DEVS as a Language for Design of Adaptive IoT Systems

Bernard P Zeigler, Laurent Capocchi, Jean-François Santucci

DEVS as a Language for Design of Adaptive IoT Systems

B.P Zeigler[1*], L. Capocchi[2] and J.F Santucci[2]

[1*]RTSync Corp., 6909 W. Ray Road, Chandler, 85226, AZ, USA.
[2]SPE UMR CNRS 6134, University of Corsica "Pasquale Paoli", Campus Grimaldi, CORTE, 20250, FRANCE.

*Corresponding author(s). E-mail(s): zeigler@rtsync.com;
Contributing authors: capocchi@univ-corse.fr; santuci@univ-corse.fr;

Abstract

As Internet of Things (IoT) applications evolve toward greater capability, the requirements for adaptive behaviors become more salient. Complex adaptive concepts implemented in IoT-enabled systems are still emerging. The Discrete Event System Specification (DEVS) formalism enables modular construction and dynamic structure changes to support adaptive behavior. Here we will show that DEVS has the necessary model expressiveness and development continuity properties to serve as an IoT Design language. We will illustrate with several examples to show the wide applicability to IoT systems. We will demonstrate the added value of the System Entity Structure (SES) to enhance expressiveness, scalability, and flexibility in IoT system design, making it a powerful tool for managing complexity and enabling efficient simulation and deployment. In addition, we will formulate conditions that are necessary and sufficient for IoT system development and show how DEVS-based model-driven engineering methodology helps to meet them.

Keywords: IoT, DEVS, SES, IoT System Design, Modular Design, IoT Simulation Frameworks

1 Introduction

Internet of Things (IoT) applications have been evolving toward greater capability [1]. As the abilities of such systems to exert control in user environments have increased, the requirements for the ability to adapt to environmental dynamics become more

prominent [2]. However, although Complex Adaptive System (CAS) concepts have existed for some time [3, 4], implementations in IoT-enabled systems are still emerging. The challenge in ubiquitous computing applications is to dynamically adapt the behaviors of components in a dynamic execution environment. The Discrete Event System Specification (DEVS) formalism [5] enables component-based, hierarchical modular construction, and dynamic changes in structure to support adaptive behavior specifically for IoT systems [6, 7].

We propose that essential requirements for a language to support the development of IoT systems are *model expressiveness* and *model continuity* for all stages of the process including design, simulation, and implementation (see Figure 1). Basically, model expressiveness is the ability to express required functionality in a form that can be easily coded, simulated, and verified. Model continuity refers to the ability to transition the "same" description from stage to stage in the development process, where by "same", we mean that little or no modification is needed in such transitions.

In this paper, we will show that DEVS has the necessary model expressiveness and continuity properties to serve as an IoT Design language. We will illustrate with several examples to show wide applicability. We will demonstrate the added value of the System Entity Structure (SES) to enhance expressiveness, scalability, and flexibility in IoT system design, making it a powerful tool for managing complexity and enabling efficient simulation and deployment.

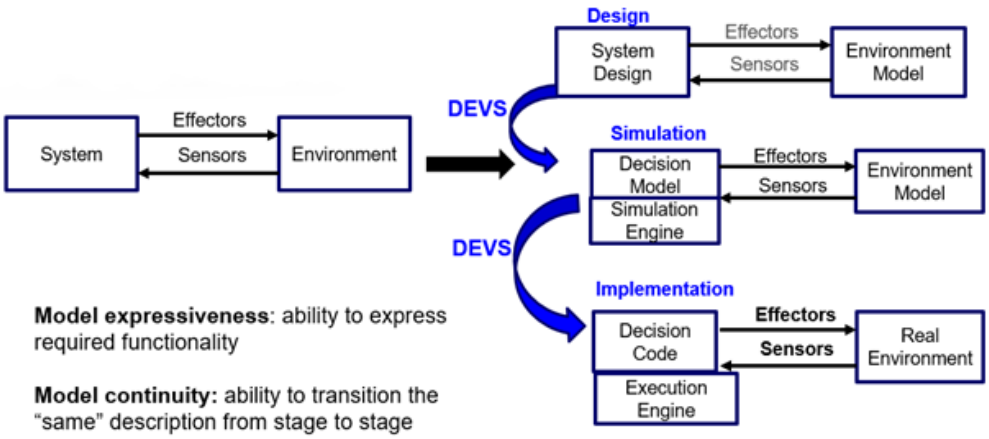

Fig. 1 Stages and the requirements for an IoT Design Language. The DEVS formalism provides the model expressiveness needed for IoT functional design and supports the model continuity needed to transition from stage to stage.

Figure 1 illustrates the stages and the requirements for an IoT Design Language. On the left side of Figure 1 we conceptualize an IoT system as interacting with its environment through sensors and effectors (also called actuators), and on the right side, three basic stages of the development of such a system are shown as Design, Simulation, and Implementation. In the design stage, the focus is on system architecture, often using

high-level design tools [8], with the environment and the sensor/effector interaction minimally represented. Model continuity allows transferring the designed model to be evaluated in the simulation stage and implemented in the implementation stage with minimal changes. Currently, an IoT design language that supports these requirements is lacking [9]. On the other hand, DEVS model expressiveness is manifested in its support of hierarchical and modular model construction. These characteristics enable the implementation and verification of smaller models, which can then be coupled to form a complete model. In this process, a family of models emerges in which the models can be reused and integrated with multiple other models. Moreover, DEVS allows for the simulation of these models alongside a set of well-developed models, targeting both discrete-event and continuous systems [10]. Furthermore, DEVS supports model continuity in which only the simulator needs to be changed to run from abstract time to real-time [11, 12] while the model remains substantially the same.

The remainder of this paper is organized as follows. Section 3 provides a review of the DEVS formalism and the SES, which serve as foundational frameworks for the modeling and simulation (M&S) of complex systems. Section 3 presents requirements for IoT design and development and summarizes how DEVS serves as a language to help users address these requirements.

Section 4 presents a series of illustrative examples demonstrating the application of DEVS in designing IoT systems. These examples include: home automation, the CAIDE architecture, the Actuation Conflict Management (ACM) framework, and swarm-based systems. Section 5 offers a discussion on how the presented examples highlight the expressiveness of DEVS for capturing dynamic and complex behaviors and its continuity in seamlessly transitioning simulation models to real-world implementations.

Section 6 discusses how model-driven engineering provides a methodology to address IoT requirements and can enhance DEVS ability to support them.

Finally, Section 7 concludes the paper by summarizing the key findings and outlining future directions for further leveraging DEVS in IoT system design.

2 Background in DEVS and SES

DEVS Atomic and Coupled Models specify Mathematical Systems [13], one of the earliest forms of general system specification that combined both the automaton formalism of computer science and the dynamic systems models of control theory [14]. DEVS Atomic model specify the dynamic input/output and state behavior of atomic components (elements that are not further decomposed in the model).

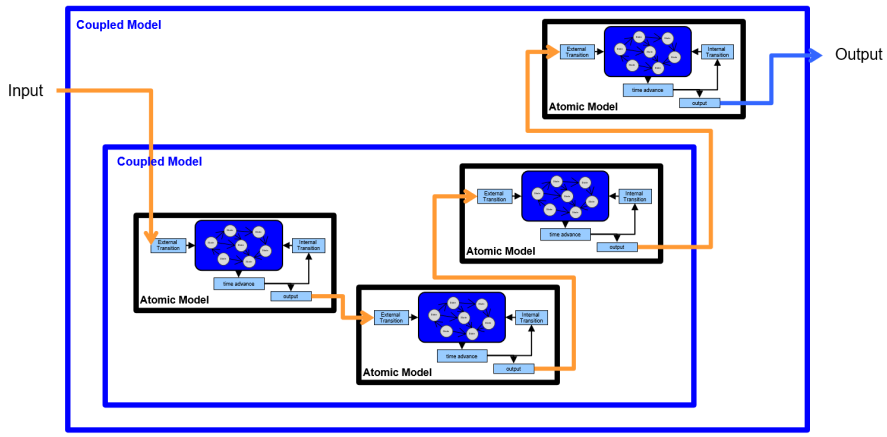

Fig. 2 Illustration of a coupled model with one input and one output, comprising an atomic model and a secondary coupled model that includes three additional atomic models

DEVS Coupled models include components and their couplings - connections from output ports of components to input ports of other components. As illustrated in Figure 2, component models can themselves be coupled models (as well as atomic models) leading to hierarchical structures. *Closure under coupling of DEVS models* is an important theorem that justifies the confidence in managing the complexity of hierarchical models. This states that the resultant of coupling well-defined DEVS models is itself a well-defined DEVS model, i.e, is equivalent in its input/output behavior as a DEVS atomic models [5]. As we will show, such *Hierarchical Modular Composition* supports the model construction needed for development of CAS such as advanced IoT systems.

The System Entity Structure (SES) is a declarative knowledge representation scheme that characterizes the structure of a family of models in terms of decompositions, component taxonomies, and coupling specifications. Formalized by a set of axioms [12], the SES is used to define and construct hierarchical modular DEVS models. As an ontology for M&S, it concentrates a relatively few basic relations as follows:

- **Aspect** expresses a way of decomposing a system into components and is relation between the parent and the children. For instance, IoTSmartApp in Figure 3 is an **entity** composed of Sensors and Actuators. Sensors and Actuators are each represented as components (called multi-entities) that are decomposed into one or more Sensor and Actuator, respectively. An aspect holds the **coupling** relations that will connect the components (children) to create a coupled model for the parent. An entity that has no aspects (decompositions) is the smallest indivisible element and is represented by an atomic DEVS model.
- **Specialization** expresses the variants that a component can assume within a decomposition. Smoke detector, WaterLeakageDetection or ThermalSensor can

replace any of the Sensors and Smartphone, WindowController or AirConditioner can replace any of the Actuators in Figure 3.

