the coupled component. 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). The coupling allows message passing between components. DEVS has the property of closure 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. The atomic and coupled models are executed according to atomic and coupled simulator protocols, respectively [9].

2.1.3. DEVS-suite simulator

The DEVS-Suite simulator (https://sourceforge.net/projects/devs-suitesim/) is an open-source and free tool for the parallel DEVS formalism [11,15,16]. It is one of the most commonly used simulators for developing and simulating parallel DEVS models . The modeling part supports implementing DEVS atomic and coupled models that have hierarchical tree-structures. The $ViewableAtomic$ and $ViewableDigraph$ Java classes allow visualization of the simulatable atomic and coupled components, respectively.

Both atomic and coupled models can receive input and send output Java entities (messages) 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 behavior of every atomic model as well as the transmissions of the messages in every coupled model.

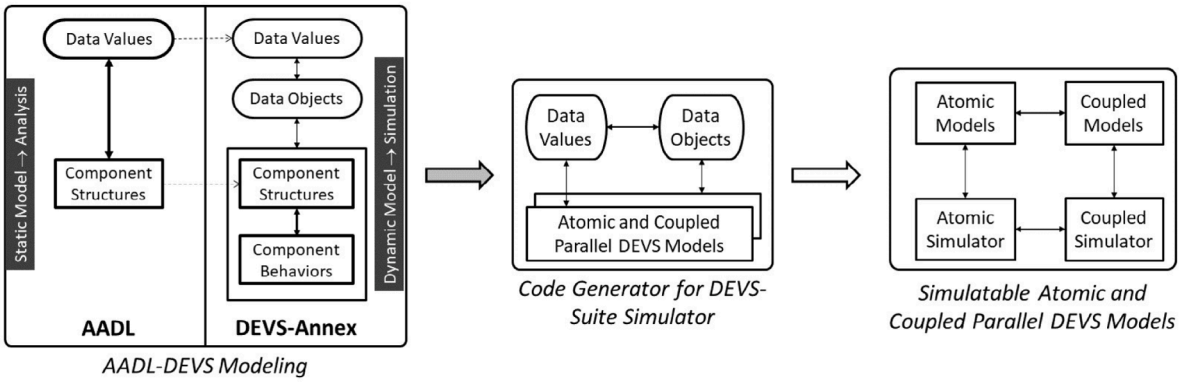


Fig. 1. An illustration of the AADL-DEVS modeling supported with code generation and simulation.

2.2. Architecture analysis & design language

Architecture Analysis & Design Language (AADL) [8] is an SAE International standard language for the architectural description of embedded systems [17]. It is a Model-based Engineering (MBE) approach which promotes *analyzable* architecture development and dependability prediction. Architectural modeling is realized through component specification of the *application software* and the *execution platform*. Component *Type* and *Implementation* classifiers, corresponding to system entities are instantiated and connected together to form the system architecture model. More details are provided in Section 3.3.

The application software may contain *process*, *data*, *subprogram*, *thread*, and *thread group* components [18]. The process component represents a protected memory space shared among thread subcomponents. A data component represents a data type with its characteristics, a subprogram represents a block of executable code. A thread represents an entity that executes a sequential instruction flow.

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 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.

2.2.1. Open source AADL tool environment

To support AADL specifications, the Open Source AADL Tool Environment (OSATE) [10] is developed using the Eclipse plugin framework. It supports textual, graphical, XML Metadata Interchange (XMI) specification formats, and a set of analysis plugins. This tool supports modeling and analysis of real-time embedded systems. In order to support extensive and focused model analysis, the AADL core language support extensions through annexes and properties.

3. AADL-DEVS framework

This framework is developed using the AADL, a semi-formal architecture description language, and DEVS, a discrete-event modeling formalism [19,20]. The result supports a combined static analysis and dynamic behavior. In this framework, the AADL is used to define structure specification, whereas the DEVS is used to define behavior specification. The nodes in AADL can be hierarchically combined using connections and bindings to define combined software and hardware specifications. DEVS lends itself to specify reactive hierarchical components, each having input and output structure with encapsulated behavior. It should be noted that the AADL structural specification is more elaborate compared to DEVS. The AADL input and output ports are typed and may be used to combine software and hardware components. In DEVS, compared to AADL, the input and output data can be arbitrary messages and can be paired with input and output ports. Such differences are accounted for in the AADL-DEVS framework and its realization, the OSATE-DEVS-Suite tool. Details are provided in Sections 3.2 and 5.1.

The AADL-DEVS framework follows a three-step process depicted in Fig. 1. In the first step, the AADL-DEVS model can be developed using the AADL and the DEVS Annex (DA). In the second step, the AADL-DEVS models are converted to partial code for the DEVS-Suite simulator. In the third step, the generated partial code should be completed to support simulations. The AADL-DEVS model shows that the component structures are specified according to AADL and then mapped to component structures according to DEVS. The behaviors for the DEVS models are defined according to the DA, DEVS behavior specification following the AADL language. In the AADL-DEVS framework illustration, the dotted arrows show mapping relationships (*e.g.,* AADL component structures are mapped to atomic and coupled DEVS model structures). The solid arrows show the elements that need to be used together

(*e.g.,* Data Values and Data Objects). The block arrows show the development flow from specifying AADL component structure to simulatable parallel DEVS models.

The AADL-DEVS modeling framework in the above process has two parts – (i) the AADL language and (ii) the DA sublanguage. The OSATE is extended with the DA to support the DEVS-Suite simulator. First, the structural syntax of atomic and coupled DEVS models are specified in OSATE. Second, the behavioral syntax of the atomic DEVS model is added using OSATE. The DA structure specification accounts for the syntax of OSATE and DEVS-Suite primitive and compound data types. Furthermore, the data types in DEVS-Suite, unlike their counterparts in OSATE, have behaviors. In addition, the DA sublanguage supports defining the internal and external functions of atomic models as time-based state machines. Other functions that are supported in the DA sublanguage are time advance and test input functions.

The code generator in the above process transforms AADL-DEVS models into parallel atomic and coupled DEVS-Suite implementations. In order to generate code for the simulator, the input and output primitive and compound data types in OSATE are mapped to their counterparts in the DEVS-Suite simulator. This requires two steps (see Section 4.1). A set of generic I/O data types in the mold of the AADL I/O data types are defined in the DA. The code generation engine is extended to transform the DA I/O data defined in OSATE to their counterparts defined in the DEVS-Suite simulator. The generated code may be manually extended as needed.

The AADL Behavior Annex (BA) provides abstractions for generic and composite component types and implementation [8]. The dynamics for these components are specified using modes (states) and mode (state) transitions. Alternative behaviors for a component can be defined in terms of a finite number of state transitions with non-zero duration for modes and events. The state transitions are encapsulated within components and may be specialized using inheritance. In DEVS, this kind of behavioral modeling is supported with key differences [21]. First, for each state change, the transition either strictly depends on the state or on input and state. The behavior of these models are defined in terms of a logical clock, not the passage of time controlled by target computing platform. Second, internal and external transitions can occur simultaneously. These state transitions are ordered where one consumes zero logical time. Concurrency in simulation is possible because time is a logical artifact. Third, DEVS has an abstract protocol that is solely responsible for time management and input/output communication needed for hierarchical models. As in BA, models can be inherited by other models using object-orientation [21]. Therefore, AADL with DA, compared with the AADL with BA, directly lends itself to developing models and simulating them. The rest of this section describes the structure and data modeling using AADL with the DA sublanguage.

3.1. Component structure modeling

In AADL, the structure of a system is defined as 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*. Features are communication ports for *Data*, *Event*, and *Event Data* for transmitting data, control, and control and data, respectively. Port communication is typed and directional. For example, an *in* port receives data/control data and an *out* port sends data/control data while an *in out* port can send and receive the 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. Software components are mapped onto execution platform components, *e.g.,* 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 a type classifier of a component.

AADL with annexes can provide support for both static and dynamic architectural modeling [22]. 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 multiple architecture models. Dynamic architectures are realized through modal behavior of the system. Models contain component and connection configuration for different operational as well as error modes [23].

