DOI: https://doi.org/10.15276/aait.08.2025.11
UDC: 004.421.2:004.415.5

Method for incremental control of consistency between structural and behavioral views of software architecture

Nataliia O. Komleva[1)]
ORCID: https://orcid.org/0000-0001-9627-8530; komleva@op.edu.ua. Scopus Author ID: 57191858904
Maksym I. Nikitchenko[1)]
ORCID: https://orcid.org/0009-0007-9560-7057; maksym.nikitchenko@gmail.com
[1)] Odesa National Polytechnic University, 1, Shevchenko Ave. Odesa, 65044, Ukraine

ABSTRACT

The development of software engineering poses a challenge for researchers to maintain the integrity of models stored simultaneously in a lightweight text format and in a formally rich metadata view. The existence of two views ensures broad compatibility with developer tools and accurate reproduction of semantics, but creates the risk of discrepancies between structural and behavioral descriptions. The relevance of the research is determined by the need for methods that prevent the accumulation of contradictions without significantly affecting the speed of design iterations. The aim of this work is to provide a theoretical justification for an incremental approach that can guarantee the consistency of a metamodel with two views during any successive changes. To achieve this goal, a generalized metamodel has been formed that distinguishes between a structural view for static entities and a behavioral view for dynamic aspects. A correspondence relationship has been introduced between views, which describes pairs of equivalent elements and sets rules for their mutual consistency. The set of rules is formalized in the language of object invariants. Incrementalism is ensured by localizing changes: after editing, only those fragments that are directly involved in the modification are checked, so that the time spent remains proportional to the volume of the updated part. The result of applying the method is to prove the correctness of the proposed restrictions, which excludes the possibility of inconsistent model states. An analytical assessment of the complexity of the procedure confirms a linear dependence on the number of changed elements, which indicates the suitability of the approach for industrial-scale models. A demonstration control example, built on a representative domain, showed that the method detects inconsistency immediately after a single edit and proposes a sequence of actions sufficient to eliminate it without involving outside expertise. As a result, the work proposes a new formal methodology for maintaining consistency between views of a single model, which comprehensively combines localized verification with a declarative description of dependencies. The practical significance is manifested in the reduction of error correction costs, increased reliability of documentation, and the ability to integrate the method into modern modeling and continuous development environments, making it a promising tool for the development and maintenance of large corporate systems.

Keywords: Model consistency; incremental verification; model synchronization; metamodel; ontological constraints, reliability.

***For citation*: Komleva N. O., Nikitchenko M. I. "Method for incremental control of consistency between structural and behavioral views of software architecture". *Applied Aspects of Information Technology*. 2025; Vol. 8 No. 2: 162–177. DOI: https://doi.org/10.15276/aait.08.2025.11**

INTRODUCTION

Modern software projects require effective approaches to system modeling due to their increased complexity and distribution. The Unified Modeling Language (UML) is one of the most common tools for formalizing requirements and designing software architecture [1]. As the scale of models grows, the question of optimal storage of UML descriptions and their integration with various development tools arises. The standard model exchange format is XML Metadata Interchange (XMI), the official notation of the Object Management Group for serializing UML data. The XMI format provides full detail of elements in accordance with the UML 2.5.1 specification [2], but is known for its complexity and redundant syntax. Instead, JavaScript Object Notation (JSON) is a lightweight text data format that is increasingly being used to store models due to its compact syntax and ease of use in data exchange, including in version control systems. However, the direct application of JSON to UML models is complicated by the need to display complex structure and relationships, which is well supported by XMI. Neither of these formats provides a one-size-fits-all solution: XMI guarantees formal precision [3], and JSON guarantees integration flexibility, so the search for combined approaches is relevant in the field of modeling. In particular, a previous work proposed a UML model representation based on a combination of JSON and XMI formats [4]. This model provides for storing the basic structures of the model (classes, attributes, relationships) in the form of JSON, and complex behavioral diagrams in the form of nested XMI fragments.

To prevent out-of-sync between both views, a consistency checking (validation) system was provided. In general, the detailed scheme is shown in Fig. 1.

For the purposes of this study, the reliability of a UML model is interpreted as its ability to maintain internal integrity, logical consistency, and predictable behavior throughout its life cycle, in particular in cases of unforeseen changes to structural components (classes, attributes, relationships) and/or behavioral aspects (state diagrams, activities, sequences, etc.), gradual expansion of functionality, or migration of the model between tools. A reliable model localizes and timely signals any inconsistencies between views, preventing them from escalating into cascading errors, which ensures that defects can be quickly eliminated without compromising overall consistency. Thus, the model acts as the only reliable source of architectural information and minimizes labor costs for finding and synchronizing hidden discrepancies.

The issue of maintaining model consistency between the two views is relevant because identifying and eliminating inconsistencies between different model representations is vital to preventing errors and software defects that may be caused by inconsistencies between UML diagrams or their parts [5]. Existing studies propose numerous consistency rules for specific types of UML diagrams, but the problem of ensuring consistency and interconnection between these diagrams within a single model remains open [5]. This is especially true for incremental (step-by-step) changes when the model evolves and it is necessary to guarantee its integrity without full verification each time [6]. In traditional tools, consistency control is mostly limited to a single environment or type of artifact [7]. For example, UML model checking tools usually assess consistency only between diagrams within a model or between model and code, but do not cover different representation formats of the same model. Therefore, it is hypothesized that the use of formal validation methods will allow maintaining the consistency of UML model with two views by combining the advantages of XMI and JSON. **The research hypothesis** is that the use of formal verification (in particular, Object Constraint Language (OCL) constraints and SAT solvers) to verify the two views of the model will ensure timely detection of inconsistencies and prevent out-of-sync without significant performance impact. The verification should be performed incrementally, with each model change, which will maintain the integrity of the system in real time. Accordingly, the relevance of the work is due to the practical need to accelerate development and reduce the risks associated with incomplete or inconsistent UML models in the project documentation in the context of rapid development iterations.

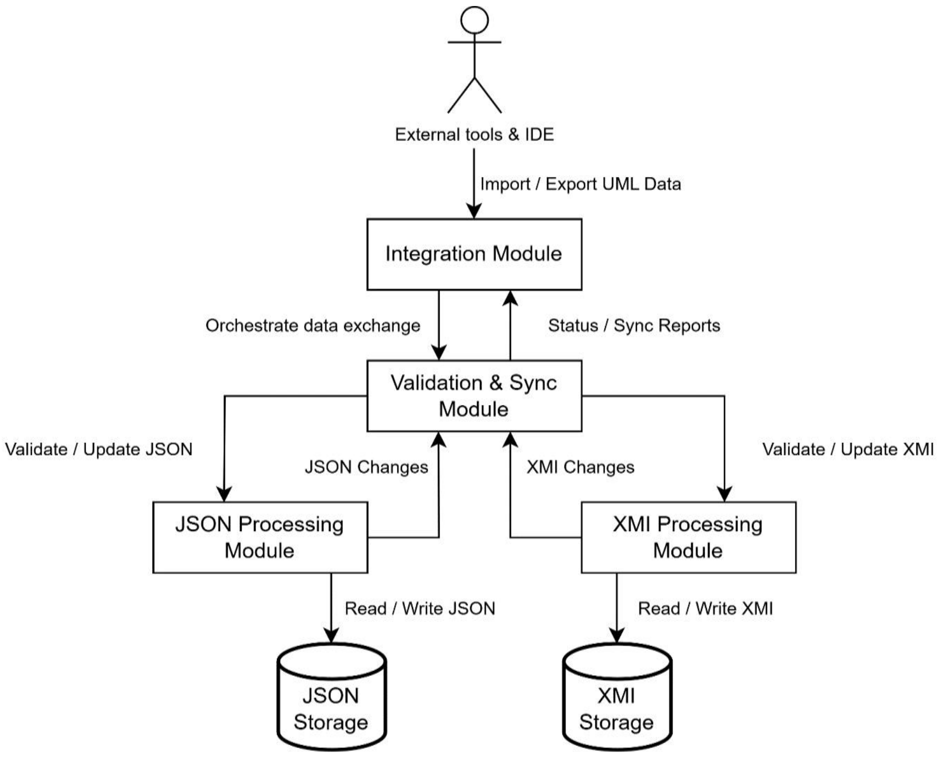

Fig. 1. **Architectural model of the system**
Source: compiled by the [4]

