Towards Dynamic Classification in Domain Modeling with Jjodel

Maurice H. ter Beek[1], Antonio Bucchiarone(✉)[2], Alfonso Pierantonio[2], and Bran Selic[3]

[1] FMT Lab, CNR–ISTI, Pisa, Italy
maurice.terbeek@isti.cnr.it
[2] SWEN, Università degli Studi dell'Aquila, L'Aquila, Italy
{antonio.bucchiarone,alfonso.pierantonio}@univaq.it
[3] Malina Software, Ottawa, ON, Canada
selic@acm.org

Abstract. This work addresses the limitations of traditional object-oriented classification in representing evolving systems. Conventional classification enforces rigid hierarchies that hinder dynamic reclassification, concurrent viewpoints, and transient or overlapping states, thus limiting their usefulness for systems that exhibit change, context sensitivity, or adaptation. We propose a modeling notation that extends UML class diagrams with declarative behavioral dynamics directly embedded in structural information. Unlike approaches that separate structure and behavior, our notation unifies them in a framework suitable for both conceptual/domain modeling and evolutionary, context-aware dynamics. The notation is fully defined, including abstract syntax, diagrammatic syntax, semantics, and simulation, within the Jjodel platform, ensuring rigor and tool support. Its structural nature enables the declarative specification of dynamics without imperative constructs. Moreover, the notation can be connected to frameworks for formal verification and model checking, enabling analysis of dynamic properties. Applicability is illustrated through a context-aware scenario in which enriched structural models can be simulated, reasoned about, and eventually verified.

Keywords: Dynamic Classification · Domain Modeling · Evolving Systems · Jjodel

1 Introduction

As software systems continue to evolve toward greater autonomy, adaptivity, and integration with the physical world, the challenges of modeling both their structure and behavior become increasingly pronounced. Within the *domain modeling* realm, the focus has traditionally been on the structural aspects of systems, identifying entities, relationships, and constraints that define the problem domain [10, 11]. Behavioral aspects, in contrast, are often introduced only in later phases of the modeling process or are expressed through separate, dedicated formalisms such as state machines, statecharts, or activity diagrams [14, 33].

Although these behavioral notation has proven effective in many engineering contexts, they offer limited abstraction capabilities and often require cumbersome refinement steps to bridge the conceptual gap between high-level domain understanding and executable specifications [1, 19].

Modern cyber-physical and context-aware systems, like autonomous drones, self-organizing sensor networks, or digital twins, expose the shortcomings of this separation between structure and behavior. These systems must dynamically adapt to fluctuating environmental conditions, operational contexts, and user requirements [22]. Their behavior is no longer static or neatly modular, but fluid, overlapping, and continuously evolving. Capturing such dynamics within a coherent, analyzable model remains one of the central challenges of software and systems engineering, particularly when domain models are expected to serve as enduring conceptual anchors throughout system evolution.

The engineering mechanism that has historically enabled abstraction and manageability is the *divide-and-conquer* principle: decomposing complexity into discrete, hierarchically organized components. This approach has long guided domain modeling practices, where systems are conceptualized in terms of entities and their relations, and behavioral specifications are subsequently layered on top [18]. Although effective in traditional, well-bounded domains, this separation falters when systems must continuously interact with—and adapt to—their environment. In nature, modularity exists, yet its boundaries are diffuse and dynamic: a single biological entity may participate simultaneously in multiple overlapping processes. Similarly, in engineered systems, components often assume context-dependent roles whose behavioral facets cannot be captured by rigid statically defined hierarchies [29].

Object-oriented classification, one of the cornerstones of both programming and domain modeling, illustrates this limitation. It supports abstraction and reuse through structural and behavioral encapsulation, but enforces a form of behavioral rigidity: once instantiated, the class membership of an object and associated operations are typically fixed. This assumption clashes with the needs of adaptive systems, where entities evolve, acquire or lose capabilities as contexts change [7]. For example, an autonomous drone can transition from an exploratory to a safety critical operational mode as weather conditions deteriorate, invoking distinct behavioral regimes while maintaining its identity as the same domain entity. Such transitions are continuous and context-sensitive, yet traditional class-based domain models remain bound to static binary notions of membership and state [21].

The motivation for this work is to overcome these long-standing limitations by rethinking domain modeling as an inherently dynamic endeavor, one in which structure and behavior co-evolve and can be expressed uniformly. We argue that classification should not be regarded as a static, structural property of objects but as a *dynamic, viewpoint-dependent construct* capable of evolving with the system it represents. In this view, domain models become living artifacts: they not only describe the ontology of the system, but also encode its potential for behavioral transformation [38].

Contributions. This paper introduces a modeling notation that extends traditional domain modeling formalisms, specifically UML class diagrams, with *declarative behavioral dynamics* embedded directly in structural specifications. Unlike conventional approaches that isolate behavioral descriptions in auxiliary state-based models, our notation unifies structure and behavior within a single declarative framework suitable for both conceptual modeling and context-aware evolution. The notation is formally defined through abstract syntax, diagrammatic conventions, semantics, and executable simulation, and has been implemented within the *Jjodel* platform [6][4] to ensure rigor and tool support. By embedding behavioral variation at the structural level, the approach preserves the expressiveness of domain modeling while enabling the specification and verification of dynamic, context-dependent phenomena without resorting to imperative constructs. The effectiveness of the approach is demonstrated through a context-aware scenario illustrating how enriched domain models can be simulated, reasoned about and verified within an integrated modeling environment.

Outline. The remainder of this paper is organized as follows. Section 2 discusses the background and related work, focusing on the limitations of traditional classification models and previous attempts to introduce dynamic perspectives. Section 3 presents the proposed dynamic classification notation and its metamodel. Section 4 describes the workbench and implementation within the Jjodel environment. Section 6 explores formal bridges to verification frameworks. Finally, Section 7 concludes the paper with a discussion and directions for future work.

2 Background and Related Work

The notion of *classification* has long served as a foundational abstraction in both conceptual modeling and programming. It enables software engineers to organize entities into classes according to common properties and behavior, simplifying reasoning about complex systems through modularization and reuse. This abstraction has proved remarkably effective within the divide-and-conquer paradigm that underpins classical software and systems engineering. By encapsulating functionality and structure within well-defined units, traditional classification has contributed to the clarity, maintainability, and reliability of software systems.

However, as modern software becomes increasingly interconnected with the physical world, through cyber-physical systems, Internet-of-Things (IoT) infrastructures, and digital twins, its underlying models must accommodate the fluid and evolving nature of the entities they represents. The crisp modular boundaries of traditional engineering abstractions (cf., e.g., [4]) rarely correspond to the continuity, overlap, and dynamism found in socio-technical systems. In such contexts, conventional forms of classification appear increasingly restrictive and often misleading.

[4] https://www.jjodel.io/

In most of the main object-oriented languages, including Java, C#, and C++, each object is an instance of a single fixed class throughout its lifetime. Class membership is determined by the static set of structural and behavioral features defined by that class. This rigid binding prevents an accurate representation of the mutable entities whose properties evolve over time. For instance, an autonomous drone may alter its behavior in response to changing environmental conditions, reducing speed and adjusting its flight altitude under strong winds, switching sensors and control modes in low visibility, or returning autonomously to base when battery levels are critical. Each of these operational contexts entails a different configuration of capabilities and constraints, yet the underlying entity remains the same drone. Such *dynamic reconfiguration* cannot easily be modeled if class membership remains immutable, since the system must effectively transition between distinct behavioral states while maintaining a continuous identity.

Existing techniques provide only partial workarounds. Multiple inheritance allows an entity to aggregate features from several parent classes, but it introduces well-known semantic complications such as the *diamond inheritance* problem and the potential propagation of irrelevant or conflicting features (cf., e.g., [23,34,36]). The *State* design pattern [12] offers another means of representing changing behavior, yet it does so indirectly by externalizing state management and obscuring the intrinsic connection between the entity and its dynamic roles. Both mechanisms highlight a conceptual gap: the inability of current programming languages to treat reclassification as a first-class modeling capability rather than a workaround.