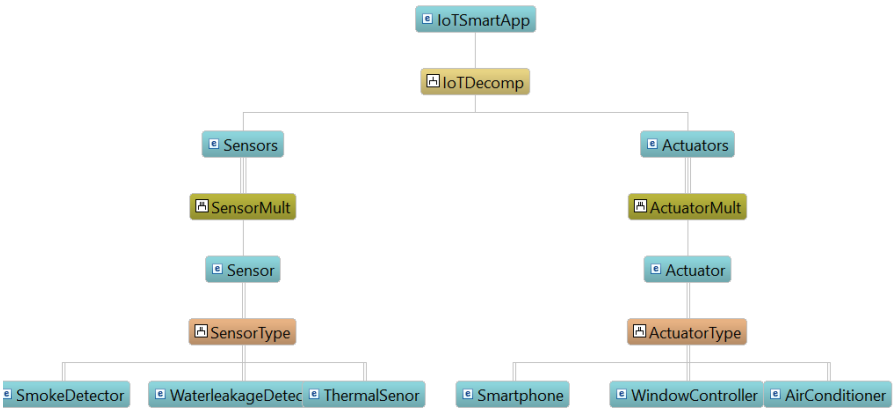

Fig. 3 IoT smart App SES design. Sensors and Actuators are defined as specialization. A sensor entity can be smoke detector or thermal sensor. A actuator can be smartphone or air conditioner.

The corresponding constrained natural language of the SES description depicted in Figure 3 appears as follows:

```
From the IoTDecomp perspective, IoTSmartApp is made of Sensors and Actuators!
From the SensorMult perspective, Sensors is made of more than one Sensor!
From the ActuatorMult perspective, Actuators is made of more than one Actuator!
Sensor can be SmokeDetector,WaterleakageDetection,or ThermalSenor in SensorType!
Actuator can be Smartphone, WindowController, or AirConditioner in ActuatorType!
```

The SES specifies hierarchical coupled models and makes it easier to create them. Indeed, tools exist that help users define SES's in a constrained form of natural language as well interfaces to prune such models[12]. A specific model is selected from the SES family of models by selecting from the available choices in a process called *pruning*. This results in several Pruned Entity Structures (PES), which can then be automatically converted to simulatable DEVS models, thus enabling comparison of alternative architectures [15].

Two other properties of DEVS are very interesting for IoT system design and will be expanded on later:

- *DEVS Universality*: DEVS models can represent a wide variety of system types including continuous, discrete, finite state, etc. [16]. For instance, when it comes to addressing service access conflicts in IoT systems, components are introduced in DevOps flow to manage actuation conflicts, and they are typically specified using particular formalisms such as ECA (Event-Condition-Action). ECA rules are a widely used language for the high level specification of controllers in adaptive systems, such as Cyber-Physical Systems and smart environments, where devices

equipped with sensors and actuators are controlled according to a set of rules. The DEVS Universality simplify the specification of this specific description. Due to the DEVS universality these description formalisms can be reduced to DEVS.
- *Dynamic Structure DEVS*: a type of DEVS model that can change its own structure while running [17, 18]. The ability to change a model's structure during its simulation is highly interesting in the IoT domain, where a system can update the list of available devices (sensors or actuators) upon discovering a new context.

3 Requirements for IoT Design and Development

The identified traits of the DEVS formalism—modularity, event-driven behavior, concurrency, adaptability, scalability, and model continuity—are essential and sufficient for dealing with the complexities of IoT systems design and development. IoT systems are inherently heterogeneous, with diverse devices and components that must interact seamlessly. IoT systems are characterized by several critical requirements:

- Heterogeneity: IoT systems typically comprise a diverse array of devices, sensors, and actuators, each with distinct communication protocols, data formats, and computational capabilities. An effective modeling formalism must accommodate this heterogeneity and enable seamless interaction between these diverse components.
- Real-Time and Event-Driven Behavior: Many IoT applications require real-time processing of sensor data, with decision-making based on specific events. For example, a smart thermostat must respond immediately to temperature fluctuations, while a smart city traffic management system must process sensor inputs in real time to optimize traffic flow.
- Concurrency and Synchronization: The dynamic and distributed nature of IoT systems often necessitates the concurrent operation of multiple devices or subsystems. These devices must interact and synchronize effectively to ensure that the system operates cohesively, without conflicts or performance degradation.
- Adaptability: IoT systems must be able to adapt to changing environmental conditions, such as the failure of components, the introduction of new devices, or network disruptions. A modeling approach must therefore support flexible system structures capable of responding to such dynamic conditions.
- Scalability: As IoT systems grow in size and complexity, the ability to scale the system without introducing instability or performance issues becomes critical. A modeling formalism must support scalable architectures that can accommodate the addition of new components or the expansion of existing ones.
- Model-to-Execution Continuity: An IoT system must transition smoothly from the design phase to the simulation and execution phases. This requires that the behavior modeled during the design phase is accurately reflected in the real-world execution of the system.

DEVS's modular and hierarchical approach addresses this heterogeneity by enabling the design of complex systems as compositions of simpler, reusable models. Furthermore, IoT systems often require real-time responses to sensor data, and DEVS's event-driven nature allows it to model the time-based behaviors that are critical for

such applications. The ability of DEVS to handle concurrent events and synchronize multiple interacting processes makes it ideal for modeling the distributed, concurrent nature of IoT systems, where multiple devices must operate in parallel and remain synchronized. Additionally, IoT systems must be adaptive to changing conditions, such as device failures or network disruptions, and DEVS supports dynamic model structures that allow for this flexibility. DEVS also facilitates scalability by allowing models to grow incrementally, an important feature for large-scale IoT systems. Finally, DEVS's model continuity ensures that the system's behavior remains consistent throughout the development lifecycle, from design to simulation and real-world execution. This continuity is crucial for ensuring that IoT systems perform as intended when deployed.

Secondary studies on IoT systems highlight these same traits as essential, reinforcing that DEVS's modularity, real-time event handling, concurrency management, adaptability, scalability, and smooth transition across stages are not only sufficient but minimal for modeling IoT systems effectively. While other formalisms, such as state machines, Petri nets, and UML, may address some aspects of IoT design, they lack the comprehensive support for concurrency, event-driven behavior, and model continuity that DEVS offers. Thus, DEVS provides a robust and minimal set of traits that make it an ideal candidate for IoT system design and simulation.

4 DEVS Design Principles for IoT Systems: Practical Examples

This section is dedicated to presenting several examples highlighting the enhanced expressiveness and continuity offered by DEVS in the field of IoT systems.

4.1 DEVS Architecture for Home Automation

Faizel and Wainer [10] provide a DEVS specification of a home automation architecture and validate its effectiveness through a detailed case study that integrates multiple sensors and actuators. They developed a series of models, each responsible for a specific functional aspect of the device. These functionalities include polling one or multiple sensors, transmitting the data to other nodes, employing advanced data-sharing algorithms to the sensor data. These models were then combined to form the complete model intended to be executed on each device. A network of nodes running these models showed that the nodes could successfully combine sensor readings from their sensors into a single value and exchange messages with other nodes to "agree" on common values. Various simulation scenarios with different configurations were conducted, and the results aligned with the desired behavior of the models. To demonstrate the applicability of the developed models in practice, a home automation application was created consisting of multiple sensors and actuators. The authors showed that their approach exploited the capability of DEVS to reuse models in both simulation and deployment on hardware. This is achieved via DEVS model continuity first using a DEVS simulator to verify the correctness of the models prior to their deployment and then transforming the models and the associated DEVS simulator to real-time implementation. More specifically, the home automation architecture uses Cadmium version 2, a C++ implementation of DEVS [19]. Multiple C++ classes,

with their state variable types, were defined to correspond to each model. These models were used for simulation purposes and later flashed into micro-controller firmware alongside a DEVS simulator for deployment. We note that all of these models had identical C++ code for both deployment on the device and simulation, except that the message broker model was replaced with an actual commercial off-the-shelf implementation and the client models were slightly modified to make them suitable for deployment on the hardware. Details are described in [10].

In [20], the authors demonstrate the suitability of the DEVS formalism for modeling synchronous automata and verifying execution strategies in the context of IoT system design. They validate their approach using a pedagogical case study: the development of an application to control room lighting. In this work, the behavior of a DEVS model is represented as specifications of a finite state automaton. However, these DEVS specifications encapsulate both the state automaton and the execution machine. The key advantage of using DEVS lies in its flexibility to define multiple strategies through distinct DEVS model specifications.

The traditional IoT system design process typically involves: (i) defining the behavior of IoT components in a library (ii) designing the coupling between components in the library and (iii) executing the resulting coupling. If errors are detected, the designer must redefine component behaviors, particularly those of the execution machine, to handle time conflicts within the ambient system.

In [20], the authors propose an alternative approach based on DEVS M&S. Instead of waiting until the implementation phase to identify potential conflicts, we advocate for an initial phase where the behavior of IoT components and execution machines is modeled and simulated using DEVS. Once the simulations yield successful results, the designer can confidently implement the system behavior within an IoT framework. We will also discuss the advantages of DEVS in managing access conflicts in IoT systems in Section 4.3.

We review how the features of this application illuminate DEVS mode expressiveness and continuity:

- Model expressiveness:
  - Employs hierarchical, modular construction to achieve incrementally verifiable functionality
  - Interacts with Sensors and Actuators in a dynamic environment
- Model continuity:
  - DEVS Simulation engine was expressed in programming language (Cadmum++, Python) for development
  - DEVS models were converted to firmware for real-time execution

4.2 DEVS IoT System: Real-time Monitoring, Management, Forecasting

Developed and implemented using DEVS, Cloud-Based Analysis and Integration for Data Efficiency (CAIDE) was applied to Solar Irradiance Sensor Farms [21]. The system design is based on DEVS, Model Based Systems Engineering (MBSE) [22]

and an IoT infrastructure with the objective to deploy and analyze solar plants in dynamic environments. The system can manage multiple sensor farms and simultaneously improve predictive models in real-time by adapting and re-training models to keep forecasts accurate and updated.