ANALYSIS OF THE LITERATURE AND PROBLEM STATEMENT

The problem of software model consistency is widely covered in the literature. There are different types of consistency of UML models: internal (between elements of the same diagram), inter-diagram (between different types of diagrams, such as classes and states), model-code, etc. In the context of Model-Driven Engineering (MDE) [8], many techniques have been developed to ensure the consistency of artifacts. Systematic reviews confirm that violation of consistency between diagrams is a common phenomenon that negatively affects the quality of models, in particular, [9] draws attention to the lack of researchers' attention to inter-diagram and semantic consistency: existing methods mainly cover syntactic aspects and individual notations, without ensuring full consistency of complex models. For example, a study has shown that when moving to short Continuous Integration/Continuous Delivery (CI/CD) iterations, most existing modeling tools do not provide adequate consistency control, and therefore automated methods remain critical [10]. In [11], the literature was systematically analyzed and more than 100 rules for the consistency of UML diagrams were identified, of which 52 were later recognized as mandatory for all UML models [12]. These rules include checking the correctness of the relationships between classes and objects in different diagrams, the correspondence between state and sequence diagrams, etc. Subsequently, the same authors applied the selected rules to analyze open UML models in practice [13], confirming that rule violations often occur even in professionally developed models. Other researchers also emphasize that inconsistencies in UML models are common and require automatic control [5]. A review of approaches to checking the consistency of behavioral models was conducted and it was concluded that despite the large number of proposed rules, methods for their practical implementation are not sufficiently developed [5], [14]. An example of an integrated approach to automated detection of deficiencies in UML class models is described in [15].

One of the key areas of research is consistency between different types of models. A systematic review of this modeling has shown the lack of common terminology and standard solutions to maintain consistency between different types of artifacts [16]. We also consider the Vitruvius approach, which introduces a single base model from which various system representations are projected. Such a projection architecture allows to automatically synchronize changes in all related model views by formally defining the concept of consistency between them [17]. This increases consistency in view-based development, as all partial views of the system remain tied to a common source of truth. Thus, maintaining model consistency remains an urgent scientific task.

Another aspect of consistency is managing consistency across large models and integrating tools. XMI, as a standard format for saving UML models, guarantees compatibility with various Computer-Aided Software Engineering (CASE) tools, but is not suitable for manual analysis or quick review of changes. In addition, as the size of models grows, XMI files become very large and slow to process. In response to these challenges, researchers are experimenting with alternative ways to store models: using JSON, relational and non-relational (NoSQL) databases, graph repositories, etc. For example, a multi-database model storage system NeoEMF has been developed that supports several NoSQL storages (graph, column, key-value) instead of the standard XMI [18]. Experimental results showed a significant increase in the performance of queries to large models: in some scenarios, accessing model elements through the NeoEMF graph database was faster, while a similar operation with XMI space required downloading the entire file and did not fit into memory. Thus, the use of flexible storage formats (such as JSON or graph databases) in combination with a formal metamodel allows for increased scalability of MDE solutions. At the same time, this creates a new challenge - to maintain consistency between different formats of representation of the same model. Existing works only partially address this issue: for example, OGC Best Practice 2018 describes the rules for encoding UML metamodels into JSON schemas, but they do not regulate the consistency of changes between JSON and XMI representations of the model. There are also several tool solutions (EMF JSON, MongoEMF, etc.) that allow you to synchronously save a model in two formats, but the consistency control is up to the developers.

An important aspect is model merging and conflict management when combining different fragments. In team projects, it is often necessary to merge changes made by several developers in parallel or to integrate different partial models (versions) into a single system model. This task is accompanied by consistency conflicts when the merged changes are incompatible. A recent mapping study, which covered 105 scientific papers, systematized techniques for detecting and resolving conflicts in model merging [19]. According to its results, the most common approaches to detecting conflicts are based on violation of holistic constraints, overlapping changes, or change patterns (patterns). At the same time, there is a lack of integrated tools to fully support this process in practice. Thus, manual intervention by experts is often required to achieve post-merger consistency, which slows down development. There are proposals in the literature to increase the level of automation of this process through artificial intelligence. For example, reinforcement learning algorithms have been experimentally applied to automatically select conflict resolution actions in UML class diagrams after version merging [20]. In their approach, the artificial intelligence agent learns from examples of successful resolution of typical conflicts and suggests the optimal sequence of corrections, personalized to specific model quality indicators. Despite the promise of such solutions, they are still at the research stage and are not integrated into standard MDE tools.

Existing model checking tools and frameworks provide validation mainly within a homogeneous representation. Some works have proposed to transform state diagrams into formal models (e.g., Petri nets or first-order logic) to verify the compatibility of behavioral diagrams. To verify the behavioral aspects, it was proposed to translate UML State Machine into colored Petri nets using the Isabelle/HOL validator, which allowed to formally prove the properties of states [21]. An alternative way to formalize statechart diagrams through temporal Petri nets is described in [22], where the focus is on reproducing the time constraints of transitions. Additionally, a method for building a model of a computing process based on a Petri net is proposed in [23], which confirms the effectiveness of this formalism for the early detection of structural errors and loops. In addition, 84 coevolution patterns were proposed, which use colored Petri nets to incrementally track changes in classes, objects, activities, states, and sequences, localizing the impact and quickly identifying potential discrepancies between diagrams [24]. We should also mention the areas devoted to incremental model consistency checking. Traditional validation tools perform a global model check after making changes, which can be slow and inconvenient. In addition, it is known that repeated full validation of a large UML model after each change is a computationally expensive task [6]. Instead, modern approaches seek to provide live validation, i.e., immediate detection of inconsistencies in the process of model editing. For example, [25] proposed an incremental algorithm for checking OCL constraints in a UML model by automatically converting them into SQL queries. This approach localizes the verification only to the elements that have changed, which significantly reduces the time for detecting errors compared to a complete re-check of the entire model.

Formal specification and verification methods are also widely used to ensure model consistency. The basic mechanism is the imposition of formal constraints (invariants), for example, in the OCL language, followed by verification of their fulfillment. A number of tools (e.g., UML USE, etc.) allow you to identify logical contradictions between model elements based on OCL rules. However, formal verification can go beyond OCL: the literature describes approaches to translate UML/OCL models into formal problem statements that can be solved automatically. Thus, in [26], it was proposed to display UML state diagrams together with OCL constraints in the Web Ontology Language (OWL) to check their consistency by means of logical inference in the descriptive logic environment. Automatic OWL analysis allows you to detect contradictions between requirements and model elements, generating a logical explanation for each conflict. Another direction is the use of SAT/SMT solvers and algebraic methods: the study [27] presented a formal consistency model based on graphs and structure-data descriptions that provides both consistency checking and consistency preservation during model evolution. Their proposed system of formal rules allows both to detect all integrity violations in multi-species UML models and to automatically maintain consistency when updating (by propagating changes or calling repair operations). A special mention should be made of [28], where the combination of triple graph grammars with linear integer programming guarantees finding a consistent transformation between two models and demonstrates scalability for industrial scenarios. The application of strict formal constraints increases the reliability of validation: the researchers emphasize that without such a framework, most existing methods do not guarantee the full correctness of models.

The analysis of modern CASE tools shows that although they declare full support for UML 2.x, none of them eliminates the key discrepancy between the static description of "what is" and the behavioral description of "what the system does": structural and behavioral artifacts are stored in different formats or proprietary repositories and are only partially or not formally checked for compliance [29]. This is one of the main arguments for the need for a metamodel with two views.

