

Methodology for Rapid Development of PAIS Based on Ontological Modeling and Category Theory

Borivoj Bogdanović¹, Vidan Marković¹, Milan Segedinac², and Zora Konjović¹

¹ Singidunum University, Danijelova 32

11010 Belgrade, Serbia

borivoj.bogdanovic@gmail.com (corresponding author),
{vidan.markovic, zkonjovic}@singidunum.ac.rs

² University of Novi Sad, Faculty of Technical Sciences

Trg Dositeja Obradovica 4, Novi Sad 21000, Serbia
milansegedinac@uns.ac.rs

Abstract. Model-Driven Architecture (MDA) structures software development through computation-independent, platform-independent, and platform-specific models. Its practical value depends on machine-readable models and transformations that preserve relevant structural properties while producing executable code. This paper proposes a methodology for the rapid development of Process-Aware Information Systems (PAIS) based on ontological modeling and category theory. Ontology-Driven Code Generation (ODCG) is formalized as a deterministic, multi-phase transformation from the BAB Core ontology, through a platform-specific ontology and library, to source code. Model levels are represented as categories, while transformations are represented as functors. The main contribution is a categorical formalization of stack-parametrized multi-target generation, together with a conditional guarantee of schema-level structural preservation under explicitly stated verification assumptions, including mapping completeness, injectivity, quotient compatibility, and absence of unintended collapse. The methodology distinguishes the categorical specification, in which functoriality follows by construction, from its operational implementation, whose conformance is established through explicit engineering checks. The approach is implemented and evaluated for the Java platform through the BABj ontology and library by generating code for an operational business information system during its life cycle.

Keywords: Model-Driven Engineering, Ontology, Category Theory, Code Generation, PAIS

1. Introduction

Abstraction is crucial in software engineering for managing complexity, and there is a constant tendency to introduce ever-higher levels of abstraction into the software development process, ranging from implementation to modeling. Model-Driven Architecture (MDA) [50] is based on the principle of using modeling languages and specifying systems at different levels of abstraction: a Computation Independent Model (CIM) for representing the environment and system requirements, a Platform Independent Model

(PIM) that describes the system in a technology-neutral manner, and a Platform Specific Model (PSM) that adds details related to a specific platform [19]. The vision of MDA is that, through automated transformations, a PIM can be translated into a corresponding PSM and further into executable code on the target platform, thereby ensuring that once-defined system logic can be semi-automatically implemented on different technology stacks, with a reduction in manual programming. Such an approach poses two basic tasks: (1) a machine-readable representation of the domain/information system model at different levels of abstraction, and (2) the transformation of models so represented to the level of executable code, with the fundamental requirement of preserving the selected schema-level structure and explicitly represented mapping distinctions during the transformation process.

Ontologies are inherently suitable for representing PIMs because they allow for the definition of the system domain at an abstract level, independent of implementation. If represented via standards such as RDF and OWL, they are machine-readable and have formally defined semantics, making them suitable for sharing and reusing domain knowledge. This paper explores an approach in which such an abstract domain ontology (core ontology) is transformed into a technology-specific ontology (analogous to a PSM), from which program code is subsequently generated. The transformation from a general model (domain ontology) to concrete code is viewed here as a functorial process, while the alignment between the abstract schema-level structure and its platform-specific realization is modeled through natural transformations.

The proposed framework represents the PIM through an ontology named BAB Core (Business Application Builder Core), which contains domain knowledge about PAIS, including entities, processes, tasks, roles, and user interfaces. A platform-specific ontology and accompanying library provide the implementation details required for a particular technological environment. In the present work, this platform-specific realization is BABj, which maps abstract BAB Core concepts to Java/Jakarta EE and Vaadin artifacts, such as JPA entities, Vaadin views, and EJB services. BABj is the only implemented and evaluated target stack considered in this paper [9,11,10].

The primary scientific contribution of the approach proposed in this paper is a foundational mechanism for transformation from model to model up to the level of generated code, while preserving selected schema-level structure and explicitly represented mapping distinctions across the transformation chain under explicitly stated verification assumptions. The practical result of the research is a repeatable and systematic code-generation process by which schema-level application structure and selected architectural realization rules, formally defined through ontologies, are translated into the Java-based BABj architecture in a controlled manner. Domain-specific method bodies remain explicit extension points and are not claimed to be fully synthesized by the present transformation. Since the transformer uses established Java/Jakarta EE patterns and the BABj library, adherence to consistent architectural practices is ensured across BABj-based projects.