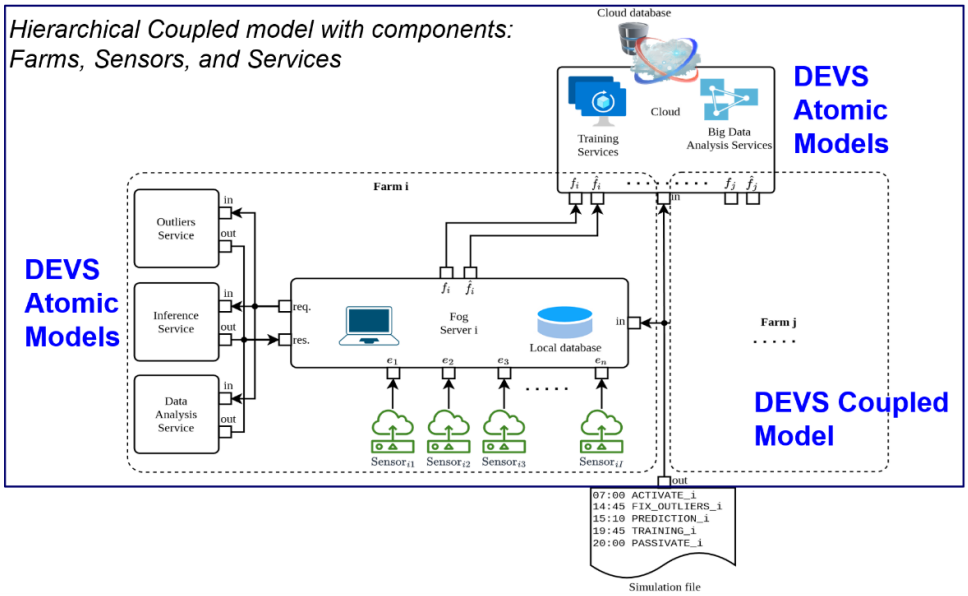

Fig. 4 IoT Solar System Architecture designed and implemented using DEVS syooirt for expressiveness and continuity

As illustrated in Figure 4, the built system is an implemented DEVS hierarchical coupled model whose top-level components are Farms and cloud-based training and big data services. Farms consist of atomic models that perform data processing services, solar sensors, and a central Fog server. The underlying DEVS architecture enables components to be executed in sequential, parallel, or distributed architectures, assuring scalability. The working system was demonstrated in a complex scenario composed of several solar irradiance sensor farms connected to a centralized management system CAIDE has important implications for deployment of solar plants and future of renewable energy sources.

As above, we review how the features of this application illuminate DEVS mode expressiveness and continuity:

- Model expressiveness:
  - DEVS supports expression of the CAIDE architecture which is layered with sensor, Fog, and cloud layers, consistent with IoT Layered Architecture.
  - DEVS atomic models express the required temporal interaction with solar sensors.

  - DEVS modularity provides the flexible basis to support the variable functionality required for AI/ML analysis and retraining.

- Model continuity:
  - The DEVS Simulation engine was employed in Python for development.
  - A DEVS real-time execution engine continues to implement essentially the same model that resulted from the initial design.

4.3 Conflict Management in IoT System DevsOps

In the management of IoT conflict, validation is crucial to identify conflicts by analyzing events and potential actions on actuators managed by concurrent smart applications. Challenges intensify as smart applications increasingly control shared IoT devices, particularly actuators that translate commands into physical effects. A key challenge is managing actuation conflicts, which arise when multiple applications compete for access to shared actuators (direct conflicts) or influence shared physical properties (indirect conflicts).

Designers must address actuation conflicts during the design phase rather than leaving them to end users. The objective is to implement Actuation Conflict Management (ACM) mechanisms that detect and resolve direct and indirect conflicts early (Figure 5). Validation plays a critical role in verifying ACM properties by simulating IoT application events and actuator actions. This process ensures that ACM specifications are robust and effective in real-world scenarios, particularly for managing direct and indirect conflicts.

The proposed approach work builds upon the M&S approach outlined in recent research [23], proposing a method that closely integrates M&S with the design of IoT systems (Figure 5). The methodology leverages the DEVS formalism within the DEVSimPy multi-platform framework [24, 25], facilitating robust and flexible system Design. DEVSimPy is an advanced wxPython General User Interface for the M&S of systems based on the DEVS formalism. With DEVSimPy, a system can be modeled by interconnecting atomic and coupled DEVS models instantiated from libraries.

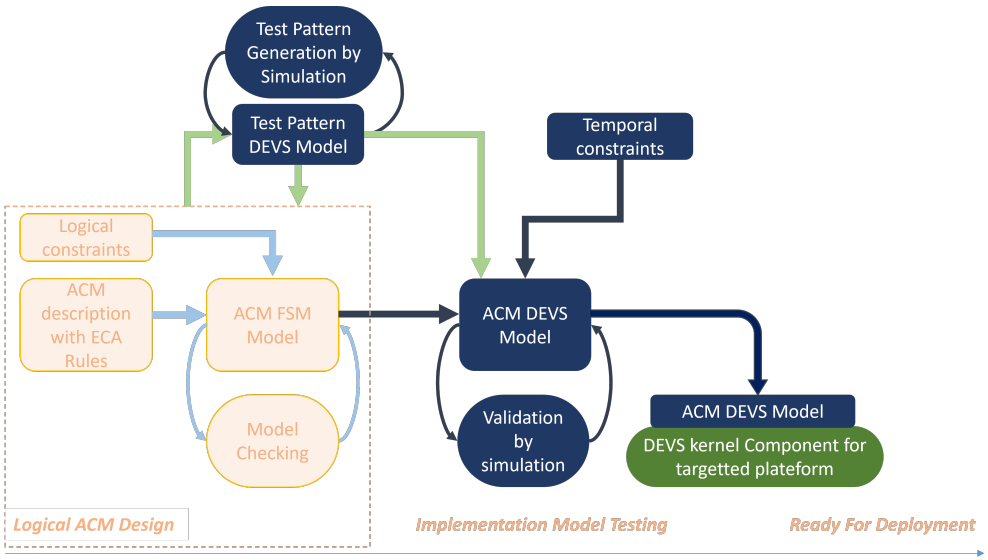

Fig. 5 DEVS M&S inside the custom ACM Design process with its three levels: The *Logical ACM Design*, *Model validation* and *Model Deployment*

Figure 5 shows the custom ACM Design process that include the following levels:

- Logical ACM Design: At this stage, the logical properties of custom ACMs (e.g., completeness, safety, liveness) are formally verified using methods like Model Checking [26, 27]. Custom ACMs are defined as finite-state machines (FSM) provided by the designer.
- Model Validation: This level validates the effects of conflict resolution on the environment through DEVS simulation. The DEVS ACM model incorporates temporal properties (e.g., event delays, state durations) derived from the ACM FSM.
- Model Deployment: This level allows custom DEVS ACMs *temporal properties* to be formally verified through different *asynchronous execution machine strategies* associated with the ACM FSM. DEVS formalism is used to simulate the different implementation strategies. At this level, different hardware platforms with different asynchronous timing specifications can be experimented thanks to the DEVS validation and different middleware/EDGE solutions (node-red [28], ThingML [29], etc.) can be also proposed using an automatic implementation of the DEVS simulation kernel in the targeted middleware.

The proposed approach incorporates an innovative ACM mechanism to identify and resolve conflicts arising from spatial and temporal competition among application flows. Conflicts in IoT systems occur when safety properties (relating to actuators or the environment) are violated. For instance, conflicts arise when multiple application flows attempt to control the same actuator, impacting one of its features. These conflicts can be classified as direct (Figure 6(a)) when targeting a single device or indirect (Figure 6(b)) when application flows affect environmental properties inconsistently.

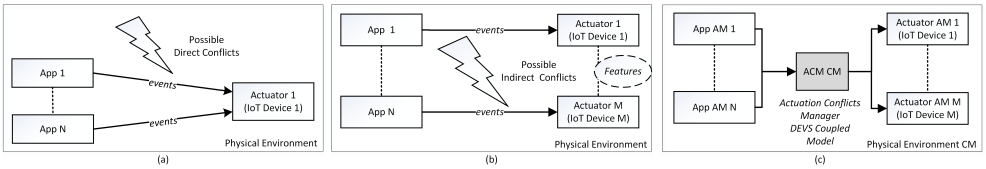

Fig. 6 IoT systems face two conflict types: (**a**) Direct conflicts, where N application flows compete for an actuator's resources; (**b**) Indirect conflicts, where N application flows affect MM devices through environmental characteristics (e.g., noise). An ACM DEVS Coupled Model is introduced to simulate and validate resolutions for both conflict types.

The proposed simulation-based approach intercepts all interactions between application flows (actions) and IoT devices (actuators) to detect potential direct and indirect conflicts. It validates resolution strategies implemented by a dedicated ACM component (Figure 6(c)).

DEVS models were utilized at an early stage of the ACM Design process, enabling simulation-based validation of the ACM mechanism prior to deployment in physical environments. The results demonstrate the expressiveness of the DEVS formalism in specifying and validating the ACM component, ensuring its seamless integration and functionality within simulated physical environments.

The ACM coupled DEVS model consists of two atomic models: a DEVS Synchronizer, which receives events from application flows and drives the evolution of the Logical Behavior based on the inputs it transmits (Figure 7).

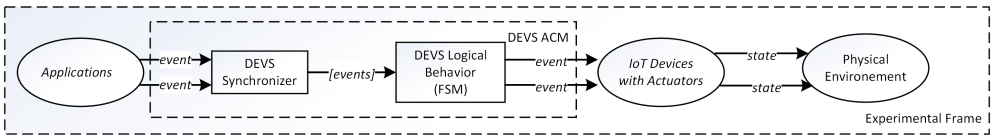

Fig. 7 The ACM component model, featuring its Synchronizer and Logical Behavior models, is embedded within a Physical Environment that includes Application flows and IoT Devices (Actuators)

Designers use the Logical Behavior model, typically modeled as a Finite State Machine (FSM), to define conflict resolution rules through state transitions and output functions. An execution engine triggers these functions based on inputs and generates outputs. While model checking can validate the FSM's logic, it may not address synchronization and temporal aspects.