3.2. Data modeling

AADL provides a Data Modeling Annex with pre-defined data types [24]. This annex has primitive data types that can be used as they are and for defining compound data types. The primitive data types include real, integer, string, and Boolean, among many others. A compound data type has multiple elements, each of which can be either of primitive or compound data types. The compound data types include Integer_Range and Real_Range. The DA code generation engine supports automatic conversion of the AADL primitive data types to primitive counterparts in the Java programming language. When a primitive data type is to be used as a message for transmission via any DEVS-Suite atomic or coupled model's input and output ports, the data type must be converted to a subclass of the entity class [11]. In the DEVS-Suite simulator, all input and output (primitive or compound) are referred to as entities. The code generation engine provides the IntEnt, DoubleEnt, and StringEnt classes corresponding to the AADL integer, real, and string data types. A Java class must exist or be constructed for any AADL compound data type, as in primitive data types, to be used as input or output messages. The Integer_Range (used to represent a finite range of integer values) and Real_Range (used to represent a finite range of real values) compound data types are provided as part of the code generation [20]. These data types are commonly needed for modeling safety–critical systems.

3.3. Behavior modeling using DEVS annex

For AADL to support discrete-event modeling and simulation, its language is extended with the DEVS Annex (DA). An earlier version of the DA was introduced in [19] with detailed syntax, and grammar with appropriate examples for each of the four sections `entities`, `variables`, `states`, and `behavior` dedicated to specify different aspects of a detailed behavior model. Based on the knowledge acquired during the development of the code generator, the `entities` section is removed as they are not used for simulation. In a DA subclause, behavior specification starts with the `variables` section, followed by the `states` section, which is then followed by `behavior` section. Declaration of these sections in DA are independent of one another (*i.e.,* for the DEVS-Suite simulator, the order in which the variables, states, and behaviors are declared is immaterial).

Considering a system is a hierarchical composition of modular components, DA enables AADL to model detailed behaviors at different abstraction levels (component and system levels) based on the concepts of the *Atomic DEVS* and the *Coupled DEVS*. Software and execution platform components can be annotated with DA subclauses to model the discrete behavior. Implemented as a plugin for the Open Source Architecture Tool Environment (OSATE), the DA is perfectly aligned with the semantics of the AADL core language to model desired monitored and controlled variables by specifying data types for ports and communicated through them. In the following, the sections `variables`, `states`, and `behavior` are briefly explained. The Extended Backus–Naur Form (EBNF) grammar with examples for the each of the DA section can be found in [19].

The `variables` section is used to declare local variables with their data types. Variable declaration in this section is supported by the classifier references (either defined in the same package, or within the scope of another package imported using `with` clause) to the appropriate AADL data components. Initial value specified after `=>` is mandatory for each variable and it can either be primitive (*i.e.,* integer, real, Boolean, or string literal) or compound consisting of more than one primitive values separated by commas and enclosed in parenthesis.

Section `states` contains definition of all the required and desired states of a particular component. Each state is defined with a name followed by the specification of the time to remain in a particular state before next transition (time advance function). The starting state is labeled as `initial` while the unlabeled states are considered as transient states with time advance set to 0.0. States with time advance function `INFINITY` indicate the final states. Starting from the initial state, the control suspends in transient states for different time advance functions and ends in a final state.

The `behavior` section contains discrete behavior specifications in terms of a state-transition system with three functions and one declaration. The confluent transition function is not defined as it is provided by default in the simulator. The external transition triggered by an input message (composed of the respective port name, the value received, and a variable to store) from a source state is modeled using the function `deltext`. It interrupts simulation to move forward to its destination state. The behavior of the external transition function is modeled as *behavior action* and depends on the current state, received input, and the time elapsed in the current state.

The function `deltint` is used to specify internal state transitions caused by the progression of the elapsed time (e). The control is moved to the next state when $e = ta$. The function `deltint` is specified with a source state identifier followed by the target state identifier and the behavior action.

The function `outfn` is used for generating output before the state change (*i.e.,* before executing the internal and external state transition functions). An output message is specified with the name of the port followed by `!` sign and the data to be transmitted. The output function can further be restricted by adding conditional expressions.

Although, not part of the DEVS formalism, stand-alone testing during simulation is modeled using declaration `intest`.[1] It is used for providing test inputs for model components during simulation.

4. Code generation for simulation

The partial code generation for the DEVS-Suite simulator is focused on transforming AADL models with DA specifications into their DEVS-Suite atomic and coupled counterparts. Although the structure of the DA sublanguage is based on the DEVS formalism, the DA specifications must be transformed to the code for the simulator. This requires generating appropriate DEVS-Suite code, as Java classes, from the AADL models. To ease behavior modeling, the DA seamlessly integrates with the AADL core language and does not require re-definition of the structural and interface elements defined in the type classifiers. The use of the AADL for structural and data modeling requires developing syntax translator from DA in OSATE to their counterparts in the DEVS-Suite simulator. To support code syntax translation, we have implemented ADCoDE — an AADL to DEVS CoDe generation Engine. This engine is implemented as a plugin for OSATE and can be activated for an implementation classifier of a component (annotated with DA specifications) in the OSATE Eclipse Outline view.

As depicted in Fig. 2, upon activation for a particular AADL component with the data, structure, and DA behavior specifications, code generation through ADCoDE is a three-step process. First, in the *data classes generation* step, appropriate Java classes (and objects of the primitive data types) are generated for the compound integer and double range data types (see section 3.3.1 in [20]). Second, 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. Third, in the *behavioral code generation*, behavioral code is generated based on the DA specifications in the

[1] The keyword `infn`, as used in an earlier version of the DEVS Annex introduced in [19], is now replaced with the `intest` to improve readability and understanding.

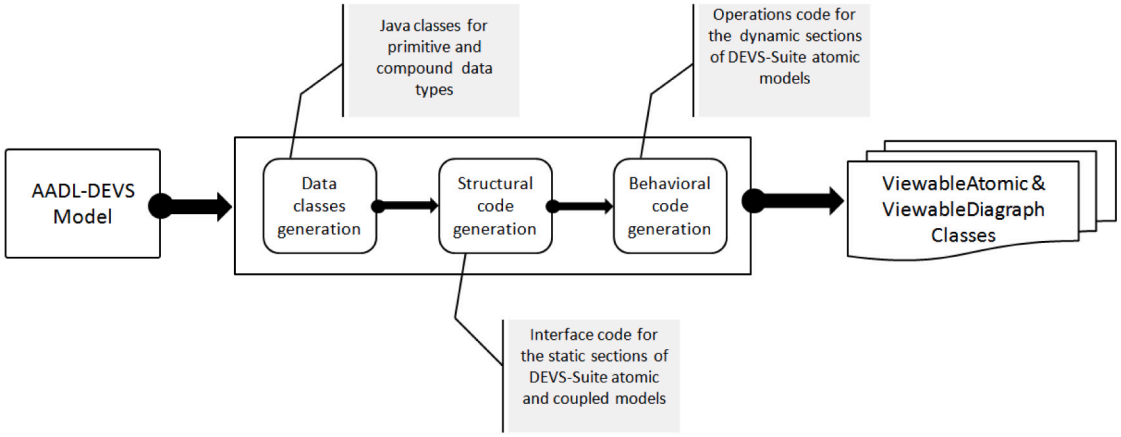


Fig. 2. AADL to DEVS CoDe Generation Engine (ADCoDE) workflow.

implementation classifier of a particular component. All the generated code segments are then combined according to the structure of the *ViewableAtomic* template provided in the simulator. Code generation for one AADL component results in one *ViewableAtomic* class. All the generated *ViewableAtomic* classes in an AADL package are included in the *Model* folder (package) while the generated data classes are organized in a separate folder named the same as the AADL package in which the data components are defined. Contents of all these folders are dynamically updated if any changes are made to the models.

ADCoDE also realizes code generation for coupled DEVS models. Upon activation for a composite AADL component with DA (for example a thread group), a model class extending the *ViewableDigraph* class is generated along with required data and model classes for each AADL subcomponent (for example thread components). This hierarchical organization of the generated simulation models improves their use and understanding.

In the following subsections, the above (i) data (`Algorithm 1`), (ii) structure (`Algorithm 2`), and (iii) behavior (`Algorithm 3`) code generation sequential steps are described in detail (see Fig. 2).