A second, equally fundamental limitation concerns the assumption of a single, dominant classification hierarchy. In practice, the same entity can be viewed from multiple legitimate perspectives, each determined by a particular set of concerns or viewpoints [17]. For example, a person might simultaneously belong to classes defined by employment status, age group, and professional qualification. Each viewpoint focuses on a different subset of features, some relevant in one context and others irrelevant in another. The principle of separation of concerns, a cornerstone of software architecture, suggests that these viewpoints should remain distinct; however, mainstream object-oriented languages enforce a unification that privileges one hierarchy over others. The resulting models lose expressiveness and become difficult to evolve when alternative perspectives are introduced.

A third limitation concerns *transient* and *overlapping* forms. Context-aware systems rarely operate within discrete, mutually exclusive modes; instead, they exhibit continuous adaptations in which capabilities emerge, diminish, or coexist over time. Consider an autonomous drone that gradually transitions between navigation modes as environmental conditions evolve: when facing light turbulence, it may only adjust its stabilization parameters, whereas in severe weather it progressively activates obstacle-avoidance and emergency-landing subsystems. During such transitions, multiple behavioral configurations can be partially active, resulting in hybrid or *in-between* operational states. Traditional classification models, based on binary feature inclusion, fail to capture these gradual

transformations. Likewise, overlapping membership, where an entity validly exhibits characteristics of several operational classes at once, is excluded by design, even though it reflects the behavior of adaptive and cyber-physical systems.

These conceptual inadequacies have practical implications. As software systems increasingly rely on adaptive, context-aware, or self-configuring mechanisms, the ability to represent entities that dynamically change classification or are described from multiple concurrent perspectives becomes crucial. Digital twin systems [20, 24] exemplify this demand: their fidelity depends on maintaining consistent yet adaptive mappings between physical and virtual states, a process that inherently requires dynamic reclassification and viewpoint management. This perspective also aligns with the needs of modern self-adaptive systems [37], where entities must reconfigure not only their control logic, but also their conceptual roles and available features depending on context (cf., e.g., [30–32], which deals with an autonomous underwater robot case study).

Several researchers have recognized these limitations. Initially, Jackson [18] criticized the rigidity of traditional classification hierarchies, arguing that they obscure alternative viewpoints and hinder flexibility. The language *Smalltalk-80* [13] supported a primitive form of dynamic reclassification through its *become* operation, allowing objects to change their class identity at runtime. However, this mechanism was not adopted by later statically typed languages due to its incompatibility with static type checking. The experimental *Fickle* language [9] extended this idea by allowing objects to migrate between classes in a controlled way, but remained largely a research prototype. The concept of *typestates* [8], formalizing object behavior as a set of state-dependent types, provided another influential contribution, anticipating later approaches to dynamic typing and behavioral modeling.

State-based modeling notations such as *Statecharts* [14] have long provided a powerful means to represent dynamic system behavior and context-dependent transitions. By hierarchically structuring states and allowing for concurrency and event-driven transitions, Statecharts extend classical finite state machines to capture complex reactive behavior in embedded and cyber-physical systems. However, despite their expressive power, Statecharts primarily focus on the *control flow* of a system rather than its *classification semantics*. They specify how an entity changes state in response to stimuli, but do not redefine its structural or behavioral features as part of those transitions. In other words, a drone modeled with Statecharts can switch between *Flying*, *Charging*, and *Emergency* states, yet its underlying class membership and associated attributes remain fixed. Integrating the expressive behavioral modeling of Statecharts with a more flexible viewpoint-based classification mechanism would enable richer representations of adaptive systems in which both control logic and conceptual identity evolve consistently over time.

Beyond programming languages, modeling standards such as the *Unified Modeling Language* (UML) [25] introduced the concept of *generalization sets* to support orthogonal classification dimensions. This construct allows modelers to organize generalizations into independent, possibly overlapping, hierarchies,

a notion closely aligned with viewpoint-based classification. However, the UML specification explicitly omits details about how such mechanisms should be realized or maintained in executable systems, and the idea has not been incorporated into any formal semantics of UML (cf., e.g., fUML [26]).

Despite these early insights, the prevailing classification model has remained largely unchanged for decades. The resulting gap between the expressive needs of modern software systems and the capabilities of current modeling and programming technologies has become increasingly apparent. As software assumes more dynamic, adaptive, and interdisciplinary roles, mirroring biological, social, or ecological systems, the need arises for frameworks that can represent multiple coexisting viewpoints, transient forms, and dynamic reclassification as intrinsic features of the modeling language itself.

The viewpoint-based approach proposed in this paper seeks to address precisely these challenges. It reframes classification not as an intrinsic property of objects, but as a *subjective construct*, a set of perspectives applied to a collection of entities to serve human reasoning and system comprehension. By allowing classification to evolve dynamically and coexist across multiple schemes, this approach aims to provide a more faithful, flexible, and semantically coherent representation of modern complex, evolving systems.

3 Dynamic Classification

As modern software increasingly operates within complex cyber-physical environments, the need to incorporate accurate models of dynamically evolving real-world phenomena becomes more acute. Traditional modeling languages such as UML [25] and SysML [27,28], while expressive for structural design, are grounded in static type systems derived from classical logic-based programming languages. These formalisms provide limited support for expressing entities whose *classification* changes over time or across contextual boundaries [33,35]. The resulting models tend to separate structural aspects from behavioral dynamics, often relegating the latter to auxiliary state machines or activity diagrams, which restricts abstraction and integration.

In contrast, the Dynamic Classification Notation (DCN) aims to unify structure and behavior by allowing entities to adopt multiple, possibly concurrent, classifications depending on the *viewpoint* applied. This capability reflects the way real-world entities are perceived and organized: classification is not intrinsic to an entity but depends on the modeling perspective and the selected discriminant property. For example, an individual can be simultaneously classified as a *parent*, an *employee*, and a *citizen*, depending on the viewpoint of interest. Similarly, a drone in an adaptive cyber-physical system can be classified as *navigating*, *avoiding collision*, or *returning to base*, depending on its current operational context. A simplified version of the metamodel for DCN is illustrated in Fig. 1. It formalizes the relationships among the core modeling constructs: *StaticClass*, *DynamicClass*, *Viewpoint*, and *Handler*.