The Synchronization policy is key to the Logical Behavior's properties. The Synchronizer DEVS model processes input events from IoT application flows by synchronizing and serializing them according to a strategy defined during design. This strategy might involve waiting for all inputs before triggering outputs, sending inputs immediately, or following specific time intervals.

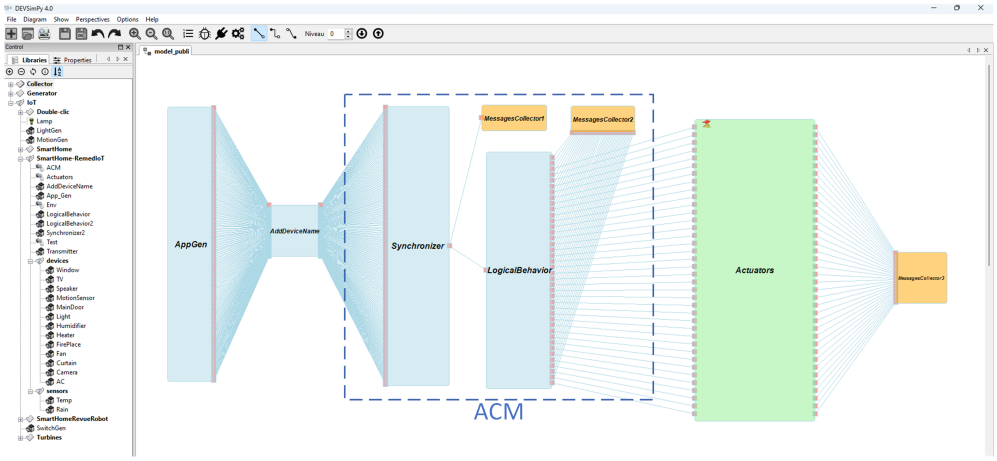

Fig. 8 The DEVSimPy model of the Smart Home scenario integrates the ACM model with the coupled application flows and actuators. The Synchronizer and LogicalBehavior models are part of the ACM coupled. model.

Figure 8) shows an example of modeling a custom ACM in a smart home scenario using DEVSimPy, focusing on conflict detection in IoT-based smart homes. The scenario involves 216 application flows controlling 37 actuators, such as windows, air conditioners, and lights. Conflicts arise when multiple applications control the same actuator, like applications 14 and 134 both controlling the TV. The total number of direct (resp. indirect) conflicts is 3124 (resp. 673). The custom ACM component detects and resolves both direct conflicts (e.g., TV commands) and indirect conflicts (e.g., ambient noise from the Speaker and TV) validating the ACM rules.

Due to DEVS continuity, the DEVS simulation engine was successfully mapped to middleware solutions such as Node-RED and ThingML, enabling straightforward integration with IoT infrastructures. The objective is to support the orchestration and deployment of IoT systems whose software components can be deployed over IoT, edge, and cloud infrastructures. The ACM simulation model can be part of the chosen deployment solution, as the DEVS simulation core is portable across any platform [11, 20]. Thanks to the DEVS formalism, the approach supports deployment across diverse hardware platforms with varying timing characteristics, underscoring its adaptability and portability. By leveraging the DEVS formalism within the DEVSimPy framework, this work bridges the gap between simulation and practical IoT implementation, ensuring that design-phase validation translates effectively to real-world applications.

Another application involving access conflicts to sensors is smart parking. Smart parking is a system that optimizes the occupancy of parking spaces by leveraging specifications that include the behavior of drivers and the dynamic interaction with sensors. One of the key challenges in this domain lies in developing a reliable model that can resolve cumulative parking conflicts, which arise when multiple drivers compete for spaces in a dynamic environment where both user behavior and sensor data are

critical. Thanks to sensor network systems, it is currently possible to provide parking spaces with sensors in order to know their availability in real time. In [30], the authors present a DEVS M&S approach dedicated to propose conflict management strategies based on the estimated travel time to reach desired places around a specific area. DEVS is used since (i) it is discrete-event oriented and therefore an effective solution to the management of time advances in simulation or in real time (ii) its hierarchical modular aspect allows you to conclude a conflict model working from the isolated user in specific models.

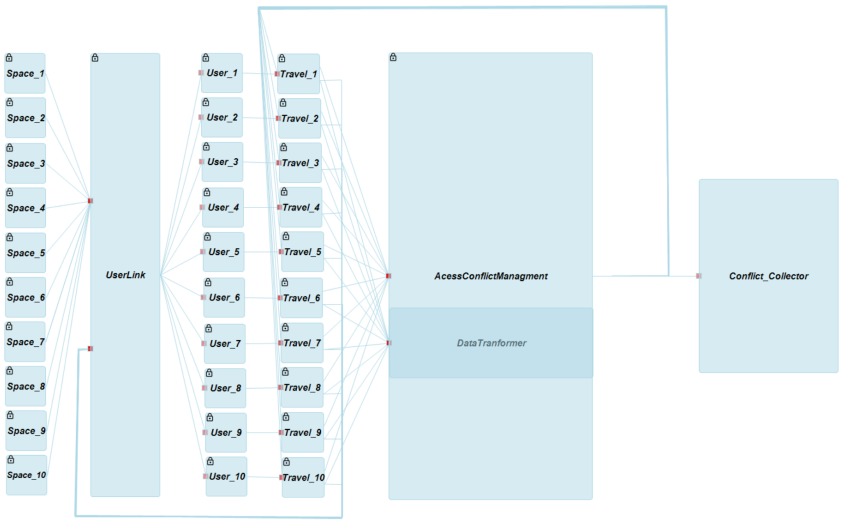

Fig. 9 The DEVSimPy simulation model integrates all the DEVS atomic models along with their interconnections. The simulation considers 10 drivers and 10 parking spaces. A Conflict_Collector model is employed to gather simulation outputs, which are subsequently used for result analysis.

Figure 9 illustrates the DEVSimPy simulation model of the smart parking system. The simulation begins with the Space atomic models (sensor), which can be grouped into a Zone coupled model. The UserLink atomic model aggregates data from sensors and transmits it to the various users (drivers). The User model processes the available parking spaces that match its criteria and forwards user-specific data to the Travel model. The "Travel" model evaluates this data and selects a parking space from the desired options based on different policies. This decision is then passed to the Access-ConflictManagement model, which resolves conflicts between users by applying various algorithms to manage competition effectively.

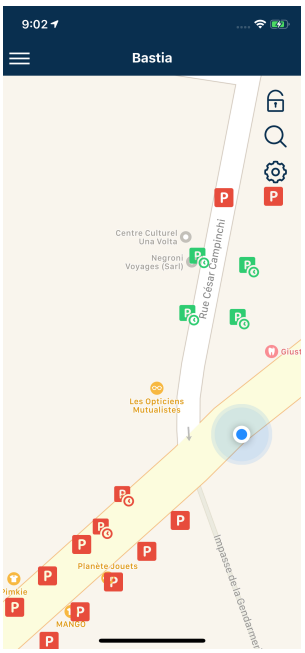

Fig. 10 Mobile app that embed the Smart parking DEVS simulation model for the Bastia city (Corsica - France) which is equipped with more than 400 presence sensors on the roads

Due to the model continuity of DEVS, the "space" models can be connected to an API providing real-time sensor activity from an actual parking lot. By also connecting users to the model (via authentication) and switching the simulation kernel to real-time mode, this simulation model can be utilized by a mobile application. This mobile application can then be used by drivers in a city to locate available parking spaces and occupy them with their vehicles. This simulation model was deployed in the mobile application (due to the DEVS Model Continuity) shown in Figure 10 for the city of Bastia (in Corsica - France).

To summarize, the features of this application that illuminate DEVS model expressiveness and continuity are:

- Model expressiveness: DEVS ability to express concurrent multiple streams of temporal events enabled simulation-based validation of the actuator coordination mechanism prior to its deployment ensuring its seamless integration and functionality within simulated physical environments.
- Model continuity:
  - The DEVS simulation engine can be mapped to middleware implementations enabling straightforward integration with IoT infrastructures.
  - DEVS supports deployment across diverse hardware platforms with varying timing characteristics, underscoring its adaptability and portability.

– DEVS bridges the gap between simulation and practical IoT implementation, enabling design-phase validation to be translated effectively to real-world applications.

4.4 DEVS IoT Development: Dynamic Structure for Adaptive Unmanned Swarm Systems

In [18] Zhang et al. emphasized the model expressiveness capability of DEVS, especially its dynamic structure feature, in application to M&S of *unmanned swarm systems* (USS). Such systems have broad applicability to a variety of domains including military, agriculture, aerospace, etc. Arguing that traditional modeling methods cannot effectively describe the dynamics of USS, they show how to apply DEVS to design, simulate, and implement such systems. The article shows how DEVS enables description of the unmanned component platforms from both behavioral and structural perspectives. In the former, DEVS atomic components model the microscopic behaviors; in the latter, DEVS coupling relationships describe the collaborative structure between unmanned platforms.