4.1. Data classes generation

AADL allows both primitive and compound data types for data modeling, hence the ADCoDE must also have the capability to map them with respect to the DEVS-Suite data types. Although the primitive data types that are pre-defined in the AADL Data Modeling Annex [24] and Base_Types, they must be mapped to their counterparts in the simulator. This requires extending data modeling in the simulator. Such an extension must be realized via inheriting from the entity class as the simulator only allows transmitting objects (messages sent and received) either entity or any of its subclasses. AADL to DEVS transformation requires the AADL data types representing input and output to be mapped to DEVS-Suite entity data types. The input and output entities are the events sent and received among atomic and coupled models. Predefined package Base_Types contains common data types as data component classifiers.

For the primitive data types, the Java entity class is extended with `stringEnt`, `booleanEnt`, `intEnt`, and `doubleEnt` data types for their AADL *string*, *Boolean*, *integer*, *float* counterparts, respectively. The core of the DEVS-Suite simulator has also been extended with the new classes `IntRange` and `DoubleRange` for the *Integer_Range*, and *Real_Range* data types, respectively. For each compound data type used in an AADL component's type or implementation classifier, a separate Java class is generated.

Algorithm 1 presents the main steps to generate data objects and classes. Here, *C.TYPE* and *C.IMPL* are the type and implementation classifiers, respectively, for an atomic AADL component *C*. Starting with the `variables` section of the DA specification, where *C.IMPL.VARS* is the set of variable declarations, for every variable *var* the data type of the variable *var.type* is extracted. If *var.type* is Base (string, Boolean, integer, float) or Range (Integer_Range, Real_Range) then the *var* is declared as an object of the appropriate extended classes for primitive and range data types.

In case of a compound data type (a user-defined data type with multiple elements of primitive and/or range types), a data class (if it has not been previously generated) is generated with the name same as *var.type*. Data type *e.type*, for every element *e*, is then extracted from a data component in the current project and the appropriate objects are declared in the same way as for *var.type*. Appropriate getters and setters are generated along with constructors for each class.

In an AADL model, data types can also be specified with ports to model the type of data to be sent or received. For every *port*, in the set of ports *C.TYPE.PORTS*, either data object or data class is generated in the same way as explained for the variable *var*.

Algorithm 1 Data objects and classes generation for AADL data types

Require: Data types specified with ports (in the type classifier *C.TYPE*) and variables (in the DEVS Annex specification in the implementation classifier *C.IMPL*) for an atomic AADL component *C*

Ensure: Generate appropriate objects and Java classes for data types

1: **for all** *var* ∈ *C.IMPL.VARS* **do**
2: **if** var.type class does not exist **then**
3: **if** *var.type* is in Base or Range **then**
4: declare *var* as an object of appropriate class from *stringEnt*, *booleanEnt*, *intEnt*, *doubleEnt*, *Integer_Range*,
5: *Real_Range*
6: **else**
7: generate Java class *var.type*
8: find data component *var.type* in the current Project
9: **for all** *e* ∈ *Elements* of *var.type* **do**
10: find *e.type* in the Property List of *e*
11: repeat steps 3 − 4 for *e.type*
12: create Getters and Setters for *e*
13: **end for**
14: create appropriate constructors for the Java class *var.type*
15: **end if**
16: **end if**
17: **end for**
18: **for all** *port* ∈ *C.TYPE.PORTS* **do**
19: find *port.type*
20: repeat steps 2 − 16 for *port.type*
21: **end for**

Algorithm 2 Structural code generation for atomic and composite AADL components

Require: Interface specification (in the type classifier *C.TYPE*) in case of an atomic AADL component *C* and subcomponents and connections specification (in the implementation classifier *C.IMPL*) in case of a composite AADL component *C*

Ensure: Generate *ViewableAtomic* Java class with structural code for each atomic (sub)component *C* and generate *ViewableDigraph* Java class for composite component *C* (if *C* is composite component)

1: **if** *C* is a composite component **then**
2: generate *ViewableDigraph* class for *C.IMPL* with required *import* and *package* info
3: declare *addInport* and *addOutport* methods to add *in* and *out* port for *C*
4: **for all** *subCom* ∈ *C.IMPL.SUBCOMPONENTS* **do**
5: **if** *subCom* is an atomic component **then**
6: do steps 17 − 24 for *subCom*
7: **else if** *subCom* is a composite component **then**
8: do steps 2 − 9 for *subCom*
9: **end if**
10: create instance of the *ViewableAtomic* class generated for *subCom* in *C*
11: generate *add* method with the instance from the previous step
12: **end for**
13: **for all** *conn* ∈ *C.IMPL.CONNECTIONS* **do**
14: generate *addCoupling* method for *conn* with *conn.source* and *conn.destination* ports for EIC, EOC, and IC
15: **end for**
16: **else if** *C* is an atomic component **then**
17: generate *ViewableAtomic* class for *C.IMPL* with required *import* and *package* info
18: **for all** *port* ∈ *C.TYPE.PORTS* **do**
19: **if** *port.in* **then**
20: generate *addInport* method for *port*
21: **else if** *port.out* **then**
22: generate *addOutport* method for *port*
23: **end if**
24: **end for**
25: **end if**

4.2. Structural code generation

The structure of an AADL component is composed of its type and implementation classifier. The type classifier contains the interfaces of a component while the implementation classifier models the internal realization.

Algorithm 2 specifies the main steps for structural code generation through ADCoDE. This algorithm uses the AADL model hierarchy starting from the highest levels to the lowest level to generated code for all atomic and coupled DEVS models. Here, *C.TYPE* and *C.IMPL* are the type and implementation classifiers, respectively, for an AADL (atomic or composite) component *C*.

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

Upon the activation for a composite AADL component (for example a thread group), ADCoDE generates code for DEVS models. Firstly, a model class extending the $ViewableDigraph$ class is generated for *C.IMPL*. Traversing through the subcomponents *C.IMPL.SUBCOMPONENTS*, separate $ViewableAtomic$ class is generated for each AADL atomic subcomponent (for example thread) by performing steps 17–24. For each composite subcomponent *subCom*, a $ViewableDigraph$ class is generated. The code generation iterates until no more subcomponents are left to be processed.

Instances of these classes are then created and added using the *add* method. The code for the input and output ports of the composite component are generated based on the input and output ports of all the subcomponents. The code generation relies on the AADL to ensure semantically identical input/output interfaces for atomic and coupled models.

Code generation for the required coupling categories (steps 13–14 of Algorithm 2) in any composite AADL component is based on its port connection specifications in the `connections` section. The external input coupling (EIC) is realized based on the connections of the composite component's input ports with the input ports of its subcomponents. The external output coupling (EOC) is realized based on the connections of the subcomponents' output ports to the composite component's output ports, while the internal coupling (IC) is realized based on the subcomponents' output ports to the input ports of other subcomponents. Appropriate *addCoupling* methods are generated for each of the specified coupling with the ports added in step 14.

4.3. Behavioral code generation

The behavior for an AADL component is specified using the DEVS Annex (DA). *Behavioral code generation*, the third step of the ADCoDE code generation, uses the external, internal, output functions and test input declarations 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. Algorithm 3 lists the steps followed by the ADCoDE for code generation. Below we describe code generation for the `variables` section and the functions and declarations used in the `behavior` section of a DA subclause.

Variable. Each local variable defined in the `variables` section has a data type and an initial value. Code is generated for each variable *var* according to Algorithm 1.

Declaration intest. The input test declarations in the simulator are defined as *addTestInput* for both atomic and coupled DEVS models. As shown in Algorithm 3 (steps 8–10), code generation for an *intest* is an *addTestInput* with an input port name and a data value.

Function deltint. It specifies the transition from source to destination states and some behavior action. The internal state transition happens when the elapsed time (e) progresses and reaches to the time advance function (ta) ($e = ta$). Internal transition function in the simulator is specified as the public method `deltint`. As shown in Algorithm 3 (steps 11–17), method *deltint* is generated to structure the code generated for various occurrences of the `deltint`. Code generation for internal transition function is based on combining all its occurrences (*C.IMPL.DELTINTS*) using a control structure (*e.g.,* if-else statement) supported in the Java programming language. For each *deltint*, 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* method. The *holdIn* is appended at the end of the *deltint* and *deltext* methods.