The remainder of the paper is organized as follows. Section 2 provides an overview of previous research on approaches to code generation from ontologies through multi-phase and formally based transformations. For the selection of papers and analysis, a framework defined by five themes was adopted, and for each of them, implications for the research in this paper are explicitly highlighted. Section 3 introduces the ontological foundations used in the paper, while Section 4 presents the category-theoretic concepts underlying the

proposed methodology. Section 5 presents the proposed methodology and the categorical transformation framework in detail. Section 6 describes the Java-based implementation of the transformer. Section 7 presents the analysis and discussion of results, and the final section provides concluding remarks and directions for future research.

2. Related Work

The subject of the research presented in this paper is a methodology for the accelerated development of Process-Aware Information Systems (PAIS), a concept explicated more than 20 years ago in [24]: “A Process-Aware Information System (PAIS) is a software system that manages and executes operational processes involving people, applications, and/or information sources on the basis of process models.” The same source classifies PAIS according to four orthogonal dimensions: (1) orientation (design vs. implementation), (2) participants involved (people and software applications), (3) process structure and predictability (unframed, ad hoc framed, loosely framed, and tightly framed), and (4) organizational scope (intra-organizational, inter-organizational), while source [25] classifies PAIS as domain-specific and domain-agnostic. Among more recent results, the monograph [33] stands out, confirming the viability of the PAIS concept, i.e., its potential for evolution leading towards Intelligent PAIS (IPAIS) with new concepts related to artificial intelligence, but also new development approaches, among which model-driven and collaborative approaches are prominent.

The PAISs of interest for our research are those that are domain-specific (business information systems), framed (tightly or ad hoc), and implementation-oriented.

The convergence of ontological modeling and software engineering has a long history and continuous development. The beginnings date back to the 1980s with the emergence of the concept of using formal knowledge structures (ontologies) for reasoning in artificial intelligence systems, expanding philosophical ideas into computational models for knowledge representation [37,13,69]. The last decade of the previous century was marked by formal methods and the Semantic Web, aiming to ensure interoperability, give models deeper meaning, and enable automated reasoning using Semantic Web standards (RDF, OWL) [7,6,44,21].

The beginning of this century (up to 2015) brought key results in the field: the idea and assessment of the possibilities of applying ontologies in software engineering [58,15,54,30,57], combining UML and ontologies [27,53,14], combining ontologies and Model Driven Engineering (MDE) approaches [65,40,4,12,66,64] and support for data access and queries [70,16,8]. In addition, during this period, the paper of Diskin and Maibaum [22] appeared, systematically introducing category theory into model-driven engineering.

The last 10 years (from 2015 to present) have been marked by research that continues to address the combination of UML and ontologies [51], model transformation in the MDE approach [34,62,43,38,1,63,39], automatic model-to-text code generation [32,67,9,11,10], the application of ontologies in data management [66,70,41,56,16,18,36], AI integration [3,48,61,46,17], infrastructure for ontology development and integration [23,28,29,26,68,20], automation of the software life-cycle [55], and ever-present new domain applications [2,45,49,60] that seek solutions and

thereby contribute to the development of the field. Finally, there are papers dealing with the application of the category theory apparatus in software engineering [71,52,42].

In the section that follows, papers directly relevant to the subject of research are singled out. The results are grouped into two categories: results dealing with the approach of code generation from ontologies through multi-phase and formally based transformations, and results dealing with the application of category theory in the field of model-driven software development. The analyzed papers are organized into the following thematic units: (1) Ontologies and Software Engineering, (2) MDE/MDA and Transformations, (3) OBDA and Semantic Web, (4) LLM/AI Support, and (5) Formalization of MDE Patterns and Synergy with Ontologies, with explicit highlighting of implications for this work.