Thus, the analysis of the literature shows that there is a significant body of work in the field of ensuring the consistency of UML models and their validation by formal methods. Previous work has focused either on the consistency of different diagrams within an XMI model, or on the consistency of the model with code, or on general rules for UML notation. Nevertheless, the analysis of modern CASE tools shows that the lack of guaranteed, automated, and efficient mechanisms for maintaining consistency between the structural and behavioral views of a UML model in a collaborative development environment leads to architectural drift, an increase in the number of defects, and significant overhead in the project. The criticality of this problem is that inconsistency errors are hidden failures. These are not syntax errors that can be detected by a compiler or schema validator, but semantic gaps that manifest themselves at later stages of the life cycle: during code generation, integration testing, or, worst of all, already in operation. For example, calling a non-existent method in a sequence diagram or using outdated data types may go unnoticed at the modeling stage, but will lead to a system failure. The cost of fixing such defects at later stages is higher than at the design stage, making the problem of ensuring consistency not just a quality issue, but a key factor in the cost-effectiveness of development. Recent formal works, such as [30], offer a theoretical framework for managing consistency between multiple models, but do not consider combined data formats.

In general, modern UML modeling tools have to combine opposing requirements: on the one hand, quick introduction of small changes to the static structure of the model, on the other hand, formal accuracy and completeness of behavior description. The serialization formats that prevail in practice provide these advantages separately. JSON demonstrates compact syntax, fast processing, and easy integration with version control systems, so it is naturally suited for storing flat hierarchies such as class or component diagrams, where relationships such as owns, typedBy, or associates are easily expressed through nested objects and simple references.

However, attempting to transfer the dynamic aspects of UML to the same format encounters a number of fundamental obstacles. First, JSON does not have a built-in system of global unique identifiers and links between elements; therefore, the order of messages in interaction diagrams or the chain of transitions in state machines has to be reproduced with artificial markers, which quickly reduces the transparency of the model. Secondly, behavioral constructs like CombinedFragment, ExecutionSpecification, or SignalEvent are recursive in nature and require many levels of nesting; in JSON, this turns into a deep tree with dozens of duplicate identifiers - unlike XMI, where such references are automatically handled by the xmi:id/xmi:idref mechanism.

A comparative assessment of the description volume confirms the gap in complexity: even with a complete list of relationship attributes, the static part in JSON takes up an order of magnitude less space and requires only schema validation, while for behavioral diagrams the number of contextual dependencies and nesting depth increases dramatically and requires an external logic engine to check for correctness. Therefore, in a pure JSON format, the model either loses its semantics or grows into an overloaded, hard-to-manage document.

At the same time, switching to a solid XMI, on the contrary, would complicate every small structural change. For example, for a typical "rename attribute" operation, the JSON fragment takes one or two lines, while the XMI fragment is much larger and includes service xmlns links, which creates unnecessary conflicts during a three-way merge. Git's text tools automatically and correctly merge such compact changes, so most everyday structural changes don't require the participation of an XMI expert.

Practice shows that a deliberate division into two views is optimal. The structure (classes, attributes, associations) is stored in a compact JSON segment, which is faster and requires less memory, is easily merged into Git, and allows developers to make common edits without diving into the formal XMI syntax. Instead, the behavior (strategic scenarios, states, message sequences) is captured in XMI fragments: this is where full compliance with the UML 2.5 specification, the ability to impose OCL constraints, and compatibility with an existing UML toolchain are required. The combined approach combines the strengths of both formats and focuses on the boundary where most critical inconsistencies between structure and behavior occur.

Accordingly, this paper aims to develop a formal framework for a UML model with two views and tools for its practical implementation that will ensure constant consistency between JSON and XMI parts. This continues and deepens the author's previous research [4], focusing not only on the architecture itself, but also on its validation algorithms.

PURPOSE AND OBJECTIVES OF THE STUDY

The aim of the paper is to formally guarantee the integral consistency of two views of a UML model using JSON and XMI by introducing a verifiable set of consistency rules and an incremental procedure for their verification. To achieve this goal, it is necessary to solve the following tasks: to formally define a metamodel with two views, consisting of a set of JSON elements and a set of XMI elements linked by a correspondence relation; to formulate a set of consistency constraints between these sets (in OCL or first-order logic) and transform them into formats suitable for automated verification (Alloy); to develop a method for incremental verification of model consistency with favorable computational complexity and to test it. These tasks will allow us to confirm or refute the hypothesis about the effectiveness of formal validation of UML models with two views.

STRUCTURAL ORGANIZATION OF THE PROPOSED METAMODEL WITH TWO VIEWS

According to the multi-level architecture of the Meta Object Facility (MOF), the development of a metamodel (level M2) is necessary because it defines the formal abstract syntax and semantics for a set of specific UML models (level M1), ensuring their homogeneity and the possibility of automated processing [31]. The metamodel defines the permissible types of elements, their attributes, relations, and OCL invariants, i.e., it acts as a "system of rules" by which models of the subject area are built and verified. Only a well-defined metamodel makes it possible to integrate different view formats (JSON/XMI) within a single formal space and guarantee the correctness of transformations between them. In addition, the metamodel serves as a contract for MDE tools: it connects to Eclipse Modeling Framework (EMF)-like frameworks and provides code generation, serialization, incremental validation, and further extension without breaking compatibility. That is why, in the context of the proposed approach, it is quite justified to talk about developing a metamodel rather than a separate UML model, since only a metamodel provides the necessary level of abstraction, formality, and reusability inherent in the MOF specification.

It is also known that there are architectural models with more views. The presence of only two views – structural and behavioral – is the result of a deliberate engineering compromise based on the fundamental distinction between "what the system is" and "what it does". It is at the interface between these two aspects that the vast majority of critical inconsistencies (signature mismatches, call errors, etc.) occur, so concentrated efforts to ensure their consistency have the greatest practical effect. In addition, each of these views requires different approaches to storage and validation: JSON provides flexible, Version Control System (VCS)-oriented management of frequent, small structure changes, while XMI provides formal rigor and OCL compatibility for complex behavioral logic. Additionally, it can be noted that the proposed two-layer strategy does not contradict the classical 4 +1 (Kruchten) model [32], but only specifies it in terms of persistence: the logical view is stored in JSON, and the process view in XMI; the remaining views (development, physical, scenarios) either indirectly rely on this data or can be integrated without breaking the chosen segregation. Therefore, limiting to two views is a deliberate choice that minimizes complexity and focuses on the problem of consistency between the static and dynamic aspects of the system.

A UML metamodel with two views is a formal representation of the UML 2.5 metamodel, which consists of a triple according to formula (1):

$$M = (M_S, M_B, \mu), \tag{1}$$

where M_S is the structural view of the metamodel (the JSON view), M_B is the behavioral view of the metamodel (the XMI view) and μ is the correspondence relation between the elements of these two views. Intuitively, M_S defines all valid entities and relationships of the static structure of the UML model (classes, attributes, associations, etc.), M_B defines all entities for describing behavior (activities, states, message sequences, etc.), and μ captures how objects of the structural view correspond to objects of the behavioral view. The metamodel was built according to the UML 2.5 specification, taking into account standard abstractions (e.g., class Class, association Association, state State, transition Transition, message Message, etc.) and using the division of the model into two views.

Entities and relations of the structural view M_S. The structural view of the metamodel describes the elements of UML structure diagrams. The main entities are classes (Class metaclass), attributes

(Property), operations (Operation), and associations (Association). Typical UML relationships are defined between these entities: a class has attributes and operations (aggregation/composition between Class and Property/Operation), an attribute has a type (a reference to a Class or a primitive), an operation can have parameters (each of which is an attribute specially marked as a parameter), an association connects two classes, etc. An entity is considered relevant for behavioral synchronization if the hasBehavior attribute= is set to true.

Formally, the structural view can be described as follows:

$$M_S = < T_S, R_s >, \quad (2)$$

where: T_S = {Class, Interface, Primitive, Enumeration, Attribute, Operation, Parameter, Association, AssociationEnd, Generalization, Package}; R_S = {owns, typedBy, hasParameter, associates, generalizes}.