Function deltext. It specifies the external transition and is composed of a port name, a received input value, and a variable to store the input value. The behavior action to be performed as a part of the external transition is defined as a string. External transitions in the simulator are realized through a public method `deltext` which accepts an object of the *message* class defined in the simulator. As messages are implemented as bags so method *messageOnPort* is used to explore the bag to find a message received on a particular port and then the required behavior actions are executed accordingly.

As shown in Algorithm 3 (steps 18–24), code generation for `deltext` function starts with *deltext* method generation, with message bag *x*, to structure the code generated for each occurrence of the `deltext`. The rest of the code generation is based on combining all the occurrences (*C.IMPL.DELTEXTS*) using a control structure (*e.g.,* if-else statement) supported by the Java programming language. For each *deltext*, the source state is identified and used in a method *phaseIs* to structure the control. The method *messageOnPort* is then used to explore the message bag on the input port specified in the function. The string containing the behavior action *deltext.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.

Algorithm 3 Behavioral code generation for an atomic AADL component

Require: Implementation classifier (*C.IMPL*) with the DEVS Annex specification of an atomic AADL component *C*
Ensure: Append *ViewableAtomic* Java class previously created for atomic AADL component *C* using Algorithm 2
1: **for all** *var* ∈ *C.IMPL.VARS* **do**
2: **if** *var.type* is Base or Range **then**
3: declare *var* as instance of *stringEnt*, *booleanEnt*, *intEnt*, *doubleEnt*, *Integer_Range*, or *Real_Range* class
4: **else**
5: declare *var* as instance of a Java class previously generated in Algorithm 1
6: **end if**
7: **end for**
8: **for all** *intest* ∈ *C.IMPL.INTESTS* **do**
9: declare *addTestInput* for *intest* with *intest.port* and *intest.value*
10: **end for**
11: generate *deltint* method for internal transitions
12: **for all** *deltint* ∈ *C.IMPL.DELTINTS* **do**
13: generate control structure for *deltint*
14: generate *phaseIs* method with *deltint.source_state*
15: generate behavior action code with *deltint.behavior_action*
16: generate *holdIn* method with *deltint.destination_state* and *deltint.destination_state.sigma*
17: **end for**
18: generate *deltext* method with message bag *x* for external transitions
19: **for all** *deltext* ∈ *C.IMPL.DELTEXTS* **do**
20: generate control structure for *deltext*
21: generate *messageOnPort* method with *x* and *deltext.port*
22: generate behavior action code with *deltext.behavior_action*
23: generate *holdIn* method with *deltext.destination_state* and *deltext.destination_state.sigma*
24: **end for**
25: generate *out* method with parameter *message* for output function
26: **for all** *outfn* ∈ *C.IMPL.OUTFNS* **do**
27: generate control structure for *outfn* with *phaseIs* and *outfn.guard_condition*
28: generate method *add* with method *makeContent*, *outfn.source_state*, and *outfn.message*
29: **end for**

Function outfn. It specifies the output messages to be transmitted through named output ports. This output generation can be further restricted through conditional expressions using the Java programming language. The output functions are specified in a public method `out` in the simulator which returns a message bag defined as a *message* class. A bag contains one or more content, each defined as an output port name and a value pair. Output values can be objects of the *entity* class or its sub-classes. The method *makeContent* is used to create the output port name and value pairs.

As shown in Algorithm 3 (steps 25–29), code generation for an `outfn` function is based on combining all defined outputs (*C.IMPL.OUTFNS*) using a control structure (*e.g.,* if-else statement) supported in the Java programming language. Method *out* is generated to structure the generated code. Then, for every *outfn*, a state is chosen and used in the method *phaseIs*. The *guard_condition*, if present, is used to further constrain the output generation. The method add of the *message* class is then used to add content to the message bag using the *makeContent* method.

4.4. Hierarchical models

A key advantage of AADL-DEVS models is their hierarchical nature. A complex multi-component system can be specified as coupled models, each of which can have other coupled models and atomic models. The implementation of the above algorithms supports code generation for such hierarchical combined AADL and DEVS models. The modeled Isolette Thermostat System (see Section 5) is replicated and used to create a hierarchical coupled model having twelve coupled models, each of which has two coupled models containing three atomic models. From the AADL perspective, the model has four layers. The topmost layer has 12 processes, each of which has 2 thread groups with each having 3 threads. This AADL-DEVS model has 108 components, out of which 72 are atomic models with the rest being coupled models. The code generation for the AADL-DEVS models for the DEVS-Suite simulator takes a few seconds on a 64-bit Windows 10 OS and Eclipse LUNA IDE with i7-5500U@2.4 GHz processor and 4 GB of memory.

5. The isolette thermostat system: An example

This section presents the use of the AADL-DEVS framework for modeling and simulation of a part of the Isolette Thermostat system, an infant incubator described comprehensively in the Requirement Engineering Management Handbook (REMH) published

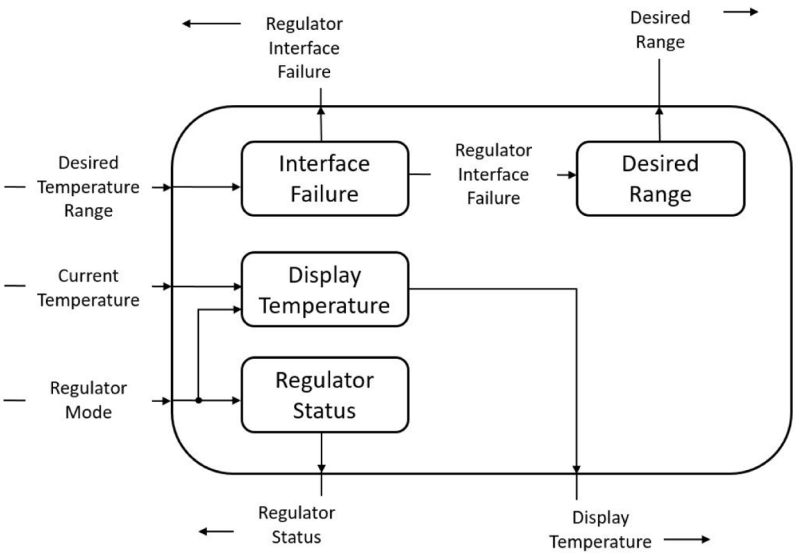


Fig. 3. Manage Regulator Interface component.

by the Federal Aviation Administration (FAA) [12]. This system serves to demonstrate the application of the proposed AADL-DEVS methodology and its realization for non-trivial time-sensitive and safety–critical systems. The *Manage Regulator Interface*, a computational sub-system of the Isolette system referred, is briefly introduced. In the following subsections, this sub-system is used to describe step-by-step combined AADL-DEVS model specification and code generation for selected data, structure, and behavior snippets (refer to [20] for a complete exposition).

As illustrated in Fig. 3, the Manage Regulator Interface obtains the Desired Temperature Range, Current Temperature, and Regulator Mode and reports back the Regulator Interface Failure, Desired Range, Regulator Status, and Display Temperature. It is further divided into four parts; *Interface Failure*, *Display Temperature*, *Regulator Status*, and *Desired Range* to manage the respective controlled and internal variables of the Isolette Thermostat system. For brevity, only the details of the data and functional requirements and architectural modeling & code generation (for DEVS-Suite) of the Display Temperature (as atomic component) and the Manage Regulator Interface (as coupled component) are described in this section. A complete set of the variables used in the Manage Regulator Interface are specified in section 4.1 in [20].

For the Display Temperature component, the variables Regulator Mode of type enumeration (with possible values Init, NORMAL, FAILED), Current Temperature of type structure with real range (68.0..105.0) and status (with possible values Init, On, Failed) elements, and Display Temperature of type integer range (68..105) are used. The Display Temperature, control variable, depends on the Regulator Mode variable. 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.

The following sections are dedicated to demonstrate model specification with code generation for the *Display Temperature* sub-component and the *Manage Regulator Interface* component. The AADL models are extended with the DEVS Annex and then transformed to executable code for the DEVS-Suite simulator. The AADL described below uses snippets from the OSATE example model [10]. A complete model of the Manage Regulator Interface component with its three sub-components is provided in [20].

