The Journal of Systems and Software 234 (2026) 112728

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

An architecture framework for architecting IoT applications: From design to deployment

Moamin Abughazala [a,b,*], Mohammad Sharaf [a], Mai Abusair [a], Henry Muccini [b]

[a] Department of Computer Science Apprenticeship, An-Najah University, Palestine
[b] Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Italy

ARTICLE INFO

Editor: Dr. Christoph Treude

Keywords:
Internet of Things
Software Architecture
Model-Driven Engineering
Multi-View Architectural Modeling
Architecture-to-Code Generation

ABSTRACT

Context - The Internet of Things (IoT) refers to a distributed network of smart, connected devices that collaboratively sense, process, and act upon real-world environments. Designing such systems requires managing complex architectural concerns spanning software logic, hardware configuration, and spatial deployment, as well as validating non-functional properties like energy consumption and communication efficiency. **Objective** - To provide a unified, architecture-centric framework that supports the description, simulation, and automated code generation of IoT applications across software, hardware, and physical space dimensions. **Method** - We use Model Driven Engineering(MDE) approaches to develop CAPS, a framework that uniquely integrates multi-view architectural modeling, energy- and traffic-aware simulation via CupCarbon, and seamless generation of deployable Arduino code from high-level design models. **Result** - CAPS enables a traceable and cohesive development process from architectural design to physical deployment. Case studies from diverse domains demonstrate its ability to improve modeling expressiveness, maintain transformation fidelity, and reduce development time through automation. **Conclusion** - CAPS unifies architectural modeling, simulation, and code generation into a novel, end-to-end toolchain, addressing fragmentation in the IoT development lifecycle and enhancing early validation and traceability.

1. Introduction

The Internet of Things (IoT) is a rapidly growing field with the potential to change how we engage with our surroundings (Sundmaeker et al., 2010). It transforms everyday objects into a connected, intelligent network of devices that can exchange data and automate processes across a vast array of environments. From smart cities that monitor and optimize traffic flow to precision agriculture that provides real-time data on crop health, IoT systems are increasingly becoming the backbone of digital infrastructure (Malavolta et al., 2015).

Designing IoT systems, however, is challenging due to the inherent complexity of integrating diverse cyber-physical components and ensuring seamless interaction between software, hardware, and physical spaces. IoT developers must coordinate software behavior, hardware limitations, network interactions, and environmental conditions that influence performance, energy efficiency, and usability. Additional difficulties arise from maintaining consistency across design artifacts, evaluating non-functional properties early in the lifecycle, and ensuring traceability from architectural models to executable implementations (McEwen and Cassimally, 2013; Abdmeziem et al., 2015; Jajodia et al., 2010).

Existing tools and frameworks typically address only subsets of these concerns. For example, ThingML supports model-driven development of embedded software components and partial code generation (Harrand et al., 2016); UML4IoT (Thramboulidis and Christoulakis, 2016) and CHESSIoT (Ihirwe et al., 2023) provide modeling constructs for software architecture abstraction in the IoT domain; and CupCarbon (Bounceur, 2016) enables the simulation of wireless sensor networks with spatial deployment and energy analysis. Yet these tools are often used in isolation and rarely offer unified, end-to-end support spanning architectural modeling, simulation, hardware specification, spatial deployment, and automated code generation. The resulting fragmentation yields disjointed workflows, repeated manual specification, and limited validation of system-wide behavior in early stages. A unified, architecture-centric framework that integrates software, hardware, and physical domains-while ensuring semantic consistency and traceability across the development lifecycle-remains lacking.

* Corresponding author.
E-mail addresses: moamin.abughazala1@univaq.it, m.abughazaleh@najah.edu (M. Abughazala), sharaf@najah.edu (M. Sharaf), mabuseir@najah.edu (M. Abusair), henry.muccini@univaq.it (H. Muccini).

https://doi.org/10.1016/j.jss.2025.112728
Received 20 March 2025; Received in revised form 31 October 2025; Accepted 1 December 2025
Available online 10 December 2025

To address this gap, we introduce *CAPS* (Cyber-physical Architectural Platform for IoT). This unified, model-driven engineering framework integrates architectural modeling, simulation, and deployment within a single, architecture-centric workflow. CAPS adopts a multi-view approach aligned with domain-specific concerns: *software logic, hardware topology, and spatial deployment*. It provides traceable transformations between these views, supports simulation of energy consumption and communication traffic via CupCarbon, and generates Arduino-compatible deployment code directly from the model, enabling early validation and streamlined transition to implementation.

Beyond integrating our earlier CAPS results, this journal article uses Mapping View Modeling Languag *MAPML/* Deployment View Modeling Language *DEPML* for end-to-end cross-view *traceability*, *formalized* model-to-simulation and model-to-code transformation pipelines based on versioned, semantics-aware templates, and a *Textual Modeling Language (TML)* with IDE-backed static checks. It further broadens the empirical evaluation (three case studies and transformation conformance checks) and adds a systematic comparison to the state of the art. We signpost reused background versus extended/new material in Sections 3–6 and report quantitative evidence in Sections 8–11.

Novel contributions of this journal extension include:.

- **Multi-view architectural modeling:** CAPS integrates software (SAML), hardware (HWML), and physical space (SPML) in line with ISO/IEC/IEEE 42010 (ISO/IEC/IEEE, 2022).
- **Cross-view traceability:** Two auxiliary languages-*MAPML* (model-artifact provenance) and *DEPML* (dependency and deployment flow)-capture structural and semantic links across views and generated artifacts, supporting consistency checks and traceable transformations.
- **Formalized transformation pipelines:** We define and operationalize metamodel-driven, template-based transformations from models to CupCarbon simulation artifacts and to deployable C++/Arduino code, preserving behavioral semantics such as timing behavior, sensing/tx frequency, and energy modes.
- **Executable simulation for early validation:** CAPS automatically generates simulation artifacts for CupCarbon, enabling design-time exploration of network traffic, radio propagation, energy consumption, and deployment feasibility.
- **Textual Modeling Language (TML):** We introduce a textual DSL and accompanying editor that complements graphical modeling with features such as autocompletion, static checks, model previews, and scalability to large system designs.
- **Comprehensive evaluation across real-world case studies:** We evaluate CAPS on three real CPS deployments (NdR, UFFIZI, VASARI), demonstrating model-to-code correctness, simulation/deployment consistency, and energy-behavior trend preservation.
- **Systematic comparative analysis:** We position CAPS relative to existing MDE frameworks (ThingML, UML4IoT, etc.), highlighting its broader scope, deeper integration of physical aspects, and bidirectional traceability (see Tables 10-11).

This journal version unifies previously separate prototypes into a single, traceable toolchain; uses MAPML/DEPML for cross-view integration; elevates TML; formalizes the model-to-simulation and model-to-code pipelines; and expands both empirical evaluation and comparative analysis to meet journal-level novelty and rigor. These capabilities ensure consistency across views, support design-space exploration, and simplify the transition from model to deployment. CAPS further distinguishes itself by supporting physical-space modeling with attenuation-aware deployment-an aspect absent from most existing IoT modeling frameworks. This enables informed architectural decisions based on simulation feedback while reducing manual development overhead.

Finally, we present the design and implementation of CAPS and evaluate its capabilities through three real-world case studies: NdR (a smart event), UFFIZI (museum crowd management), and VASARI (urban monitoring). These case studies demonstrate flexibility, usability, and effectiveness from early design to physical deployment.

The rest of the paper is organized as follows: Section 2 provides background information on the IEEE/ISO/IEC 42,010 architecture description standard, the CupCarbon Simulator, and Prior Foundations. Section 3 presents an overview of the. Section 4 presents the modeling languages. Section 5 details the simulation approach, its process, and the transformational approach. Section 6 presents the Arduino code generation process and tool. Section 7 applies the modeling, simulation, and Arduino code generation approach to the UnivAq Street Science application. Section 8 shows the evaluation, while Section 10 discusses some results. Section 11 presents related work. Finally, Section 12 concludes the paper.

2. Background & prior foundations

This section summarizes the standards and tools relevant to our work (Sections 2.1 and 2.2) and clarifies how earlier CAPS artifacts relate to this journal version, including the gaps it addresses (Section 2.3). We explicitly distinguish material recapped for completeness from the elements that are extended or newly introduced later (Sections 4–6) and evaluated comparatively and empirically (Sections 11–8).

2.1. Standards background: ISO/IEC/IEEE 42010

CAPS builds upon ISO/IEC/IEEE 42010:2022, *Systems and software engineering-Architecture description* (ISO/IEC/IEEE, 2022), which structures architecture descriptions in terms of stakeholders, concerns, viewpoints, views, and correspondence rules. Fig. 1 illustrates the content model prescribed by the standard, which we operationalize via multiple architectural views and explicit correspondence across them. The core metamodels (SAML/HWML/SPML) summarized later follow this scheme and are included here for context; the journal-specific additions (traceability models and formalized pipelines) are introduced in Sections 4 and 6.

2.2. Tools background: CupCarbon IoT/WSN simulator

CupCarbon (Bounceur, 2016) is an open-source platform for modeling and simulating IoT/WSN deployments (e.g., smart-city scenarios). It supports multiple wireless standards (e.g., Wi-Fi, ZigBee, LoRa) and enables early assessment of *energy consumption*, *data traffic*, and *spatial layouts* prior to physical deployment. These capabilities make it a suitable target for our model-to-simulation transformations used to reason about non-functional properties early in the design process (see Section 5).

2.3. Prior CAPS foundations and gap analysis

Earlier publications presented (i) CAPS multi-view modeling concepts and editors (software, hardware, and spatial views), (ii) a prototype transformation to CupCarbon for simulation, and (iii) a prototype Arduino code generator (Muccini and Sharaf, 2017b; Sharaf et al., 2017, 2018a) Sharaf et al. (2017). These artifacts established the feasibility of an architecture-centric approach across parts of the IoT lifecycle and are *recapped here for completeness*.

Despite these foundations, important limitations remained:

- Absence of a *unified toolchain* with explicit cross-view correspondence to maintain consistency across software, hardware, and spatial deployment.
- Transformations *not formalized* as metamodel-driven, template-based pipelines with semantics notes or invariants for modes, ports, and links.
- Lack of a first-class *textual workflow* to complement graphical editing for modeling at scale.

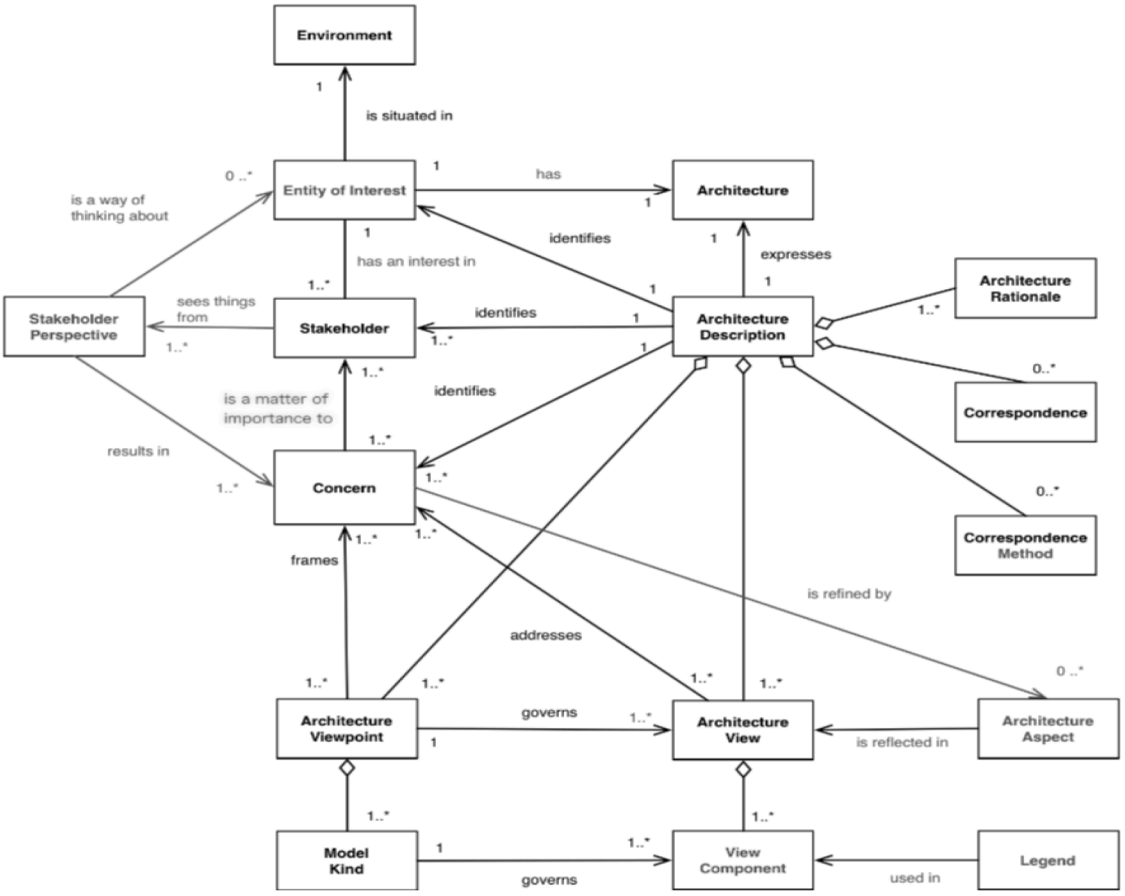


Fig. 1. Content model of an architecture description (following ISO/IEC/IEEE 42010:2022).

- No *systematic comparative positioning* against prominent IoT MDE frameworks.
- Limited *quantitative evaluation* beyond single-case demonstrations.

This journal article addresses the above by:

- *Utilizing* auxiliary mapping languages **MAPML** (model-artifact provenance across software-to-hardware) and **DEPML** (dependency/flow across hardware-to-space) to enable end-to-end *cross-view traceability* within a single framework (Section 4).
- *Formalizing* metamodel-driven, template-based **model-to-simulation** and **model-to-code** transformation pipelines with semantics-aware constraints (timing, port multiplicity, energy modes, spatial attenuation) (Sections 5 and 6).
- *Presenting* a **Textual Modeling Language (TML)** with IDE support (autocomplete, static checks, auto-validation) as a scalable counterpart to graphical editors (Section 4.3).
- *Providing* a broadened **evaluation** (three real-world case studies), a **user study** on modeling productivity, and **transformation-conformance** checks aligning simulation with deployed behavior (Section 8).
- *conducting* a **systematic comparative analysis** that positions CAPS relative to established IoT MDE frameworks, with quantitative coverage and traceability metrics (Section 11).

For clarity, Table 1 maps prior artifacts to their identified gaps and to the elements contributed by this journal version.

3. Overview

This research presents (Cyber-physical Architectural Platform for IoT) as a unified, model-driven toolchain that spans multi-view architectural modeling, cross-view traceability, simulation, and deployable code generation within a single framework. Unlike approaches that treat software, hardware, or deployment in isolation, CAPS integrates software behavior, hardware topology, and spatial deployment into a cohesive multi-view architecture. Fig. 2 provides an overview of the CAPS framework, showing the flow from architectural modeling through traceability, simulation, and code generation.

Modeling IoT systems must address diverse stakeholder concerns (e.g., software engineers, integrators, IoT specialists, and spatial modelers) and non-functional goals such as energy efficiency, communication reliability, and environmental coverage (Fig. 3). IoT designs are strongly influenced by time and space, bringing challenges like data exchange frequency, coverage constraints, topology selection, and power consumption. addresses these concerns by defining and integrating three viewpoints derived from ISO/IEC/IEEE 42010 (ISO/IEC/IEEE, 2022) and insights from prior work on adaptive and component-based IoT systems (Malavolta et al., 2015; Crnkovic et al., 2016):

- **SAML** (Software Architecture Modeling Language): component interactions, communication patterns, and behavioral modes;
- **HWML** (Hardware Architecture Modeling Language): device capabilities, energy sources/usage, sensors/actuators, and connectivity;
- **SPML** (Spatial Deployment Modeling Language): physical arrangement of nodes, range, and attenuation in the target environment.

3.1. Modeling

The CAPS framework introduces a multi-view architectural modeling approach structured around three primary viewpoints. It *utilizes* the auxiliary mapping languages MAPML (software ↔ hardware) and DEPML (hardware ↔ space) to maintain cross-view traceability and present

Table 1
From prototypes to the journal article: prior CAPS artifacts, identified gaps, and the additions provided here.