Thus, T_S is the set of types that can make up a structural view. And R_S is the set of relations or functions between elements from E_S that specify their interconnections. For example, typedBy: Attribute → (Class ∪ Primitive).

Entities and relations of the behavioral view *M_B*. The behavioral view of the metamodel describes the elements of UML dynamic diagrams that are stored in the format of XMI fragments [33]. The main meta- entities are activity diagrams (Activity metaclass), state diagrams (StateMachine and related State, Transition metaclasses) – define possible object states and transitions between them when events occur, sequence diagrams (Interaction and Lifeline, Message, ExecutionSpecification metaclasses, etc.) – model the exchange of messages between objects (class instances) in time, events and calls (SignalEvent, CallEvent, Action, etc.). These M_B elements have their own relationships: for example, a sequence diagram contains Lifelines, each Lifeline can refer to a specific class (through the represents or classifier property), a Message is associated with an operation call (through a reference to the corresponding Operation), a Transition in a state diagram can have a trigger that corresponds to an operation call event or a signal receipt, an Action in an activity can call a class operation or change an attribute value, etc.

In addition, the model stipulates that each class, attribute, or operation has a unique global ID and name (unique within its entity). This ensures that all references to the same element are associated with the same record at the structural view, which guarantees consistency: for example, if the Book class is renamed, the change will automatically propagate to all relevant XMI fragments via a single ID.

Formally, the behavioral view can be described as follows:

$$M_B = < T_B, R_B >, \quad (3)$$

where T_B = {Activity, Action, ControlFlow, StateMachine, State, Transition, Event, Interaction, Lifeline, Message, ExecutionSpecification, CombinedFragment}, R_B = {contains, triggers, represents, calls}.

Similarly to the structural view, T_B is the set of types that a behavioral view can consist of. R_B is the set of relations or functions between elements in E_B that define their interconnections.

If necessary, both views can be extended with new types and new relationships.

The **μ- correspondence** relation (μ) reflects the agreement between the elements of the structural and behavioral views of the model. In general,

$$\mu \subseteq E_S \times E_B, \quad (4)$$

where: E_S is set of all structural view elements (instances of M_S), a E_B is the set of all elements of the behavioral view (instances M_B). The tuple $(s,b)\in\mu$ means that the behavioral element b corresponds to (or is bound to) the structural element s.

For example: $(C,A)\in\mu$ can mean that activity diagram A belongs to class C (the diagram models the behavior associated with this class); $(s, l)\in\mu$ indicates that lifeline l is an instance of class s (i.e., classRef(l)=s); $(o, m)\in\mu$ means that message m models the call to operation o (formally, m.name=o.name and opRef(m)=o). Thus, μ covers all the necessary types of relationships between structural and behavioral elements.

If we think of μ as a relation, we can say that the following conditions are satisfied.

1. Identification linkage. In each pair $(s, b) \in \mu$, one element directly refers to the id of the other, which provides unambiguous tracing between views.

2. Totality with respect to structure. Every structural element that is marked as hasBehavior=true and for which behavioral granularity is possible has at least one correspondence $b\in E_B$.

3. Partiality in relation to behavior. Any behavioral fragment must have a structural anchor. Formally: $\forall b\in E_B, \exists s\in E_S : (s,b)\in\mu$.

4. Non-functionality $E_S! \to !E_B$ and functionality vice versa. One structural element can have several behavioral correspondences, while each behavioral

element has a single structural anchor. Formally: $\forall b \in E_B, \exists! s \in E_S : (s,b) \in \mu$.

To formally guarantee the consistency of the model, we introduce a system of **ontological constraints** on the relation μ. Here, ontology is a set of object types, their properties, and relationships (in the form of logical statements or OCL invariants) that determines when a model is considered consistent. In other words, it is a set of *consistency rules*. Formally, these rules can be specified as logical formulas (e.g., first-order) or as OCL constraints that must be satisfied on any instance of the model M.

Here are the key limitations of consistency.

Completeness of correspondences: For every relevant structural element, there must be a corresponding behavioral element. By "relevant" we mean one that is explicitly marked with the hasBehavior attribute= true (in this case, it is considered dynamic). For other operations (e.g., getters/setters, DTO-class service methods, factory constructors, etc.), behavioral support is not mandatory, so a violation of the rule is not recorded by the OCL invariant as follows:

```
context Class
inv BehaviorExists:
  self.operations->exists(op |
op.hasBehavior) implies
    (Interaction.allInstances()->exists(i |
        i.messages->exists(m | m.name =
op.name)) or
     StateMachine.allInstances()->exists(sm
|
        sm.transitions->exists(t |
t.trigger.operation = op)) or
     Activity.allInstances()->exists(act |
        act.actions->exists(a | a.operation
= op)))
```

This formulation creates a one-way relationship: the behavioral view is based on the structural view, but not vice versa. A class without designated dynamic operations can remain purely structural, which is consistent with the principle “structure is primary, behavior is derived”.

This selective rather than total requirement has two advantages. Firstly, it avoids false warnings for “passive” classes, such as Enumeration, where behavior is not required. Secondly, it allows you to gradually develop the model: first, set the structure, and then, as needed, specify the behavior without breaking consistency. If a stricter rule is needed in future projects (all operations must be covered by test scenarios or animations), it can be formulated as an extension of the current invariant, just change the hasBehavior predicate to be inclusive.

Unity of compliance: Any behavioral element has a single anchor at the structural view.

The OCL invariant is as follows:

```
context BehaviorElement
inv SingleAnchor:
  not self.anchor.oclIsUndefined()
```

Compatibility of types and parameters: Object types and call signatures must match between views.

The OCL invariant is as follows:

```
context Message
inv SignatureMatches:
  not self.opRef.oclIsUndefined() implies
    self.arguments->size() =
self.opRef.parameters->size() and
    self.arguments->forAll(arg |
      let idx : Integer = self.arguments-
>indexOf(arg) in

arg.type.conformsTo(self.opRef.parameters-
>at(idx).type))
```

No dangling references: No behavioral fragment should refer to a structural element that does not exist, and vice versa – a structural element cannot have references to non-existent behavioral fragments.

The OCL invariant is as follows:

```
context BehaviorElement
inv AnchorExists:
  StructuralElement.allInstances()
->includes(self.anchor)

context Class
inv ActivityReferenceValid:
  not self.activityRef.oclIsUndefined()
implies
     Activity.allInstances()
->exists(act | act.id = self.activityRef)
```

To formally guarantee consistency between the structural and behavioral views of the model, the following key invariants are defined:

BehaviorExists (invariant of completeness of correspondences).

Guarantees that each dynamic structural-view operation (hasBehavior attribute = true) has at least one behavioral implementation.

Formally:

$$\forall s \in E_S, s.\text{hasBehavior} = \text{true} \Rightarrow \\ \exists b \in E_B : (s,b) \in \mu. \quad (5)$$

AnchorExists (invariant of no dangling references).

Requires each behavioral element to have at least one structural anchor.

Formally:

$$\forall b \in E_B, \exists s \in E_S: (s, b) \in \mu. \quad (6)$$

SingleAnchor (invariant of unity of compliance).

Prohibits multiple bindings of one behavioral element to several structural elements, requiring only one anchor.

Formally:

$$\forall b \in E_B, \exists! \, s \in E_S: (s, b) \in \mu. \quad (7)$$

SignatureMatches (signature matching invariant).

Ensures that signatures match the definition of operations at the structural view and their calls in behavioral diagrams.

Formally:

$$\forall m \in \text{Message}, m.\text{opRef} \neq \text{undefined} \Rightarrow \text{SignatureMatches}(m). \quad (8)$$

These invariants are an integral part of the formal model and are used to check consistency when making changes to the model.