5.1. Structure and data modeling

The graphical AADL model of the composite Manage Regulator Interface component is depicted in Fig. 4. Based on the functional requirements to set controlled and internal variables, the architectural model consists of three `threads` which are combined into a `thread group` `manage_regulator_interface`. The thread `manage_status` is specified to model Regulator Status. The thread `manage_display_temperature`, specified for the Display Temperature, is described below. The Desired Range and the Regulator Interface Failure are specified as a single thread named `manage_interfaceFailure_desiredRange`. The structure and data models for these threads are provided in Section 5.1 of [20].

The type classifier of Listing 1 declares the input and output interfaces of the `manage_display_temperature` thread component. The input data ports `regulator_mode` receives the regulator mode and `current_temperature` receives the current temperature from

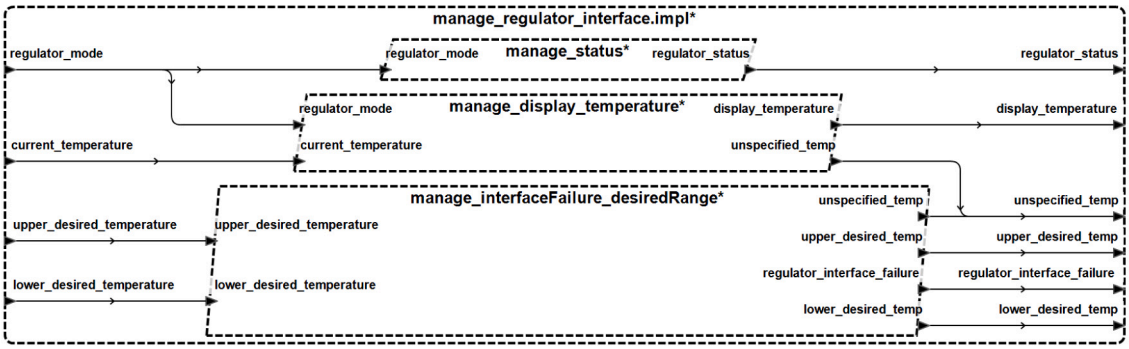


Fig. 4. Manage Regulator Interface AADL model.

Listing 1: Specification for Manage Display Temperature Component

```
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;

end manage_display_temperature;
```

the Manage Regulator Mode component. The output data port `display_temperature` has the temperature to be displayed. The output data port `unspecified_temp` has either "UNSPECIFIED" when the Regulator Mode is not Normal or nothing otherwise.[2] The `properties` section specifies that `manage_display_temperature` is a `Periodic` thread with execution period of `100 ms`.

Properties for these data types are defined using the Data Model annex. For example, the Data component `current_temperature`, as shown below, specifies the Current Temperature as a structure with two elements; `t` to represent the temperature value, and `status` to represent the status value.

```
data current_temperature
 properties
  Data_Model::Data_Representation => Struct;
  Data_Model::Element_Names => ("t", "status");
  Data_Model::Base_Type => (classifier (Iso_Types::measured_temperature_range),
                classifier(Iso_Types::valid_flag));
end current_temperature;
```

Listing 2 presents the extracts from the type and implementation classifiers specified for the `manage_regulator_interface` thread group component (see Fig. 4). Interfaces (*e.g.,* `current_temperature` and `display_temperature`) are defined in the `features` section of the type classifier to establish *EIC* and *EOC* for the coupled component in the `connections` section of the implementation classifier. Section `subcomponents` has references to the respective implementation classifiers of the thread components. For example, the instance with `manage_display_temperature` is a reference to the thread implementation `manage_display_temperature.impl`. Similar identifiers are used to aid with understandability. A complete data modeling for the coupled component `manage_regulator_interface` is presented in Section 5.1 of [20].

Listing 2: A Partial Specification of the Manage Regulator Interface Component and Implementation

```
features
current_temperature :  in  data port Iso_Types::current_temperature;
display_temperature : out data port Iso_Types::display_temperature;
...
subcomponents
manage_display_temperature: thread manage_display_temperature.impl;
...
connections
EOC5 : port manage_display_temperature.display_temperature ->display_temperature;
```

[2] The external data components, ports, and variables are defined within the scope of `Iso_Types` and imported using the AADL `with` clause (see [20])

5.2. Behavior modeling

This section describes component behavior modeling with the DEVS Annex (DA). The implementation classifier of manage_display_temperature thread annotated with DA subclause is explained in detail while the implementation classifiers of the other two threads, specified following the same method and not shown here, are completely described in Section 5.2 of [20].

The DA subclause of the implementation classifier specifies the detailed behavior of the manage_display_temperature thread component (see Listing 3). In the variables section, the variable rgm with initial value INIT is of type regulator_mode and represents the regulator mode. Variable crt is of type current_temperature with initial value 68.0 for the first element and Valid for the second element. The variable ust is of type Boolean with the initial value false and is used as a flag to indicate the unspecified temperature according to *REQ-MRI-5*. The variable pd is of type Float and is set to 100.0, representing the period of the thread. The variable unspecified_value is defined to transmit *unspecified_value* when required.

Listing 3: Specification for 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 => 100.0;
   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, (102.0, "Valid")];

**};

end manage_display_temperature.impl;
```

The states section in Listing 3 contains the declarations for the 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. The state Chk_Mode with $ta =$ pd is a transient state declared to update the variable ust based on the regulator mode. The state Set_Vars is also an instantaneous state with $ta = 0.0$. The state variables ust and Set_Vars are used for generating outputs.

In the behavior section, three deltint state transitions are specified. The first state transition has Start as the source state and Chk_Mode as the destination state. The second and third state transitions are similarly defined. The empty braces indicate no behavior is required to be specified.

Two state transitions deltext associated with two input messages and Chk_Mode as the source state are defined. One has Chk_Mode as destination state while the other has Set_Vars as the destination state. The external message used in the first state transition specifies that the message received on the input port regulator_mode is stored in variable rgm while the second state transition specifies the message received on the input port current_temperature is stored in the variable crt. No other behavior is defined for the first state transition. The second state transition contains the additional behavior defined as an *if-else* statement for setting a value for the ust variable. If the Regulator Mode is Normal, then the variable ust is assigned true otherwise it is assigned false to satisfy the requirements REQ-MRI-4 and REQ-MRI-5.

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

Listing 4: Code Snippet of Generated Data Class 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() {

  }

  ...
}
```

In Listing 3, three test input `intest` declarations are specified to provide test inputs for the `regulator_mode` input port with all possible values ``NORMAL'', ``INIT'', and ``FAILED''. One test input declaration is defined for the `current_temperature` input data port with values `(102.0, ``Valid'')`. The first element in this composite value represents the current temperature while the second element represents the status of the current temperature.

In AADL, behavior of a composite component is defined strictly by the behavior of the subcomponents and the connection between them. Thus, the implementation classifier specification of the `manage_regulator_interface` in Listing 2 has no DA subclause.

5.3. Code generation for DEVS-suite simulator

This section specifies code generation using the AADL to DEVS CoDE generation Engine (ADCoDE) as explained in Section 4. Codes are generated for one thread component `manage_display_temperature` and the thread group component `manage_regulator_interface` as examples of the atomic and coupled parallel DEVS models. Code generation for the other two thread components, completed using the same method, is not included for brevity. The generated code for these can be found in Section 5.3 of [20].

Data classes are organized in a package *Iso_Types* with the same name as the AADL file containing the data components modeling the data types while the model classes are organized in the package *Model*. The name of every model is extended with "_sim" to mark it as one that can be simulated.

Primitive and composite data. Listing 4 presents the code snippet of class `current_temperature` generated for data type Current Temperature (specified as the data component `current_temperature`) using Algorithm 1. It extends the DEVS-Suite Java class *entity* and has two private variables. The 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`.

Parallel atomic and coupled DEVS models. Listing 5 contains the excerpt of the model class for the thread `manage_display_temperature` using Algorithms 2 and 3. Defined in the package `RegulateTemperature` (which contains all the data and model classes), the class `manage_display_temperature_impl_sim` imports the packages `structuredEntities` and `Iso_Types` that contain DEVS-Suite extension and the generated data classes, respectively (not shown in Listing 5). The rest of the imports declarations are required for the DEVS-Suite simulator.

The input and output ports and the test inputs are added as `addInport`, `addOutport`, and `addTestInput` (*e.g.,* `regular_mode`, `display_temperature` and `current_temperature(102.0, ``Valid'')`).