2.1. Ontologies and Software Engineering

From the first group of papers, of interest to this research is paper [51] which provides an exhaustive overview of combining UML and ontologies, considering, among other things, ways to make bidirectional model synchronization feasible; paper [34] which deals with the relationship between metamodels and ontologies, directly supporting the use of a core ontology as a PIM layer above technology; papers [32,67,9,11,10] which deal with automatic model-to-text source code generation alongside a “best practices” library; papers on the adoption of ontological modeling in software engineering practice that identify a pronounced need for tools that bring semantic models closer to programmers (editors, linters, generators) [23,26]; and papers dealing with collaborative work and “design review” that emphasize the importance of visualization [23] and the automation of core model generation by learning from existing artifacts (code, APIs, data schemas) [3]. Mkhinini et al. in their review [51] analyze the convergence of UML and ontologies. The authors identify that, although UML offers the visual standardization necessary for engineers, it lacks the formal semantics for automated reasoning offered by ontologies. The conclusion of their research points to the necessity of hybrid approaches where ontologies take on the role of a “metamodel with semantics,” enabling verification that classic UML tools cannot provide.

The methodology proposed in this paper positions the ontology as the central Platform-Independent Model (PIM) and the Single Source of Truth (SSoT) for the domain. To ensure the consistency and completeness of this model before code generation, a supporting tool layer is introduced. Its key functions (visualization, validation, and support for semi-automated concept learning) are applied as mandatory steps before triggering transformations towards the PSM layer and final code.

2.2. MDE/MDA and Transformations (Tools, Patterns, Verification, Collaboration)

Contemporary reviews of MDE note a trend of merging with ML/analytics and collaborative workflows [20,48], which is relevant for the automatic suggestion of mapping patterns in the BABj ontology and team-based model evolution. Source [38] offers a taxonomy of transformation tools and techniques (model-to-model vs. model-to-text), while [43] summarizes transformation design patterns that can be formalized (e.g., Entity → JPA, Process → Service, View → UI-class). Here, the key issue is correctness, and results

vary from techniques for transformation verification (invariants, provable rules) [14], through the systematization of model composition essential for merging modules and sub-ontologies prior to generation [1], to specialized solutions for specific domains (e.g., digital twins) [45,49].

The methodology in this paper defines the transformation approach as a formalized, strictly controlled model-to-text (M2T) process. This process relies on two key elements: (1) explicit transformation rules ($\text{core} \rightarrow \text{BABj}$; $\text{BABj} \rightarrow \text{code}$) grounded in recognized patterns from the literature, and (2) a multi-layered verification framework encompassing invariant checking, ontology validation, and evaluation of generated code via static analysis.

2.3. OBDA and Semantic Web (Data Access, Interoperability, Practical tooling)

Another area with significant reliance on semantic technologies is Ontology-Based Data Access (OBDA), which emphasizes the integration of ontological models with relational and heterogeneous data stores [18], and where results include ways of merging semantic modeling with data lake approaches, opening space for the pipeline to generate data-access artifacts (e.g., R2RML/OBDA configurations) alongside application code [36], or to support migrations between stacks/organizations [68].

Even early research like RDFReactor [70] showed that $\text{ontology} \rightarrow \text{OO API}$ transformation is feasible and useful for developers. The EMF/Ecore line [66] established a mature practice of $\text{model} \rightarrow \text{Java}$ generation, while KOMMA/Empire/Elmo (AliBaba) [41,56] illustrate different approaches to object/triple mapping towards RDF stores.

In accordance with the cited results, the methodology proposed in this paper adopts proven concepts from the literature, such as typed API generation and the application of patterns, but differs by extending these ideas to the entire information system (entities, services/processes, user interface, and data access), thereby creating a basis to ensure, through the application of formal mapping and verification, schema-level structural and architectural consistency of the generated system with the central core ontology. Based on the cited results from the field of OBDA (Ontology-Based Data Access), the proposed methodology extends the target scope of generation beyond exclusively application code. The transformation pipeline is, therefore, designed to produce a complete data access layer as well. In this way, it is ensured that generated data-access artifacts are structurally and schema-level consistent with the central core ontology.

To position the approach against practical tooling, we compare it along two axes: the target scope (a single layer versus the whole information system) and the presence of a formal, verifiable preservation guarantee. EMF/Ecore [66] established mature model-to-Java generation but targets primarily the structural/persistence layer and offers no categorical preservation guarantee. RDFReactor [70] generates a typed object-oriented API from an ontology, again single-layer and without formal preservation claims. KOMMA and AliBaba (Elmo/Empire) [41,56] provide object/triple mapping toward RDF stores, focused on data access. Ontop [16] realizes OBDA by rewriting queries over databases without generating application code at all. Against these, the proposed method targets the whole system (entities, services/processes, UI, and data access) from a single ontological source and adds a formal, checkable structural-preservation guarantee across a multi-target pipeline. The distinction is thus not the idea of generating code from a model,

which these tools already demonstrate, but the combination of end-to-end scope with a categorical correctness argument.