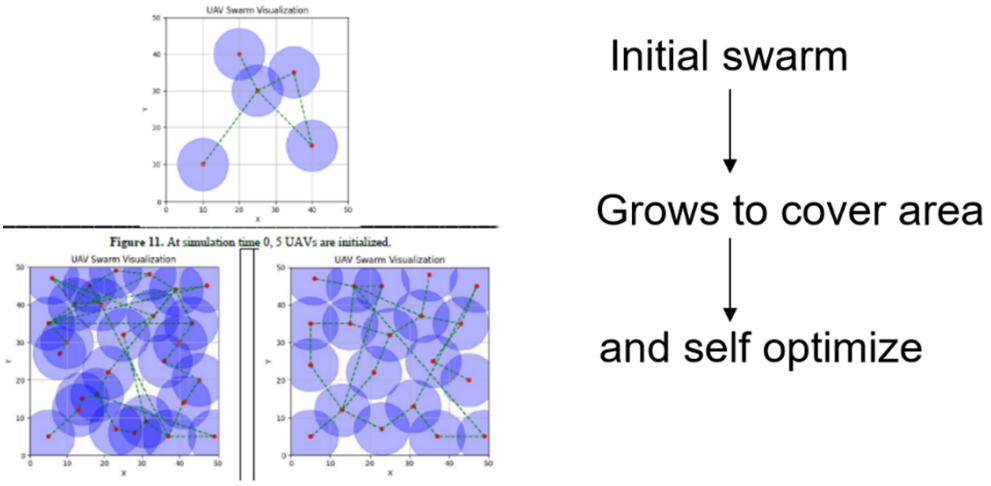

Fig. 11 Dynamic Structure DEVS Objective: minimize overlapping areas

Particularly, Figure 11 illustrates how swarm growth and self-optimization for surveillance by USS is supported by dynamic structure DEVS. The capability to dynamically add and delete sub-models and input and output ports, and to change port connection relationships enables such systems to make structural adjustments in response to environmental changes. They perform such adaptation in accordance with given goals and structure change rules during simulation operation. A DEVS-based synchronization mechanism supports implementation of coordinated actions and changes in structure as required for adaptive behavior.

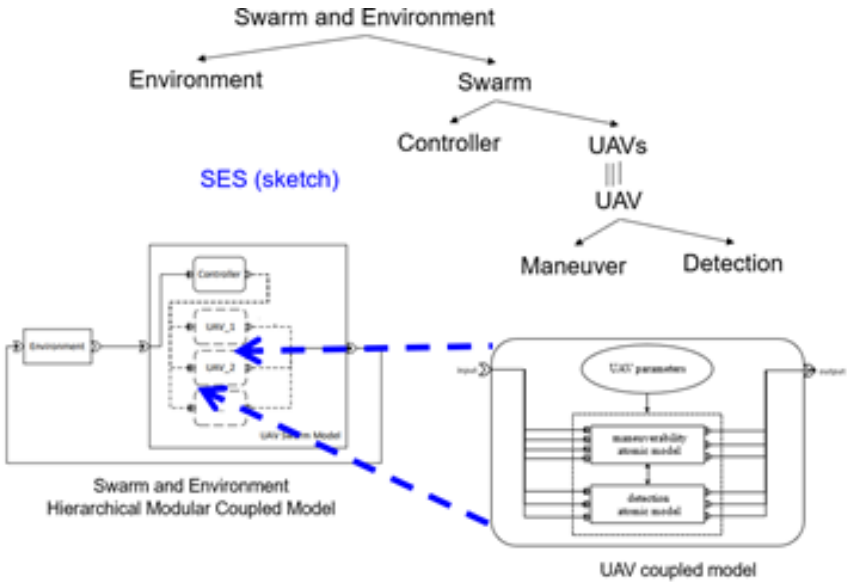

Fig. 12 Hierarchical, modular structure of an unmanned swarm system and its environment is described in the SES (top of figure) and depicted by the hierarchical coupled model (figure bottom)

The hierarchical, modular structure of the USS and its environment is sketched in Figure 12 using the SES on top and the block diagram form at the bottom. The Swarm is decomposed into a Controller and Unmanned Autonomous Vehicles (UAVs), each of which are coupled models containing maneuverability and terrain feature detection atomic models. The USS implements adaptive reconnaissance in the sense that UAVs adaptively adjust their positions in the area under surveillance, or exit the mission, following rules such as:

- When an area is not being surveyed, add a UAV.
- When a UAV is damaged by more than 50%, it must exit the mission.
- When the UAV suffers interference, it leaves the interference area.
- When UAV reconnaissance areas overlap, the more damaged UAV leaves the area.

Such rules are implemented in the model using the dynamic structure capability of DEVS. Further, the temporal properties of DEVS are well suited to model the synchronization mechanism. The latter is necessary because UAVs can take different amounts of time to finish their assigned tasks due to the heterogeneity of rule application and environmental effects such as damage and interference. Synchronization is implemented by having the controller wait for all UAVs to report that they have completed the preceding task, before issuing the order to proceed to the next task.

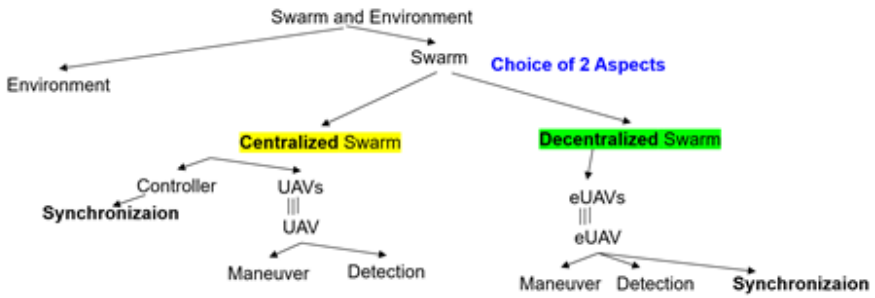

Fig. 13 Sketch of the system Entity Structure for unmanned swqrm system centralized and decentralized synchronization. This SES is described by the text in Figure 14

Figure 13 leverages the SES to express how synchronization can be implemented in a *decentralized* manner as opposed to the *centralized* one just discussed. In the latter, the central controller is absent and its functionality is implemented by the individual UAVs. While DEVS expresses the synchronization behavior required in either case, the SES expresses the alternative implementations employing different aspects (decomposition sub-trees) for the centralized and decentralized alternatives, respectively. Figure 14 illustrates such an SES description.

```
From the topLevel perspective, SwarmNEnvironment is made of UAVSwarm and Environment!
From the topLevel perspective, UAVSwarm sends  outCommand to Environment as inCommand!
From the topLevel perspective, Environment sends  outCommand to UAVSwarm as inCommand!
From the topLevel perspective, UAVSwarm sends  outStatusInformation to Environment as
inStatusInformation!
From the topLevel perspective, Environment sends  outStatusInfoCorrupted to UAVSwarm as
inStatusInfoCorrupted!

From the Centralized perspective, UAVSwarm is made of Controller, First_UAV, Second_UAV, and
Third_UAV!
From the Centralized perspective, UAVSwarm sends  inCommand to all UAV as inCommand!
From the Centralized perspective, Controller sends  outCommand to UAVSwarm as outCommand!
From the Centralized perspective, UAVSwarm sends  inStatusInfoCorrupted to Controller as
inStatusInfoCorrupted!
From the Centralized perspective, all UAV sends  outStatusInformation to UAVSwarm as
outStatusInformation!

From the Decentralized perspective, UAVSwarm is made of Second_UAV, First_UAV, and Third_UAV!
From the Decentralized perspective, all UAV sends  outStatusInformation to all UAV as inStatusInformation!
From the Decentralized perspective, all UAV sends  inCommand to all UAV as inCommand!
```

Fig. 14 Example of a SES description using constrained natural language. The text specifies a hierarchical coupled model that can be constructed from one ot the two aspects corresponding to centralized and decentralized control (shown in yellow and green, respectively.

The first line in the fragment of Figure 14 defines an aspect called *topLevel* that decomposes the overall model *SwarmNEnvironment* into *UAVSwarm* and *Environment* components. Subsequent lines for this aspect declare coupling relations that state how output ports of sender components connect to input ports of receiver components. The yellow and green highlighted lines define two different aspects for decomposing the *UAVSwarm* component corresponding to those shown in color in Figure 13, respectively. Coupling relations express in these lines express connections from a coupled model input port to one more of its components input ports (called External Input Coupling) as well as conversely, from component output ports to output ports of the parent coupled model.

4.5 Cyber-physical Systems Concepts Intergrated into IoT Systems

While IoT design focuses on communication and control of sensors and effectors, cyber-physical system design emphasizes the interaction between digital and physical components. As applications become more demanding, the two thrusts are converging toward a more complete methodology that combines their capabilities [31].

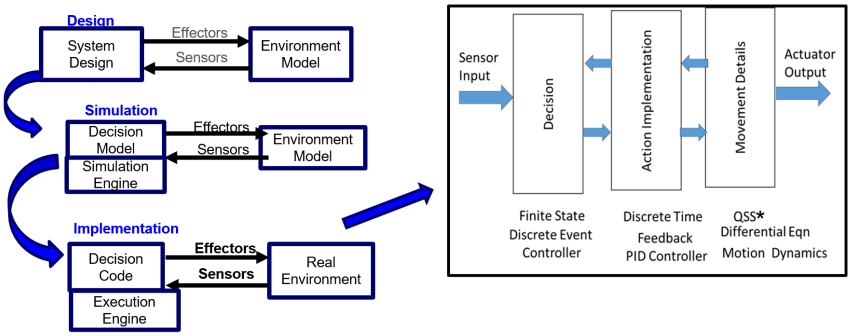

Fig. 15 Architecture that leverages DEVS capability to express building blocks for both the computational and control-theoretic functions required for intelligent cyber-physical system design

Along these lines, Figure 15 illustrates an architecture that leverages DEVS capability to express building blocks for both the computational and control-theoretic functions required for intelligent cyber-physical system design [32]. Implementation of such an architecture is discussed by [33] and exploits DEVS model continuity for efficient development of embedded controllers for robotic systems. DEVS model continuity can support design, simulation, and implementation of such design concepts for USS systems as well.

In this use case, the features that illuminate DEVS model expressiveness and continuity are:

- Model expressiveness:

  - DEVS Atomic functions capture microscopic behaviors (messages, timing, decisions)
  - DEVS atomic and coupled models support synchronization.
  - DEVS Hierarchical modular structure expresses collaborative interaction in centralized and decentralized control.
  - DEVS Dynamic structure enables the structural changes needed for adaptive behavior.