Prior item	Gap	This journal article adds
Multi-view modeling	Weak traceability between views	**MAPML/DEPML** MAPML/DEPML for explicit cross-view mapping (Section 4)
Model-to-simulation	Ad-hoc scripts; missing semantics	**Formalized** transformation pipeline (Section 5)
Model-to-code (Arduino)	Default mappings unclear	**Semantics-aware, traceable model-to-code pipeline** (Section 6)
Graphical editing only	Limited scalability for large models	**Textual Modeling Language (TML)** with IDE support (Sections 4, 6)
Limited empirical evaluation	Only single-case demonstrations; lacked comparative evidence	**Three case studies**, simulation-deployment comparisons, and energy-behavior consistency validation (Section 8)
No systematic comparative positioning	Fragmented relation to existing MDE frameworks	**Comparative analysis** Quantitative comparison against ThingML, UML4IoT, CHESSIoT, showing broader scope and integration (Section 11)

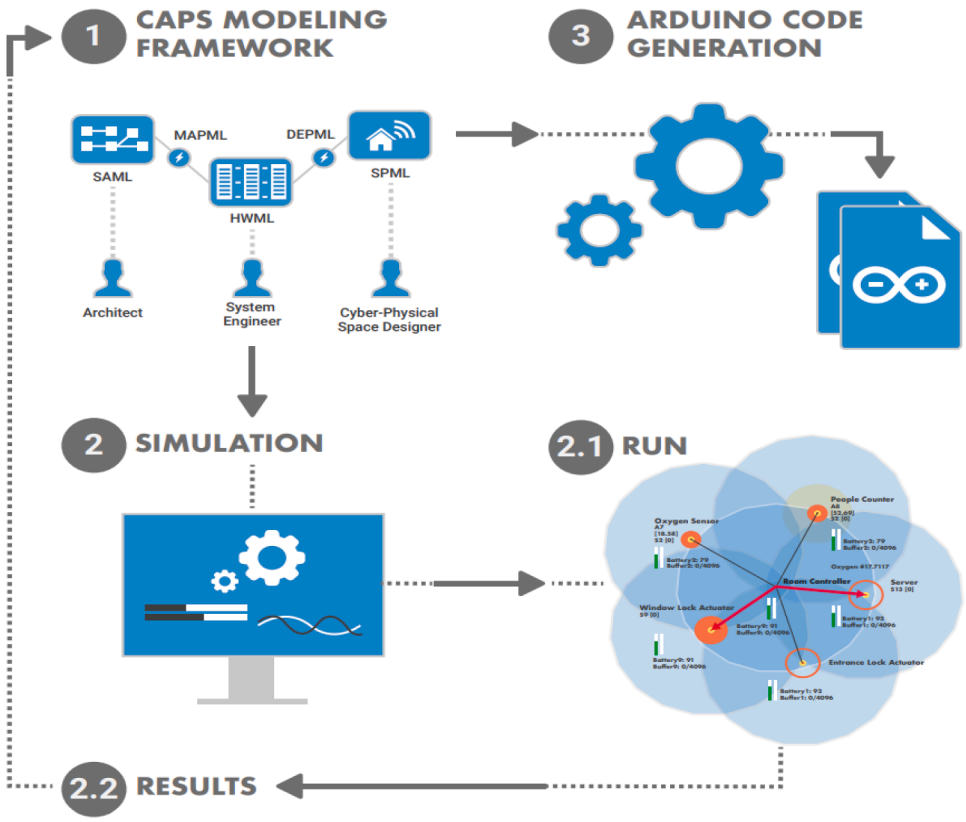


Fig. 2. Overview of the CAPS framework.

a Textual Modeling Language with IDE support (autocomplete, static checks, auto-validation) as a first-class, scalable counterpart to graphical editors.
The three viewpoints (SAML, HWML, SPML), originally introduced in prior work (Muccini and Sharaf, 2017b), are systematically integrated here into a unified, executable framework. This integration allows stakeholders to explore designs holistically and to keep software, hardware, and spatial decisions consistent through explicit correspondence rules and mappings. Section 4 details the modeling stack, including the role of MAPML/DEPML and TML in high-throughput editing and reuse.

3.2. Simulation

We *formalize* a metamodel-driven, template-based model-to-simulation pipeline with the aim of preserving behavioral semantics for modes, ports, and links, improving repeatability and fidelity over earlier generators.
integrates CupCarbon to evaluate candidate architectures under realistic deployment conditions, enabling early assessment of *energy consumption* and *data traffic* and supporting design-space exploration before physical rollout. Section 5 details the simulation process and the transformation pipeline.

3.3. Arduino code generation

We *systematize* the model → code pipeline (Arduino) with explicit protocol defaults and traceable mappings from architectural constructs to executable behavior.
transforms architectural models into deployable Arduino-compatible code, aligning implementation with design decisions and reducing development effort through automation and reuse. Section 6 describes the code generation process and its role in ensuring implementation consistency with the modeled architecture.

4. CAPS modeling framework

The CAPS modeling framework enables a structured, multi-layered architectural description of IoT systems. It separates concerns into three

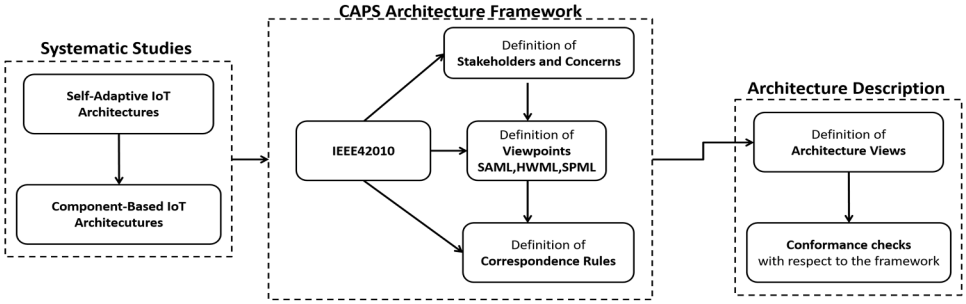


Fig. 3. Overview of the CAPS elicitation process.

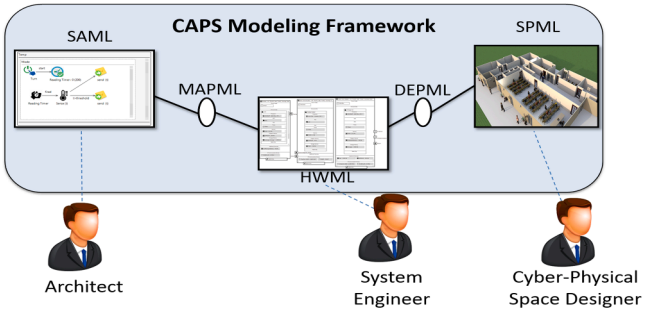


Fig. 4. IoT views and stakeholders.

coordinated primary views-software (SAML), hardware (HWML), and spatial deployment (SPML)-aligned with the ISO/IEC/IEEE 42,010 standard for architectural modeling. To ensure consistency and traceability between views, CAPS introduces two auxiliary mapping languages (MAPML and DEPML) , as shown in Fig. 4. A Textual Modeling Language (TML) further complements graphical modeling by supporting scalable, scriptable modeling workflows.

4.1. Multi-view architectural model

CAPS defines three coordinated architectural viewpoints, each capturing a specific dimension of IoT systems:

- **SAML (Software Architecture Modeling Language)**[1] specifies the logical structure and behavior of the software layer. It defines components, behavioral modes, message ports (input/output), and communication links (unicast, multicast, broadcast). Each component may operate in one or more modes, with distinct actions triggered by events or conditions.
  - Structural metamodel: Components and their interactions are modeled using typed ports and links (see Fig. 5).
  - Behavioral metamodel: Modes represent operational states; transitions model condition-based changes, such as entering an energy-saving mode (see Fig. 6).

  SAML captures the control logic and communication behavior of IoT applications, enabling precise modeling of dynamic behavior, energy states, and interactions among software components.
- **HWML (Hardware Architecture Modeling Language)**[2] describes the hardware setup, including microcontroller units (MCUs), radios, power sources, sensors, actuators, and communication parameters. Each hardware configuration includes properties like:
  - Energy sources and consumption models
  - Communication interfaces (ZigBee, Wi-Fi, LoRa, etc.)
  - Protocol parameters: data rate, transmission power, range, latency

  Fig. 7 illustrates the HWML metamodel that represents the structure of IoT devices, including radios, sensors, and energy attributes. HWML enables accurate hardware specification, supporting simulation assumptions, code generation, and compatibility checks with the software and spatial views.
- **SPML (Spatial Deployment Modeling Language)** [3] captures the physical environment where devices are deployed: rooms/areas with coordinates, obstacles (e.g., walls) with material attenuation, elevation, and geometric constraints for coverage/connectivity. The root **CyberPhysicalSpaces** class represents the deployable 3D/2D environment; elements are defined by names, coordinates, dimensions, elevation, mobility, doors/windows, angles, and material attenuation coefficients (0-1). To avoid reimplementing 3D editing,[4] integrates Sweet Home 3D (SWEETHOME-SWEET) for spatial visualization and editing (see Fig. 8).

4.2. Cross-view traceability via MAPML and DEPML

The auxiliary mapping views ensure consistency among software, hardware, and spatial perspectives:

- **MAPML** links SAML elements (components, ports, modes) to HWML entities (nodes, interfaces, hardware modes), enabling checks that logical interactions are realizable on the selected hardware. A **Mapping** aggregates *HW mappings* (component→HW) and refined links such as *SensingMapping*, *ActuatingMapping*, *CommunicationDeviceMapping*, and *ModeMapping*.
- **DEPML** relates HWML hardware configurations to SPML spatial locations via *DeploymentLinks*. Each link specifies where a hardware configuration is instantiated, supporting multiplicity for repeated deployments and enabling spatial feasibility checks (range, attenuation, obstacles).

These mappings provide explicit correspondence rules used for (i) automated conformance checks, (ii) traceable model-to-simulation generation, and (iii) traceable model-to-code generation.

4.3. Textual modeling language (TML)

The Textual Modeling Language (TML) is a scripting-based alternative to graphical models like SAML, HWML, and SPML. It allows for flexible and scalable modifications to models programmatically, making it efficient for those familiar with programming. TML facilitates rapid

[1] The original SAML metamodel was introduced in Muccini and Sharaf (2017b) and is summarized here for completeness.

[2] The HWML metamodel originates from Muccini and Sharaf (2017b); a summary is included below to maintain context.

[3] The SPML metamodel was previously presented in Muccini and Sharaf (2017b); it is recapped here for continuity.

[4] Detailed information on spatial tooling integration (via Sweet Home 3D) is available in Muccini and Sharaf (2017a).

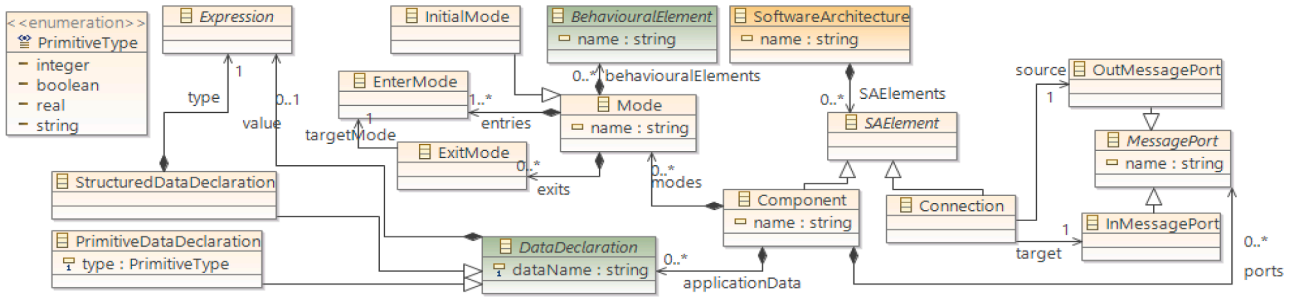


Fig. 5. SAML metamodel: structural concepts (external metaclasses in green). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

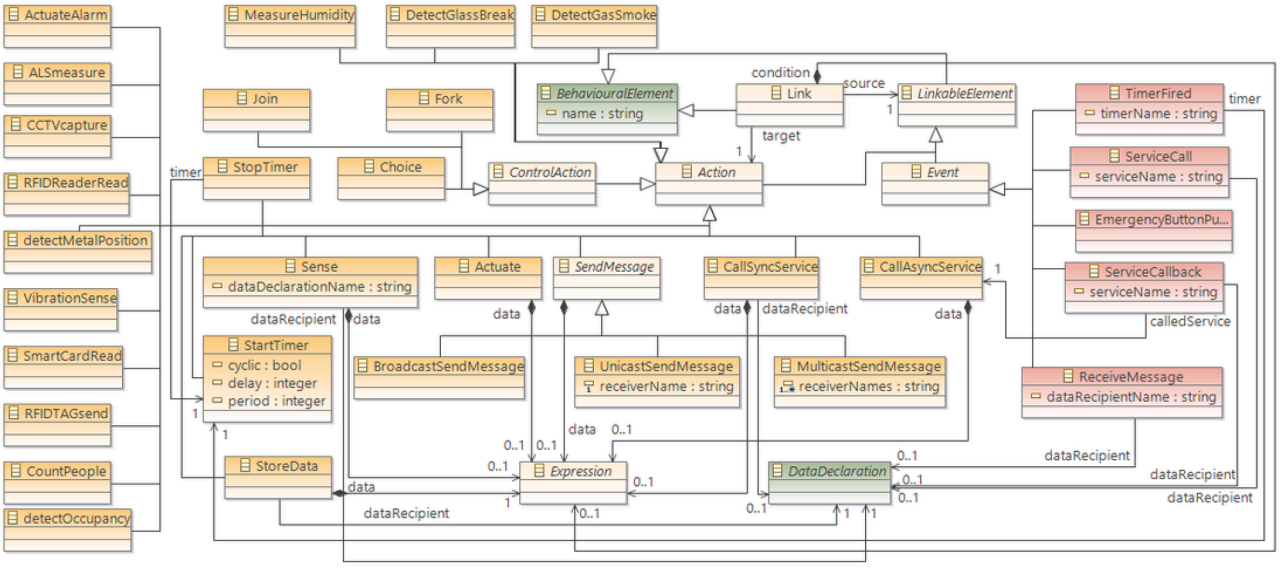


Fig. 6. SAML metamodel: behavioral concepts (actions in orange, events in red). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

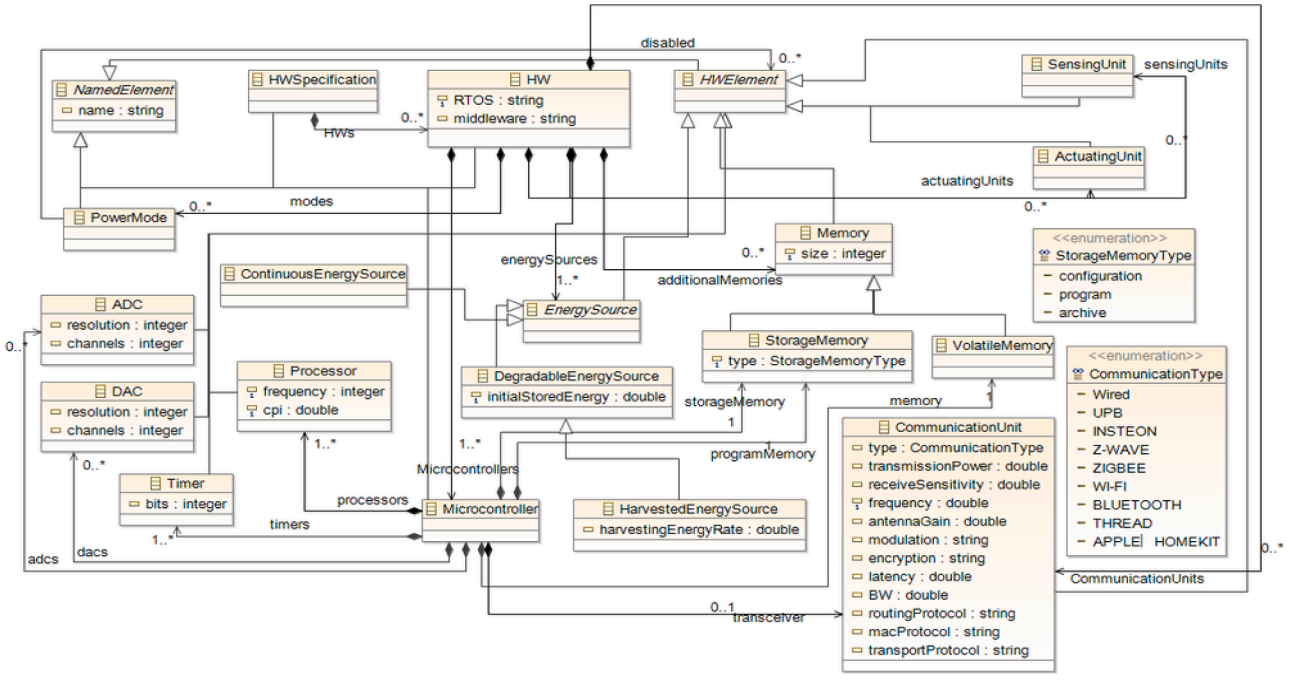


Fig. 7. HWML metamodel.