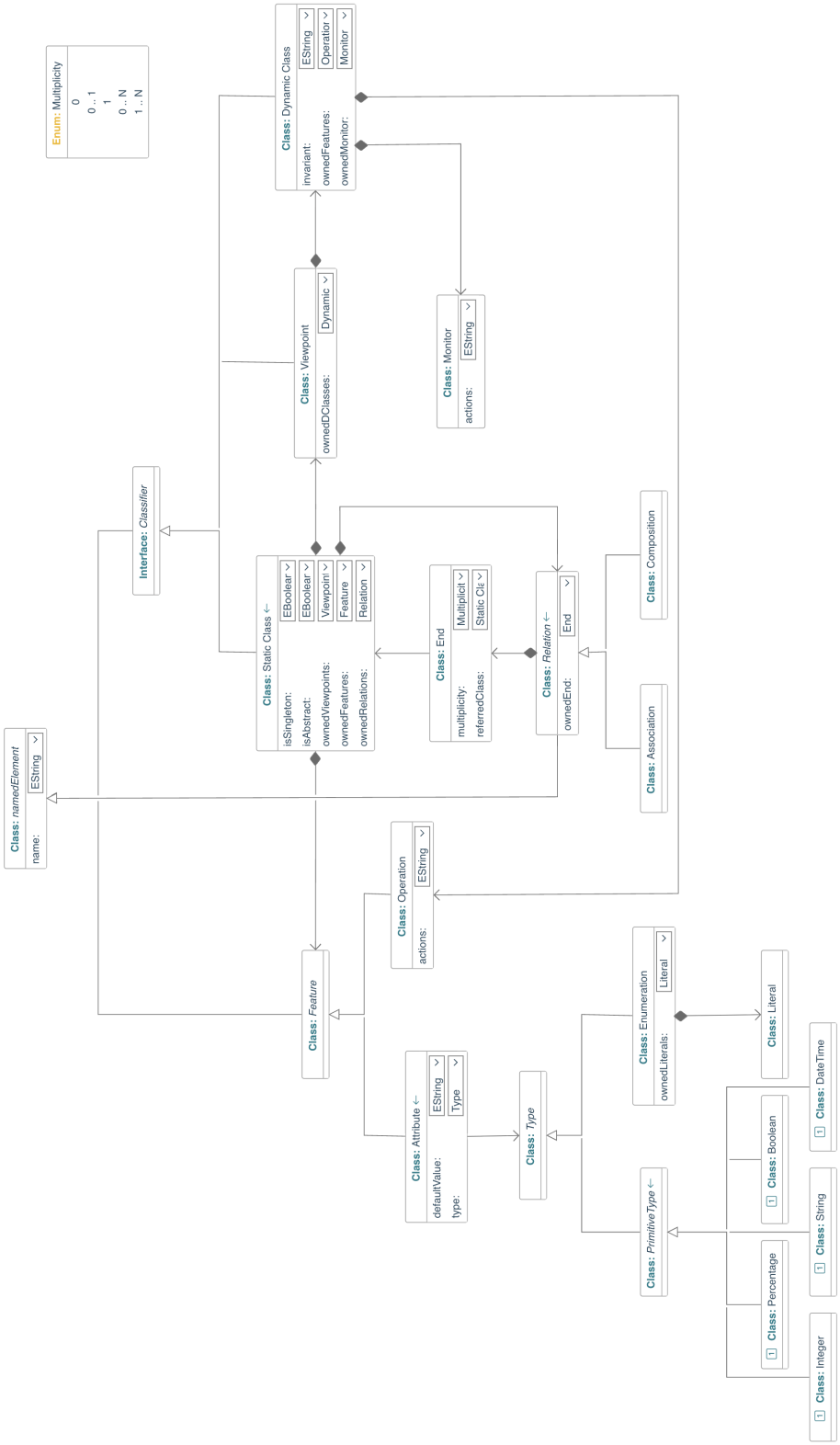

Fig. 1. The metamodel of the Dynamic Classification Notation (DCN).

The metaclass `StaticClass` represents the enduring structural concept of a phenomenon being modeled. Its instances embody entities that may assume different dynamic classifications over time. Each `StaticClass` aggregates a collection of `Features` (attributes and operations), `Relations`, and one or more `Viewpoints`. A `Viewpoint` denotes a modeling perspective that defines a set of possible `DynamicClasses`, each corresponding to a distinct contextual or temporal state of the same static entity.

A `DynamicClass` encapsulates the properties, invariants, and behavior that are valid when the represented phenomenon occupies a specific classification. Its `invariant` (an `EString` expression) specifies the conditions under which the classification is valid. The `ownedFeatures` and `ownedMonitor` references define the attributes, operations, and behavioral responses that are active within a given classification. The `Monitor` metaclass, in turn, enables the specification of the effects that changes in one aspect of the system have on dependent quantities. For instance, an increase in engine power can influence other performance parameters, such as speed or battery consumption. This mechanism directly integrates behavioral dynamics into the structural representation at the domain level, providing a declarative means to capture interdependencies among evolving features.

The model supports multiple concurrent viewpoints, enabling entities to be classified simultaneously according to different discriminants. For instance, in a biological domain, a `StaticClass` representing a *Frog* may have one `Viewpoint` describing its *life cycle* (*Tadpole*, *Adult*), and another representing its *health state* (*Healthy*, *Infected*). Each `DynamicClass` within these viewpoints activates or deactivates specific `Features` and `Operations` accordingly. For example, a frog in the *Tadpole* classification has a tail but no limbs, while in the *Adult* classification it has limbs but no tail, both modeled as features that dynamically appear or disappear based on the current classification.

Formally, the metamodel integrates declarative behavior for dynamic classification directly into structural models, bridging the gap between traditional domain modeling and behavioral specification. This approach provides a unified foundation for conceptual, executable, and verifiable representations of dynamic, adaptive systems.

4 Workbench and Implementation

The Dynamic Classification Notation (DCN), introduced in the previous section, has been implemented within *Jjodel* [6][4], a model-driven platform that supports the engineering of domain-specific modeling languages (DSMLs) through the explicit specification of their syntax, semantics, and validation rules. Jjodel integrates these aspects within a unified metamodeling framework, providing both design-time modeling and runtime execution capabilities that preserve the correspondence between abstract language definitions and executable behavior.

4.1 Defining a Modeling Language in Jjodel

A modeling language in Jjodel is defined by specifying:

- an *abstract syntax*, expressed as a metamodel capturing the domain concepts and their relationships;
- a *concrete syntax*, which defines how those concepts are represented and edited visually or textually; and
- a *semantic mapping*, which provides an interpretation of model elements, enabling simulation or analysis.

Each of these components is defined declaratively within the Jjodel workbench. The platform also integrates constraint checking through OCL-style invariants and offers validation and testing facilities for constraints [5].

For the DCN, the abstract syntax corresponds to the metamodel described in Section 3. The metamodel is specified as an `Ecore` model[5] that defines the key metaclasses `StaticClass`, `DynamicClass`, `Viewpoint`, `Feature`, and `Monitor`. These elements define the structural and dynamic aspects of the modeling language. Once defined, the metamodel is registered in the Jjodel repository and linked to a corresponding graphical concrete syntax.

4.2 Concrete Syntax of the Dynamic Classification Notation

The concrete syntax of the DCN employs a diagrammatic representation designed to clearly distinguish structural, contextual, and dynamic dimensions. An example of a DCN model rendered in Jjodel is shown in Fig. 2.

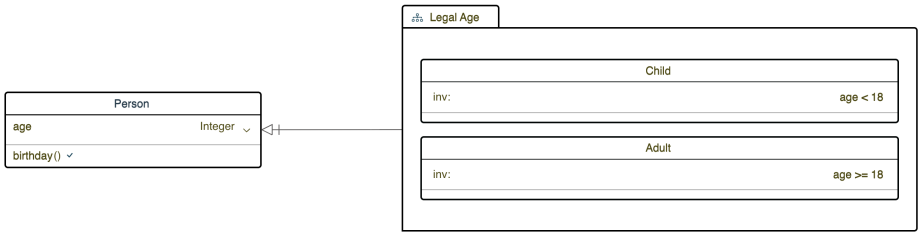

Fig. 2. Example of a DCN model: a `Person` with its `Legal Age` viewpoint specification. Static classes are shown in gray, viewpoints in blue, and dynamic classes in white.

In concrete syntax, static classes represent the enduring structural elements of the system domain (e.g., `Person`). These correspond to instances of the `StaticClass` metaclass. Viewpoints (blue containers) express contextual perspectives from which a static class can be classified dynamically, such as `Legal Age`.

[5] https://eclipse.dev/emf/

Dynamic classes (white boxes within viewpoints) represent specific classifications under each viewpoint (e.g., `Child`, `Adult`). Each dynamic class specifies its validity condition (`invariant`), active features, and possible behavior through `Monitors`.

4.3 Integration with Object Diagrams

To support simulation and analysis, the DCN metamodel is paired with the object diagram metamodel illustrated in Fig. 3, which defines the structure of runtime configurations.