The internal transition function `deltint` has a control structure with the method `phaseIs` to map the internal transition functions defined in Listing 3. For example, if the `phaseIs(``Start'')` (*i.e.,* the current state is `Start`), the destination state changes to `Chk_Mode` with $ta = pd$.

The external transition function `deltext` in Listing 5, is generated for the external transition functions defined in Listing 3 and a control structure provided in the DEVS-Suite simulator. As shown, if the `phaseIs(``Chk_Mode'')` and a message is received on the `regulator_mode` input data port, the content of the message is traversed using the method `messageOnPort` and the last examined value is set for the `rgm` variable. The control stays in `Chk_Mode` for `pd` time units.

Listing 6 contains code snippet of generated model class for the respective thread group `manage_regulator_interface`. In the package `RegulateTemperature`, the `manage_regulator_interface_impl_sim` class extends the `ViewableDigraph` with the required input and output ports specified in Listing 2.

Three instances (*e.g.,* `manage_display_temperature`) are generated for the already generated *ViewableAtomic* classes. The input and output ports extracted from the atomic models are added using the `addInport` and `addOutport` methods. All the test

Listing 5: Generated ViewableAtomic Model Class for *manage_display_temperature_impl*

```
...

public class manage_display_temperature_impl_sim extends ViewableAtomic {

  private current_temperature crt = new current_temperature(68.0, "Valid");
  ...
  public manage_display_temperature_impl_sim(String name) {
    addInport("regular_mode");
    addOutport("display_temperature");
    ...
    addTestInput("current_temperature", new current_temperature(102.0, "Valid"));
    ...
  }

  public void deltint() {
    if (phaseIs("Start")) {
      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());
        }
      }
    }

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

    if (phaseIs("Set_Vars"))
     {
       ...
     }
  }
}
```

input ports are also extracted from *ViewableAtomic* classes and added using the `addTestInput` method. For example, in the `addTestInput(‘‘regulator_mode’’, new stringEnt(‘‘NORMAL’’))`, the input port `regulator_mode` can accept the value defined as `new stringEnt(‘‘NORMAL’’)`, which is an instance of the `stringEnt` class. All the required `addCoupling` methods are then generated for the external input and external output connections (EICs and EOCs) specified in the implementation classifier of the `manage_regulator_interface` thread group. Note that there are no internal connections since the *Interface Failure* and *Desired Range* sub-components in the *Manage Regulator Interface* component are specified as a non-decomposable thread.

6. Simulation using DEVS-suite

The Manage Regulator Interface model developed in the AADL-DEVS environment can be simulated in the DEVS-Suite simulator. The componentized visualization of the Manage Regulator Interface function is depicted in Fig. 5. 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 two output messages; these are compound DEVS-Suite data types. The output events, shown in Fig. 5, are due to the coupled model receiving simultaneously inputs on the test input ports `current_temparature`, `regulator_mode`, and `upper_desired_temperature`. Defining, conducting, and evaluating simulation experiments for the Manage Regulator Interface model are outside the scope of this paper.

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_display_temperature` 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 value of the display temperature. When the value of the received input event is `NORMAL`, `manage_display_temperature` is set to `Set_Vars` (see Listing 5). Since sigma for processing the external events is 0.0, the output event `68.0` (current temperature value modeled with `cVal` of *DoubleRange* element `t` of variable `crt`) is dispatched immediately on the `display_temperature` output port.

Fig. 5 shows the simulation result when the `manage_display_temperature` atomic component receives the temperature `102.0` via the `current_temperature` input port through test input declaration (see Listing 5). After the dispatching of the output, the phase

Listing 6: Generated ViewableDigraph Model Class for *manage_regulator_interface_impl*

```
...
public class manage_regulator_interface_impl_sim extends ViewableDigraph() {

  public manage_regulator_interface_impl_sim() {
    super("manage_regulator_interface");

    ViewableAtomic manage_display_temperature = new
      manage_display_temperature_impl_sim("manage_display_temperature");

    add(manage_display_temperature);
    ...
    addInport("current_temperature");
    addOutport("regulator_status");
    ...
    addTestInput("regulator_mode", new stringEnt("NORMAL"));
    addCoupling(manage_display_temperature, "display_temperature", this, "display_temperature");
  }
}
```

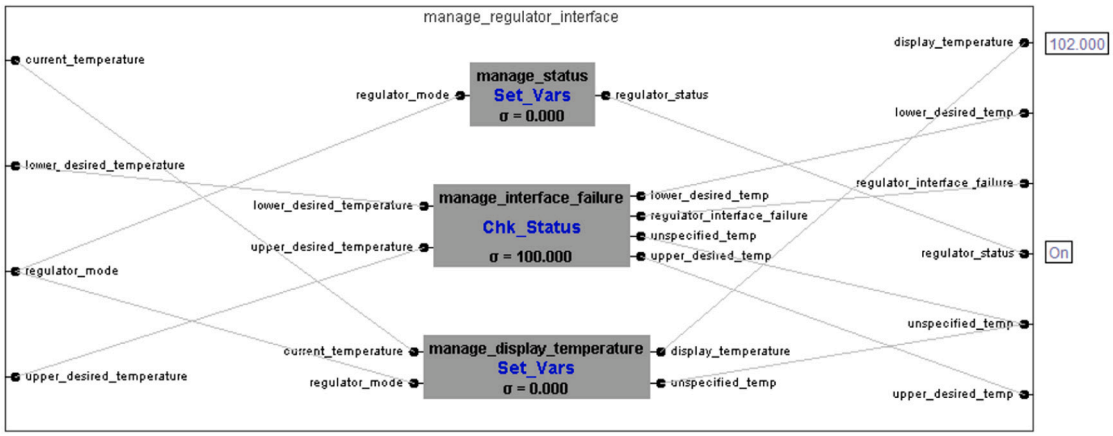


Fig. 5. Manage Regulate Interface Coupled DEVS model.

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''))`.

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 Fig. 5, 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 models can simultaneously receive their input events and execute their external transition functions in parallel. In this step, each model's 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.

The coupled model `manage_regulate_interface` shown in Fig. 5 corresponds to the *manage_regulator_interface.impl* specified in Listing 2. Its atomic models corresponding to the sub-components specified in the `subcomponents` section. The inputs `new stringEnt(''NORMAL'')` and `new current_temperature(102.0, ''Valid'')` are injected to the coupled model's input ports `regulator_mode` and `current_temperature`, respectively. These inputs are transmitted through two external input couplings to the input ports of the `manage_status` and `manage_display_temperature` atomic models.

The `On` value on the output port `regulator_status` of the `manage_status` atomic model is transmitted via an external output coupling to the output port `regulator_status` of the `manage_regulator_interface` coupled model. Similarly, the `102.000` value on the output port `display_temperature` of the `manage_display_temperature` atomic model is transmitted via an external output coupling to the output port `display_temperature` of the couple model `manage_regulator_interface`.

A complete set of AADL-DEVS models for Isolette (see Section 5) with corresponding Java classes automatically generated using the ADCoDE tool, and the example DEVS-Suite models are available at https://github.com/ehah/AADL-DEVS-Framework.

7. Related works

Numerous efforts, spanning a wide range of approaches, frameworks, and tools, are related to the contributions presented in this paper in varying degrees. These existing works are the outcomes obtained by many researchers from different communities, usually with overlapping needs and interests. Some works focus on architecture and design specifications with the aim of using them to build actual software-centric systems. Other researchers aim to develop architecture and design specifications for creating simulations of actual systems. We describe a summary of the related works in relation to this paper from two overlapping aspects. The first focuses on the frameworks purposed to develop architectural and design models systematically. The second considers selected model development environments that support automatic code generation.

Modeling Frameworks: The developments of structural and behavioral specifications are supported by a variety of modeling approaches and frameworks. Among these is CHARMY, a framework for iterative modeling and validation using model checking and simulation [25]. The UML state diagrams and SPIN can be used for behavior specification and model checking. Early works have utilized meta-programmable and model-integrated computing for virtual systems-of-systems evaluations [26].

Hybrid system modeling of the AADL models with Simulink/Stateflow has been explored [27]. The structure is modeled in AADL, the discrete behavior is modeled using the BLESS annex, and the continuous behavior is modeled in Simulink. The AADL Inspector proprietary software provides schedulability analysis and simulation of AADL models with MARZIN simulation engine. Schedulability analysis and simulation of AADL models can be achieved using the Furness toolset [28]. SystemC based simulation of AADL models is also explored in frameworks such as AADS that is developed for POSIX and lending itself to multiple RTOS [29].