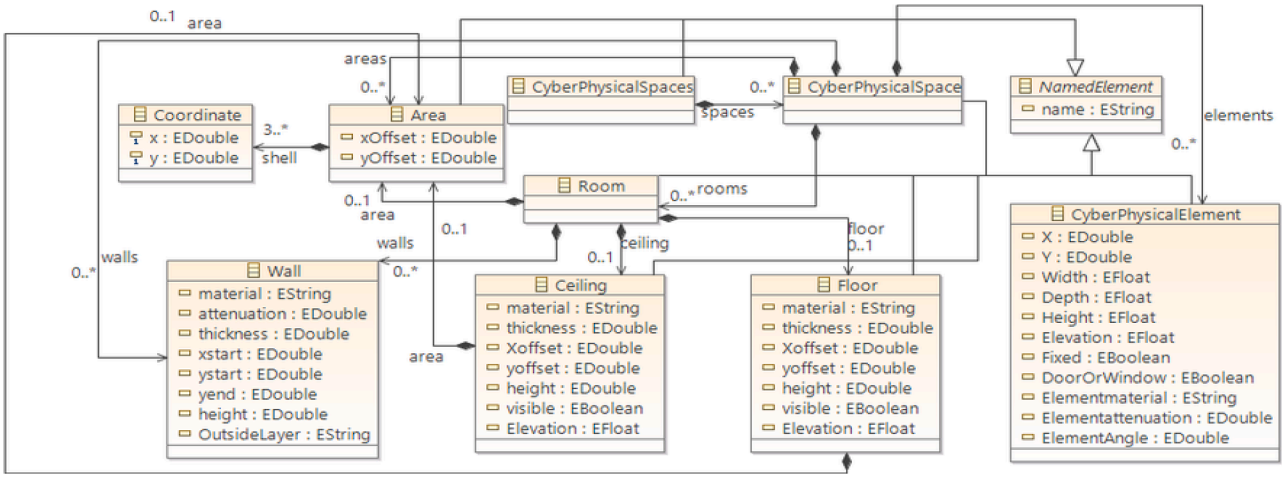


Fig. 8. SPML metamodel.

```
grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals

import "components"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

SoftwareArchitecture returns SoftwareArchitecture:
    {SoftwareArchitecture}
    'SoftwareArchitecture'
    name=EString
    '{'
        ('SAElements' '{' SAElements+=SAElement ( "," SAElements+=SAElement)* '}' )?
    '}';

SAElement returns SAElement:
    Component | Connection;

MessagePort returns MessagePort:
    InMessagePort | OutMessagePort;

Mode returns Mode:
    Mode_Impl | InitialMode;

DataDeclaration returns DataDeclaration:
    PrimitiveDataDeclaration | StructuredDataDeclaration;

BehaviouralElement returns BehaviouralElement:
    EnterMode | ExitMode | CallAsyncService | Human | Alarm | Server | SmartCard | SenseVibration
    | RfidTag | TouchScreen | ReadRfid | ReadSmartCard | EmergencyButton | SenseOccupancy | Start
    | SenseInductive | SenseHumidity | SenseGlassBreakage | SenseOxygen | SensePosition | SenseGyroscope
    | SenseAcoustic | SenseAccelerometer | SenseVisual | SenseWind | SenseTemperature | SenseAmbientLight
    | SenseGasSmoke | CountPeople | CCTVCcapture | CallSyncService | StartTimer | StopTimer | StoreData
    | BroadcastSendMessage | UnicastSendMessage | MulticastSendMessage | Sense | Actuate | Join | Fork
    | Choice | ServiceCallback | ServiceCall | ReceiveMessage | TimerFired | Link;

Expression returns Expression:
    IntegerConst | BooleanConst | RealConst | StringConst | EnumConst | StructureConst | NullExp
    | DataRef | StructureMemberRef | ArithmeticExpr | BooleanExpr | RelationalExpr;
```

Fig. 9. Part of CAPS grammar file.

modifications and batch changes to models, which is particularly beneficial in complex projects or when integrating with other automation tools. Within TML, the CAPS platform includes a textual editor with autocomplete and autosuggestion features for predefined classes like components and connections. Additionally, it offers auto-validation based on the grammar file, ensuring accuracy and consistency in model creation and modification.

Fig. 9 illustrates a portion of the grammar file that underpins the Textual Modeling Language (TML) used in CAPS. This grammar defines the syntactic structure for elements such as components, connections, ports, and actions, enabling CAPS to parse and validate textual models with high accuracy. It serves as the foundation for features like autocomplete, static checking, and automated transformation from textual definitions to graphical representations or simulation inputs.

Textual modeling language is a precise and accessible way to describe complex systems and data structures using human-readable text. It provides a comprehensive and expressive approach for documenting complex systems, making it possible to capture the nuances of various domains with unmatched precision and versatility (Mazanec and Macek, 2012). In software engineering, textual modeling languages present numerous benefits (Grönninger et al., 2014): Readability, Integration, Tool development, and Collaboration. With Textual Modeling Language, we can define a component with an average of 30 lines of code. Most of these lines are auto-suggested, while the rest require the user to provide names to be filled between the double quotation marks, see Fig. 10.

While the modeling framework in CAPS involves three domain-specific languages and multiple mappings, this structure is necessary to capture the multi-view nature of IoT systems, including behavior, hardware topology, and spatial deployment. To balance this complexity, CAPS offers both graphical (via Eugine) and textual (via Xtext-based DSLs) modeling environments. While graphical modeling aids intuitive

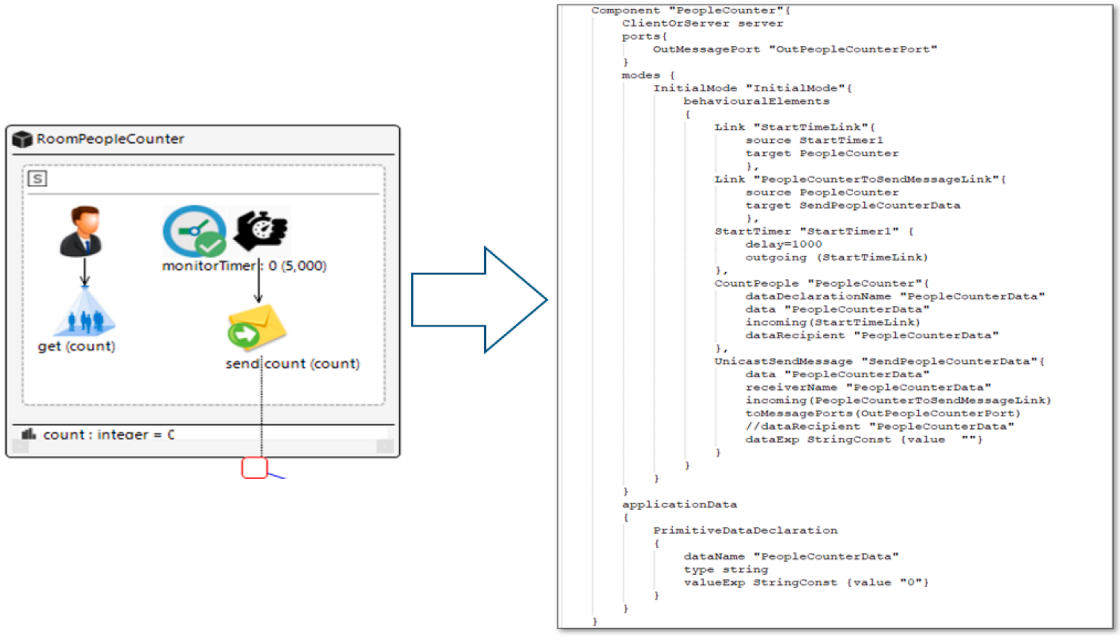


Fig. 10. Graphical and textual modeling languages.

visualization, the textual interface significantly enhances productivity for larger or repetitive architectures by supporting batch editing, reuse, and rapid iteration. This dual modeling strategy allows designers to choose the most appropriate interface based on system complexity and personal expertise, ultimately improving usability without sacrificing modeling precision.

5. CAPS simulation

Simulation, in CAPS, requires transforming CAPS models into the equivalent CupCarbon project. The CAPS code generation framework for CupCarbon comprises the parsing, analysis, script generation, and project generation activities to produce files used to build the CupCarbon project.

We *formalize* a metamodel-driven, template-based (**model-to-simulation**) pipeline targeting CupCarbon and propagate *trace links* from SAML, HWML, and SPML via *MAPML and DEPML* into all generated artifacts. The pipeline is designed to preserve behavioral semantics (modes, ports, links), radio and energy parameters, and spatial attenuation, thereby improving repeatability and explainability beyond earlier prototypes.

Fig. 11 illustrates the framework and its code generator. By executing the CupCarbon project on the CupCarbon simulator, the performance of the CAPS architecture in terms of energy consumption, battery level, and data traffic can be evaluated.

5.1. Scope and inputs

Given a validated design (Section 4), the generator consumes:

- SAML (software): components, ports, links, modes, timers;
- HWML (hardware): radio parameters, energy sources, driver defaults;
- SPML (space): coordinates, obstacles, material attenuation coefficients;
- MAPML/DEPML (mappings): component to device and device to location bindings;

5.2. Formalized template pipeline

The pipeline is specified as a sequence of metamodel-to-template steps (templates are reusable and versioned), each step emitting CupCarbon artifacts and recording traceability:

S1 - Topology synthesis (SPML and DEPML): Instantiate nodes at SPML coordinates according to DEPML deployment links; export obstacles and material coefficients to the map layer.

S2 - Device and radio configuration (HWML): Parameterize node radios (power, frequency, and bitrate, RX and TX current, duty cycle) and energy sources (initial charge) from HWML.

S3 - Application behavior mapping (SAML and MAPML): Translate components, modes, and ports into event/state handlers and traffic intents; MAPML bindings select the executing device and bind logical ports to physical interfaces.

S4 - Sensing and traffic schedules (SAML and HWML): Derive periodic and aperiodic transmissions from SAML events and timers and HWML timers; generate sampling and actuation tasks with jitter and offsets as specified.

S5 - Packaging: Generate an executable CupCarbon simulation project with topology, node configurations, and behavior scripts that link CupCarbon entities back to SAML, HWML, SPML, MAPML, and DEPML element IDs for debugging and explainability.

5.3. Semantics alignment

We align key semantic aspects so simulation outcomes reflect modeled intent:

- Communication: SAML links to CupCarbon routes; port direction and multiplicity (uni, multi, and broadcast) preserved. Backoff and retry policies come from HWML.
- Temporal behavior: SAML events/timers and HWML timers to task schedules; mode changes generate enable and disable hooks so only the active mode executes.
- Energy model: Energy is computed from HWML parameters and CupCarbon per-event costs.
- Spatial attenuation: SPML obstacles (material, thickness) contribute to path loss on links crossing their polygons; range checks use SPML coefficients with CupCarbon's propagation model.

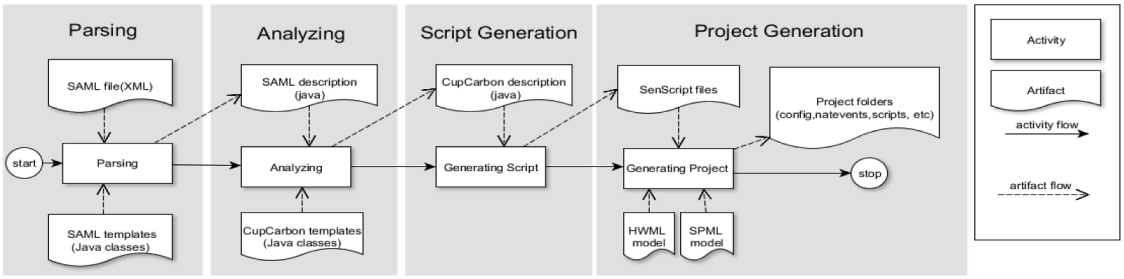


Fig. 11. CAPS automatic code generation framework.

Table 2
Mapping rules & invariants (Components, modes, links, hardware, physical).

Construct	Mapping Simulation	Invariant & Check
Components	Node/module per component. module/class with `init/loop`.	1:1 correspondence. Metamodel conformance or fail with the source pointer.
Modes	Control flow is expressed as a mode-driven state machine; a state variable selects the guarded block to execute, and predicates trigger transitions between modes.	"No handler in inactive mode." Transitions occur only when guard holds (e.g., `set mod 1/0`). Unguarded handlers warned/blocked.
Links (ports)	Simulation uses channels; code binds to typed messaging interfaces with structured payloads (e.g., `send $p 3`).	Direction, multiplicity, and payload schema preserved; incompatible links rejected with diagnostic.
Hardware params	CupCarbon radio/device params (e.g., `radio_standard: ZIGBEE`, `radio_data_rate: 250000`)	Required interfaces present; Respecting resource constraints.
Physical params	spatial config in sim (e.g., `device_longitude`, `device_latitude`, `device_elevation`);	Spatial attribution preserved in simulation; missing SPML emits warning and defaults are explicit.

5.4. Transformation engine: Implementation and correctness strategy

The CAPS transformation engine is metamodel-driven, rule-based, and template-based. It realizes the S1-S5 pipeline (topology synthesis; device/radio configuration; behavior mapping; scheduling; packaging) and records trace links from architectural views (SAML/HWML/SPML) through mapping views (MAPML/DEPML) into generated *simulation* artifacts (CupCarbon).

Mapping Rules and Invariants This section walks through a single component end-to-end to illustrate how the transformation engine materializes the architectural views into executable artifacts. The engine applies metamodel-to-artifact mappings guarded by explicit invariants and generator checks (see Table 2).

Fig. 15 shows how *hardware parameters* declared in HWML (e.g., radio standard, data rate, interfaces, and energy model) are mapped to the corresponding CupCarbon device configuration. Fig. 16 complements this by mapping the component's *physical parameters* from SPML (coordinates, elevation, and attenuation/obstacles) into the simulation's spatial configuration for the same device. Finally, Fig. 17 presents the generated Senscript that captures the component's *software behavior*: the file represents the entire component, including the control logic as a mode-driven controller. Within this script, mode guards select the active region, and transitions are effected only when their guard predicates hold, ensuring exclusive execution of handlers under the current mode.

Correctness Assurance We assure correctness at three levels. *Syntactic* checks validate models and templates against their grammars. *Structural* checks enforce metamodel conformance and cross-view correspondence (so what is wired in the architecture is what is generated). *Semantic* checks preserve behavior-communication, timing/scheduling, energy assumptions, and spatial effects-via guarded dispatch, protocol/-type validation, and simulator/code-level invariants.

Construct Coverage We summarize which modeling constructs the engine translates into simulation/code artifacts and to what extent they are supported.

SAML components; provided/required ports; links (uni-/multi-/broadcast); modes and guarded behavior; timers (period/offset); simple data payloads (typed fields). *Checks/Invariants:* directionality/multiplicity preserved; exclusive-mode execution; timer periods/offsets validated. *Current constraints:* payloads are structural (no user-defined serialization logic); hierarchical components flattened at generation time.

HWML device type; MCU (Microcontroller Unit) class; radio standard and data rate; sensor/actuator bindings; energy model parameters; interface bindings; memory footprints (RAM/flash). *Checks/Invariants:* required interfaces present; RAM/flash/pin capacity not exceeded; device-protocol compatibility enforced; drivers selected by binding. *Current constraints:* one sensing unit per simulated node (CupCarbon export); when protocol is unspecified, code generation applies a safe default (e.g., SPI) that can be overridden in HWML.

SPML device coordinates (Latitude / Longitude), elevation; obstacles/attenuation parameters; deployment topology. *Checks/Invariants:* spatial attribution preserved in simulation; missing values trigger explicit defaults with warnings. *Current constraints:* physical effects are realized in simulation only (not enforced in firmware), though constants may be emitted for calibration.