The metamodel and the correspondence relation μ are implemented in such a way that all the above constraints can be set declaratively. For this purpose, OCL invariants were used to describe the consistency rules and their subsequent formal verification by means of Alloy. It is known that Alloy is a language and tool for analyzing structures based on the Boolean formulae feasibility problem (SAT) [34], [35]. Each constraint is represented as an Alloy-assertion, and the sets E_S and E_B are the signatures of an Alloy model. The relation μ is described through a binary relational attribute between two signatures. For example, the rule "Unity of compliance". In Alloy, it is expressed by a quantifier formula on the set of model facts. If a constraint is violated, Alloy finds a counterexample – a specific set of objects s, b that does not meet the requirement. This allows you to localize the inconsistency. It should be noted that using the SAT method requires limiting the search area (model scopes), but in our case, the scopes correspond to the actual number of elements in the model, which is known and relatively small, which corresponds to the practical performance limit of Alloy 4.2; for models > 5000 elements, it is advisable to use a sharding strategy or SAT accelerator. Thus, the formal consistency rules have been transformed into a form suitable for automatic verification by SAT/Alloy. It is also worth noting that the current translation covers a subset of OCL (without collect/select on nested collections); operators outside the subset are marked as not-supported and require manual reformulation. For a detailed overview of the syntax and expressive capabilities of OCL, on which these rules are based, see [36].

For example, “Unity of compliance” can be represented as follows:

```
-- declaration of basic signatures
sig StructuralElement {}

sig BehaviorElement {
  -- each behavioral element has exactly
ONE anchor structure
  anchor: one StructuralElement
}
```

This approach combines the clarity of the specification at the UML metamodel level (via OCL) with the rigor of the logical analysis offered by Alloy (the SAT solver checks whether constraints are met on all possible combinations of elements within the specified limits).

Preserving the order and UML profile extensions is planned in separate map-tables; in this article, we limit ourselves to the basic elements of the UML 2.5 specification.

A METHOD FOR CHECKING THE CONSISTENCY OF MODELS WITH TWO VIEWS

The proposed method is based on a formal UML metamodel with two views [1]. The method is focused on incremental localized consistency checking, which ensures guaranteed consistency of the model during its gradual modification.

Let's define the stages of the method.

1. Change detection and model view identification

A change is identified as being related to one of the metamodel views, M_S or M_B. This is determined based on the structure of the identifiers or the editing context. According to the structure of the sets E_S (instances of M_S) and E_B (instances of M_B), determines which of the subsets has changed and localizes the corresponding subgraph in the correspondence graph $\mu \subset E_S \times E_B$.

2. Synchronization when changing the structural view (JSON → XMI)

If a structural element is changed, the method performs a direct check and synchronization of the impact on the behavioral view.

Consider the subcases.

2.1. Adding a structural element

When adding the element $s \in E_S$, the method checks whether the rule of completeness of correspondence is fulfilled:

$$\forall s \in E_S, s.\mathrm{hasBehavior} = \mathrm{true} \implies \exists b \in E_B : (s, b) \in \mu. \tag{9}$$

If there is no match, a new behavioral element b is created that satisfies the requirement of existence in the set E_B. This guarantees the fulfillment of the OCL invariant *BehaviorExists*.

2.2. Deleting a structural element

For each b that has a relationship $(s,b) \in \mu$, a relevance check is performed. If the dependency is critical, the element b is removed, otherwise it is marked as invalid. This supports the fulfillment of *the AnchorExists* invariant.

2.3. Modification of a structural element

The method evaluates which properties of the element s have been changed (e.g., type, name, attributes) and performs a partial check of the corresponding consistency constraints for the associated b, such that $(s,b) \in \mu$.

An example of type compatibility checking for operation calls (8). Here, *SignatureMatches*(m) means that:

– the number of arguments in the message m is equal to the number of parameters of the corresponding operation;

– the type of each argument corresponds to or is compatible with the type of the corresponding parameter of this operation.

This ensures that the *SignatureMatches* invariant is respected for parameter types and calls.

3. Synchronization when changing the behavioral view (XMI → JSON)

3.1. Adding a behavioral element

For each new $b \in E_B$, the method sets the anchor $s \in E_S$ in accordance with the requirements of the *SingleAnchor* invariant, according to (7).

If there is no anchor, a proposal is generated to create or bind the corresponding element at the structural view.

3.2. Deleting a behavioral element

It checks whether there are any structural elements that depend solely on the removed behavioral element. The presence of such elements may indicate the formation of invalid structural fragments.

3.3. Modifying a behavioral element

A localized check of the compatibility of parameters, types, and correspondences between views is performed. In case of inconsistencies, recommendations or suggestions for automatic correction are generated.

4. Localized verification of OCL invariants

After synchronization operations, including possible automatic corrections, the method checks the consistency of the model by applying OCL invariants. The check is not performed for the entire model, but only for the set of Affected Elements – elements that are directly or indirectly related to the changed element through the correspondence relation μ and other structural and behavioral dependencies. This approach significantly reduces the scope of verification and computational costs by limiting the analysis to only relevant parts of the model.

Each OCL invariant is local in nature and is checked based on data about adjacent model elements (for example, class invariants are checked by their attributes and relationships, message invariants are checked by related operations and parameters). If all the invariants are met for the elements of the Affected Elements set, the change is considered accepted, and the model returns to the consistent state. If at least one of the invariants is violated, the algorithm records the conflict and generates a corresponding message for the user.

5. Conflict handling and notifications

If any residual conflicts are detected, the system immediately alerts the user in real time. The conflict message contains a description of the problem (for example, “The message calls an operation that does not exist” or “The argument type does not match the type of the operation parameter”) and suggests possible steps to resolve it: “Create an operation with this signature in the class”, “Change the argument type to the correct one”, or “Delete the duplicate class”. The user can choose one of the proposed options, after which the system automatically makes the appropriate changes to the model. After that, the invariants are rechecked, and if the conflict is resolved, the message disappears.

If automatic conflict resolution is not possible (for example, when duplicated classes have significant differences and it is impossible to determine which one should be left), the system records the problem and passes it on to the model engineer for consideration.

The following notations are used in the following: N is the total number of elements of the UML model; k is the power of the set of changed elements Δ in the current revision, with $k \ll N$; d is the maximum number of trace edges μ incident to each element. The graph μ is stored in a two-way hash index, which provides access to adjacent vertices in $O(1)$.

The incremental algorithm performs for each $s_i \in \Delta$ a recheck of only local invariants, and also extends the result to the dependent behavioral element b_j, the number of which is limited to d. Thus, the total number of elementary operations does not exceed $k \cdot d$, and if d is constant (sparse trace graph), the asymptotic is $O(k)$.

Theorem 1. Let Δ be the set of changed elements of the UML model, $|\Delta|=k$, and d be the upper bound on the degree of each vertex in the trace graph μ. Then the running time of the incremental consistency procedure satisfies the estimate $T(k)=O(k \cdot d)$.

Proof. For each element s_i of the set Δ, no more than d local invariant checks and one status propagation operation to the corresponding b_j are performed. Since all auxiliary structures (indexes, caches) provide $O(1)$ access, the total complexity does not exceed $k \cdot d$; hence, $T(k) = O(k \cdot d)$.

It follows from the above that when $k=1$, the amortized complexity of a single event approaches $O(1)$, while in general it grows linearly with the number of changes. A full model check covering all N elements requires $O(N \cdot d)$ steps and serves as a baseline for comparison.

For a scenario where the entire UML model is stored only in XMI format, localized incremental verification covers one representation, and its time is denoted as $T_1(k)=O(k \cdot d)$. In the dual JSON and XMI view, verification consists of three sequential operations: (i) structural view validation, (ii) behavioral view validation, and (iii) verification of μ-relations between views. The first two operations have the same upper bound of $O(k \cdot d)$, while the third has $O(k)$, since the μ relation is supported by a bidirectional hash index with $O(1)$ access. Therefore, $T_2(k)=O(k \cdot d+k) = O(k \cdot d)$.

As a result, the presence of two formats does not change the order of complexity compared to the single-format XMI model; the difference is limited to a constant multiplicative overhead that does not affect the scalability of the method.