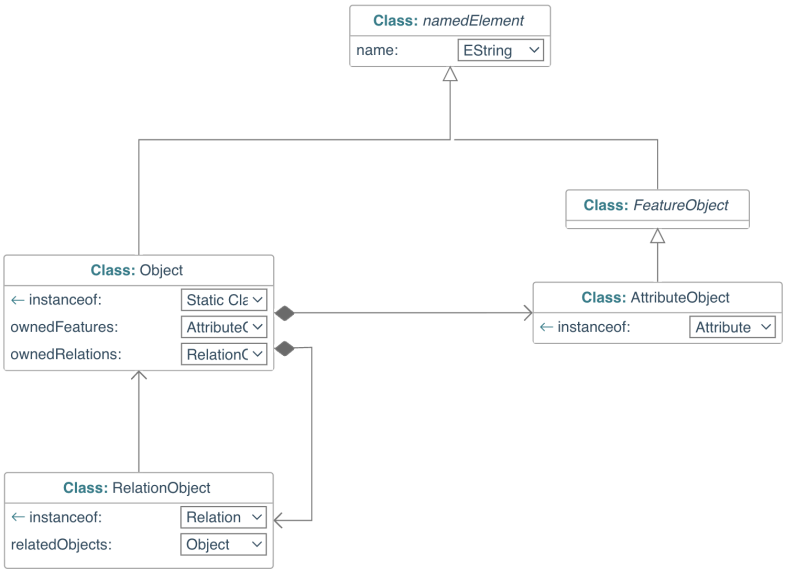

Fig. 3. The metamodel of object diagrams.

This metamodel specifies the structure of instance-level representations that conform to DCN models (i.e., extended class diagrams). To ensure that object instances remain consistent with their corresponding classes in the DCN metamodel, the approach adopts a *multi-view specification*. In this metamodel, the metaclass `Object` denotes an `instanceof` a `StaticClass` and aggregates its `ownedFeatures` (attributes) and `ownedRelations`. The symbol ← preceding the reference name indicates that it is a *cross-link reference*, i.e., a reference targeting a metaclass defined in another metamodel. Accordingly, the `instanceof` references in `Object`, `AttributeObject`, and `RelationObject` point to `StaticClass`, `Attribute`, and `Relation` in the DCN metamodel, respectively. Together, these elements define runtime configurations that conform to both the object diagram metamodel and the extended class diagram specified by the DCN.

In this setting, each object diagram must conform to:

1. the *object diagram metamodel*, ensuring syntactic correctness (objects, links, and attribute values are well-typed); and
2. a specific *extended class diagram*, which is itself an instance of the DCN metamodel, ensuring semantic consistency with the modeled viewpoints and classification rules, as illustrated in Fig. 4.

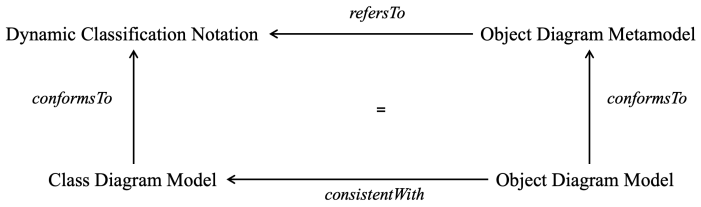

Fig. 4. Conformance and consistency relationships between class and object diagrams.

This dual conformance relation allows object diagrams to represent the dynamic state of a DCN model at runtime. For example, in Fig. 5, a Mario Nintendo instance of `Person` currently satisfies the invariant of the `Adult` dynamic class from the `LegalAge` viewpoint, as his `age` is 20. Conversely, `Luigi Nintendo` is classified as a `Child`, since his `age` is 13. Such configurations are automatically validated within Jjodel, ensuring moreover that runtime instances respect both structural and dynamic constraints. The operation `birthday()` is defined as `age = age + 1` and it can be invoked on each `Person` instance to trigger reclassification when the corresponding invariant condition changes.

This integration between structural modeling (through static and dynamic classes) and instance-level configurations (through object diagrams) provides a seamless workflow: modelers can define, visualize, and simulate dynamic classification behavior directly within the Jjodel workbench, ensuring coherence between conceptual and executable models.

In the next section, we illustrate the method through a more extended case study.

5 Case Study: Specification of an Electric Drone

To demonstrate the applicability of the DCN, we consider a case study based on the specification of an *electric drone*. The system exemplifies a cyber-physical architecture that combines autonomous behavior, adaptive control, and continuous interaction with the physical environment. The specification highlights how the DCN supports multiple viewpoints (structural, contextual, and behavioral) within a unified modeling framework formalized in the Jjodel platform.

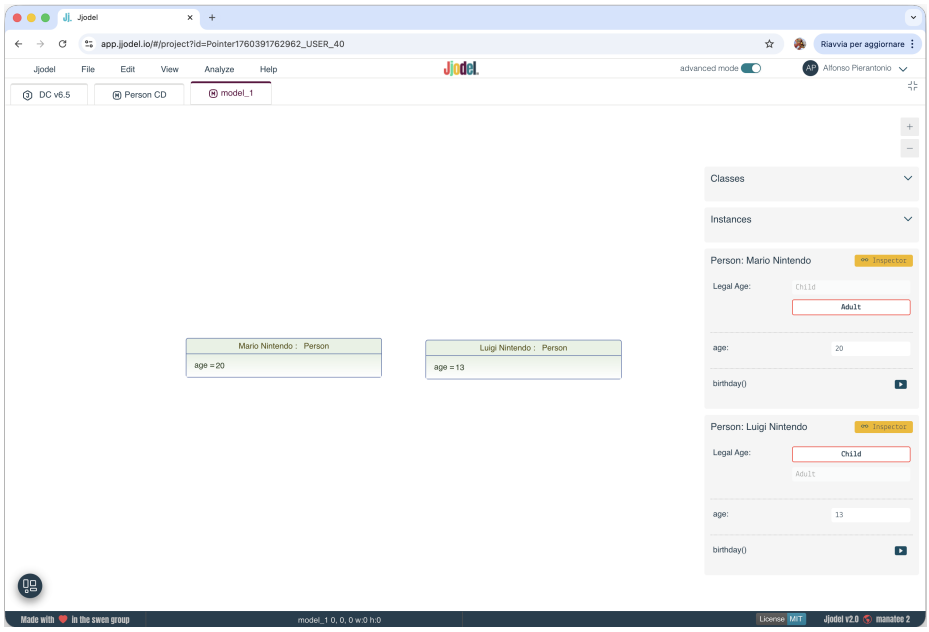

Fig. 5. Simulation of the `Person` domain in Jjodel. The model dynamically updates classifications (`Child` or `Adult`) based on the `LegalAge` viewpoint and current attribute values.

5.1 DCN Class Diagram Specification

The DCN class diagram shown in Fig. 6 models the structural and dynamic features of the drone and its supporting components. The central element is the `Drone` class, which encapsulates measurable physical properties such as `distance`, `speed`, `power`, `battery`, and `vmax`. The following two viewpoints are defined to represent the operational and environmental contexts of the drone.

Mode: describes the operational life cycle of the drone through four dynamic classes: `Charging`, `Ready`, `Flying`, and `Empty`. Each dynamic class specifies the invariants that determine when it is active. For example, the drone is in `Charging` mode when `distance = 0`, `speed = 0`, and `battery < 100`, while the `Flying` mode is maintained when `battery > 0` and `speed > 0`. Behavioral dynamics, such as `rampUp()` or `rampDown()`, are expressed as declarative operations that modify relevant attributes. Please note that while `Charging`, `Ready`, and `Flying` are mutually exclusive, `Empty` overlaps `Charging` to denote that the battery is completely empty.

Alert: defines the environmental classification of the operating context based on the weather conditions detected by the onboard and external sensors. Three dynamic classes (`Safe`, `Moderate`, and `Adverse`) are distinguished by

invariant conditions over variables such as wind speed, temperature, humidity, and rain intensity. These classifications support adaptive decisions, e.g., slowing down or returning to base in adverse weather.