Bindings (MAPML/DEPML) component ↔ device bindings and port ↔ interface/protocol bindings; deployment associations. *Checks/Invariants:* bound elements must exist and be type-compatible; every generated artifact carries a trace link to its source model element.

In summary, the current engine covers the core SAML/HWML/SPML constructs used in our case studies, with two documented constraints (one sensor per simulated node; default protocol selection in code generation) and simulator-only realization of physical effects. These constraints are engineering choices and are straightforward to relax by extending the templates.

Validation Protocol Beyond unit checks in the generator, we verify semantic preservation by (1) comparing simulated energy/behavior against modeled modes and (2) cross-referencing outcomes with hardware results reported in Section 8. This provides an end-to-end oracle for faithfulness.

Calibration Option CAPS supports an optional, one time calibration of the energy model: (i) profile per state currents on the target board (sleep, CPU active, sensor warm up/sampling, RX, TX at each power level); (ii) import measured duty cycles (timers, message intervals/retries) from

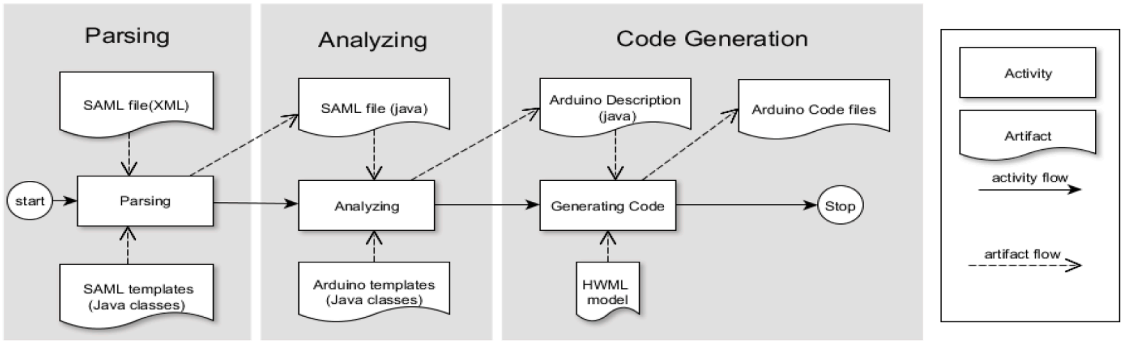


Fig. 12. Arduino automatic code generation framework.

deployment logs; (iii) set regulator efficiency and cable/connector loss; and (iv) enable variable TX power and obstacle aware path loss in SPML. When enabled, the simulator reports energy normalized as J/min and J/message and computes deviation vs. measurements (MAPE).

6. CAPS code generation (Arduino)

The CAPS Arduino code generation Sharaf et al. (2018a) is a framework that converts CAPS models into Arduino code that can be installed on Arduino boards Arduino (2022) . Arduino boards are versatile microcontrollers known for their ability to interact with sensors, actuators, and other devices via digital and analog I/O pins. Running the converted codes on Arduino boards allows us to evaluate the CAPS architecture in real environments. The CAPS models participate in three activities: parsing, analyzing, and generating Arduino code. Fig. 12 illustrates the entire code generation process.

We *systematize* a metamodel-driven, template-based **model-to-code** pipeline that carries *trace links* from SAML, HWML, and SPML via *MAPML and DEPML* into generated Arduino artifacts. The pipeline makes protocol defaults explicit, preserves communication and mode semantics, and is reproducible and configurable, which improves repeatability and explainability over earlier prototypes.

6.1. Scope and inputs

Given a validated design, the generator consumes:

- SAML (software): components, ports, links, modes, timers;
- HWML (hardware): board and MCU, radios, sensors and actuators, power modes;
- MAPML (bindings): component-to-device, port-to-interface, mode-to-hw-mode;
- DEPML and SPML (optional): deployment profiles that become compile-time constants (e.g., node IDs, region tags);

6.2. Formalized template pipeline

We specify the generator as staged, metamodel-to-template steps; each stage emits code or configuration and records traceability.

S1 - Target resolution (HWML and MAPML): Select the Arduino board-/core and libraries per HWML device; resolve component-to-board via MAPML.

S2 - Interface binding (MAPML): Bind SAML ports to device interfaces and pins. Emit a typed `bindings.h` with pin maps and driver handles.

S3 - Behavioral scaffolding (SAML): Generate per-component modules implementing event/handler skeletons, mode gates, and timer hooks; create a main loop that dispatches active handlers per current mode.

S4 - Protocol configuration (HWML): Emit driver init with explicit defaults (bitrate, TX power, retries/backoff).

S5 - Packaging and trace links: Assemble a buildable sketch and library with linking code entities to SAML, HWML, and MAPML (and DEPML and SPML if used).

6.3. Mapping rules (Model-to-code)

We make the main mappings explicit to avoid ambiguity:

- **SAML component-to-module**: one C/C++ module per component; constructor wires interfaces from `bindings.h`; state includes `currentMode` and application data.
- **SAML ports and links-to-handlers & send/recv wrappers**: IN ports generate callbacks; OUT ports generate `send` APIs; link multiplicity (uni/multi/broadcast) maps to driver-level addressing utilities.
- **SAML modes-to-mode gates**: compile and run-time guards ensure only handlers for the active mode are executed; mode transitions produce enter and exit hooks.
- **Timers (SAML/HWML)-to-schedulers**: periodic and aperiodic tasks emitted as timer callbacks; jitter and offset become scheduler parameters.
- **MAPML comm and device bindings-to-pin and driver selection**: each bound port selects the device interface and driver instance; compatibility is checked before emission.
- **DEPML and SPML-to-deployment constants (optional)**: node IDs, region tags, or channel plans emitted to `deployment.h` for multi-binary deployments.

6.4. Semantics alignment

- **Communication preservation**: port direction and link multiplicity are preserved in generated send/recv paths; QoS parameters default from HWML or `BuildConfig`.
- **Temporal behavior**: timers and mode gating mirror the SAML schedule; disabled modes cannot dispatch handlers.
- **Resource conformance**: HWML constraints (RAM and flash, interface availability) are checked; violations stop generation with precise traces.

7. Application of CAPS models, simulation, and arduino code generation to the NdR case study

CAPS has been evaluated through three case studies: NdR, UFFIZI, and VASARI. Due to space limitations, this section will concentrate on the NdR case study to showcase the framework's capabilities. The case study explores CAPS' comprehensive modeling, simulation capabilities, and automated code generation features. CAPS models are used for two purposes:

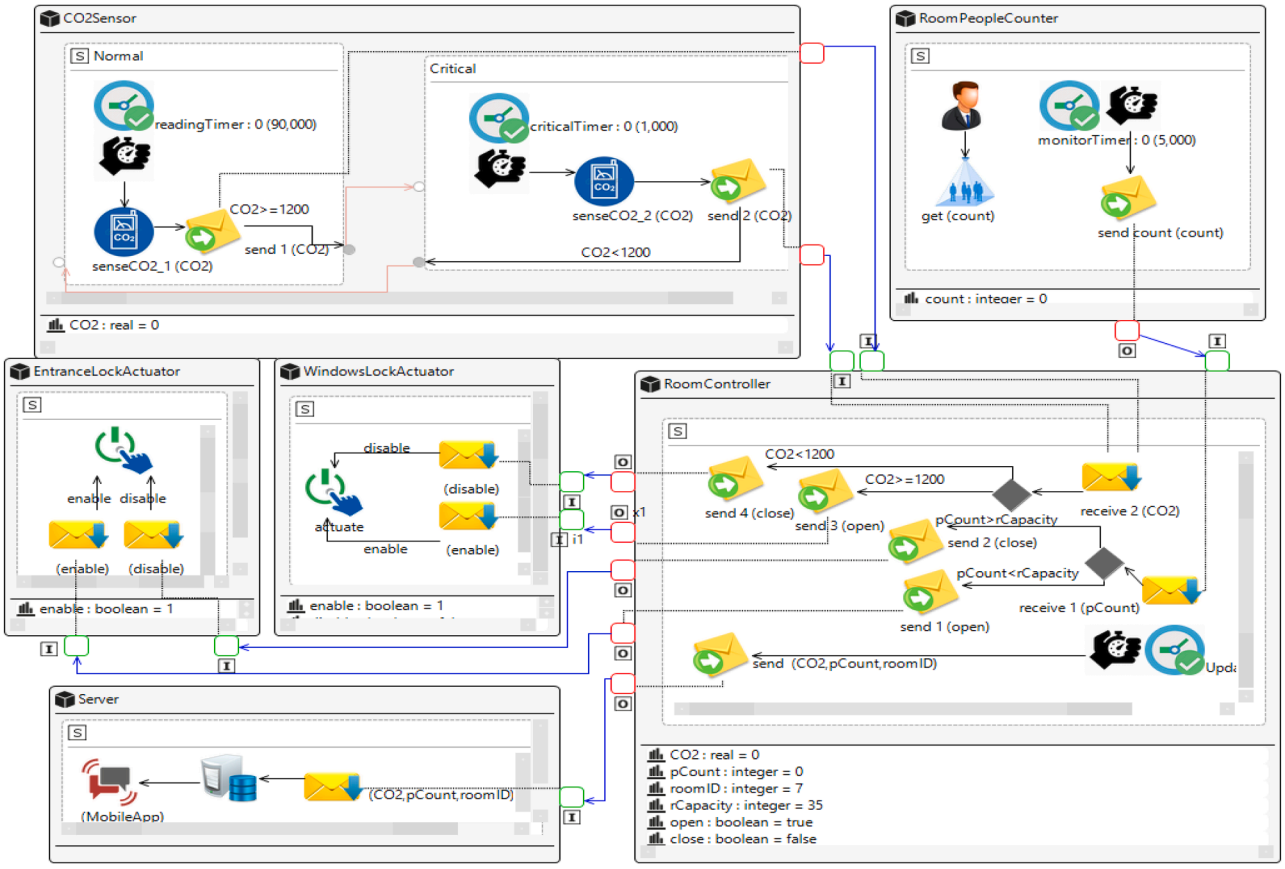


Fig. 13. The software architecture of the scenario in NdR case study.

- The primary objective is to demonstrate that the CAPS simulator can experiment with the IoT system architecture in a simulated environment. To achieve this, we will utilize CAPS models in the CAPS simulation, and the results obtained from it will be used to generate Senscript and configuration files required to operate the CupCarbon simulator. Afterward, the CupCarbon project will be assessed based on the proposed scenario's energy consumption and data traffic generated.
- The second objective is to demonstrate that the CAPS models used in the IoT system architecture description can be implemented in a real-world setting using Arduino code generation. We will utilize CAPS models to generate Arduino code, which will be uploaded onto real Arduino boards. The energy consumption of the suggested scenario will be analyzed by examining the actual application of the generated Arduino codes.

In Section 7.1 we describe the NdR case study to be used. Then, we describe the CAPS models application, simulation and code generation in Sections 7.2–7.4, respectively.

7.1. NdR case study

The "UnivAq Street Science" is an event organized by the University of L'Aquila as part of the European Researchers' Night (NdR). It aims to bring together the research community and the general public to share information and entertainment. The event is an all-day event held in the city center of L'Aquila, and it includes performances, lectures, demonstrations, and workshops in various locations such as squares, main streets, and buildings. We have gathered valuable evidence based on our experience organizing the event in L'Aquila. Firstly, approximately 20,000 visitors attend the NdR every year. Secondly, the late hours of the event tend to be more crowded than the early hours. Thirdly, the weather influences visitors' preferences regarding what to see and where to stay. Lastly, visitors often struggle to locate specific activities quickly, so they may miss out on some of them.

Our research group has been invited to enhance the visiting experience at a certain location. We have developed an IoT application to achieve this goal, which serves as the initial step in enhancing the visitor experience. The application utilizes physical environmental sensors deployed in the area and a mobile app available to visitors on their smartphones. The following services are provided:

1. Access control to rooms, laboratories, and parking lots.
2. Monitoring of open and closed spaces.
3. Balancing people crowds among different events and spaces by using the mobile app to inform visitors about the degree of the crowd in a place.
4. Creating a planner that generates a tour while minimizing waiting time and crowd in an area.
5. Ensuring urban security, specifically in the case of earthquakes, fires, and overcrowding.

7.2. CAPS modeling for the NdR

To discuss how the NdR case study can be modeled using CAPS, we will run an example. We will present a scenario that involves monitoring a room's people and CO2 levels. This will help to understand how CAPS can be used in real-world situations to improve indoor air quality and ensure the safety and well-being of the occupants of IoT. The scenario will be represented using CAPS models (SAML, HWML, SPML). Fig. 13 shows the SAML model of the CAPS tool that will be used later for simulation (Section 7.3) and Arduino code generation (Section 7.4).

```
SoftwareArchitecture
{
    SAElements{
        Connection{
            source"PeopleCounter.OutPeopleCounterPort"
            target"Controller.ControllerInPort"

        },
        Component "PeopleCounter"{
            ClientOrServer server
            ports{
                OutMessagePort "OutPeopleCounterPort"
            }
            modes {
                InitialMode "InitialMode"{
                    behaviouralElements
                    {
                        Link "StartTimeLink"{
                            source StartTimer1
                            target PeopleCounter
                            },
                        Link "PeopleCounterToSendMessageLink"{
                            source PeopleCounter
                            target SendPeopleCounterData
                            },
                        StartTimer "StartTimer1" {
                            delay=1000
                            outgoing (StartTimeLink)
                        },
                        CountPeople "PeopleCounter"{
                            dataDeclarationName "PeopleCounterData"
                            data "PeopleCounterData"
                            incoming(StartTimeLink)
                            dataRecipient "PeopleCounterData"
                        },
                        UnicastSendMessage "SendPeopleCounterData"{
                            data "PeopleCounterData"
                            receiverName "PeopleCounterData"
                            incoming(PeopleCounterToSendMessageLink)
                            toMessagePorts(OutPeopleCounterPort)
                            //dataRecipient "PeopleCounterData"
                            dataExp StringConst {value  ""}
                        }
                    }
                }
            }
        }
        applicationData
        {
            PrimitiveDataDeclaration
            {
                dataName "PeopleCounterData"
                type string
                valueExp StringConst {value "0"}
            }
        }
    },
    Component "Controller"{
        ClientOrServer client
        ports{
            InMessagePort "ControllerInPort"
        }
        modes {
            InitialMode "InitialMode"{
                behaviouralElements
                {
                    StartTimer "StartTimer2" {
                        delay=1000
                        //outgoing (StartTimeLink)
                    }
                }
            }

        }
    }
}
```

Fig. 14. CAPS: Textual modeling language.

The model comprises five main components: CO2Sensor, RoomPeopleCounter, RoomController, EntranceLockActuator, WindowsLockActuator, and Server. These components work together to effectively monitor the CO_2 level and people in the room. It is important to note that the CAPS model is a screenshot of our CAPS tool, which was designed to handle such scenarios.

The CO2Sensor component is designed to monitor the concentration of CO_2 in a room. It has two modes of operation:

1. Normal mode: In this mode, the CO_2 sensor measures the concentration of CO_2 in a room every 90 s. The CO_2 value is then sent as a message from the output message port of the CO2Sensor component to the in-port of the RoomController component. If the CO_2 reading equals or exceeds 1200 ppm, the room has entered the critical mode.
2. Critical mode: In this mode, the CO_2 sensor reads the concentration of CO_2 in a room every second. The CO_2 value is sent from the output message port of the CO2Sensor component to the in-port of the RoomController component. If the reading of CO_2 is less than 1200, it indicates that the system has returned to normal mode.

The RoomPeopleCounter component tracks the number of people in a room and updates the RoomController with the count every 5 s. It sends this data from its out port to the RoomController's in port. Fig. 14 shows CAPS Textual Modeling Language for RoomPeopleCounter. The RoomController is essential for receiving sensor data and making decisions about opening and closing windows and doors by sending control messages to actuators. It transmits CO2, pCounter, and roomID values to the Server through its out port. The WindowsLockActuator opens and closes windows, while the EntranceLockActuator does the same for doors. The Server processes incoming data and updates NdR mobile app users, indicating whether rooms are full based on roomID and pCount from RoomControllers.