- Model continuity:
  - DEVS supports definition of building blocks and architectural patterns for intelligent hybrid cyber-physical system design.
  - DEVS Execution engines can be implemented in diverse technologies such as virtualization, hardware, embedded system, and bioware.

5 Discussion

In the following we summarize the properties of the DEVS formalism that were elucidated in the examples given above to support the claims for its validity as an IoT design language based on its model expressiveness and continuity.

5.1 DEVS Properties for Model Expressiveness and Model Continuity

DEVS is a strong candidate for the expressiveness of IoT system design due to its ability to model heterogeneous, dynamic, and event-driven environments while ensuring modularity, hierarchy, and formal validation. IoT systems are inherently complex and distributed, involving diverse components that operate at different time scales and require flexible interaction mechanisms. DEVS offers key advantages that align well with these characteristics:

- **Modeling of Asynchronous and Event-Driven Behavior**: IoT systems rely on asynchronous interactions among devices, sensors, and actuators. DEVS, as a discrete-event formalism, naturally represents systems where state changes occur at discrete time instants, making it well-suited for capturing real-world IoT dynamic [34]. It represents state changes at discrete time intervals, making it adept at modeling real-world dynamics in IoT environments [35];
- **Hierarchical and Modular Structure**: DEVS enables hierarchical composition of models, allowing IoT architectures to be designed in layers—such as edge, fog, and cloud computing—while maintaining encapsulation and interoperability between components [36]. This modularity enhances the reusability of models across different applications [34].
- **Separation of Concerns (Structure vs. Execution)**: IoT systems often require separating functional behavior from execution strategies. DEVS achieves this through its atomic models (defining component behavior) and coupled models (specifying interactions and execution flow), providing a clear separation between computation and communication [37].

- **Support for Concurrency and Synchronization**: IoT components often involve multiple interacting subsystems that require concurrent processing. DEVS inherently supports parallel discrete-event simulation (P-DEVS), making it suitable for modeling concurrency, synchronization mechanisms, and conflict resolution in distributed IoT systems [38].
- **Adaptability to Dynamic Environments**: IoT applications demand adaptability due to changing conditions and evolving requirements. Dynamic Structure DEVS (DS-DEVS) extends DEVS by allowing on-the-fly reconfiguration, which is essential for modeling adaptive behavior in IoT networks [39]. Adaptive decision-making frameworks, such as the one proposed by Wang et al., utilize layers that sense, decide, and execute actions based on dynamic conditions [40].
- **Validation through Discrete-Event Simulation**: A critical aspect of IoT system design is verifying whether execution strategies remain conformant with the intended functional model while incorporating real-world constraints. DEVS provides a rigorous simulation-based validation framework, allowing designers to test control strategies, real-time constraints, and system reliability before deployment [39].
- **Interoperability with Other Modeling Approaches**: IoT system design often integrates multiple modeling paradigms, such as synchronous automata, Petri nets, and state machines [41]. DEVS can coexist with and complement these models, making it a flexible bridge for heterogeneous system design.

By capturing both system structure and execution dynamics, while enabling modularity, adaptability, and validation, DEVS emerges as a powerful and expressive framework for IoT system modeling and simulation.

Table 1 lists properties of the DEVS formalism related to model expressiveness, defined as its ability to express functional aspects of IoT systems.

Table 1 Model Expressiveness: Ability to Express Required Functionality in IoT systems

DEVS Properties	**Expressiveness Features**
DEVS atomic model functions	capture microscopic behaviors (messages, timing, decisions), express temporal interaction with sensors and actuators
DEVS hierarchical modular construction	supports incrementally verifiable functionality, expresses collaborative interaction in centralized and decentralized control, expresses the IoT architecture which is layered with sensor, Fog, and cloud layers
DEVS modularity	provides flexible support for the variable functionality required for AI/ML model analysis and retraining
DEVS temporal properties	express concurrent multiple streams of temporal events enabling simulation-based validation of the coordination and synchronization mechanisms
DEVS dynamic structure	enables structural changes needed for adaptive behavior
DEVS system-theory basis	supports definition of building blocks and architectural patterns for intelligent hybrid cyber-physical system design

Model continuity refers to the ability to transition a system description seamlessly across different stages of development—design, simulation, and execution—with

minimal or no modifications. This is particularly important in IoT system development, where models need to remain consistent across heterogeneous platforms and real-world constraints. DEVS provides a formal and modular approach that supports model continuity in the following ways:

- Transition from Design to Simulation: DEVS provides a formal specification that allows IoT models to be directly simulated without reinterpreting their structure or behavior. The same model used in design can be executed in a discrete-event simulation environment, ensuring that functional behaviors (e.g., message passing, event synchronization, timing constraints) are validated early. The hierarchical and modular nature of DEVS allows developers to incrementally refine their models while preserving core behavioral properties. For example, IoT system architects can design DEVS models representing sensor interactions, data aggregation, and processing logic, then test these models in a simulation engine before deployment.
- Transition from Simulation to Execution: DEVS enables migration from simulated environments to real-world execution by transitioning from abstract simulation time to real-time execution. DEVS models can be mapped to real-time platforms, ensuring that the timing, coordination, and decision-making behaviors observed in simulation are maintained during execution. Through real-time DEVS (RT-DEVS), the same IoT models can be integrated into embedded systems, middleware, and cloud environments without major alterations. For example, a DEVS-based traffic monitoring system tested in a simulation environment can be directly deployed onto real-world IoT infrastructure while maintaining its event-driven behavior [42].
- Support for Diverse Implementation Platforms: DEVS models can be executed across a wide range of hardware and software platforms, including: (i) Embedded systems (IoT devices, microcontrollers) (ii) Edge and fog computing environments (iii) Cloud-based IoT platforms and (iv) Distributed simulation frameworks [43]. This adaptability ensures that the same IoT model can be scaled and reused across multiple deployment scenarios. For example, a DEVS-based smart grid model can be tested in a cloud-based simulation environment and later deployed onto real-time distributed IoT systems while preserving model fidelity.
- Model Validation and Conflict Resolution: DEVS simulation helps verify and validate execution strategies, ensuring that an IoT system's operational behavior remains consistent with its design [44]. At the Operational Model level, DEVS supports conflict actuation management, helping resolve issues such as resource contention, sensor conflicts, and dynamic adaptation. For example, a smart building IoT system modeled in DEVS can simulate conflicting temperature control settings before deployment, ensuring smooth operation.
- Dynamic Adaptation and Evolution: IoT systems can reconfigure themselves autonomously, reducing the need for human intervention. This self-management is vital in complex environments, as highlighted in studies on self-adaptive software systems [45]. Through Dynamic Structure DEVS (DS-DEVS), models can adapt to environmental changes in real-time, allowing IoT systems to be self-reconfigurable. This ensures that model continuity extends beyond initial deployment, supporting evolution and updates without requiring full redesigns. For example, an IoT-based

disaster response system modeled with DS-DEVS can dynamically adjust communication patterns and resource allocation in response to changing emergency conditions.

Table 2 lists transitions from the Design stage to the Simulation stage and from the latter to the Execution stage for which DEVS model continuity supports development of IoT systems. The table also considers the diversity of implementation that DEVS can work within.

Table 2 Model Continuity: Ability to Transition the "Same" Description from Stage to Stage

Inter-stage Transitions	**DEVS Model Continuity Features**
Migration from Design to Simulation	The DEVS Simulation engine is coded in a variety of programming and higher level languages for design and simulation
Migration from Simulation to Execution	DEVS Simulation engine can be transformed from its abstract time base to real-time bases and DEVS models can be converted to hardware or middleware forms for real-time execution
Diversity of implementation media	The DEVS simulation engine can be mapped to middleware implementations enabling straightforward integration with IoT infrastructures, DEVS supports deployment across diverse hardware platforms with varying timing characteristics, underscoring its adaptability and portability, DEVS Execution engines can be implemented in diverse technologies such as virtualization, hardware, embedded systems, and bioware

They are other modeling formalisms besides DEVS that aim to achieve expressiveness and continuity in IoT systems, including Class Diagrams, State Machines, and Petri Nets [46]. While each formalism has its strengths, they may not fully address the unique challenges of IoT systems in the same way that DEVS does. Let's explore how some of these popular formalisms compare to DEVS in terms of expressiveness, continuity, and support for IoT system features:

- Class Diagrams (UML): Class Diagrams are commonly used in Unified Modeling Language (UML) to represent the structure of systems through classes, attributes, operations, and relationships between classes. They are valuable for defining the static structure of IoT systems, particularly for object-oriented design and database schema representation [47].
  - Strengths: Static structure definition: Ideal for capturing the hierarchical relationships and data organization within an IoT system (e.g., sensor data models, device classes). Widely adopted: A well-understood formalism, especially in enterprise and software system design.
  - Limitations compared to DEVS: Lack of Temporal Dynamics: Class diagrams do not inherently model time-dependent behavior or event-driven interactions that are crucial in IoT systems. This makes them less suitable for modeling asynchronous events and temporal dependencies. Limited Reactivity: Class diagrams are static and do not easily model reactive behavior—the ability of a system to

respond to external stimuli or events in real time. No explicit support for Execution Models: Class diagrams do not specify how an IoT system behaves over time, which limits their support for execution strategies or model validation in dynamic, event-driven environments.