The drone interacts with a `WeatherAssessmentUnit` (WAU), which aggregates several types of `Sensor` objects: `WindSensor`, `TempSensor`, `RainSensor`, and `HumiditySensor`. Each sensor provides readings of physical quantities, such as wind speed, temperature, rain, and humidity, modeled by instances of the `PhysicalQuantity` class. The `SensorStatus` enumeration defines possible sensor states (`OK`, `Fault`, `Calibrating`), allowing fault-tolerant behavior to be specified and verified within the DCN framework.

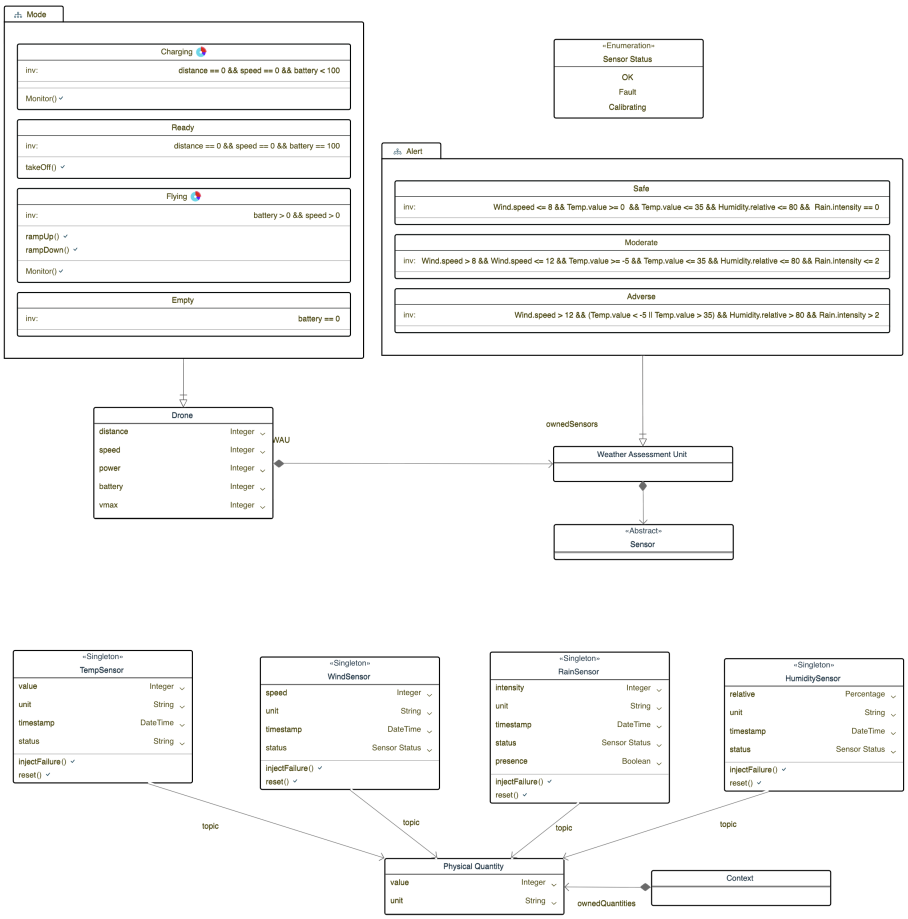

Fig. 6. DCN class diagram of the electric drone system. The model integrates operational (`Mode`) and environmental (`Alert`) viewpoints to describe adaptive behavior.

5.2 Runtime Configuration and Object Diagram

The corresponding object diagram, depicted in Fig. 7, represents a snapshot of a drone runtime configuration.

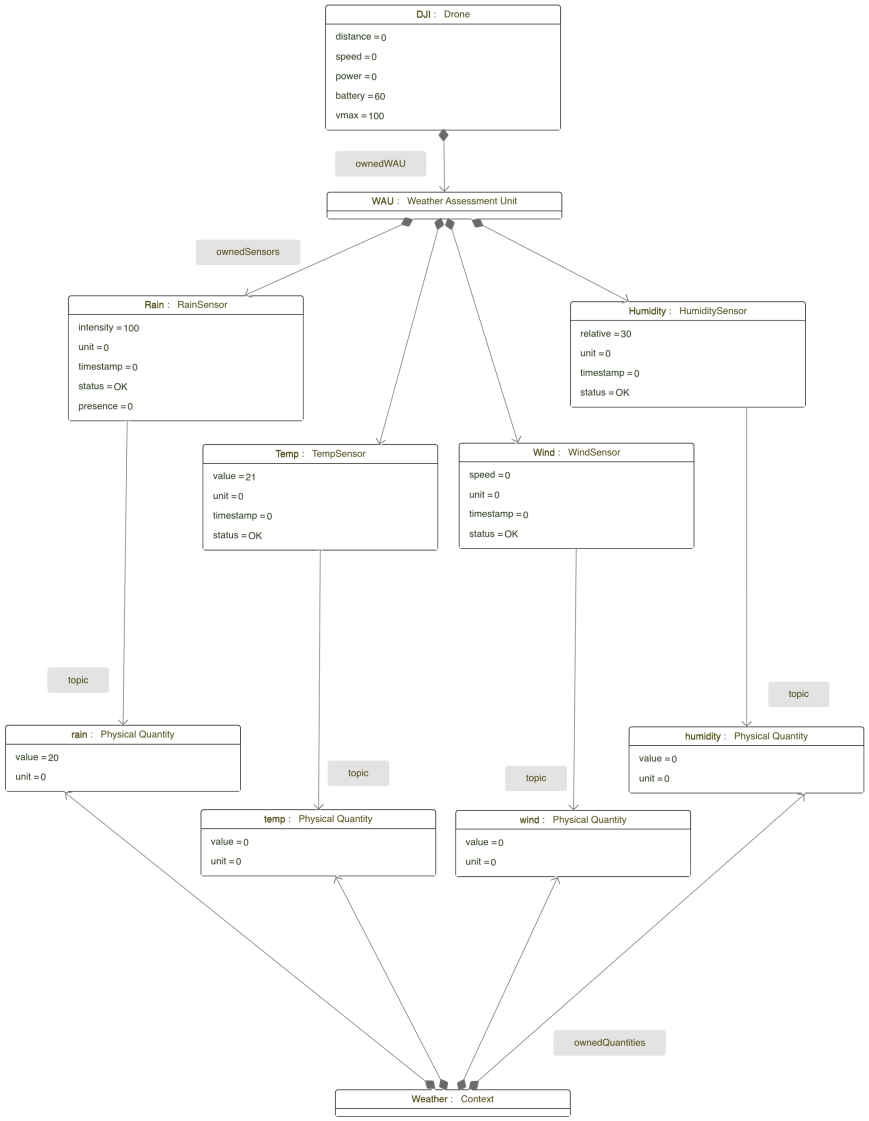

Fig. 7. Object diagram representing a runtime configuration of the electric drone system. The configuration conforms to both the object diagram metamodel and the DCN-based extended class diagram.

In Fig. 7, an instance `DJI:Drone` interacts with its associated `WeatherAssessmentUnit` and sensor instances. The `TempSensor` records a temperature of 21°C, while the `HumiditySensor` reports 30% relative humidity. The wind and rain sensors indicate nominal conditions (`speed = 0 m/s`, `intensity = 0 mm/h`), which collectively satisfy the invariant of the `Safe` dynamic class under the `Alert` viewpoint. Each object instance in the diagram conforms simultaneously to (i) the object diagram metamodel, ensuring that objects, attributes, and relations are well-typed and correctly linked, and (ii) the extended class diagram (the DCN model), ensuring semantic consistency with the operational and environmental viewpoints as illustrated in the commutative diagram in Fig. 4. This multi-level conformance enables simulation and analysis of the adaptive behavior of the drone in Jjodel. For example, when the `battery` level decreases below the threshold or when environmental conditions violate the `Safe` invariants, the system may transition dynamically to another classification (`Charging` or `Adverse`) without requiring re-instantiation. This runtime adaptability exemplifies how the DCN unifies structure and behavior in a single declarative framework.