Event-driven simulation of AADL models is supported using ADeS tool; an Eclipse plugin for OSATE [30]. DEVS simulation is also used with AADL for verification of TT-Ethernet modeled using AADL [13]. The use of the DEVS modeling is advocated for AUTOSAR [31], a standard for automotive software architecture [32]. The research demonstrates simulating safety–critical component models for controllers operating DC motors of an electronic control unit. It describes the use of hierarchical DEVS models a power window case study [33] and co-simulation for deployment purposes.

Compared to these approaches, the proposed AADL-DEVS framework is grounded in combining the AADL and DEVS modeling methods. A first prototype of the DA supporting logical-time simulation using the DEVS-Suite simulator is developed as a plugin for OSATE [19]. The DA provides a basis for combined static and dynamic testing, verification, and validation. From the simulation standpoint, 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) [34,35] modeling formalism. Such an extension can support simulation under real-time constraints defined in the model and enforced by the DEVS-Suite simulator's host computing platform [6]. The Constrained-DEVS models can be specified and model checked using the DEVS-Suite simulator [36].

Code Generation Tools: One of the goals of Model-Based Engineering (MBE) is to support code generation at various stages of system model development. A syntax-compliant code skeleton is generated for a target programming language for implementation or the source models are filtered to generate specific constructs to enable analysis/simulation using a particular tool. For example, different Architecture Description Languages can be used to generate code for requirement validation and system implementation [37]. Considering DEVS models, they can be translated to code for target simulators (*e.g.,* EMF-DEVS [38], DEVS Natural Language (DNL) [3], and [39]). Ptolemy, based on the Actor Model, is a modeling and simulation tool supporting C-code generation for RTOS [1]. Nonetheless, automatic code generation from hybrid design specifications remains challenging, especially as model complexity grows.

The AADL and Simulink models are translated to C language code and then manually combined with communication code annotations for co-simulation [27]. A framework for automatic code generation from architectural specification, with different views, for situational-aware CPS is proposed in [40]. The executable code, generated through model transformations, is simulated using CupCarbon for power consumption and traffic load analysis. Other exemplar studies proposing different frameworks for self-adaptive systems [41,42], building adaptive web Applications [43], concern-driven modularity measurement [44], and service-oriented embedded systems [45].

Studies investigating code generation from AADL models are more closely related to our work. The OSATE, a realization of the AADL in the Eclipse framework, enables the development of plugins and tools for code generation. An example is the code generator for Ravenscar Profile restriction on architecture models with the OCARINA tool-suite [46]. The generated C and Ada code are used for schedulability and safety analyses. RAMSES is proposed and developed for automatic code generation for different operating systems [47].

System-level co-simulation of integrated systems with Polychrony is studied in [48]. The synchronous sub-set of AADL with Simulink can be exploited for functional behavior modeling while the system-level architecture is modeled using AADL. Automatic code generation without human intervention, for example in C language, is a key aspect of MBE [49]. Respective code is automatically generated for the AADL process, thread, sub-program, and data components. Template-based code generation can be exploited to facilitate multi-platform execution [50]. Rules are defined for code generation for different object platforms based on predefined templates. Code can be generated from AADL models based on model transformation with Model-driven Architecture methodology for a Real-time Operating System (RTOS) [51].

Considering the above approaches and frameworks, code generation in the AADL-DEVS framework is for simulation needs. Java code, compliant with the DEVS-Suite simulator, 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 AADL type classifier, while the behavioral code is generated for different sections of the DA subclause.

8. Conclusion and future work

Systems that have time-sensitive and safety–critical computational components are pervasive. Due to their high degree of composing components, the development of such systems is challenging. To help with satisfying this need, we have presented a methodology for developing combined AADL and DEVS specifications powered with code generation supporting discrete-event simulation. The core AADL is extended with the DEVS Annex for behavior modeling and simulation. The OSATE supported with an annex for parallel DEVS with implementation in DEVS-Suite simulator and ADCoDE results in automating the development of time-based models and simulating them. Such models play a key role in the analysis and design of many complex dynamical systems.

Future work includes using the DEVS Annex for modeling and simulating systems such as Transactive energy and automotive transportation. Toward this goal, one future work is on the capability for the combined AADL and DEVS Annex models to be simulated in real-time, in addition to logical-time. Another direction is to support continuous behavior modeling and simulation in the mold of the DEVS Annex. These research topics should aid in developing, simulating, and evaluating the architectures and designs of system-of-systems that must satisfy accurate time and safety requirements.

CRediT authorship contribution statement

Ehsan Ahmad: Conceptualization, Methodology, Software. **Hessam S. Sarjoughian:** Conceptualization, Methodology, Software.

Data availability

No data was used for the research described in the article.

Acknowledgment

We are thankful to all the reviewers who provided us with helpful critiques and suggestions on an earlier version of this paper.

References

[1] C. Ptolemaeus (Ed.), System Design, Modeling, and Simulation using Ptolemy II, 2014, Ptolemy.org. URL http://ptolemy.org/books/Systems.
[2] R. Alur, Principles of Cyber-Physical Systems, MIT Press, 2015.
[3] B.P. Zeigler, H.S. Sarjoughian, Guide to Modeling and Simulation of Systems of Systems, second ed., in: Simulation Foundations, Methods and Applications, Springer, 2017.
[4] J. Sztipanovits, T. Bapty, X.D. Koutsoukos, Z. Lattmann, S. Neema, E.K. Jackson, Model and tool integration platforms for cyber-physical system design, Proc. IEEE 106 (9) (2018) 1501–1526, http://dx.doi.org/10.1109/JPROC.2018.2838530.
[5] Y. Lei, W. Wang, Q. Li, Y. Zhu, A transformation model from DEVS to SMP2 based on MDA, Simul. Model. Pract. Theory 17 (10) (2009) 1690–1709.
[6] H.S. Sarjoughian, S. Gholami, T. Jackson, Interacting real-time simulation models and reactive computational-physical systems, in: 2013 Winter Simulations Conference, WSC, IEEE, 2013, pp. 1120–1131.
[7] H.S. Sarjoughian, Restraining complexity and scale traits for component-based simulation models, in: 2017 Winter Simulation Conference, WSC, IEEE, 2017, pp. 675–689.
[8] P. Feiler, D. Gluch, Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, Addison-Wesley, 2012.
[9] B.P. Zeigler, H. Praehofer, T.G. Kim, Theory of Modeling and Simulation, second ed., Academic Press, Inc. Orlando, FL, USA, 2000.
[10] OSATE, Open source AADL tool environment, version 2.3.4, 2018, https://github.com/osate/examples/tree/master/isolette. (Accessed 06 August 2020).
[11] ACIMS, DEVS-suite simulator, version 6.1, 2021, https://acims.asu.edu/software/devs-suite.
[12] D.L. Lempia, S.P. Miller, Requirement Engineering Management Handbook, Tech. Rep. DOT/FAA/AR-08/32, Federal Aviation Administration, 2009.
[13] T. Robati, A. El Kouhen, A. Gherbi, J. Mullins, Simulation-based verification of avionic systems deployed on IMA architectures, in: MoDELS'15, 2015, URL https://hal.inria.fr/hal-01211242.
[14] A.W. Wymore, Model-Based Systems Engineering, CRC Press, 1993.
[15] H.S. Sarjoughian, S. Sundaramoorthi, Superdense time trajectories for DEVS simulation models, in: 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.
[16] M.B. McLaughlin, H.S. Sarjoughian, DEVS-scripting: A black-box test frame for DEVS models, in: 2020 Winter Simulation Conference, WSC, IEEE, 2020, pp. 2196–2207.
[17] SAE International, SAE AS5506C, architecture analysis & design language (AADL), 2019.
[18] P. Feiler, D. Gluch, Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, Addison-Wesley, 2012.
[19] E. Ahmad, H.S. Sarjoughian, A behavior annex for AADL using the DEVS formalism, in: 2019 Spring Simulation Conference, SpringSim, 2019, pp. 1–12, http://dx.doi.org/10.23919/SpringSim.2019.8732894.
[20] E. Ahmad, H.S. Sarjoughian, An AADL-DEVS Framework for Cyber-Physical Systems Modeling and Simulation Supported with an Integrated OSATE and DEVS-Suite Tools, Tech. rep., Arizona State University, 2020, URL https://acims.asu.edu/wp-content/uploads/sites/18/2020/03/AADL_DEVS_Framework.pdf.
[21] B.P. Zeigler, H.S. Sarjoughian, V. Au, Object-oriented DEVS, in: Enabling Technology for Simulation Science, Vol. 3083, SPIE, 1997, pp. 100–111.
[22] E. Ahmad, B.R. Larson, S.C. Barrett, N. Zhan, Y. Dong, Hybrid annex: An AADL extension for continuous behavior and cyber-physical interaction modeling, Ada Lett. 34 (3) (2014) 29–38, http://dx.doi.org/10.1145/2692956.2663178, URL http://doi.acm.org/10.1145/2692956.2663178.
[23] S. Procter, P. Feiler, The AADL error library: An operationalized taxonomy of system errors, ACM SIGAda Ada Lett. 39 (1) (2020) 63–70.
[24] SAE International, Architecture analysis & design language (AADL) annex volume 2: Annex B: Data modeling annex annex D: Behavior model annex annex F: ARINC653 annex AS5506/2, 2019.
[25] P. Pelliccione, P. Inverardi, H. Muccini, CHARMY: A framework for designing and verifying architectural specifications, IEEE Trans. Softw. Eng. 35 (03) (2009) 325–346, http://dx.doi.org/10.1109/TSE.2008.104.
[26] J. Sztipanovits, G. Karsai, Model-integrated computing, IEEE Comput. 30 (4) (1997) 110–111, http://dx.doi.org/10.1109/2.585163.