- State Machines: State Machines (or Finite State Machines, FSM) are widely used to model discrete states and state transitions based on input events. They are effective for describing control flow and sequential behavior, which makes them applicable to certain types of IoT systems (e.g., simple control systems, state-based devices) [48].
  - Strengths: Clear Representation of Control Flow: Good for modeling sequential logic and finite state transitions, which are common in IoT devices (e.g., a smart thermostat with states like "heating," "cooling," and "idle"). Simple and Intuitive: Easy to understand and implement, making them suitable for small systems or components with straightforward behaviors.
  - Limitations compared to DEVS: Limited Modularity: While state machines can model transitions, they lack the modular design inherent to DEVS. Complex IoT systems that involve multiple interacting components may become difficult to manage using only state machines. No Support for Concurrency: Traditional state machines are inherently sequential and do not handle concurrent events well, which is a core feature in IoT systems where multiple components interact simultaneously. Lack of Hierarchical Abstraction: DEVS allows for hierarchical modeling, which enables nested behavior and system decomposition—this is particularly useful in IoT systems that have multiple layers (e.g., sensor networks, cloud services, edge devices). State machines generally do not support this level of abstraction.
- Petri Nets: Petri Nets are a graphical and mathematical formalism used to model concurrent, asynchronous, and distributed systems. They have been used in modeling communication protocols, process control, and IoT systems [49].
  - Strengths: Concurrency and Synchronization: Petri nets are strong in modeling parallelism, concurrency, and synchronization of events, which is crucial in IoT systems where multiple devices and sensors may operate simultaneously. Well-Suited for Event-Driven Systems: They handle event-driven behaviors well and can model complex resource-sharing and token-passing mechanisms, which are common in IoT systems.
  - Limitations compared to DEVS: Lack of Modularity: While Petri nets can model concurrency, they do not support the modular composition of IoT systems in the same way DEVS does. They can be complex to manage when dealing with large systems with many interacting components. Limited Focus on Execution Models: Petri nets model state transitions and events, but they do not inherently support execution strategies, such as mapping a model to real-time platforms or handling issues like timing constraints or adaptive reconfiguration. Partial Support for Dynamic Structure: While Petri nets can model system dynamics, they do not inherently support dynamic structure changes or self-adaptation in the same way DEVS with DS-DEVS does.

- SysML (Systems Modeling Language): SysML, an extension of UML, is used for modeling complex systems of systems, and it includes state diagrams, activity diagrams, and block definition diagrams. SysML is frequently used in engineering and embedded systems [50].
  - Strengths: Supports Complex Systems: SysML is suited for representing multi-domain systems (e.g., electrical, mechanical, and software components), which is useful in large IoT systems. State Transitions and Behavior Modeling: Like UML state machines, SysML can represent state-based behaviors.
  - Limitations compared to DEVS: Limited Simulation Support: SysML does not natively include simulation capabilities as part of the formalism. For IoT systems, DEVS provides simulation and validation tools that allow for dynamic, event-driven analysis. Lack of Real-Time Behavior Modeling: SysML does not inherently support the real-time execution of systems as DEVS does. IoT systems often require not just simulation but also direct mapping to real-time execution environments, which DEVS provides seamlessly.

Table 3 compares different formalisms for expressiveness and model continuity in IoT system design.

Formalism	**Strengths**	**Limitations Compared to DEVS**
Class Diagrams	Good for static structure modeling	Lacks temporal dynamics, reactivity, and execution models
State Machines	Effective for sequential control and finite states	No support for concurrency, hierarchical design, or modularity
Petri Nets	Excellent for concurrency and synchronization	Limited modularity, no inherent support for dynamic structures or execution strategies
SysML	Useful for complex system-of-systems modeling	Lacks native simulation support and real-time execution capabilities

Table 3 Comparison of Different Formalisms with DEVS

6 How Model-driven Engineering Methodology and DEVS Help Meet IoT Requirements

In the context of IoT systems, model-driven engineering (MDE) [51] plays a significant role in supporting the principles of expressiveness and continuity. Regarding expressiveness, MDE allows for the use of Domain-Specific Languages (DSLs), which provide tailored abstractions for different IoT components such as sensors, actuators, and communication protocols. These DSLs enable designers to express complex IoT behaviors at a higher level, thus making system design more intuitive and aligned with the functional requirements of the system. Additionally, MDE supports modular design through the use of formal methods, such as UML, SysML, and DEVS, which

enable the decomposition of complex IoT systems into smaller, more manageable components. This modularity enhances system expressiveness by providing a flexible and scalable approach to modeling large, distributed systems. Moreover, MDE facilitates the use of formal validation methods, ensuring that the models accurately capture the system's dynamic and event-driven behaviors, which are crucial for the temporal and reactive nature of IoT systems.

In terms of continuity, MDE enables the seamless transition of models through different stages of the development process. Model refinement allows for progressive detailing of an abstract model, transforming it from high-level design to a more concrete, executable form. MDE also supports model transformations, which automatically convert models from one stage to the next (e.g., from design to simulation, or from simulation to execution), ensuring that the same system description can be carried through the entire development lifecycle without introducing inconsistencies. This ability to maintain consistency across stages is further reinforced by traceability mechanisms, which link different levels of abstraction and ensure that changes made at one stage are reflected in the subsequent stages. Additionally, MDE facilitates simulation and execution integration, allowing for models to be simulated early in the development process and then directly executed on hardware or middleware with minimal modification. Finally, MDE supports automatic code generation, which converts high-level models into executable code, enabling the direct deployment of IoT systems onto real-time platforms without the need for significant rework. Thus, MDE ensures that models maintain their integrity as they transition from design to deployment, reducing the effort required for system implementation and enhancing the overall continuity of the development process.

In summary, MDE provides a powerful framework to help implement the application of DEVS as a language for IoT development. MDE provides a methodology to apply the above-described DEVS-based modeling and simulation capabilities to address the challenges of expressiveness and continuity in IoT system design. By allowing for high-level abstractions, modular design, and seamless transitions between stages of development, DEVS and MDE significantly enhance the efficiency and accuracy of IoT system development, from initial design through to real-world deployment.

7 Conclusion

In this paper, we have demonstrated that the DEVS formalism, coupled with the system Entity Structure, SES, possesses the necessary expressiveness and continuity to serve as a robust design language for IoT systems. Through illustrative examples, such as home automation, solar sensor farm management, conflict resolution mechanisms, and dynamic unmanned swarm systems, we highlighted how DEVS supports adaptive and complex behaviors with hierarchical modularity, synchronization, and dynamic structure capabilities. Furthermore, the continuity inherent in DEVS enables seamless transitions from design and simulation to real-world implementation, making it a powerful tool for bridging the gap between conceptual models and operational systems.

Looking ahead, we envision expanding DEVS applications to encompass even more diverse IoT domains, emphasizing its scalability, flexibility, and integration potential with emerging technologies. By leveraging these strengths with the support of MDE methodology, DEVS is poised to play a pivotal role in the evolution of intelligent, adaptive systems within the rapidly advancing IoT ecosystem.

References

[1] Fortino, G., Savaglio, C., Spezzano, G., Zhou, M.: Internet of things as system of systems: A review of methodologies, frameworks, platforms, and tools. IEEE Transactions on Systems, Man, and Cybernetics: Systems **51**(1), 223–236 (2021) https://doi.org/10.1109/TSMC.2020.3042898

[2] Javed, A., Malhi, A., Kinnunen, T., Främling, K.: Scalable iot platform for heterogeneous devices in smart environments. IEEE Access **8**, 211973–211985 (2020) https://doi.org/10.1109/ACCESS.2020.3039368

[3] Booker, L., Forrest, S., Mitchell, M., Riolo, R.: Perspectives on Adaptation in Natural and Artificial Systems. Oxford University Press, Oxford, United Kingdom (2005). https://doi.org/10.1093/oso/9780195162929.001.0001

[4] Zhang, R., Sun, B.: Complex adaptive system theory, agent-based modeling, and simulation in dominant technology formation. Journal of Systems Engineering and Electronics **35**(1), 130–153 (2024) https://doi.org/10.23919/JSEE.2023.000160

[5] Zeigler, B.P., Muzy, A., Kofman, E.: Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations. Academic Press, San Diego, CA (2018)

[6] Zeigler BP, T.M. Mittal S: Mbse with/out simulation: State of the art and way forward. Systems **6**(4) (2018) https://doi.org/10.3390/systems6040040

[7] Zeigler, B.P., Mittal, S., Traoré, M.K.: Fundamental requirements and devs approach for modeling and simulation of complex adaptive system of systems: Healthcare reform. In: Proc. of the Symposium on Modeling and Simulation of Complexity in Intelligent, Adaptive and Autonomous Systems, Baltimore, MD (2021)

[8] Fattahi, A.: IoT System Design Process and Main Components, pp. 95–161 (2023). https://doi.org/10.1002/9781119787686.ch5

[9] Arslan, S., Ozkaya, M., Kardas, G.: Modeling languages for internet of things (iot) applications: A comparative analysis study. Mathematics **11**(5) (2023) https://doi.org/10.3390/math11051263

[10] Alavi Fazel, I., Wainer, G.: Discrete event system specification for iot applications. Sensors **24**(23) (2024) https://doi.org/10.3390/s24237784

[11] Hu, X., Zeigler, B.P., Couretas, J.: Devs-on-a-chip: implementing devs in real-time java on a tiny internet interface for scalable factory automation. In: 2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat.No.01CH37236), vol. 5, pp. 3051–30565 (2001). https://doi.org/10.1109/ICSMC.2001.971984

[12] Zeigler, H. B.P.; Sarjoughian: Guide to Modeling and Simulation of System of Systems. Springer, New York, NY (2017)

[13] Wymore, A.W.: A Mathematical Theory of Systems Engineering: The Elements. Krieger, Huntington, NY (1967)

[14] Kaplan, W.: Topics in mathematical system theory (rudolf e. kalman, peter l. falb and michael a. arbib). SIAM Review **12**(1), 157–158 (1970) https://doi.org/10.1137/1012030 https://doi.org/10.1137/1012030

[15] Bulcão-Neto, R., Teixeira, P., Lebtag, B., Graciano-Neto, V., Macedo, A., Zeigler, B.: Simulation of iot-oriented fall detection systems architectures for in-home patients. IEEE Latin America Transactions **21**(1), 16–26 (2023)

[16] Samuel, K.G., Bouare, N.-D.M., Maïga, O., Traoré, M.K.: A devs-based pivotal modeling formalism and its verification and validation framework. SIMULATION **96**(12), 969–992 (2020) https://doi.org/10.1177/0037549720958056 https://doi.org/10.1177/0037549720958056