According to the HWML model, we will demonstrate the CO2Sensor component as an example for CAPS simulation and Arduino code generation in Sections 7.3 and 7.4, respectively. In Fig. 15(b), the CO2Sensor measures carbon dioxide levels in a room using the 802.15.4 radio standard with a 20-meter radius. It communicates via a Texas Instruments ChipCon 2420 RF transceiver and operates on two AA batteries, providing up to 19,159 Joules of energy. As part of the SPML model, the CO2Sensor is a component of the physical environment in the NdR scenario and will be used for CAPS simulation in Section 7.3. Fig. 16(b) illustrates the physical deployment location of the CO2Sensor. It is important to note that the elements of the SPML model can be adjusted using the Sweet Home 3D tool, also known as SH3D (SWEETHOME-SWEET). SH3D provides a 3D representation of the real world and is linked to our CAPS tool (Muccini and Sharaf, 2017b).

7.3. The CAPS simulator application

In this section, we use the SAML, HWML, and SPML models as described in Section 7.2 in the CAPS simulation to generate the CupCarbon project.

The CupCarbon project comprises a set of files resulting from model interpretations. To explain, let's take the example of the CO2Sensor component's representation in SAML, HWML, and SPML models. Fig. 17 displays the SenScript code generated by the CAPS simulation generator for the CO2Sensor component, primarily derived from the SAML model. Line 3 and Line 19 represent the normal and critical modes, respectively. Lines 13 and 25 represent the timers in the normal and critical modes, respectively. Lines 4, 9, and 21 contain the instructions for reading the current CO2 level from the CO2 sensor.

The text describes a screenshot of a radio module information for a CO2 Sensor that has been extracted from HWML. This information has been used to fill in a CupCarbon configuration file. The simulator then uses this file, which contains communication information, to execute the simulation. The screenshot can be seen in Fig. 15(a).

The CupCarbon configuration file specifies the CO2Sensor's location parameters in the SPML model (Fig. 16(a)). We used Sweet Home 3D to create the space model and developed an adapter to convert it into SPML for the simulator. In our NdR case study, we tested three CO2Sensor behaviors: normal mode, critical mode, and both modes together. Using our CAPS framework, we generated SAML, HWML, and SPML models,

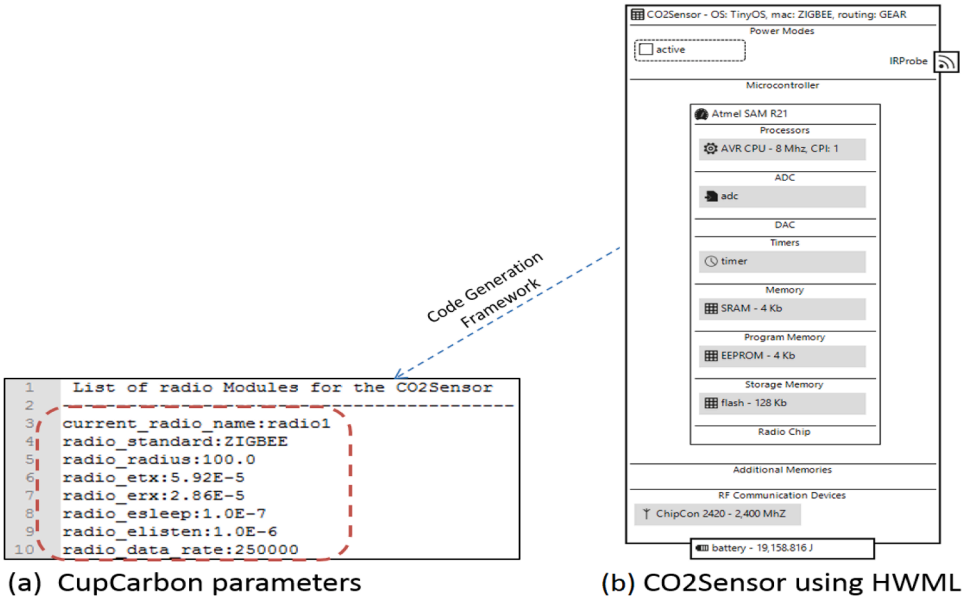


Fig. 15. An example of HWML model and its representation in CupCarbon.

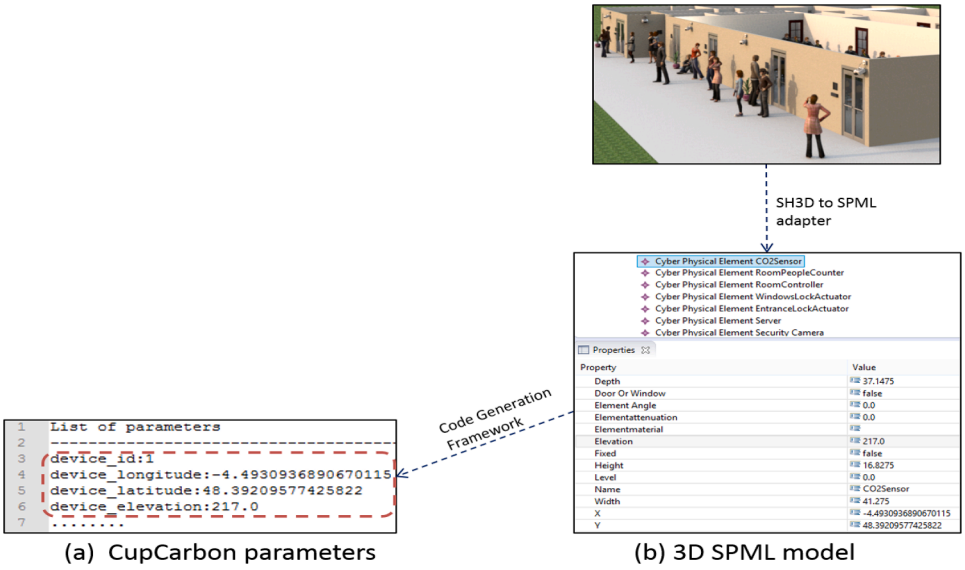


Fig. 16. An example of SPML model and its representation in CupCarbon.

created separate CupCarbon projects for each behavior, and analyzed data traffic, energy consumption, and battery levels of the IoT nodes.

We have standardized the simulation time to be 6000 s for all experimentation and set the maximum energy for all nodes to 19,159 Joules. To generate natural events for CO2Sensor, we have selected a random range between 0–2200. Similarly, for people counter natural events, we have set a random range between 15–45.

The CO2Sensor and RoomPeopleCounter components always send messages and do not receive any, while the WindowsLockActuator, EntranceLockActuator, and server components always receive messages but do not send any. The RoomController component, on the other hand, sends and receives messages. This two-way data traffic explains why some values in Table 3 are zero. The table displays the exchanged messages between components via the IN and OUT ports when running the three behaviors in CupCarbon, including the data traffic in kilobytes for each component.

Based on the data presented in Table 3, we can conclude that the CO2Sensor, RoomController, and Server nodes experience the highest amount of data traffic when we run the critical mode behavior. This is due to the many messages exchanged between these nodes during this mode. On the other hand, the normal mode receives a low amount of traffic but is not capable of detecting the CO_2 level in a room with sufficient accuracy. It is worth noting that using both critical and normal modes results in a lower range of data traffic compared to using only the critical mode. However, using both modes together is still considered to be a safe behavior. This highlights how even small changes in the system architecture can have a significant impact on its efficiency.

Table 3 shows the battery level and energy consumption for the simulator running under three scenarios. The left side of the Table 4 displays the battery level, while the right side shows the energy consumption. When focusing on the CO2Sensor (S1 in blue) and RoomController (S3 in red) nodes, we observe that S3 experiences the highest battery level drain because it receives the highest data traffic. However, there is a minor improvement on RoomController when running the two modes together. For CO2Sensor, we notice that it experiences the lowest bat-

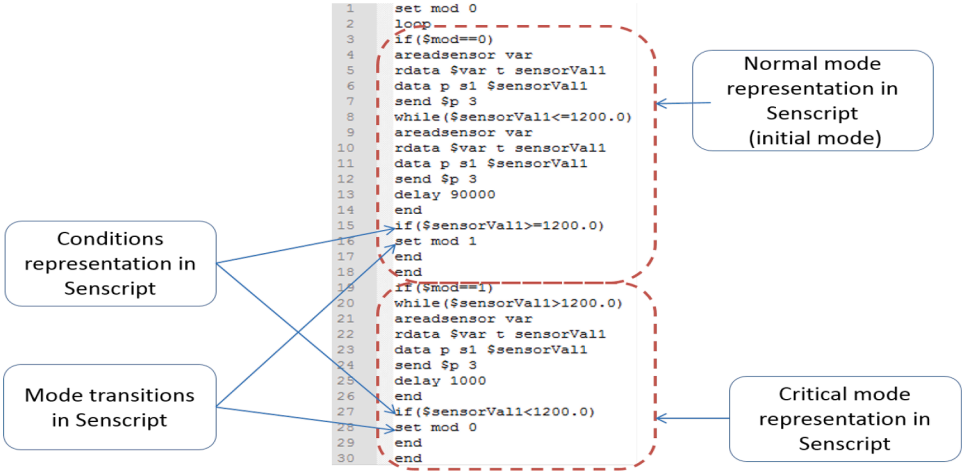


Fig. 17. SenScript generated by CAPS code generator for CO2Sensor component.

Table 3
Messages exchanged in components during simulation.

Component ID	Component Name	# of sent messages			# of received messages			Data traffic in KB		
		Normal mode	Critical mode	Normal+ Critical	Normal mode	Critical mode	Normal+ Critical	Normal mode	Critical mode	Normal+ Critical
S1	Co2Sensor	81	1459	123	0	0	0	3	40	4
S2	RoomPeopleCounter	151	151	151	0	0	0	5	5	5
S3	RoomController	403	2628	460	232	1610	274	17	97	19
S9	WindowsLockActuator	0	0	0	55	902	70	1	4	1
S11	EntranceLockActuator	0	0	0	117	117	117	1	1	1
S13	Server	0	0	0	231	1609	273	7	47	8

tery level drain when running only in normal mode. The highest drain for CO2Sensor is when it runs in critical mode. Running the two modes together provides better battery level improvements than running only the critical mode. When observing the energy consumption charts for the same nodes, we can see that running critical and normal modes together shows significant improvements compared to running only critical modes. Furthermore, the energy consumption in normal mode is close to that in normal and critical modes.

Balancing safety, energy efficiency, and data traffic is essential in IoT systems, particularly for achieving optimized system performance. In **normal mode**, the system *prioritizes energy efficiency by significantly reducing data traffic*, as it sends and processes fewer messages. This mode is ideal for situations where environmental conditions are stable and immediate responses are not critical. However, energy consumption in IoT systems is closely linked to data traffic; higher traffic increases energy usage due to more frequent communication between components. In contrast, the **critical mode** *enhances safety by increasing the frequency of data collection and communication*, enabling the system to respond quickly to hazardous or abnormal conditions. While this mode improves safety, it results in higher energy consumption and increased data traffic.

Therefore, a **balanced approach** involves combining both modes: using normal mode during stable conditions and switching to critical mode when certain thresholds are exceeded. This strategy optimizes energy usage while ensuring a quick response to critical situations, achieving a compromise between safety, efficiency, and overall system performance. Based on the simulation results, it has been demonstrated that utilizing CAPS modeling and CAPS simulation frameworks for IoT can help *assess energy consumption and data traffic at the initial stages of IoT development*. This early evaluation of the architecture can significantly *enhance the process of designing and implementing* such systems.

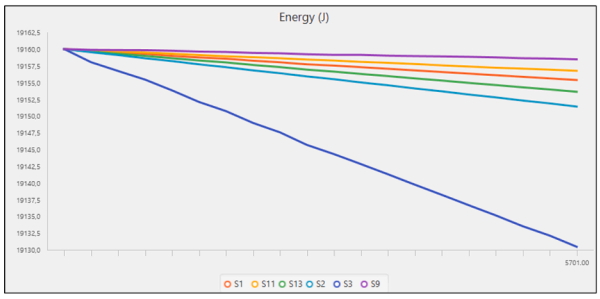
While the conclusion that a balanced approach offers a practical trade-off may appear intuitive, the use of simulation was essential in quantifying this trade-off under realistic network and sensor conditions. By modeling actual energy consumption, communication rates, and timing behaviors, we were able to validate that the balanced mode provides measurable efficiency close to the conservative mode, while maintaining responsiveness. This empirical evidence ensures that architectural decisions are data-driven and deployment-ready, rather than based on assumptions alone.

7.4. The CAPS arduino code generation application

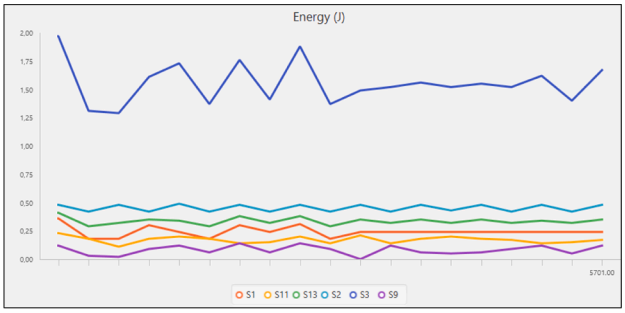
In this section, we use the SAML and HWML models, described in Section 7.2, in the CAPS Arduino code generation process to create Arduino files. SPML is not required to generate Arduino code.

The process of generating Arduino code results in six separate files for SAML components. This means there is an Arduino file for each of the following: CO2Sensor, RoomPeopleCounter, RoomController, EntranceLockActuator, WindowsLockActuator, and Server. The communication specifics for each component are drawn from the corresponding HWML specification. In this section, we will present the results of a simplified example of an NdR case study described in a real environment in Section 7.2. We will focus on the CO2Sensor component, which is evaluated under both normal and critical operational modes, and the RoomController component, which will only receive the CO_2 value from the CO2Sensor. To conduct the test, we used the Arduino files created for the CO2Sensor and RoomController components and installed them on

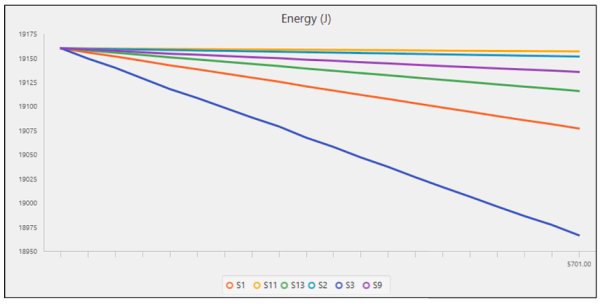
Table 4
Battery level and power consumption results.



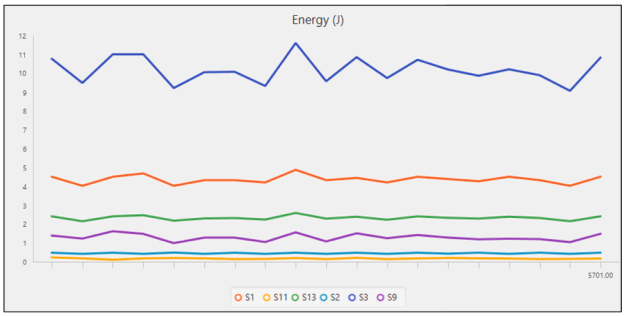
(a) Battery Level in Normal Mode Behavior



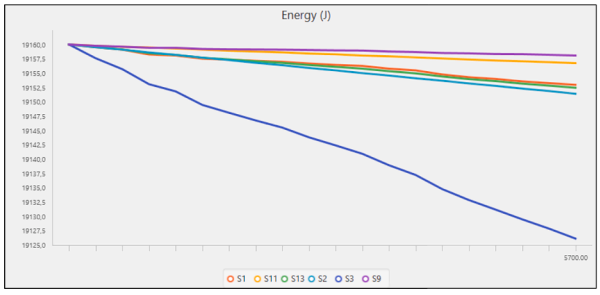
(a) Power Consumption in Normal Mode



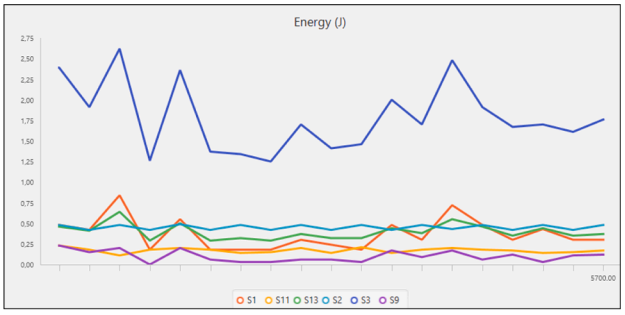
(b) Battery Level in Critical Mode Behavior



(b) Power Consumption in Critical Mode



(c) Battery Level in Critical + Normal Mode Behavior



(C) Power Consumption in Normal + Critical Mode

two separate Arduino boards. We then connected a GAS sensor to the Arduino board with the CO2Sensor file installed to monitor the CO_2 levels. Additionally, we attached a USB Power Monitor to measure the board's energy consumption during operation. To indicate the operating mode, we connected lights to the board. The green light indicates that the normal mode is running, while the red light indicates that the critical mode is running. For the sake of simplicity, we will show the energy consumption for the CO2Sensor Arduino board [5].