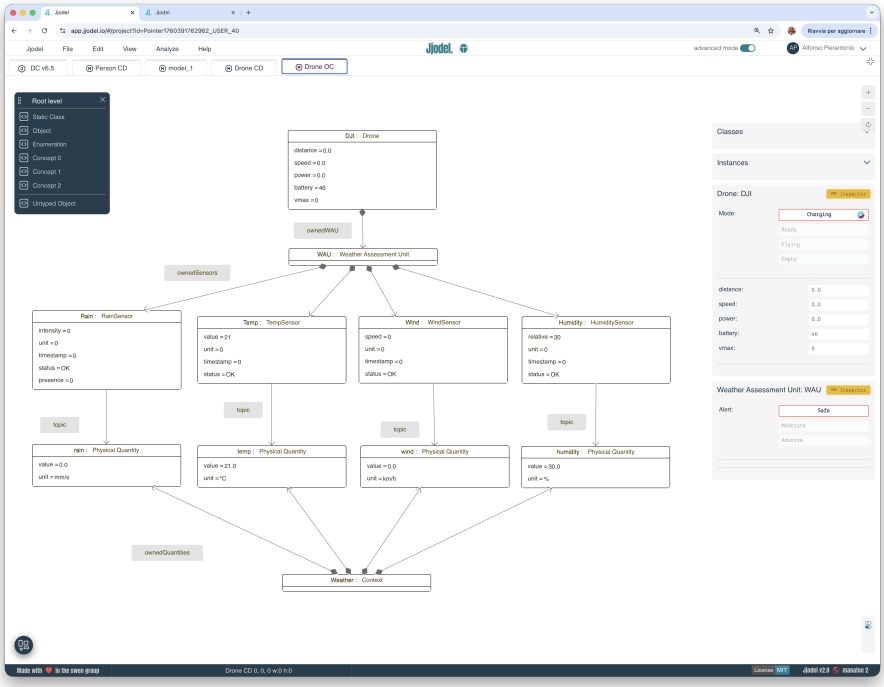

Fig. 8. Runtime execution of the electric drone model in the *Jjodel* platform.

Figure 8 shows the runtime execution environment of the electric drone model within the *Jjodel* platform. The central pane displays the instantiated object diagram corresponding to the previously introduced DCN specification. At the top of the hierarchy, the `DJI:Drone` object represents the drone instance, characterized by its structural attributes such as `distance`, `speed`, `power`, `battery`, and `vmax`, as specified in the corresponding `Drone` static class in Fig. 6. The drone owns a single `WeatherAssessmentUnit (WAU)`, which in turn aggregates four sensor instances: `WindSensor`, `TempSensor`, `RainSensor`, and `HumiditySensor`. Each sensor reports measured quantities through the corresponding `PhysicalQuantity` instances, which are ultimately related to a `Weather:Context` object representing the environmental conditions.

The right-hand inspector panel shows the currently active dynamic classifications for each viewpoint. In this configuration, the drone is in the `Charging` mode under the `Mode` viewpoint, as indicated by the red-highlighted selection. In fact, the associated attributes confirm that the drone is stationary (`distance = 0`, `speed = 0`, `battery = 80`). When the drone is in the `Charging` state, the evolution of the `battery` attribute is governed by the `Monitor` defined in the corresponding dynamic class.

Both `Monitor` and `Operation` elements are specified in a similar way through `actions`; however, while operations can be explicitly invoked by the user or by external events, monitors are continuously evaluated to simulate the ongoing impact of the current state on the attributes of the system.

In the user interface, the presence of a small round icon next to the active dynamic class name indicates that a `Monitor` is currently running, continuously updating the system state according to its declarative specification.

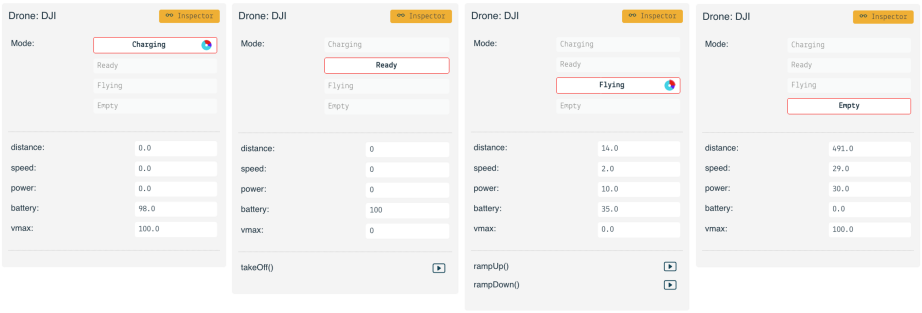

Fig. 9. Different Classifications of `DJI:Drone`.

Figure 9 details the inspector view as the drone transitions through its four operational modes under the `Mode` viewpoint. In the first panel, the drone is in the `Charging` state, with its `battery` gradually increasing under the effect of the running monitor. Once the battery reaches 100%, the invariant of the `Charging`

class is no longer satisfied, and the system automatically reclassifies the drone into the `Ready` state. From this configuration, invoking the `takeOff()` operation activates the `Flying` mode, where drone speed and distance progressively increase as power consumption drains the battery. When the `battery` level eventually reaches zero, the invariant of the `Flying` class is violated, and the drone transitions to the `Empty` state, completing its operational cycle.

The user interface supports interactive manipulation of attribute values, allowing modelers to adjust environmental variables or operational parameters and observe in real time how DCN invariants trigger automatic reclassification. This illustrates how Jjodel provides an executable environment for simulating and validating dynamic classification models at runtime.

Overall, this case study illustrates how the DCN supports the modeling and simulation of adaptive cyber-physical systems. Moreover, the Jjodel workbench provides modelers with a means to capture multiple contextual perspectives while preserving formal rigor and executability. The study is preliminary to potential verification, as discussed in the next section.

6 Formal Bridging to Verification Frameworks

The declarative semantics of the DCN lends itself well to formal verification. DCN models capture both structure and dynamics in a uniform constraint-based form, facilitating automatic translation into formal models suitable for model checking. In this section, we sketch how such a mapping could be defined and applied to the case study, relate it to existing approaches from the literature, and outline directions for future work.

6.1 Mapping DCN Models to Formal Models

The DCN notation and models can be connected to state machines to enable the analysis of dynamic properties through formal verification, for instance, by means of model checking. Concretely, state machines should properly reflect the dynamicity of the subclasses of the static class corresponding to the state machine. The behavior (transitions) of the state machines is extracted from the operations that are available in the static class and in its dynamic subclasses, with pre- and postconditions suitably constraining the behavior and allowing state changes based on the validity of feature constraints defined in the form of predicates (logic).

Each DCN model implicitly defines a state space whose states correspond to configurations of dynamic classifications across all viewpoints. More concretely, a state represents a consistent assignment of attribute values, active dynamic classes, and satisfied invariants for all entities in the model. Transitions arise from declarative operations or monitors whose pre- and postconditions describe admissible state changes. Formally, these can be captured as a Kripke structure $\mathcal{K} = (S, s_0, \delta, L)$, in which:

- S is the finite set of reachable DCN configurations;

- $s_0 \subseteq S$ is the set of initial configurations defined by the model initialization;
- $\delta \subseteq S \times S$ is the transition relation induced by operation invocations or monitor evaluations; and
- $L : S \to 2^{\mathrm{AP}}$ labels each state with the atomic propositions (AP) that correspond to the satisfied invariants or the viewpoint predicates.