Conflict handling mechanisms. The method's policy is to automate the correction as much as possible, but retain control over the model by the user. If the detected conflict is unambiguously resolvable according to the business logic (for example, a duplicate record – you can delete one; a renamed class – you should rename it in all diagrams), the tool applies the fix itself and only notifies you of it. In subtler cases, the user is given a choice. For example, when two duplicate entities are found, the system can offer to merge them, keep one (and which one), or keep both, but then you need to manually distinguish them (rename or clarify). The system traces all changes, so the history of edits is saved (you can cancel the automatic correction if it turned out to be undesirable).

REFERENCE EXAMPLE

To verify the performance of the proposed model and method, a simplified demonstration example is considered. The scenario is a UML model of the conditional subject area "Smart Home – Device", presented at two views in accordance with the developed metamodel.

At the structural view, the model includes two classes. The first of them is *SmartHome*, which describes a smart home object with an *address* attribute of type *String*, a collection of *devices*, and *a toggleAll()* operation. The second class is *Device*, which represents an individual device; it has the *id* (string identifier) and *status* (boolean flag on/off) attributes, as well as the *toggleStatus()* operation, which is marked as dynamic (the *hasBehavior* attribute *is = true*).

At the behavioral view, the model captures a minimal set of dynamic scenarios, with all key elements tied to structural identifiers. For the *Device* class, a *DeviceLifecycle* state diagram was created that models switching between the *Idle* and *Active* states. For the *SmartHome* class, there is a *ToggleAll* sequence diagram in which the *SmartHome* object sends *toggleStatus()* messages *to* all devices. The correspondence relation μ matches the identifiers of the model elements: *SmartHome* has identifiers J_1 (structural view) and X_1 (behavioral view), *Device* has J_2 and X_2, and the *status* attribute has J_3 and X_3. In the initial state, the model is consistent: each structural element has a single behavioral counterpart, and all operation signatures are consistent across views.

To simulate the real situation of model editing, two typical out-of-synchronization typical of complex systems are deliberately introduced. After that, we demonstrate the application of the developed method of incremental consistency checking, which allows to automatically detect and eliminate such violations.

Below is a sequence of steps that illustrates the application of the methodology using a benchmark example.

1. **Change Δ_1.** The developer adds a new dynamic operation checkBattery() to the **Device** class: int, which receives **the JcheckBattery** identifier. However, at the behavioral view, the

corresponding diagram or action that implements this operation is not created. This leads to a violation of the *BehaviorExists* invariant, which requires that each dynamic operation has at least one mapping in the behavioral model. The proposed method automatically detects a change in a structural element, identifies the absence of a corresponding behavioral fragment through the correspondence relation μ, and then initiates the auto-synchronization process. A new action-call checkBattery() is created in the **DeviceLifecycle** state diagram, assigned the identifier **XcheckBattery**, and a pair (JcheckBattery, XcheckBattery) is added to the μ relationship. After that, the *BehaviorExists* invariant is executed, and the model returns to the consistent state.

2. **Application of the method after Δ_1**. The incremental mechanism detects the change of the **JcheckBattery** element at the structural view, analyzes the set of correspondences in the relation μ and detects the absence of the corresponding behavioral element. The method activates the autosynchronization procedure and creates the missing fragment in the behavioral view, which ensures the fulfillment of the formal constraint *BehaviorExists*. As a result, full correspondence between structural and behavioral elements for the added operation is achieved without involving additional expertise.

3. **Change Δ_2.** The project developer makes changes to the **ToggleAll** behavior diagram and adds the newState: Boolean parameter *to the toggleStatus()* message. At the structural view, the toggleStatus() operation description remains parameterless, which violates the *SignatureMatches* invariant, which requires full signature matching between operation definitions and corresponding messages in diagrams. As a result, there is a discrepancy between the parameter sets that needs to be eliminated.

4. **Using the method after Δ_2**. The method detects a change in the behavioral fragment **Xmsg** that describes the toggleStatus() message. The structural operation ***JtoggleStatus*** is found through the correspondence relation μ, after which the parameter lists are compared. If a discrepancy is found, the method activates the correction procedure: in the **Device** class, the newState: Boolean parameter is added *to the* ***toggleStatus()*** operation. After updating the signature, the *SignatureMatches* invariant is executed again, and the model returns to the consistent state.

Consistency control. After both scenarios are completed, the model is re-validated to ensure that all key invariants are met: *BehaviorExists*, *SignatureMatches*, and *SingleAnchor*. Localized verification reveals the absence of dangling links, and all structural and behavioral elements have correct correspondences in terms of μ. Analytical evaluation confirms that, compared to a full $O(N \cdot d)$ check for monolithic XMI, the incremental approach reduces complexity to $O(k \cdot d)$, where k is the number of changed elements, which allows for effective consistency maintenance even in industrial-scale models.

DISCUSSION OF THE RESULTS

The literature analysis and practical experience of using UML in large teams show that the most common consistency issues arise when different types of diagrams are edited: renamed or omitted methods, mismatched operation signatures, duplicate elements in several packages, etc. Without automatic control, such errors accumulate and become noticeable only at the later stages of development (code generation, integration testing), which significantly increases the cost of corrections.

The proposed model with two views, combined with formal consistency constraints, potentially allows detecting such discrepancies at early stages of modeling. It is assumed that incremental validation will run in the background and provide the developer with immediate feedback, minimizing the impact on the workflow. The formal definition of rules eliminates the risks of subjective interpretation of the validation, ensuring the unambiguity of the results. This increases the reliability of project documentation – consistency is maintained automatically and continuously, so the risk of defects due to out-of-sync diagrams is significantly reduced.

The scientific novelty of the work is that for the first time a metamodel and method for ensuring the consistency of UML descriptions in the two views using JSON and XMI formats are formally defined. Unlike previous attempts to implement similar models, where synchronization between formats is outlined only conceptually, this study proposes a clear algorithmic verification mechanism that acts as a "converter-controller" between two model views. The theoretical analysis and case study confirmed that such integration combines the advantages of both formats, allowing the development team to use convenient modeling tools without fear of losing data accuracy or consistency. The previous UML consistency rules (e.g., matching state diagrams with class diagrams, checking the integrity of requirements and design models, etc.) take on a new dimension in our metamodel – they can be applied in the context of multiple description formats.

The limitation of this method is the need to initially identify the corresponding elements between the existing JSON and XMI descriptions. Another direction of development is to expand the set of consistency checks: for example, controlling the consistency between the model and the code. It is also planned to integrate the prototype system into a development environment for further empirical evaluation on industrial cases. In general, the results of the study demonstrate the high efficiency of the formal method for maintaining model consistency. The proposed metamodel and method have significant potential to improve the quality and reliability of software projects without significantly complicating development processes.

Further research involves empirically testing these findings on real projects and quantifying the performance of incremental validation.

CONCLUSIONS

The article presents a formal metamodel and method for ensuring the consistency of a UML model that is simultaneously stored in JSON and XMI formats. A metamodel with two views, with a clear correspondence relation between elements of different views and a system of formal constraints (OCL → Alloy) describing the necessary conditions for their consistency is developed. Based on this model, an incremental method of ensuring consistency is proposed that automatically tracks changes and synchronizes both views of the UML model. The scientific novelty of the obtained results lies in the combination of flexible JSON and strict XMI through a single formalized mechanism, taking into account the properties of OCL constraints and the capabilities of SAT analysis, which allows maintaining the integrity of the model in real time. It is shown that the application of this approach allows to use the strengths of each of the presented formats, while, thanks to the method of ensuring consistency, potential errors are detected at early stages and do not accumulate, resulting in a reduction in the complexity of corrections in the later phases of the software life cycle and, in general, full consistency between different views is achieved. The proposed methodology extends the early work on the described format [4] and lays the groundwork for the practical use of JSON and XMI formats within the same model without the risk of out-of-sync. Quantitative evaluation confirmed that the transition from full validation $O(N{\cdot}d)$ to incremental $O(k{\cdot}d)$ reduces the complexity of consistency checking by an order of magnitude for a typical ratio of $k \ll N$. The risk factors include the scalability of Alloy analysis for models > 5000 elements, incomplete coverage of OCL operators, and the need to integrate the change log and 3-way-merge into the IDE; these aspects will be the focus of future research.