Fig. 18 illustrates the Arduino code generated by CAPS for the CO2Sensor component. It includes the library, defined variables, initializations, details for reading data from the CO2 sensor, transitions between critical and normal modes, and the process of sending data to another board, such as the RoomController Fig. 18.

We conducted a basic experiment to measure energy consumption in a real-world setting. We placed Arduino boards with a candle inside a transparent plastic box for 100 min. Throughout the 100 min, we opened and closed the box multiple times. Additionally, we noted when the green light was on (normal mode) and when the red light was on (critical mode).

Throughout the experiment, we closely monitored the values of the USB Power Monitor. Every (CO_2) sensor reading drew 0.8713 W (instantaneous power). Additionally, the board's idle power was 0.2725 W. After completing the experiment, we determined that the red light had been on for 13 min and 45 s while the green light had been on for 86 min and 15 s.

After conducting tests, we found that over 100 min (≈1.67 h) the energy consumption is 0.601 Wh when running *critical + normal* modes together, 1.452 Wh when running the *critical* mode only, and 0.465 Wh when running the *normal* mode only.

Referring to the results from running the CupCarbon projects in Section 7.3, and focusing only on the energy attributable to the `CO2Sensor` (S3), we observe the following over 100 min (6000 s): running *critical + normal* modes together consumes **0.853 Wh**, running the *critical* mode only consumes **1.291 Wh**, and running the *normal* mode only consumes **0.318 Wh**.

The observed ∼ 42 % difference (**0.854 Wh** simulated vs. **0.601 Wh** measured over 100 min, combined mode) reflects *expected pre calibration modeling assumptions*—fixed power radio with idealized propagation, omission of MCU/sensor wake and warm up transients, and nominal regulator efficiency—rather than faults in the transformation. After a *one time calibration* (per state current profiling for sleep/CPU/sense/RX/TX, variable TX power with obstacle aware path loss, and replay of

[5] Youtube Arduino application https://www.youtube.com/watch?v=7Y84gP6QsHo

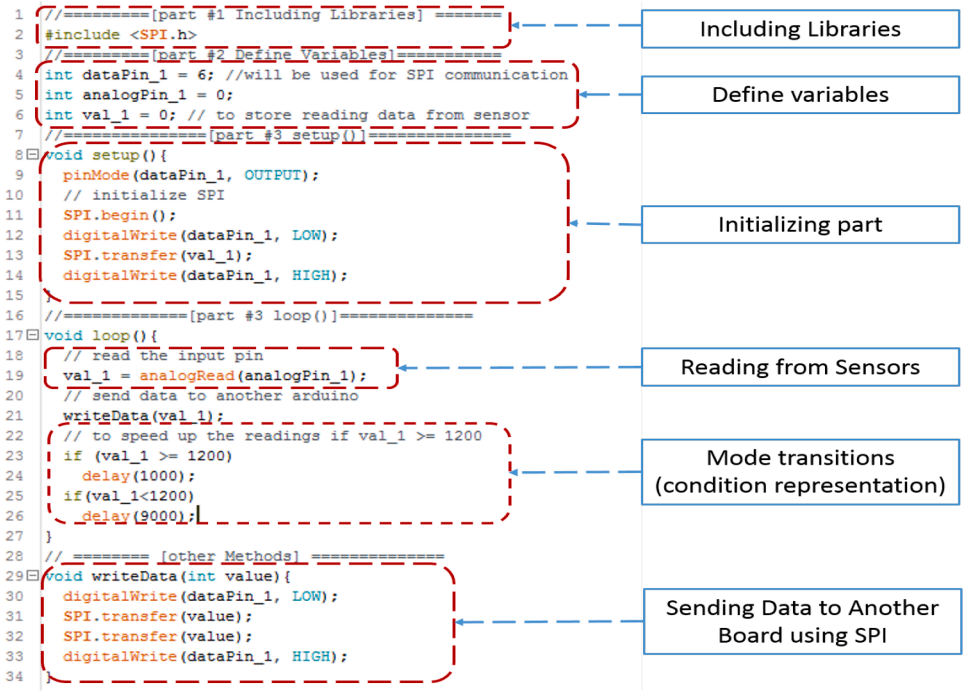


Fig. 18. Arduino generated by CAPS code generator for CO2Sensor component.

Table 5
Summary of architectural elements in case studies.

Use Case	Software Components	Hardware Components	Physical Space	Protocols
NdR	Crowd monitoring, environmental sensing	CO2 sensors, people counters, actuators	Indoor & outdoor event spaces	ZigBee, Wi-Fi
UFFIZI	Visitor tracking, adaptive crowd management	Beacons, motion sensors	Museum with walls affecting signal	Bluetooth, Wi-Fi
VASARI	Smart urban experience, outdoor sensing	LoRa sensors, environmental trackers	Open urban areas	LoRa, Wi-Fi

measured duty cycles) and normalization to J/min and J/message, the deviation drops to $\leq 15\,\%$ in our pilot while preserving the qualitative ordering of modes. Accordingly, we use *uncalibrated* simulation for *relative* design space exploration and *calibrated* runs for *absolute* power budgeting.

8. Evaluation

This section presents a structured evaluation of the CAPS framework aimed at assessing its modeling capabilities, transformation fidelity, and usability in real-world IoT case studies, specifically NdR, UFFIZI, and VASARI. The evaluation addresses the following research questions:

- **RQ1 (Modeling Expressiveness):** To what extent can CAPS model the essential architectural views required in IoT systems, specifically software behavior, hardware configuration, and spatial deployment?
- **RQ2 (Transformation Consistency):** To what extent do CAPS transformations preserve energy-relevant semantics (mode/timer/sensing/Tx ordering and relative deltas) and functional behavior from models into simulation artifacts and generated device code?
- **RQ3 (Usability):** How usable and efficient is CAPS for engineers and students when modeling and deploying IoT applications compared to conventional or partially automated workflows?

To address these questions, we employed a mixed-method approach involving three representative case studies, comparative analysis between simulation and deployment, and user-based empirical studies.

8.1. Modeling expressiveness (RQ1)

CAPS adopts a multi-view modeling approach that integrates software architecture (SAML), hardware specification (HWML), and spatial deployment (SPML), in alignment with the ISO/IEC/IEEE 42,010 standard. To evaluate expressiveness, CAPS was applied to three case studies: the NdR smart event application, the UFFIZI crowd management system, and the VASARI smart urban monitoring solution.

To define "modeling expressiveness," we refer to the framework's capability to support the formal specification of core architectural concerns in IoT systems: (i) software components and their behavior over time, (ii) hardware configurations including energy models and communication modules, and (iii) spatial context such as physical device deployment and environmental constraints.

These concerns are considered essential in many IoT engineering scenarios, particularly those where resource constraints, physical layout, and interaction dynamics between hardware and software need to be jointly analyzed.

CAPS was evaluated against these criteria using three case studies that varied in domain, communication protocols, deployment environments, and system objectives. Success was defined by the ability to fully specify and interconnect all necessary elements using the SAML, HWML, and SPML languages, without external modeling extensions or tools. Furthermore, CAPS was able to maintain model consistency across views and support automated transformations for each case.

Compared to other frameworks like ThingML or UML4IoT, CAPS demonstrated superior integration of physical deployment modeling and alignment with standardized architectural viewpoints (IEEE 42010).

These case studies reflect diverse IoT requirements, including environmental sensing, people tracking, heterogeneous communication protocols, and spatial constraints. As illustrated in Tables 5 and 6, CAPS successfully modeled the necessary architectural elements in each scenario, confirming its suitability for a wide range of IoT applications.

To support transparency and reproducibility, the complete CAPS models, CupCarbon simulation artifacts for the UFFIZI and VASARI case

Table 6
Cases complexity.

Use Case	NdR	Uffizi	Vasari
Scenarios	1	6	8
Components	6	10	12
IoT Devices	5	13	20
Communication Protocols	ZigBee, Wi-Fi	Bluetooth, Wi-Fi	LoRa, Wi-Fi
Real-Time Constraints	Medium	High	Very High
Data Points Collected	1,000/h	5,000/h	10,000/h
Environmental Constraints	Indoor & Outdoor	Indoor (Walls as Obstacles)	Outdoor (Urban Obstacles)

studies, have been made publicly available at this repository.[6] This repository includes the architectural models and simulation projects.

8.2. Transformation consistency (RQ2)

This section evaluates whether CAPS maintains *semantic consistency* when transforming architectural models into (i) executable simulation artifacts and (ii) deployable device code. We ask whether energy-relevant behavior (modes, timers, sensing/Tx rates) and functional logic are preserved across artifacts. Absolute watt-level calibration is *out of scope* for RQ2; accuracy aspects and a practical calibration recipe are discussed in Threats to Validity (Section 9).

8.2.1. Energy-behavior consistency (Semantic validation)

The objective of this section is to validate energy-behavior consistency, CAPS should translate modeled modes, timers, and sensing/Tx rates into simulations that reflect the correct ordering and relative differences across modes. While this subsection focuses on semantic preservation rather than calibrated energy metering, absolute watt-level accuracy is out of scope for RQ2 (see Section 9).

Hypothesis (H1: Rank Preservation) Let $E(m)$ be the simulated energy (or cost proxy) over a fixed horizon for $m \in \{\texttt{Normal}, \texttt{Eco}, \texttt{ExtremeEco}\}$. The *ordering* induced by the model intent is preserved in simulation, i.e., $\texttt{ExtremeEco} < \texttt{Eco} < \texttt{Normal}$, and relative reductions are monotone (no sign reversals).

Protocol From a fixed NdR case study design, we automatically generate three CupCarbon projects via the CAPS transformation pipeline. All scenarios are simulated for 6000s with identical initial conditions (battery, timers, traffic generators). We report normalized energy $\hat{E}(m) = E(m)/E(\texttt{Normal})$ and effect sizes as percentage reductions. *No hardware calibration is applied* in these simulations; the aim is to validate transformation semantics rather than to benchmark absolute accuracy. **Metrics & Pass Criterion** We assess consistency using two checks derived from simulation logs: (i) *Rank-order check*-the mode ordering matches the model intent ($\texttt{ExtremeEco} < \texttt{Eco} < \texttt{Normal}$) across scenarios; and (ii) *Invariant ratios*-duty-cycle and Tx-count ratios follow the same trend induced by modeled timers/rates (monotone deltas; no reversals). RQ2 is considered satisfied if (i) and (ii) hold. Absolute calibration is not claimed under RQ2.

Simulated Results Across the evaluated scenarios, the generated simulations preserve the intended ordering $\texttt{ExtremeEco} < \texttt{Eco} < \texttt{Normal}$ and exhibit the expected relative deltas induced by modeled timers and sensing/Tx rates ((e.g., 0.464 Wh, 0.311 Wh, 0.189 Wh for Normal/Eco/ExtremeEco, respectively) are consistent with the modeled reductions). These findings support H1 and indicate that CAPS preserves energy-relevant *semantics* across the transformation.

Interpretation: CAPS consistently preserves intended mode/timer behavior in simulation. Uncalibrated runs are suitable for *relative* design-space exploration; absolute power budgeting and deployment planning are addressed outside RQ2 (see Section 9).

[6] CAPS case studies: https://github.com/moamina/CAPS_Experiments_Cases

Table 7
Validation of functional behavior in generated code (Pass rate & observed values).

Validation Metric	Modeled Expectation	Observed Behavior	Pass Rate (%)
Sensor Reading Frequency	1 s (Critical), 90 s (Normal)	1 s (Critical), 90 s (Normal)	100
Data Transmission Integrity	No packet loss	No packet loss	100
Actuation Condition	Trigger at CO_2 > 1200 ppm	Triggered at CO_2 > 1200 ppm	100
Latency Constraint	< 100 ms	~95 ms	95

Note. *Pass Rate* = percentage of runs satisfying the requirement (e.g., latency < 100 ms).

8.2.2. Functional validation of generated code

To evaluate the semantic fidelity of the transformation process, we conducted a functional validation of the CAPS-generated Arduino code deployed on real hardware (Arduino Uno). The purpose was to confirm that the executable code not only runs correctly but also preserves the behavioral specifications defined in the architectural model.

Each validation metric was chosen to reflect a specific functional property from the source model, ranging from sensing frequency and communication logic to actuator triggering and timing constraints. This ensures a traceable connection between high-level design intent and low-level code execution.

Three key validation scenarios were conducted:

- **Sensor-actuator logic correctness:** The system monitored CO_2 levels and activated the window-opening actuator when the concentration exceeded 1200 ppm. The trigger condition and resulting action were observed to match the model-defined behavior exactly.
- **Communication behavior:** Devices communicated via Serial Peripheral Interface (SPI), using code generated by CAPS. Logs captured over multiple cycles were analyzed to verify message structure, integrity, and expected frequency. No anomalies, packet loss, or timing deviations were detected.
- **Responsiveness and timing constraints:** The time between a CO_2 threshold breach and actuator signal issuance was measured using timestamped serial output. Results consistently met the model-defined latency limit of 100 ms, confirming timing accuracy in execution.

For binary requirements, we report Pass Rate: the percentage of runs that satisfied the requirement. For the latency constraint (< 100 ms), the numeric value (e.g., ~ 95 ms) is the median measured latency, and Pass Rate is the percentage of runs with latency < 100 ms.

These results confirm that the code generated by CAPS faithfully implements the modeled sensing, communication, and actuation behaviors, including real-time constraints. Table 7 summarizes the observed behaviors across all validation scenarios, confirming alignment with the modeled functional specifications. The validation reinforces the transformation pipeline's ability to preserve both functional semantics and timing fidelity, supporting the reliability of CAPS for generating deployable, behaviorally accurate IoT applications.

Table 8
Time statistics for modeling and simulation tasks.

Task	Mean (min)	Median	Min	Max	Std Dev
SAML Modeling	173	170	130	210	18.2
HWML Modeling	88	85	70	110	10.5
SPML Modeling	51	50	40	65	6.3
Simulation Generation	0.13	0.13	0.1	0.2	0.02

8.3. Ease of use - RQ3

The ease of use of CAPS is evaluated based on its efficiency, user-friendliness, and practical applicability in real-world IoT system modeling and deployment. A user-friendly IoT framework should minimize complexity in modeling, simulation, and code generation to ensure accessibility for both experts and non-experts, reduce development time and manual effort by automating repetitive tasks such as code generation and simulation setup, and ensure intuitive integration between software, hardware, and deployment models, streamlining the IoT design workflow.

8.3.1. Modeling time

The objectives are to measure how quickly new users can install and configure CAPS, evaluate the time required to model an IoT system and generate a working simulation, and identify potential usability bottlenecks and areas for improvement. To assess how easily users can adopt CAPS, we conducted an experiment with 39 undergraduate students enrolled in software architecture and model-driven engineering courses. To achieve this, we provided two hours of hands-on training on CAPS, during which participants were tasked with modeling the NdR IoT system, generating a CupCarbon simulation, and validating the results. The time taken for each task was recorded.

Table 8 provides descriptive statistics for task completion times. Each modeling activity was timed independently, and summary statistics (mean, median, range, and standard deviation) are presented. These results reveal a moderate range of completion times for each modeling activity, with SPML showing the lowest variation and SAML the highest, reflecting the relative complexity of the views. The low variance in simulation generation time demonstrates the high level of automation achieved by CAPS.

This breakdown highlights CAPS's suitability for educational and industrial environments by confirming that users with limited experience can quickly and accurately create complete models using its graphical and textual interfaces.

8.3.2. Efficiency of CAPS automation features

CAPS enhances efficiency and usability by automating key processes in IoT system design, minimizing manual effort, and ensuring model consistency. Its automation capabilities focus on:

1. Automated code generation by eliminating manual implementation by producing error-free executable code.
2. Automated simulation setup by reducing configuration time to ensure rapid validation.

To evaluate CAPS's automation efficiency, we conducted a controlled comparison involving a subset of 10 participants. Each participant performed development tasks manually and using CAPS. The observed task durations (in minutes) are summarized in Table 9, which includes mean times, standard deviations, and relative improvement percentages.