This mapping preserves the semantics of DCN invariants: a transition between two states is permitted only if the target configuration satisfies all invariant predicates of its newly active dynamic classes. The declarative form of DCN operations (expressed as attribute updates and constraints) eliminates the need for explicit control-flow constructs, simplifying the generation of the corresponding transition relation.

We plan to define model transformations that consistently map DCN class diagrams into state machines that represent the behavior of objects according to a given viewpoint. These transformations will establish the foundation for automated analysis, allowing DCN specifications to be systematically connected to verification back ends (in Jjodel). To this aim, we will follow relevant related approaches from the literature.

6.2 Model Checking and Property Specification

Once we have translated the DCN models into Kripke structures, the resulting formal model can be analyzed using well-known temporal logic-based formal verification tools such as mCRL2[6], NuSMV[7], UPPAAL[8], or UMC[9]. UMC is the UML model checker [2] from the KandISTI family of model checkers [3] developed together with Alessandro Fantechi by the FMT Lab at CNR–ISTI. UMC considers UML models defined as a set of concurrently executing UML state machines describing the dynamic aspects of a system component's behavior, considering both its state-based (e.g., values of object attributes) and its event-based (i.e., related to the executed actions) aspects, represented by a doubly-labelled transition system (L^2TS) or Kripke structure whose transitions are labeled with events.

Typical properties that could then be verified for the case study include the following examples.

Safety: "*A drone never flies with* `battery = 0`".
Liveness: "*Whenever the drone is in* `Charging` *mode and its* `battery = 100`, *it eventually becomes* `Ready`".
Consistency: "*At any time, exactly one dynamic class per viewpoint is active*".

These properties can be expressed in (action-based) CTL, LTL, or (modal) μ-calculus, depending on the specific target verification framework. The translation of invariants and monitors into logical formulae enables us to directly check these properties over the generated transition system.

[6] https://mcrl2.org/
[7] https://nusmv.fbk.eu/
[8] https://uppaal.org/
[9] https://fmt.isti.cnr.it/umc/

6.3 Relation to Existing Formal Approaches

A promising conceptual correspondence can be drawn with the HELENA approach [15,16], which provides a formal foundation for modeling ensemble-based systems centered on the first-class notion of a *role* that expresses the capabilities that a specific instance of a system component requires when participating in a concrete ensemble. To appreciate the analogy, one can think of a component instance as corresponding to a static class, a role to one of its dynamic classes, and a concrete ensemble to a viewpoint. According to this approach, components can change their roles when needed and participate in multiple ensembles simultaneously by playing different roles, thereby focusing on the capabilities required to collaborate in a particular ensemble composition. The typical component behavior of performing a certain role is modeled in terms of Kripke structures, and dynamic logic is used to specify the properties of the ensembles.

Our vision extends this idea by embedding such role- and viewpoint-based dynamics directly within the structural semantics of the modeling language. Unlike UML/fUML or Statecharts, where behavioral semantics are imperative and often detached from structure, DCN's declarative embedding enables a direct and analyzable correspondence between domain-level constraints and execution semantics. This positions DCN as a bridge between conceptual modeling and formal analysis: high-level structural models can be simulated, validated, and formally verified without rewriting them in a dedicated behavioral formalism.

6.4 Research Agenda

We plan to define a model-to-model transformation from DCN metamodel instances to a formal intermediate representation compatible with model checkers. The transformation should:

1. Identify the set of relevant viewpoints to bound the state space;
2. Encode attribute domains and invariants as logical variables and constraints;
3. Generate transition relations from declarative operations and monitors; and
4. Export the resulting structure to the verification back ends.

This line of research aims to provide bidirectional traceability between the DCN model and its formal counterpart, allowing verification results to be visualized directly within the Jjodel environment. Ultimately, this bridge will support the formal analysis of adaptive and context-aware systems, such as the drone case study, enabling rigorous reasoning about their dynamic reclassification and context-dependent behavior.

7 Conclusion and Future Work

The principal contribution of this work lies in its declarative integration of dynamics into structural models. By embedding behavioral invariants and context-dependent transformations directly within UML-like class diagrams, the DCN

enables modelers to reason about evolution and adaptation without relying on imperative or state-based auxiliary formalisms. This yields three key advantages. First, *expressiveness*: modelers can represent overlapping roles, transient states, and concurrent viewpoints using a single unified formalism. Second, *traceability and analyzability*: since dynamic properties are encoded as logical invariants, models remain suitable for both simulation and formal verification. Third, *tool support and usability*: the Jjodel workbench allows users to define, visualize, and execute DCN models interactively, ensuring consistency between conceptual and runtime representations.

The case study on an adaptive electric drone system demonstrates that the approach naturally supports context-aware systems whose behavior depends on environmental factors. Within Jjodel, these dynamics are simulated declaratively through monitors and operations, allowing entities to reclassify themselves automatically when invariant conditions change. This interactive environment bridges conceptual modeling, simulation, and potential verification, embodying the principles of living models or digital twins for cyber-physical systems.

Although promising, the approach also raises several challenges. Declarative specifications can become complex and computationally intensive when models involve numerous viewpoints, constraints, or dynamically interdependent features. Managing the *scalability of invariant evaluation and reclassification at runtime* will require optimization techniques such as incremental constraint solving and selective evaluation strategies. Moreover, *human comprehensibility* remains crucial: as models grow in size, maintaining readability of overlapping viewpoints and dynamic rules may require novel visualization or abstraction mechanisms. Ongoing work in Jjodel is exploring layered representations and semantic zooming to address these issues.

From a methodological perspective, the integration of the DCN into existing model-driven engineering (MDE) workflows requires guidelines for when and how to apply dynamic classification relative to traditional state machines, fUML behavior, or DSMLs. The relationship between declarative reclassification and more conventional imperative semantics represents a fertile ground for further exploration.

Compared with Statecharts and fUML, the DCN adopts a constraint-based rather than control-based view of behavior. This declarative stance favors analyzability and direct linkage to model checking but sacrifices explicit sequencing semantics. In contrast to viewpoint modeling or role-based frameworks such as HELENA, the DCN integrates these concepts directly within the structural layer, maintaining coherence between roles, invariants, and class features. This synthesis positions the DCN as a bridge between conceptual modeling (concerned with expressing domain knowledge) and formal analysis (concerned with verifying behavioral properties).

Future work will pursue two directions. First, the ongoing formalization of the DCN semantics will define systematic transformations from DCN models to intermediate representations suitable for verification back ends, enabling automated reasoning about safety, liveness, and consistency properties within model-

checking tools such as UMC, UPPAAL, NuSMV, or mCRL2. Second, we plan to expand Jjodel's support for hybrid simulation, combining declarative dynamics with real-time data streams to study digital twin scenarios and adaptive systems in operation.

Furthermore, once formal verification is in place, our approach would be particularly well suited for the modeling and analysis of railway systems, the favorite playground of Alessandro Fantechi, since they are cyber-physical, safety-critical, and context-dependent and demand formal verification.

Ultimately, the vision behind Jjodel and the DCN is to enable models that evolve with their systems; not static blueprints but dynamic representations that capture how software and its environment co-adapt over time. This paradigm holds promise for advancing MDE toward the next generation of context-aware, self-adaptive, and verifiable systems.

Acknowledgments. The first two authors are grateful to Alessandro Fantechi for a lasting friendship that began during their years working together in the Formal Methods and Tools (FMT) Lab at CNR–ISTI in Pisa.

This paper, in particular Section 6, benefited from discussions with Rolf Hennicker several years ago on viewpoint-based classifications and transient diagrams.