[27] H. Zhan, Q. Lin, S. Wang, J.-P. Talpin, X. Xu, N. Zhan, Unified graphical co-modelling of cyber-physical systems using AADL and simulink/stateflow, in: P. Ribeiro, A. Sampaio (Eds.), Unifying Theories of Programming, Springer International Publishing, Cham, 2019, pp. 109–129.
[28] S. Gui, L. Luo, Y. Li, L. Wang, Formal schedulability analysis and simulation for AADL, in: 2008 International Conference on Embedded Software and Systems, IEEE Computer Society, Los Alamitos, CA, USA, 2008, pp. 429–435, http://dx.doi.org/10.1109/ICESS.2008.63, URL https://doi.ieeecomputersociety.org/10.1109/ICESS.2008.63.
[29] R. Varona-Gómez, E. Villar, AADL simulation and performance analysis in systemC, in: 2009 14th IEEE International Conference on Engineering of Complex Computer Systems, 2009, pp. 323–328, http://dx.doi.org/10.1109/ICECCS.2009.11.
[30] J.-F. Tilman, R. Sezestre, A. Schyn, Simulation of system architectures with AADL, in: Embedded Real Time Software and Systems, ERTS2008, Toulouse, France, 2008, URL https://hal-insu.archives-ouvertes.fr/insu-02269764.
[31] M. Staron, D. Durisic, Autosar standard, in: Automotive Software Architectures, Springer, 2017, pp. 81–116.
[32] J. Denil, P. De Meulenaere, S. Demeyer, H. Vangheluwe, DEVS for AUTOSAR-based system deployment modeling and simulation, Simulation 93 (6) (2017) 489–513.
[33] S.M. Prabhu, P.J. Mosterman, Model-based design of a power window system: Modeling, simulation and validation, in: Proceedings of IMAC-XXII: A Conference on Structural Dynamics, Society for Experimental Mechanics, Inc., Dearborn, MI, 2004, sn.
[34] H.S. Sarjoughian, S. Gholami, Action-level real-time DEVS modeling and simulation, Simulation 91 (10) (2015) 869–887, http://dx.doi.org/10.1177/0037549715604720.
[35] S. Gholami, H.S. Sarjoughian, Action-level real-time network-on-chip modeling, Simul. Model. Pract. Theory 77 (2017) 272–291, http://dx.doi.org/10.1016/j.simpat.2017.06.004.
[36] S. Gholami, H.S. Sarjoughian, Unified property evaluations of constrained-DEVS models for simulation and model checking, in: 2021 Annual Modeling and Simulation Conference, ANNSIM, IEEE, 2021, pp. 1–12.
[37] A. Bucchiarone, D.D. Ruscio, H. Muccini, P. Pelliccione, From requirements to code: An architecture-centric approach for producing quality systems, 2009, arXiv:0910.0493.
[38] H.S. Sarjoughian, A.M. Markid, EMF-DEVS modeling, in: 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, URL http://dl.acm.org/citation.cfm?id=2346635.
[39] M. Cristiá, D.A. Hollmann, C.S. Frydman, A multi-target compiler for CML-DEVS, Simulation 95 (1) (2019) http://dx.doi.org/10.1177/0037549718765080.
[40] M. Sharaf, M. Abughazala, H. Muccini, M. Abusair, An architecture framework for modelling and simulation of situational-aware cyber-physical systems, in: ECSA, 2017.
[41] M.T. Moghaddam, E. Rutten, P. Lalanda, G. Giraud, IAS: An IoT architectural self-adaptation framework, in: A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya, O. Zimmermann (Eds.), Software Architecture, Springer International Publishing, Cham, 2020, pp. 333–351.
[42] J. Ehlers, W. Hasselbring, A self-adaptive monitoring framework for component-based software systems, in: I. Crnkovic, V. Gruhn, M. Book (Eds.), Software Architecture, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 278–286.
[43] A. Agrawal, T.V. Prabhakar, Towards a framework for building adaptive app-based web applications using dynamic appification, in: D. Weyns, R. Mirandola, I. Crnkovic (Eds.), Software Architecture, Springer International Publishing, Cham, 2015, pp. 37–44.
[44] C. Sant'Anna, E. Figueiredo, A. Garcia, C.J.P. Lucena, On the modularity of software architectures: A concern-driven measurement framework, in: F. Oquendo (Ed.), Software Architecture, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 207–224.
[45] P. Newman, G. Kotonya, A runtime resource-management framework for embedded service-oriented systems, in: Software Architecture, Working IEEE/IFIP Conference on, IEEE Computer Society, Los Alamitos, CA, USA, 2011, pp. 123–126, http://dx.doi.org/10.1109/WICSA.2011.24, URL https://doi.ieeecomputersociety.org/10.1109/WICSA.2011.24.
[46] G. Lasnier, B. Zalila, L. Pautet, J. Hugues, Ocarina : An environment for AADL models analysis and automatic code generation for high integrity applications, in: F. Kordon, Y. Kermarrec (Eds.), Reliable Software Technologies – Ada-Europe 2009, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 237–250.
[47] S. Rahmoun, A. Mehiaoui-Hamitou, E. Borde, L. Pautet, E. Soubiran, Multi-objective exploration of architectural designs by composition of model transformations, Softw. Syst. Model. 18 (1) (2019) 107–127, http://dx.doi.org/10.1007/s10270-017-0580-2.
[48] H. Yu, Y. Ma, Y. Glouche, J.-P. Talpin, L. Besnard, T. Gautier, P.L. Guernic, A. Toom, O. Laurent, System-level co-simulation of integrated avionics using polychrony, in: Proceedings of the 2011 ACM Symposium on Applied Computing, SAC '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 354–359, http://dx.doi.org/10.1145/1982185.1982263.
[49] C. Zhang, X. Niu, B. Yu, A method of automatic code generation based on AADL model, in: Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence, CSAI '18, ACM, New York, NY, USA, 2018, pp. 180–184, http://dx.doi.org/10.1145/3297156.3297172, URL http://doi.acm.org/10.1145/3297156.3297172.
[50] K. Hu, Z. Duan, J. Wang, L. Gao, L. Shang, Template-based AADL automatic code generation, Front. Comput. Sci. 13 (4) (2019) 698–714, http://dx.doi.org/10.1007/s11704-017-6477-y.
[51] M. Brun, J. Delatour, Y. Trinquet, Code generation from AADL to a real-time operating system: An experimentation feedback on the use of model transformation, in: 13th IEEE International Conference on Engineering of Complex Computer Systems, Iceccs 2008, 2008, pp. 257–262, http://dx.doi.org/10.1109/ICECCS.2008.19.