The implementation of the validator prototype is promising

In the next work, we plan to implement a prototype validator in Python using the PyEcore library and the Alloy SAT solver. It is expected to support full and incremental verification modes, integration with popular IDEs and collection of static performance statistics. The scalability of the Alloy solver and full integration with the IDE API remain critical success factors, which will be the subject of further work. Experimental evaluation on real projects should confirm the effectiveness and scalability of the proposed solution.

REFERENCES

1. Fowler, M. “UML Distilled: A Brief Guide to the standard object modeling language”. *Boston: USA; Pearson Education Limited*. 2018.

2. Chonoles, M. J. “OCUP 2 Certification Guide: Preparing for the OMG Certified UML 2.5 Professional Foundation Exam”. *1-st ed. Amsterdam – Boston: Morgan Kaufmann (Elsevier)*. 2018. ISBN 978-0-12-809640-6.

3. “Object Management Group (OMG). XML Metadata Interchange (XMI) Version 2.5.1 Specification”. 2015. – Available from: https://www.omg.org/spec/XMI/2.5.1. – [Accessed: June 2025].

4. Nikitchenko, M. I. “Two-Tier UML Architecture Based on Hybrid JSON and XMI Format”. *Scientific Notes of Taurida National V. I. Vernadsky University, Series: Technical Sciences*. 2025; 2 (1): 157–162. DOI: https://doi.org/10.32782/2663-5941/2025.1.2/23.

5. Tang, G., Jiang, J. & Wen, H. “Consistency analysis of UML models”. In: *Proceedings of the 29th International Conference on Distributed Multimedia Systems*. 2023. DOI: https://doi.org/10.18293/dmsviva2023-187.

6. Clarisó, R., González, C. A. & Cabot, J. “Incremental Verification of UML/OCL Models”. *Journal of Object Technology*. 2020; 19 (3): 3:1. DOI: https://doi.org/10.5381/jot.2020.19.3.a7.

7. Tröls, M. A., Marchezan, L., Mashkoor, A. & Egyed, A. “Instant and Global consistency checking during collaborative engineering”. *Software and Systems Modeling*. 2022; 21 (6): 2489–2515. DOI: https://doi.org/10.1007/s10270-022-00984-4.

8. Brambilla, M., Cabot, J. & Wimmer, M. “Model-driven software engineering in practice”. *Cham: Switzerland; Springer*. 2017. p. 123–140. DOI: https://doi.org/10.1007/978-3-031-02549-5_8.

9. Bashir, R. S., Lee, S. P., Khan, S. U. R., Chang, V. & Farid, S. “UML Models Consistency Management: Guidelines for Software Quality Manager”. *International Journal of Information Management*. 2016; 36 (6): 883–899, https://www.scopus.com/record/display.uri?eid=2-s2.0-84975132479&origin=resultslist. DOI: https://doi.org/10.1016/j.ijinfomgt.2016.05.024.

10. Jongeling, R., Ciccozzi, F., Carlson, J. & Cicchetti, A. “Consistency management in industrial continuous model-based development settings: A reality check”. *Software and Systems Modeling*. 2022; 21 (6): 1511–1530, https://www.scopus.com/record/display.uri?eid=2-s2.0-85128275171&origin=resultslist. DOI: https://doi.org/10.1007/s10270-022-01000-5.

11. Torre, D., Labiche, Y., Genero, M. & Elaasar, M. “A systematic identification of consistency rules for UML diagrams”. *Journal of Systems and Software*. 2018; 144: 121–142. DOI: https://doi.org/10.1016/j.jss.2018.06.029.

12. Torre, D., Labiche, Y., Genero, M. & Elaasar, M. “How consistency is handled in model-driven software engineering and UML: An Expert-Opinion Survey”. *Software Quality Journal*. 2023; 31 (1): 1–54. DOI: https://doi.org/10.1007/s11219-022-09585-2.

13. Torre, D., Labiche, Y., Genero, M. & others. “UML Consistency Rules: A Case Study with Open-Source UML Models”. In: *Proceedings of the 18th International Conference on Software Engineering and Formal Methods (SEFM 2020)*. 2020. p. 130–140. DOI: https://doi.org/10.1007/978-3-030-58768-0_8.

14. Muram, F. U., Tran, H. & Zdun, U. “Systematic review of software behavioral model consistency checking”. *ACM Computing Surveys*. 2017; 50 (2): 1–39. DOI: https://doi.org/10.1145/3037755.

15. Kungurtsev, O. & Novikova, N. “Identification of class models imperfection”. *Herald of Advanced Information Technology*. 2020; 3 (2): 13–22. DOI: https://doi.org/10.15276/hait.02.2020.1.

16. Cicchetti, A., Ciccozzi, F. & Pierantonio, A. “Multi-view approaches for software and system modelling: A Systematic Literature Review”. *Software and Systems Modeling*. 2019; 18 (6): 3207–3233, https://www.scopus.com/record/display.uri?eid=2-s2.0-85061733132&origin=resultslist.
DOI: https://doi.org/10.1007/s10270-018-00713-w.

17. Klare, H., Kramer, M., Langhammer, M., Burger, E. & Reussner, R. “Enabling Consistency in View-Based System Development: The Vitruvius Approach”. *Journal of Systems and Software*. 2021; 171: 110827, https://www.scopus.com/record/display.uri?eid=2-s2.0-85092430945&origin=resultslist. DOI: https://doi.org/10.1016/j.jss.2020.110815.

18. Daniel, G., Sunyé, G., Tisi, M., Madiot, F. & Cabot, J. “NeoEMF: A Multi-Database Model Persistence Framework for Very Large Models”. *Science of Computer Programming*. 2018; 149: 26–52. DOI: https://doi.org/10.1016/j.scico.2017.08.002.

19. Sharbaf, M., Zamani, B., Sunyé, G. & Barbier, F. “Conflict Management Techniques for Model Merging: A Systematic Mapping Review”. *Software and Systems Modeling*. 2022; 21 (6): 2687–2713, https://www.scopus.com/record/display.uri?eid=2-s2.0-85140063127&origin=resultslist.
DOI: https://doi.org/10.1007/s10270-022-01050-9.

20. Sharbaf, M., Zamani, B., Sunyé, G. & Barbier, F. “Automatic Resolution of Model Merging Conflicts Using Quality-Based Reinforcement Learning”. *Journal of Computer Languages*. 2022; 71: 101123. DOI: https://doi.org/10.1016/j.cola.2022.101123.

21. Meghzili, S., Chaoui, A., Strecker, M. & Kerkouche, E. “On the Verification of UML State-Machine Diagrams to Colored Petri Nets Transformation Using Isabelle/HOL”. In: *Proceedings of the 2017 IEEE International Conference on Information Reuse and Integration (IRI)*. 2017. p. 419–426. DOI: https://doi.org/10.1109/iri.2017.63.

22. Noulamo, T., Tanyi, E., Nkenlifack, M., Lienou, J.-P. & Djimeli Tsajio, A. B. “Formalization Method of the UML Statechart by Transformation toward Petri Nets”. *IAENG International Journal of Computer Science*. 2018; 45: 505–513.

23. Paulin, O., Komleva, N., Marulin, S. & Nikolenko, A. "Method for Constructing the Model of Computing Process Based on Petri Net". *Applied Aspects of Information Technology*. 2019; 2 (4): 260–270. DOI: https://doi.org/10.15276/aait.04.2019.1.

24. Rajabi, B. A. & Lee, S. P. "Coevolution patterns to detect and manage UML diagrams changes". *International Journal of Computing*. 2019; 18 (4): 471–482, https://www.scopus.com/record/display.uri?eid=2-s2.0-85085195001&origin=resultslist.
DOI: https://doi.org/10.47839/ijc.18.4.1617.