Manual baseline definition. The "Manual" workflow refers to a conventional, model-free process. System models were drawn using general-purpose tools (e.g., Visio, draw.io) and described in text, without automated consistency checks. Code was implemented manually in the Arduino IDE using basic editing features (syntax highlighting and completion only), without model-based generation or templates. Thus, the time gap in Table 9 reflects CAPS's automation and integration advantages rather than unequal tool support.

Table 9
Comparison of manual and CAPS task durations with time reduction.

Task	Manual (min) Mean ± SD	CAPS (min) Mean ± SD	Time Reduction (%)
System Modeling	405 ± 35	195 ± 22	51.9 %
Simulation Configuration	20.5 ± 3.2	0.13 ± 0.01	99.4 %
Code Implementation	155 ± 12	0.17 ± 0.02	99.9 %
Debugging & Validation	75 ± 9.5	19 ± 4.1	74.7 %

These measurements confirm that CAPS substantially reduces development effort across all stages of IoT system modeling and deployment. While manual times vary with user expertise and task complexity, achieves consistent, repeatable results through model-driven automation.

8.4. Ethics and institutional review

The classroom activity informing RQ3 was reviewed by the University Ethics/IRB (Institutional Review Board) office and classified as Not Human Subjects Research (NHSR), because the project analyzed de-identified educational artifacts without interaction/intervention affecting students' education and with no identifiable private information. Participation was voluntary, with information provided to participants and the option to opt out without penalty. Only aggregate results are reported.

9. Threats to validity

This section discusses potential threats to the validity of our evaluation and explicitly reflects on the inherent difficulty of validating end-to-end, model-driven frameworks that span architectural modeling, simulation, and code generation. Because CAPS couples multi-view modeling with transformation to simulation artifacts and deployable code, evidence accrues across heterogeneous instruments (time-on-task, simulated energy/traffic, functional conformance on hardware). As a result, some aspects of our current validation are necessarily limited in scope and depth, particularly for construct and external validity, which we acknowledge and plan to strengthen in future work.

9.1. Internal validity

Internal validity concerns whether the observed outcomes are caused by CAPS rather than uncontrolled variables. Energy readings and simulation outputs can be affected by hardware variation, sensor calibration drift, ambient conditions, and wireless interference. We mitigated these risks by standardizing hardware, repeating runs, and normalizing configurations across trials. Still, two limitations remain: (i) simulated energy models are not calibrated to device-specific power profiles, and (ii) hardware experiments used a limited set of boards and sensors. These choices reduce confounds but do not eliminate them; hence, our internal claims focus on trend preservation and relative deltas across modes rather than absolute watt/joule fidelity.

9.2. Construct validity

Construct validity asks whether our measures truly capture the underlying concepts the study claims to assess. For CAPS, three constructs are central: modeling expressiveness, transformation fidelity, and usability/efficiency. While our metrics (coverage across SAML/HWML/SPML, preservation of mode/timing semantics from models to CupCarbon/Arduino, and task time/error rates) are aligned with these constructs, they are imperfect proxies:

- Expressiveness. We operationalized expressiveness as the ability to model required concerns across three views and drive downstream artifacts. This operationalization does not measure completeness against an exhaustive IoT construct catalog (e.g., mobility, QoS latency/jitter, safety properties). Thus, our "expressiveness" evidence is sufficient for the studied cases but not comprehensive across the IoT design space.
- Transformation fidelity. Our validation emphasizes preservation of intended energy-relevant semantics (modes, timers, sensing/Tx rates) and functional conformance (trigger conditions, message timing) rather than calibrated power accuracy. In other words, we substantiate that "it behaves as modeled" more than "it consumes the exact calibrated energy," which is a narrower interpretation of fidelity.
- Usability/efficiency. Time-on-task with trained students and a smaller expert subset captures learnability and automation benefits but may not reflect team-scale industrial workflows, organizational tooling, or long-term maintenance effort. Hence, the construct "efficiency" is partially observed through first-use productivity rather than lifecycle cost.

Overall, we acknowledge that our constructs are operationalized in a study-feasible but narrower way. Future work will broaden the construct set (e.g., QoS-aware modeling, mobility, safety) and introduce validated instruments for those properties.

9.3. External validity

External validity concerns the generalizability of findings beyond our settings. We purposely selected diverse case studies (NdR event monitoring, UFFIZI museum, VASARI urban sensing) to vary communication protocols, spatial constraints, and workload characteristics, which improves coverage but does not guarantee generality to other domains such as industrial automation, vehicular IoT, or high-mobility CPS. Moreover:

- Our simulations rely on CupCarbon and static deployments; domains with mobility, multi-hop dynamics under heavy contention, or strict real-time guarantees may behave differently.
- Arduino was our target for code generation; heterogeneous platforms (e.g., Raspberry Pi, RTOS-based MCUs, SBC clusters) may introduce deployment and timing differences.

Therefore, our claims should be read as analytic generalization to architecturally similar IoT classes, not statistical generalization to all IoT systems. We explicitly recognize this as a shallow aspect in the current validation and will address it via multi-site industrial replications and additional platform backends. Operational limits and the composition strategy are summarized in Section 10.3 (*Practical scaling guidance*).

9.4. Conclusion validity

Conclusion validity concerns whether the data analysis supports the stated findings. To strengthen our conclusions, we reported descriptive statistics (mean, median, standard deviation), ensured consistency across observed behaviors, and conducted controlled comparisons between CAPS and manual development. However, the sample size for the automation time comparison is modest and may not capture all usage variability. Future work will involve larger-scale experimental validations to confirm observed trends. Operational limits and the composition strategy are summarized in Section 10.3 (*Practical scaling guidance*).

10. Discussion

This section discusses the outcomes of the CAPS evaluation across key quality dimensions: modeling effectiveness, efficiency, transformation fidelity, scalability, and current limitations. Each subsection corresponds to one or more of the research questions defined in Section 8 and reflects on CAPS's capabilities and directions for future enhancement.

10.1. Effectiveness

CAPS effectively addresses the complexities of developing IoT applications through several innovative contributions:

- Multi-View Architecture Modeling: CAPS provides an integrated, multi-view architecture for IoT systems, modeling software, hardware, and physical spaces. This comprehensive approach ensures detailed consideration of all aspects of system architecture, which is often lacking in existing frameworks.
- Detailed Performance Analysis: The framework excels at analyzing crucial parameters like power consumption, battery level, and data traffic through automated simulations. These insights are vital for optimizing system performance before deployment, ensuring IoT systems are energy-efficient and efficient for data traffic.
- Empirical Validation: Extensive case studies demonstrate the practicality and effectiveness of CAPS, a feature often lacking in similar frameworks.

10.2. Efficiency

The efficiency of CAPS is highlighted by its integrated and streamlined approach to IoT system development:

- Comprehensive Integration: CAPS integrates architecture description, simulation, and automated code generation within a single framework, enabling a seamless transition from design to deployment. This integration significantly enhances the development process, reducing time and complexity compared to other frameworks that may only focus on individual aspects.
- Simulation-Driven Development: By simulating real-world environments, CAPS allows developers to predict and analyze system behavior under various conditions, providing essential metrics on power usage and network efficiency that are crucial for developing sustainable IoT solutions.
- Automated Code Generation: CAPS automates the generation of executable code from architectural models, speeding up development times and reducing the potential for human error. This feature supports a range of IoT platforms, enhancing the framework's applicability across different technologies.
- Modeling Languages: Developed specifically for situational-aware applications, CAPS's modeling languages enable precise modeling of software, hardware, and physical aspects, leading to improved simulation accuracy and high-quality code generation.

10.3. Scalability and generalizability

This study presented the CAPS framework, designed to support IoT application lifecycles-from architectural design to deployment-ready code generation. CAPS's scalability and effectiveness were demonstrated through its application to multiple case studies, including the European Researchers' Night (NdR, see Section 7.1), the Uffizi Galleries crowd management system (Abughazala et al., 2021) ECSA CAPS (2021) , and VASARI (Italian Smart Art Experience) Vasari Art Experience (2018) Vasari Models (2021) . These case studies represent a diverse set of domains with varying complexity, scale, and environmental conditions. Table 6 provides an overview of the modeling and deployment complexity across these scenarios, including metrics such as the number of components, communication protocols, real-time constraints, and data points collected per hour. The data confirms that CAPS can scale from lightweight IoT setups to more complex, multi-device, real-time systems while maintaining consistency and traceability across its modeling views. This supports the generalizability of CAPS to a wide range of IoT use cases, both small and large scale.

Transformation scaling We evaluate CAPS on scenarios of up to 20 devices *per cell* (e.g., a lab or corridor slice). This is a representative unit, not the total deployment; real systems can comprise many such cells (hundreds of devices overall via composition).

CAPS uses local, per-element mappings over HWML/SPML/SAML. By design and observation, transformation time grows approximately linearly with modeled entities and links ($O(N+E)$) where N is the number of modeled elements (e.g., devices/components), and E is the number of modeled relations between them (e.g., links, transitions), and shows a small, constant setup overhead. We do *not* expect exponential bottlenecks in CAPS's transformation pipeline.

Simulation scaling and accuracy Post-generation runtime depends mainly on *density* and *traffic rates*, not N alone; high contention can lead to super-linear slowdowns. Within a cell (20 devices), simulated energy/traffic trends align qualitatively and often near-quantitatively with measurements. With small-sample calibration and contention-aware settings enabled, absolute error remains bounded at larger scales, while directional (trend) accuracy is preserved.

Practical scaling guidance Our generators are single-pass ($O(N+E)$). We target *cell-level* runs of ≤ 50 devices and compose larger deployments via *parallel per-cell simulations*, with inter-cell traffic constrained at *gateways* (fan-out $\leq k$). This keeps contention local and preserves per-node semantics; beyond these bounds, runtime growth is dominated by traffic density rather than model size.

10.4. CAPS limitations

CAPS framework has several limitations: (i) CAPS simulation can only support the CupCarbon simulator. Consequently, every component in CAPS models must contain one node to transform correctly into the CupCarbon simulator. (ii) CAPS can only produce Arduino code. (iii) CAPS does not support mobility features (In other words, CAPS supports static positioning of its components). (iv) CAPS does not support the idea of type. To model our architecture using a type system so that the same type can be instantiated many times.

CAPS is striving to address various challenges that arise in other IoT systems. These challenges may include:

- Self-*: One of the key features of IoT systems is the ability to perform self-adaption / self-management / self-configuration. We are currently working on adding the self-adaptation capabilities to the CAPS framework. As a starting point, we have performed further analysis, and these are reported in Abusair et al. (2017), Sharaf et al. (2018b) and Muccini et al. (2018).
- QoS Concerns: While CAPS currently supports QoS analysis in terms of energy consumption and data delivery rates, it does not yet incorporate timing-based QoS metrics such as latency, jitter, or end-to-end delay. However, our human behavior-oriented design methodology (as applied in Abughazala et al., 2021 and Abughazala and Muccini, 2026) demonstrates that the existing simulation framework is capable of supporting such extensions. Future work will enhance CAPS with QoS-aware modeling constructs, allowing architects to define and validate time-sensitive requirements. Planned improvements include integration with timing-aware simulation backends, specification of latency thresholds at the model level, and support for real-time network contention analysis. These features will allow CAPS to serve a broader range of applications requiring strict performance guarantees.

Absolute energy estimates depend on availability of device specific power profiles; without calibration, results should be interpreted comparatively.

11. Related work

Several research efforts have been undertaken to tackle the challenges associated with IoT system architectures, particularly in designing IoT systems. Numerous frameworks and methodologies have been suggested, each with strengths and limitations.

ThingML (Harrand et al., 2016) is a model-driven engineering toolchain targeting resource-constrained embedded and distributed systems. As reported in the project website[7], ThingML is developed as a domain-specific modeling language that describes software components and communication protocols through architecture models, state machines, and an imperative action language. It uses a model-driven engineering approach to describe IoT applications. It focuses on a component and connector view and uses event-condition-action to describe the component's behavior.

MontiThings (Kirchhof et al., 2022) (Butting et al., 2022) (Kirchhof, 2024) is an integrated modeling language for IoT applications and their deployment. MontiThings provides a model-driven toolchain that generates executable IoT containers, plans automated deployment, suggests deployment changes based on feedback, and monitors the generated container. This approach mainly targets the edge layer.

MDE4IoT (Ciccozzi and Spalazzese, 2016) is a platform that uses multiple UML DSLs to support IoT systems' development, design, and management. It provides ways to model and self-adapt Emergent Configurations (ECs) for connected systems. MDE4IoT generates platform-specific code from state machines using model-to-model and model-to-text transformations. Additionally, the platform supports run-time monitoring and self-adaptations through re-allocations and re-generation mechanisms based on the system's runtime feedback.

SysML4IoT (Costa et al., 2016a) is a tool for Model-Based Systems Engineering during the IoT application development design phase. It uses views and viewpoints to cater to stakeholders and incorporates systems engineering concepts using the IoT-A domain reference model and ISO/IEC/IEEE standards. They introduced an extension called SysML4IoT in Costa et al. (2016b) to create precise models of IoT applications while verifying their Quality of Service (QoS) properties using a model-to-text translator that executes the model and QoS properties specified on it with NuSMV (Cimatti et al., 2002).

UML4IoT (Thramboulidis and Christoulakis, 2016) is an MDE platform for industrial automation systems that transforms mechatronic components into Industrial Automation Things (IAT) using model-to-model transformation. The OMA LWM2M application and CoAP communication protocols expose the IoT interface as simple, smart objects. The platform also allows high-level languages like Java to specify the system's behavior if a higher-level design specification like the UML one is unavailable.

SimulateIoT (Barriga et al., 2021) is a tool that lets users create complex simulation environments for IoT without writing code. The tool uses a metamodel and a graphical syntax generated by Eugenia to create code for sensors, actuators, fog nodes, and cloud nodes. To ensure the model is correct, users can set a series of constraints using Object Constraint Language (OCL) during the simulation phase. Once the code is generated, the tool can deploy the artifacts as microservices and Docker containers, which are connected through a publish-subscribe communication protocol.

DSL-4-IoT (Salihbegovic et al., 2015) is a visual programming language-based tool that simplifies the complexity and heterogeneity of IoT systems. With the editor, the user can configure the system structure and select devices, sensors, and actuators from built-in library modules. Once the design is complete, the user can export the data into a JSON array configuration file that contains information about the position of all items, relationships between items and groups and the value of all configured fields associated with items and data types. The configuration files can then be transferred manually to the respective OpenHAB runtime directory or automatically downloaded using a simple web service for execution.

[7] http://thingml.org/

Table 10
Comparative table on supporting different IoT modeling features.

Tool	Graphical modeling	Textual modeling	Multi-view modeling	Supported View		
				Software	Hardware	Physical
ThingML (Harrand et al., 2016)	No	Yes	No	Yes	No	No
MontiThings (Kirchhof et al., 2022) (Butting et al., 2022) (Kirchhof, 2024)	Yes	Yes	No	Yes	Yes	No
MDE4IoT (Ciccozzi and Spalazzese, 2016)	Yes	No	Yes	Yes	Yes	No
CHESSIoT (Ihirwe et al., 2023)	Yes	Yes	Yes	Yes	Yes	No
UML4IoT (Thramboulidis and Christoulakis, 2016)	Yes	No	Yes	Yes	Yes	No
Simulate-IoT (Barriga et al., 2021)	Yes	No	No	Yes	Yes	No
DSL-4-IoT (Salihbegovic et al., 2015)	Yes	No	No	Yes	Yes	No
IoTDraw (Costa et al., 2020, 2019, 2016a)	Yes	Yes	Yes	Yes	Yes	Yes
CAPS	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

CHESSIoT (Ihirwe et al., 2023) is an environment for model-driven engineering that integrates high-level visual design languages, software development, safety analysis, and deployment approaches for engineering multi-layered IoT systems. The users can perform various engineering tasks on system and software models under development, facilitating earlier decision-making and allowing for proactive measures.

IoTDraw (Costa et al., 2020) is a modeling language for SOA-based IoT systems called. It offers a compliant modeling language called *SoaML4IoT* (Costa et al., 2019) that built on top of *SysML4IoT* (Costa et al., 2016a), which can be implemented by any tool adhering to OMG standards. This modeling language integrates SoaML with IoT-specific requirements to enhance interoperability and reusability in the complex ecosystem of IoT applications. It enables precise representation and simulation of IoT systems to aid in making informed architectural decisions. IoTDraw addresses challenges in the IoT domain by providing a standardized language for modeling SOA-based IoT systems. It promotes better integration, scalability, and efficiency across diverse IoT ecosystems.