The research presented in this paper is partially funded by the European Union under the Grant Agreement No 101189664. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency (HADEA). Neither the European Union nor the granting authority can be held responsible for them.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. acatech – National Academy of Science and Engineering (ed.): Cyber-Physical Systems: Driving force for innovation in mobility, health, energy and production. Springer (2011). https://doi.org/10.1007/978-3-642-29090-9
2. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. Science of Computer Programming **76**(2), 119–135 (2011). https://doi.org/10.1016/J.SCICO.2010.07.002
3. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: States and Events in Kand-ISTI: A Retrospective. In: Margaria, T., Graf, S., Larsen, K.G. (eds.) Models, Mindsets, Meta: The What, the How, and the Why Not?, LNCS, vol. 11200, pp. 110–128. Springer (2019). https://doi.org/10.1007/978-3-030-22348-9_9
4. Bencomo, N., Cabot, J., Chechik, M., Cheng, B.H.C., Combemale, B., Wąsowski, A., Zschaler, S.: Abstraction Engineering. arXiv (2024). https://doi.org/10.48550/ARXIV.2408.14074
5. Bucchiarone, A., Di Rocco, J., Di Vincenzo, D., Pierantonio, A.: From OCL to JSX: declarative constraint modeling in modern SaaS tools. arXiv (2025). https://doi.org/10.48550/ARXIV.2509.17629

6. Bucchiarone, A., Di Rocco, J., Di Vincenzo, D., Pierantonio, A.: Modeling in Jjodel: towards bridging complexity and usability in model-driven engineering. Software and Systems Modeling pp. 1–25 (2025). https://doi.org/10.1007/s10270-025-01324-y
7. Cardozo, N., Mens, K.: Programming language implementations for context-oriented self-adaptive systems. Information and Software Technology **143**(C), 106789 (2022). https://doi.org/10.1016/J.INFSOF.2021.106789
8. DeLine, R., Fähndrich, M.: Typestates for objects. In: Odersky, M. (ed.) Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04). LNCS, vol. 3086, pp. 465–490. Springer (2004). https://doi.org/10.1007/978-3-540-24851-4_21
9. Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., Giannini, P.: More Dynamic Object Reclassification: $Fickle_{||}$. ACM Transactions on Programming Languages and Systems **24**(2), 153–191 (2002). https://doi.org/10.1145/514952.514955
10. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley (2004)
11. France, R.B., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. In: Proceedings of the ISCE Workshop on the Future of Software Engineering (FOSE'07). pp. 37–54. IEEE (2007). https://doi.org/10.1109/FOSE.2007.14
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
13. Goldberg, A., Robson, D.: Smalltalk-80: the language and its implementation. Addison-Wesley (1983)
14. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming **8**(3), 231–274 (1987). https://doi.org/10.1016/0167-6423(87)90035-9
15. Hennicker, R.: Role-Based Development of Dynamically Evolving Esembles. In: Fiadeiro, J.L., Ţuţu, I. (eds.) Revised Selected Papers of the 24th IFIP WG 1.3 International Workshop on Recent Trends in Algebraic Development Techniques (WADT'18). LNCS, vol. 11563, pp. 3–24. Springer (2018). https://doi.org/10.1007/978-3-030-23220-7_1
16. Hennicker, R., Klarl, A.: Foundations for Ensemble Modeling – The Helena Approach: Handling Massively Distributed Systems with ELaborate ENsemble Architectures. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. LNCS, vol. 8373, pp. 359–381. Springer (2014). https://doi.org/10.1007/978-3-642-54624-2_18
17. International Standard ISO/IEC/IEEE 42010: Systems and software engineering — Architecture description (December 2011). https://doi.org/10.1109/IEEESTD.2011.6129467
18. Jackson, M.: Object-Orientation: Classification Considered Harmful. In: Proceedings of NordDATA'91. pp. 107–121 (1991), https://www.researchgate.net/publication/246987164_Object-Orientation_Classification_Considered_Harmful
19. Jureta, I.J., Borgida, A., Ernst, N.A., Mylopoulos, J.: The Requirements Problem for Adaptive Systems. ACM Transactions on Management Information Systems **5**(3), 17:1–17:33 (2014). https://doi.org/10.1145/2629376
20. Kritzinger, W., Karner, M., Traar, G., Henjes, J., Sihn, W.: Digital twin in manufacturing: A categorical literature review and classification. IFAC-PapersOnLine

51(11), 1016–1022 (2018). https://doi.org/10.1016/J.IFACOL.2018.08.474, Proceedings of the 16th IFAC Symposium on Information Control Problems in Manufacturing (INCOM'18)
21. Lara, J., Guerra, E., Sánchez Cuadrado, J.: When and how to use multilevel modelling. ACM Transactions on Software Engineering and Methodology **24**, 1–46 (2014). https://doi.org/10.1145/2685615
22. Lee, E.A.: The Past, Present and Future of Cyber-Physical Systems: A Focus on Models. Sensors **15**(3), 4837–4869 (2015). https://doi.org/10.3390/s150304837
23. Malayeri, D., Aldrich, J.: CZ: Multiple Inheritance Without Diamonds. In: Arora, S., Leavens, G.T. (eds.) Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09). pp. 21–40. ACM (2009). https://doi.org/10.1145/1640089.1640092
24. National Academy of Engineering and National Academies of Sciences, Engineering, and Medicine: Foundational Research Gaps and Future Directions for Digital Twins. National Academies Press (2024). https://doi.org/10.17226/26894
25. Object Management Group: OMG Unified Modeling Language (OMG UML) (December 2017), https://www.omg.org/spec/UML/2.5.1/PDF
26. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (fUML) (February 2021), https://www.omg.org/spec/FUML/1.5/PDF
27. Object Management Group: OMG Systems Modeling Language (SysML). Part 1: Language Specification (September 2025), https://www.omg.org/spec/SysML/2.0/Language/PDF
28. Object Management Group: OMG Systems Modeling Language (SysML). Part 2: SysML v1 to SysML v2 Transformation (September 2025), https://www.omg.org/spec/SysML/2.0/Transformation/PDF
29. Oliver, D., Kelliher, T., Keegan, J.: Engineering Complex Systems with Models and Objects. McGraw-Hill (1997)
30. Päßler, J., ter Beek, M.H., Damiani, F., Dubslaff, C., Johnsen, E.B., Tapia Tarifa, S.L.: Feature-Oriented Modelling and Analysis of a Self-Adaptive Robotic System. Formal Aspects of Computing **37**(4), 32:1–32:39 (2025). https://doi.org/10.1145/3709159
31. Päßler, J., ter Beek, M.H., Damiani, F., Johnsen, E.B., Tapia Tarifa, S.L.: A Configurable Software Model of a Self-Adaptive Robotic System. Science of Computer Programming **240**, 103221 (2025). https://doi.org/10.1016/J.SCICO.2024.103221
32. Päßler, J., ter Beek, M.H., Damiani, F., Johnsen, E.B., Tapia Tarifa, S.L.: Analysing Self-Adaptive Systems as Software Product Lines. Journal of Systems and Software **222**, 112324 (2025). https://doi.org/10.1016/J.JSS.2024.112324
33. Rumpe, B.: Modeling with UML: Language, Concepts, Methods. Springer (2016). https://doi.org/10.1007/978-3-319-33933-7
34. Sakkinen, M.: Disciplined Inheritance. In: Cook, S. (ed.) Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP'89). pp. 39–56. Cambridge University Press (1989)
35. Selić, B., Pierantonio, A.: Fixing Classification: A Viewpoint-Based Approach. In: Margaria, T., Steffen, B. (eds.) Proceedings of the 10th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'21). LNCS, vol. 13036, pp. 346–356. Springer (2021). https://doi.org/10.1007/978-3-030-89159-6_22
36. Singh, G.B.: Single Versus Multiple Inheritance in Object Oriented Programming. OOPS Messenger **5**(1), 34–43 (1994). https://doi.org/10.1145/182078.182085

37. Weyns, D.: An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective. Wiley (2020)
38. Whittle, J., Hutchinson, J., Rouncefield, M.: The State of Practice in Model-Driven Engineering. IEEE Software **31**(3), 79–85 (May/June 2014). https://doi.org/10.1109/MS.2013.65