25. Oriol, X. & Teniente, E. "Incremental Checking of OCL Constraints with Aggregates Through SQL". In: *Proceedings of the 34th International Conference on Conceptual Modeling (ER 2015)*. 2015; 9381: 199–213, https://www.scopus.com/record/display.uri?eid=2-s2.0-84951803713&origin=resultslist. DOI: https://doi.org/10.1007/978-3-319-25264-3_15.

26. Lu, S., Tazin, A., Chen, Y., Kokar, M. & Smith, J. "Ontology-Based Detection of Inconsistencies in UML/OCL Models". In: *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development MODELSWARD*. 2022. p. 194–202, https://www.scopus.com/record/display.uri?eid=2-s2.0-85146603066&origin=resultslist.
DOI: https://doi.org/10.5220/0010814500003264.

27. Wen, H., Wu, J., Jiang, J., Tang, G. & Hong, Z. "A Formal Approach for Consistency Management in UML Models". *International Journal of Software Engineering and Knowledge Engineering*. 2023; 33 (5): 733–763, https://www.scopus.com/record/display.uri?eid=2-s2.0-85153971381&origin=resultslist.
DOI: https://doi.org/10.1142/S0218194023500134.

28. Leblebici, E., Anjorin, A. & Schürr, A. "Inter-Model Consistency Checking Using Triple Graph Grammars and Linear Optimization Techniques". In: *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE 2017)*. 2017; 10202: 191–207, https://www.scopus.com/record/display.uri?eid=2-s2.0-85016392599&origin=resultslist.
DOI: https://doi.org/10.1007/978-3-662-54494-5_11.

29. Nikitenko, M. I. "Analysis of formats for storing UML models in modern CASE tools". In: Proceedings of the IV International Scientific Conference "Technology and Society: Interaction, Influ-ence, Transformation.". 2025; p. 208–212. DOI: https://doi.org/10.62731/mcnd-20.06.2025.

30. Stünkel, P., König, H., Lamo, Y. & Rutle, A. "Comprehensive Systems: A Formal Foundation for Multi-Model Consistency Management". *Formal Aspects of Computing*. 2021; 33 (6): 1067–1114. DOI: https://doi.org/10.1007/s00165-021-00555-2.

31. Overbeek J.F. "Meta Object Facility (MOF): Investigation of the State of the Art". Master's thesis. Enschede: University of Twente; 2006. 92 p. – Available from: https://essay.utwente.nl/57286.

32. Kruchten P. "Architectural Blueprints – The '4 + 1' View Model of Software Architecture". IEEE Software. 1995; 12(6): p. 42–50. DOI: https://doi.org/10.1109/52.469758.

33. Lu, L. & Kim, D.-K. "Required Behavior of Sequence Diagrams". *ACM Transactions on Software Engineering and Methodology*. 2014; 23 (2): 1–28. DOI: https://doi.org/10.1145/2523108.

34. Jackson, D. "Alloy: A Language and Tool for Exploring Software Designs". *Communications of the ACM*. 2019; 62 (9): 66–76. DOI: https://doi.org/10.1145/3338843.

35. Bagheri, H. & Malek, S. "Titanium: Efficient Analysis of Evolving Alloy Specifications". In: *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2016)*. Seattle, USA. 2016. DOI: https://doi.org/10.1145/2950290.2950337.

36. Cabot, J. & Gogolla, M. "Object Constraint Language (OCL): A Definitive Guide". In: *Formal Methods for Model-Driven Engineering*. Berlin: Germany; Springer. 2012. p. 58–90. DOI: https://doi.org/10.1007/978-3-642-30982-3_3.

Conflicts of Interest: The authors declare that they have no conflict of interest regarding this study, including financial, personal, authorship or other, which could influence the research and its results presented in this article
Author Nataliia O. Komleva is a member of the Editorial Board ofthis journal.This role had no influence on the peer review process or editorial decision regarding this manuscript

Received 08.04.2025
Received after revision 12.06.2025
Accepted 18.06.2025

DOI: https://doi.org/10.15276/aait.08.2025.11
УДК: 004.421.2:004.415.5

Метод інкрементального контролю консистентності між структурним і поведінковим поданням програмної архітектури

Комлева Наталія Олегівна[1)]
ORCID: https://orcid.org/http://orcid.org/0000-0001-9627-8530; komleva@op.edu.ua. Scopus Author ID: 57191858904
Нікітченко Максим Ігорович[1)]
ORCID: https://orcid.org/0009-0007-9560-7057; maksym.nikitchenko@gmail.com
[1)] Національний університет "Одеська політехніка", пр. Шевченка, 1. Одеса, 65044, Україна

АНОТАЦІЯ

Розвиток програмної інженерії ставить перед дослідниками завдання підтримувати цілісність моделей, що зберігаються одночасно у легкому текстовому та у формально насиченому метаданому поданні. Наявність двох представлень забезпечує широку сумісність з інструментами розробників і точне відтворення семантики, проте породжує ризик розбіжностей між структурним і поведінковим описами. Актуальність дослідження визначається потребою в методах, які унеможливлюють накопичення суперечностей без суттєвого впливу на швидкість ітерацій проектування. Метою роботи є теоретичне обґрунтування інкрементального підходу, здатного гарантувати узгодженість метамоделі з двома поданнями під час будь-яких послідовних змін. Для досягнення цієї мети сформовано узагальнену метамодель, що виокремлює структурне подання для статичних сутностей і поведінкове подання для динамічних аспектів. Між поданнями запроваджено відношення відповідності, яке описує пари еквівалентних елементів і задає правила їх взаємної узгодженості. Сукупність правил формалізовано мовою об'єктних інваріантів. Інкрементальність забезпечена локалізацією змін: після редагування перевіряються лише ті фрагменти, що безпосередньо залучені до модифікації, завдяки чому часові витрати залишаються пропорційними обсягу оновленої частини. Наслідком застосування методу є доведення коректності запропонованих обмежень, яке виключає можливість виникнення несумісних станів моделі. Аналітична оцінка складності процедури підтверджує лінійну залежність від кількості змінених елементів, що свідчить про придатність підходу для промислових розмірів моделей. Демонстраційний контрольний приклад, побудований на репрезентативному домені, засвідчив, що метод виявляє інконсистентність одразу після одиночної правки та пропонує послідовність дій, достатню для її усунення без залучення сторонньої експертизи. У підсумку робота пропонує нову формальну методику підтримки узгодженості між поданнями однієї моделі, яка комплексно поєднує локалізовану перевірку з декларативним описом залежностей. Практична значущість проявляється у зменшенні витрат на виправлення помилок, підвищенні надійності документації та можливості інтегрувати метод у сучасні середовища моделювання і безперервної розробки, що робить його перспективним інструментом при розробці та супроводу великих корпоративних систем..

Ключові слова: Узгодженість моделей; інкрементальна перевірка; синхронізація моделей; метамодель; онто-логічні обмеження, надійність

ABOUT THE AUTHORS

Nataliia O. Komleva – Candidate of Engineering Sciences, Associate Professor, Head of Software Engineering Department. Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044, Ukraine
ORCID: http://orcid.org/0000-0001-9627-8530; komleva@op.edu.ua. Scopus Author ID: 57191858904
Research field: Data analysis; software engineering; knowledge management

Комлева Наталія Олегівна – кандидат технічних наук, завідувач кафедри Інженерії програмного забезпечення, Національний університет «Одеська політехніка», пр. Шевченка, 1. Одеса, 65044, Україна

Maksym I. Nikitchenko – graduate student, Software Engineering Department. Odesa Polytechnic National University. 1, Shevchenko Ave. Odesa, 65044, Ukraine
ORCID: https://orcid.org/0009-0007-9560-7057; maksym.nikitchenko@gmail.com
Research field: Software Engineering

Нікітченко Максим Ігорович – аспірант кафедри Інженерії програмного забезпечення. Національний університет "Одеська політехніка", пр. Шевченка, 1. Одеса, 65044, Україна