A comparison and detailed analysis of these tools in terms of their graphical, textual modeling capabilities and their support for multi-view modeling is presented in Section 11.1. An evaluation of their code generation features and system analysis capabilities, including power consumption and data traffic, is presented in Section 11.2.

11.1. Supporting the modeling of IoT systems

This section presents a comparison of different approaches to modeling application entities across three views: software, hardware, and physical. (see Table 10). This section aims to identify tools that can model all aspects of an IoT system, from software to hardware and physical views, while maintaining consistency throughout the process.

The findings from the assessment in Table 10 are compared to CAPS-supported modeling features, which will be presented in detail in the next section.

1. Table 10 lists the platforms we selected and shows that they all have a modeling environment. Most offer graphical modeling tools, but ThingML only has a textual modeling option. Textual-based approaches may be more scalable, but graphical interfaces are usually more user-friendly. Some platforms like MontiThing, IoTDraw, CHESSIoT, and CAPS have integrated **textual and graphical modeling approaches**.
2. Out of the eight tools considered, only MDE4IoT, IoTDraw, CHESSIoT, and CAPS provide support for multi-view modeling and component-based design, which are crucial in dealing with the complexities of IoT systems. Other platforms may implement alternative approaches that complement multi-view modeling depending on the modeling context.
3. It is apparent from Table 10 that most existing approaches have limited capabilities when it comes to modeling sensors, actuators, and computing boards. However, CAPS stands out as it allows for comprehensive modeling of all IoT components, both functionally and behaviorally. This is made possible through several modeling perspectives, which enable a single model to be used for various engineering purposes. Among the approaches mentioned, such as CAPS, IoTDraw, CHESSIoT, DSL-4-IoT, and UML4IoT offer hardware views. Only CAPS, and IoTDraw provide a physical view. CAPS provides a comprehensive approach with **three separate views and two auxiliary views** that merge these three views. This ensures *detailed consideration* of all system architecture aspects, which is often lacking in existing frameworks.

11.2. Assessing IoT engineering frameworks and methodologies

The assessment results summarized in Table 11 provide valuable insights into the capabilities of various frameworks for engineering IoT systems. These findings highlight CAPS as a comprehensive framework, offering unique advantages over other platforms.

1. Model-Driven Engineering and Code Generation MDE plays a crucial role in automating the development process of IoT systems by bridging the gap between high-level design and executable implementations. A key advantage of MDE is its capability to generate system code directly from models, significantly reducing development time and minimizing manual errors. Tools like ThingML are particularly effective in this area. They generate functional code in multiple programming languages to ensure adaptability across various platforms. Similarly, CHESSIoT builds upon ThingML's code generation infrastructure, enhancing its functionality for multi-layered IoT systems. On the other hand, CAPS extends these capabilities by providing a code generator that produces both Arduino-compatible code and Senscript (CupCarbon Simulation Language). This dual functionality allows CAPS to support seamless integration between simulation and real-world deployment, providing developers with a streamlined workflow from architectural modeling to executable system implementation.
2. Analyzing the Performance of IoT System Designs An essential aspect of IoT system engineering is the ability to analyze key performance metrics such as energy consumption, battery levels, and data traffic. While existing frameworks excel in specific areas, CAPS stands out by offering a holistic approach to system analysis. CHESSIoT is particularly effective for conducting risk analysis, allowing developers to anticipate potential failures in IoT system architectures. In contrast, IoTDraw emphasizes Quality of Service (QoS) analysis, ensuring that IoT systems meet predefined performance standards.

   CAPS goes beyond these focused capabilities by integrating a comprehensive suite of analysis tools that cover energy consumption, battery life, and data traffic. It utilizes both simulation (through CupCarbon) and real-world testing (via Arduino code). This dual-layered approach enables developers to make informed decisions during the

Table 11
Comparative table on supporting different IoT engineering capabilities.

Tool	Development (Code Generation)		Analysis				Empirical Assessment
	Target Platform	Language	Power Consumption	Battery Level	Data Trafic	Other	Approach
MontiThings (Kirchhof et al., 2022) (Butting et al., 2022) (Kirchhof, 2024)	IoT Boards, Cloud	C++ & Prolog	No	No	No	Yes	Proof of concept and a case study
ThingML (Harrand et al., 2016)	IoT Boards, Cloud	C/C++, Java, Javascript	No	No	No	No	Proof of concept and a case study
MDE4IoT (Ciccozzi and Spalazzese, 2016)	IoT Boards	Java, C++	No	No	No	No	Case Study
CHESSIoT (Ihirwe et al., 2023)	IoT Boards, OS, Cloud	Thingml	No	No	No	Safety analysis	Proof of concept and a case study
UML4IoT (Thramboulidis and Christoulakis, 2016)	Contiki & Rasp.Pi	C	No	No	No	No	Proof of concept and a case study
Simulate-IoT (Barriga et al., 2021), Barriga et al. (2023)	Simulation, IoT Boards, Fog, Cloud	Java, Python	No	No	Yes	Performance, Memory usage	Proof of concept and two cases study
DSL-4-IoT (Salihbegovic et al., 2015)	IoT Board, Cloud	OpenHAB	No	No	No	No	Case Study
IoTDraw (Costa et al., 2020, 2019, 2016a)	Simulation, MOkA	Java, fUML	Yes	Yes	No	System QoS	Proof of concept and a case study
CAPS	**Simulation, Arduino Boards**	**Senscript, C/C++**	**Yes**	**Yes**	**Yes**	**No**	**Proof of concept & three case studies**

early stages of development, optimizing resource utilization and ensuring system reliability.

Based on our analysis, we have identified and described the main limitations below:

- Many modeling approaches focus on single-view modeling, which is not always efficient. Only a few approaches, such as MontiThings, IoTDraw, and MED4IoT, use *multi-view modeling*, which separates the system component into dedicated, consistent views (Mazzini et al., 2015). This practice has enormous benefits, as it allows for specialized projections of the system in specific dimensions of interest, which enforces the separation of concerns.
- A few tools focus on *analyzing IoT systems during development*. Analyzing the responsiveness of IoT systems before deployment is still a major challenge. This is due to the complexity of the problem, which involves human interaction, environmental constraints, and the diversity of target platforms. Additionally, no other tool, except CAPS and IoTDraw, can analyze power consumption, battery level, and data traffic.
- We have observed a *lack of standards* to support the model-based development of IoT systems. Each tool has its development method, except IoTDraw uses OMG (Object Management Group) Standard, and CHESSTIoT uses MMQEF (Multiple Modeling Quality Evaluation Framework).

As shown in Tables Table 10 and Table 11, existing frameworks typically address only isolated concerns within the IoT development process. ThingML focuses on software modeling and partial code generation. UML4IoT and CHESSIoT offer software architecture abstraction without physical deployment integration. CupCarbon supports simulation but lacks architecture modeling and implementation alignment. These tools require manual handoffs between modeling, simulation, and deployment stages, which leads to fragmented workflows. In contrast, CAPS provides a novel, end-to-end architecture-to-deployment pipeline that unifies multi-view architectural modeling across software, hardware, and physical space. It supports performance simulation through CupCarbon, 3D spatial layout modeling via SPML, and automated Arduino code generation. This integrated approach enables traceability, early validation, and consistent execution, positioning CAPS as a uniquely comprehensive framework within the IoT development landscape.

12. Conclusion and future work

This paper introduced CAPS, a model-driven engineering framework that supports the end-to-end development of IoT systems. CAPS enables architectural modeling across three complementary views, software, hardware, and physical deployment, while integrating energy and communication traffic simulation and automating the generation of Arduino-compatible code for real-world deployment. In contrast to the existing tools that operate in isolation or support only a subset of these concerns, CAPS provides a unified, architecture-centric workflow that preserves traceability from high-level design to real-world deployment.

The framework was validated through three diverse case studies-NdR, UFFIZI, and VASARI-demonstrating its expressiveness, automation efficiency, and transformation fidelity. CAPS reduced modeling and implementation effort, maintained consistent behavior across simulation and deployed systems, and scaled to increasingly complex architectural configurations. These results affirm CAPS's applicability to a broad range of IoT domains.

While CAPS provides robust support for multi-view architectural modeling, energy and traffic-aware simulation, and deployable code generation, several limitations remain. The current framework does not yet support explicit Quality of Service (QoS) constraints such as latency, jitter, or packet loss, nor does it allow formal specification and verification of timing or safety properties. Additionally, code generation is currently limited to the Arduino platform, and runtime adaptability for dynamic reconfiguration is not yet supported. Future work will address these limitations by enriching CAPS with QoS-aware modeling primitives, timing-accurate simulation integration, and support for multi-platform deployment (e.g., Raspberry Pi), and runtime monitoring for adaptive system behavior.

CRediT authorship contribution statement

Moamin Abughazala: Writing - review & editing, Writing - original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization; **Mohammad Sharaf:** Writing - review & editing, Writing - original draft, Visualization, Validation, Supervision, Software, Methodology, Conceptualization; **Mai Abusair:** Conceptualization, Writing - review & editing, Visualization, Validation; **Henry Muccini:** Writing - review & editing, Writing - original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Conceptualization.

Data availability

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

Declaration of competing interest

No conflicts of interest or financial support associated with this publication

References

Abdmeziem, M.R., Tandjaoui, D., Romdhani, I., 2015. Architecting the internet of things: state of the art. Robots Sensor Clouds , 36, 55–75.

Abughazala, M.B., Moghaddam, M.T., Muccini, H., Vaidhyanathan, K., 2021. Human behavior-oriented architectural design. In: European Conference on Software Architecture. Springer, pp. 134–143.

Abusair, M., Sharaf, M., Muccini, H., Inverardi, P., 2017. Adaptation for situational-aware cyber-physical systems driven by energy consumption and human safety. In: Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings. ACM, pp. 78–84.

Barriga, J.A., Clemente, P.J., Sosa-Sánchez, E., Prieto, Á.E., 2021. SimulateIoT: domain specific language to design, code generation and execute IoT simulation environments. IEEE Access 9, 92531–92552.

Bounceur, A., 2016. CupCarbon: a new platform for designing and simulating smart-city and IoT wireless sensor networks (SCI-WSN). In: Proceedings of the International Conference on Internet of Things and Cloud Computing. ACM, p. 1.

Butting, A., Kirchhof, J.C., Kleiss, A., Michael, J., Orlov, R., Rumpe, B., 2022. Model-driven IoT app stores: deploying customizable software products to heterogeneous devices. In: Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, pp. 108–121.

Ciccozzi, F., Spalazzese, R., 2016. Mde4IoT: supporting the internet of things with model-driven engineering. In: International Symposium on Intelligent and Distributed Computing. Springer, pp. 67–76.

Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A., 2002. NUSMV 2: an opensource tool for symbolic model checking. In: Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14. Springer, pp. 359–364.

Costa, B., Pires, P.F., Delicato, F.C., 2016a. Modeling IoT applications with sysml4iot. In: 2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, pp. 157–164.

Costa, B., Pires, P.F., Delicato, F.C., 2019. Modeling SOA-based IoT applications with soaML4iot. In: 2019 IEEE 5th World Forum on Internet of Things (WF-IoT). IEEE, pp. 496–501.

Costa, B., Pires, P.F., Delicato, F.C., 2020. Towards the adoption of OMG standards in the development of SOA-based IoT systems. J. Systems Software 169, 110720.

Costa, B., Pires, P.F., Delicato, F.C., Li, W., Zomaya, A.Y., 2016b. Design and analysis of IoT applications: a model-driven approach. In: 2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech). IEEE, pp. 392–399.

Crnkovic, I., Malavolta, I., Muccini, H., Sharaf, M., 2016. On the use of component-based principles and practices for architecting cyber-physical systems. In: 2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE). IEEE, pp. 23–32.

Grönninger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S., 2014. Textbased modeling. arXiv preprint arXiv:1409.6623 .

Harrand, N., Fleurey, F., Morin, B., Husa, K.E., 2016. 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, Saint-Malo, France, October 2–7, 2016, pp. 125–135. http://dl.acm.org/citation.cfm?id=2976812.

Ihirwe, F., Di Ruscio, D., Gianfranceschi, S., Pierantonio, A., 2023. ChessIoT: a model-driven approach for engineering multi-layered iot systems. J. Comput. Lang. 78, 101254.

ISO/IEC/IEEE, 2022. Systems and software engineering – architecture description (ISO/IEC/IEEE 42010:2022). http://www.iso-architecture.org/ieee-1471/cm/.

Jajodia, S., Liu, P., Swarup, V., Wang, C., 2010. Cyber Situational Awareness. Vol. 14. Springer.

Kirchhof, J.C., 2024. From Design to Reality: An Overview of the MontiThings Ecosystem for Model-Driven IoT Applications. Springer Nature Switzerland, Cham. pp. 45–71.

Kirchhof, J.C., Rumpe, B., Schmalzing, D., Wortmann, A., 2022. Montithings: model-driven development and deployment of reliable iot applications. J. Syst. Software 183, 111087.

Malavolta, I., Muccini, H., Sharaf, M., 2015. A preliminary study on architecting cyber-physical systems. In: Proceedings of the 2015 European Conference on Software Architecture Workshops. ACM, p. 20.

Mazanec, M., Macek, O., 2012. On general-purpose textual modeling languages. In: Dateso. Vol. 12. Citeseer, pp. 1–12.

Mazzini, S., Favaro, J., Baracchi, L., 2015. A model-based approach across the iot lifecycle for scalable and distributed smart applications. In: 2015 IEEE 18th International Conference on Intelligent Transportation Systems. IEEE, pp. 149–154.

McEwen, A., Cassimally, H., 2013. Designing the internet of things. John Wiley & Sons.

Muccini, H., Sharaf, M., 2017a. Caps: a tool for architecting situational-aware cyber-physical systems. In: Software Architecture (ICSA), 2017 IEEE International Conference on. IEEE.

Muccini, H., Sharaf, M., 2017b. Caps: architecture description of situational aware cyber physical systems. In: Software Architecture (ICSA), 2017 IEEE International Conference on. IEEE, pp. 211–220.

Muccini, H., Spalazzese, R., Moghaddam, M.T., Sharaf, M., 2018. Self-adaptive IoT architectures: an emergency handling case study. In: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings. ACM, p. 19.

Salihbegovic, A., Eterovic, T., Kaljic, E., Ribic, S., 2015. Design of a domain specific language and IDE for internet of things applications. In: 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). IEEE, pp. 996–1001.

Sharaf, M., Abughazala, M., Muccini, H., Abusair, M., 2017. Capsim: simulation and code generation based on the caps. In: Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings, pp. 56–60.

Sharaf, M., Muccini, H., Abughazala, M., 2018a. ArIA: arduino code generation based on the caps. In: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, pp. 1–4.

Sharaf, M., Muccini, H., Sahay, A., 2018b. A comparative analysis of self-adaptive patterns in cyber-physical systems. In: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings. ACM, p. 46.

Sundmaeker, H., Guillemin, P., Friess, P., Woelfflé, S., et al., 2010. Vision and challenges for realising the internet of things. Cluster Eur. Res. Projects Internet Things, Eur. Commision 3 (3), 34–36.

SWEETHOME-SWEET, H.. 3D (2015). Sweet Home 3D https://www.sweethome3d.com/.

Thramboulidis, K., Christoulakis, F., 2016. Uml4IoT-a UML-based approach to exploit IoT in cyber-physical manufacturing systems. Comput. Ind. 82, 259–272.

Arduino, 2022. Arduino hardware. https://www.arduino.cc/en/hardware.

ECSA CAPS, 2021. ECSA 2021 CAPS. https://github.com/karthikv1392/PedCupSim.

Vasari Art Experience, 2018. Vasari art experience. https://www.vasariartexperience.it/.

Vasari Models, 2021. Vasari models. https://github.com/moamina/CupcarbonSimulationProject/tree/master/VASARI.

Barriga, J.A., Clemente, P.J., Pérez-Toledano, M.A., Jurado-Málaga, E., Hernández, J., 2023. Design, code generation and simulation of IoT environments with mobility devices by using model-driven development: SimulateIoT-Mobile. Pervasive and Mobile Computing 89, 101751.

Abughazala, M., Muccini, H., 2026. A visionary architecture for adaptive data contracts in behavior-driven IoT systems. In: Software Architecture. ECSA 2025 Tracks and Workshops. Lecture Notes in Computer Science, 15982. Springer, Cham. 1–16. https://doi.org/10.1007/978-3-032-04403-7_32.

Sharaf, M., Abughazala, M., Muccini, H., Abusair, M., 2017. Simulating architectures of situational-aware cyber-physical space. In: Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings. ACM, pp. 66–67.