[17] Uhrmacher, A.M.: Dynamic structures in modeling and simulation: a reflective approach. ACM Trans. Model. Comput. Simul. **11**(2), 206–232 (2001) https://doi.org/10.1145/384169.384173

[18] Zhang, W., Li, Q., Xu, X., Li, W.: Modeling and simulation of unmanned swarm system based on dynamic structure devs. Journal of Physics: Conference Series **2755**(012021), 1–18 (2024) https://doi.org/10.1088/1742-6596/2755/1/012021

[19] Belloli, L., Vicino, D., Ruiz-Martin, C., Wainer, G.: Building devs models with the cadmium tool. In: 2019 Winter Simulation Conference (WSC), pp. 45–59 (2019). https://doi.org/10.1109/WSC40007.2019.9004917

[20] Sehili, S., Capocchi, L., Santucci, J.F., Lavirotte, S., Tigli, J.Y.: Discrete event modeling and simulation for iot efficient design combining wcomp and devsimpy framework. In: Proceedings of the 5th International Conference on Simulation and Modeling Methodologies, Technologies and Applications. SIMULTECH 2015, pp. 26–34. SCITEPRESS - Science and Technology Publications, Lda, Setubal, PRT (2015). https://doi.org/10.5220/0005538300440052 . https://doi.org/10.5220/0005538300440052

[21] Risco-Martín, J.L., Prado-Rujas, I.-I., Campoy, J., Pérez, M.S., Olcoz, K.: Advanced simulation-based predictive modelling for solar irradiance sensor farms.

Journal of Simulation **0**(0), 1–18 (2024) https://doi.org/10.1080/17477778.2024.2333775

[22] Zhang, L., Zhao, C.: Modeling and Simulation Based Systems Engineering. WORLD SCIENTIFIC, Singapore (2023). https://doi.org/10.1142/12960 . https://www.worldscientific.com/doi/abs/10.1142/12960

[23] Capocchi, L., Santucci, J.-F., Tigli, J.-Y., Gomnin, T., Lavirotte, S., Rocher, G.: Actuation conflict management in internet of things systems devops: A discrete event modeling and simulation approach. In: Rey, G., Tigli, J.-Y., Franquet, E. (eds.) Internet of Things, pp. 189–206. Springer, Cham (2025)

[24] Capocchi, L.: DEVSimPy. https://github.com/capocchi/DEVSimPy. Software available on GitHub (2024)

[25] Capocchi, L., Santucci, J.F., Poggi, B., Nicolai, C.: Devsimpy: A collaborative python software for modeling and simulation of devs systems. In: 2011 IEEE 20th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, pp. 170–175 (2011). https://doi.org/10.1109/WETICE.2011.31

[26] M, C.J.E., Orna, G., Daniel, K., Doron, P., Helmut, V.: Model Checking. MIT Press, Cambridge, MA (2018)

[27] Fang, Z., Fu, H., Gu, T., Qian, Z., Jaeger, T., Hu, P., Mohapatra, P.: A model checking-based security analysis framework for iot systems. High-Confidence Computing **1**(1), 100004 (2021) https://doi.org/10.1016/j.hcc.2021.100004

[28] Widyawati, D.K., Ambarwari, A., Wahyudi, A.: Design and prototype development of internet of things for greenhouse monitoring system. In: 2020 3rd International Seminar on Research of Information Technology and Intelligent Systems (ISRITI), pp. 389–393 (2020). https://doi.org/10.1109/ISRITI51436.2020.9315487

[29] Harrand, N., Fleurey, F., Morin, B., Husa, K.E.: Thingml: A language and code generation framework for heterogeneous targets. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. MODELS '16, pp. 125–135. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2976767.2976812 . https://doi.org/10.1145/2976767.2976812

[30] Dominici, A., Capocchi, L., De Gentili, E., Santucci, J.-F.: Discrete event modeling and simulation of smart parking conflict management. In: 24th International Congress on Modelling and Simulation. Modsim'21, pp. 246–252. Modelling and Simulation Society of Australia and New Zealand, Sydney, Australia (2021). https://doi.org/10.36334/modsim.2021.E3.dominici

[31] Kate, C.: Internet of Things and Beyond: Cyber-Physical Systems. IEEE. https://iot.ieee.org/articles-publications/newsletter/may-2016/internet-of-things-and-beyond-cyber-physical-systemsy (2016)

[32] Zeigler, B.: Devs-based building blocks and architectural patterns for intelligent hybrid cyberphysical system design. Information **12**(12), 531 (2021)

[33] Castro, R., Marcosig, E.P., Giribet, J.I.: Simulation model continuity for efficient development of embedded controllers in cyber-physical systems. Complexity Challenges in Cyber Physical Systems: Using Modeling and Simulation (M&S) to Support Intelligence, Adaptation and Autonomy, 8193 (2019)

[34] Fazel, I.A., Wainer, G.: A devs-based methodology for simulation and model-driven development of iot. In: Guisado-Lizar, J.-L., Riscos-Núñez, A., Morón-Fernández, M.-J., Wainer, G. (eds.) Simulation Tools and Techniques, pp. 3–17. Springer, Cham (2024)

[35] Rainey, L.B., Holland, O.T. (eds.): Emergent Behavior in System of Systems Engineering: Real-World Applications, 1st edn. CRC Press, ??? (2022). https://doi.org/10.1201/9781003160816

[36] Risco-Martín, J.L., Mittal, S., Henares, K., Cardenas, R., Arroba, P.: xdevs: A toolkit for interoperable modeling and simulation of formal discrete event systems. Software: Practice and Experience **53**(3), 748–789 (2023) https://doi.org/10.1002/spe.3168 https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3168

[37] Capocchi, L., Santucci, J.-F., Fericean, J., Zeigler, B.P.: Devs model design for simulation web app deployment. In: 2022 Winter Simulation Conference (WSC), pp. 2154–2165 (2022). https://doi.org/10.1109/WSC57314.2022.10015469

[38] Trabes, G.G.: Efficient devs simulations design on heterogeneous platforms (2023) https://doi.org/10.22215/etd/2023-15536

[39] Lee, E., Seo, Y.-D., Kim, Y.-G.: Self-adaptive framework with master–slave architecture for internet of things. IEEE Internet of Things Journal **9**, 16472–16493 (2022) https://doi.org/10.1109/JIOT.2022.3150598

[40] Wang, Y., Zheng, L., He, J., Cui, Z.: Adaptive IoT Decision Making in Uncertain Environments . In: 2023 IEEE International Conference on Smart Internet of Things (SmartIoT), pp. 265–269. IEEE Computer Society, Los Alamitos, CA, USA (2023). https://doi.org/10.1109/SmartIoT58732.2023.00048 . https://doi.ieeecomputersociety.org/10.1109/SmartIoT58732.2023.00048

[41] An, H., Park, W., Park, S., Lee, E.: Logical space composition of iot for a scalable and adaptable smart environment. In: 2024 International Conference on Information Networking (ICOIN), pp. 614–618 (2024). https://doi.org/10.1109/ICOIN59985.2024.10572086

[42] Earle, B., Bjornson, K., Ruiz-Martin, C., Wainer, G.: Development of a real-time devs kernel: Rt-cadmium. In: 2020 Spring Simulation Conference (SpringSim), pp. 1–12 (2020). https://doi.org/10.22360/SpringSim.2020.CPS.002

[43] Risco-Martín, J.L., Mittal, S., Fabero, J.C., Malagón, P., Ayala, J.L.: Real-time hardware/software co-design using devs-based transparent ms framework. In: Proceedings of the Summer Computer Simulation Conference. SCSC '16. Society for Computer Simulation International, San Diego, CA, USA (2016)

[44] Hwang, K., Lee, M., Han, S., Yoon, J., You, Y., Kim, S., Nah, Y.: The devs integrated development environment for simulation-based battle experimentation. Journal of the Korea Society for Simulation **22**, 39–47 (2013) https://doi.org/10.9709/jkss.2013.22.4.039

[45] Matusek, D.: Towards resilient execution of adaptation in decentralized self-adaptive software systems. In: 2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), pp. 74–75 (2022). https://doi.org/10.1109/ACSOSC56246.2022.00036

[46] Alhirabi, N., Rana, O., Perera, C.: Security and privacy requirements for the internet of things: A survey. ACM Trans. Internet Things **2**(1) (2021) https://doi.org/10.1145/3437537

[47] Reggio, G.: A uml-based proposal for iot system requirements specification. In: Proceedings of the 10th International Workshop on Modelling in Software Engineering. MiSE '18, pp. 9–16. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3193954.3193956 . https://doi.org/10.1145/3193954.3193956

[48] Xiao, R., Wu, Z., Wang, D.: A finite-state-machine model driven service composition architecture for internet of things rapid prototyping. Future Generation Computer Systems **99**, 473–488 (2019) https://doi.org/10.1016/j.future.2019.04.050

[49] da Silva Fonseca, J.P., de Sousa, A.R., Souza Tavares, J.J.-P.Z.: Modeling and controlling iot-based devices' behavior with high-level petri nets. Procedia Computer Science **217**, 1462–1469 (2023) https://doi.org/10.1016/j.procs.2022.12.345 . 4th International Conference on Industry 4.0 and Smart Manufacturing

[50] Escamilla-Ambrosio, P.J., Robles-Ramírez, D.A., Tryfonas, T., Rodríguez-Mota, A., Gallegos-García, G., Salinas-Rosales, M.: Iotsecm: A uml/sysml extension for internet of things security modeling. IEEE Access **9**, 154112–154135 (2021) https://doi.org/10.1109/ACCESS.2021.3125979

[51] Doddapaneni, K., Ever, E., Gemikonakli, O., Malavolta, I., Mostarda, L., Muccini, H.: A model-driven engineering framework for architecting and analysing wireless sensor networks. In: 2012 Third International Workshop on Software Engineering

for Sensor Network Applications (SESENA), pp. 1–7 (2012). IEEE