2.4. LLM/AI for Ontologies and Generation (Assistance, Qualitative Checks)

Recent research indicates that LLMs accelerate OE (Ontology Engineering): concept proposal, consistency checking, assistance with mapping and documentation [46,61], and thus its application in various domains. In parallel, a review of ontology embeddings [17] opens the possibility for patterns and code artifacts to be selected based on semantic similarity and experience from previous projects.

As a response to contemporary papers on the application of LLM technologies in ODE, the methodology conceptually envisions the possibility of introducing AI-assisted mapping through a human-in-the-loop (HITL) process. In such a mode of operation, the LLM would propose BABj mapping patterns (with explanations), while the engineer would retain explicit control through a mandatory validation step before applying the rules. This would accelerate the cognitively demanding mapping process while preserving formal control and the deterministic behavior of the transformation layer.

2.5. Domain Applications and Industry

Finally, systematic literature reviews (SLR) and reviews by domain, e.g., manufacturing [60], show that ontologies provide measurable value in complex industrial scenarios. This suggests that the core $\rightarrow$ stack $\rightarrow$ code approach has the potential for cross-domain use, where domain specificities are captured through the core model and exploited through different target stacks.

To examine the potential for cross-domain application identified in the literature review, the methodology is evaluated through a case study from a complex industrial domain. The purpose is to illustrate the operation of the core $\rightarrow$ stack $\rightarrow$ code approach in an industrially relevant scenario and to assess the architectural consistency and engineering quality of the generated artifacts.

2.6. Formalization of MDE Patterns and Synergy with Ontologies

Progress in the field of Model-Driven Engineering increasingly relies on rigorous mathematical formalization to overcome the problems of ad-hoc transformations. Diskin and Maibaum, in their paper [22] which is considered fundamental for MDA application, argue that category theory in MDE serves not merely as an “abstract support”, but as a source of concrete design patterns for model management. They show that key engineering tasks, such as model merging and transformation, can be directly modeled as categorical constructions. This is of essential importance for ensuring consistency and traceability in complex transformation chains, as it suggests that model integrity is maintained not by procedural scripts, but by structural mappings that preserve information.

Complementary to this, Kovalyov [42] proposes a comprehensive category-theoretic approach to designing software systems, emphasizing the ability of categories to unify heterogeneous engineering technologies. His work sets the foundations for treating different aspects of a software system (data, processes, UI) through a unified formal framework, which directly supports the idea of a technology-neutral core (like BAB Core) that is projected via functors onto specific technological stacks.

The categorical approach gives precise semantics to the proposed transformations. Mordecai and associates [52] offer a vocabulary for MBSE cycles, which can support the formalization of functors realizing Core-to-Stack and Stack-to-Code transformations, while their composition models the complete transformation pipeline. Zhu with co-authors [71] demonstrates the application of categories in verification, where implementation respects design in concurrent systems. Analogously, commutativity of diagrams is required during refactoring (rename/split/merge) and preservation of invariants (cardinalities, mandatory fields, completeness of process transitions).

A closely related and complementary line of work targets the automatic generation of domain behavior from requirements and knowledge-representation models. In particular, [59] derives application logic from requirement and knowledge models rather than only structural skeletons. Our work is orthogonal in scope: whereas [59] focuses on synthesizing behavior, we provide a formally specified, structure-preserving, multi-target transformation backbone. The two directions are combinable: the behavioral synthesis of [59] could populate the explicit method-body extension points (“hooks”) that our pipeline leaves open, while our categorical framework would supply the structural and multi-platform guarantees.

The methodology presented in this paper directly concretizes theoretical insights from the cited literature. Relying on Diskin’s concept of categorical design patterns, the proposed transformation system models code generation not as mere textual substitution, but as a functor mapping the structure of the ontology into the structure of the target language. This achieves the mathematical guarantee of structural correctness advocated by Kovalyov. At the same time, acknowledging the conclusions of Mkhinini et al., this paper uses ontology as a super-model that complements the shortcomings of traditional UML modeling, providing the semantic basis necessary for the automated generation of schema-level application structure and selected architectural realization artifacts. The proposed methodology thus defines formal, measurable criteria for the verification of transformations, directly derived from the application of category theory. The purpose of these criteria is to verify that the generated code remains faithful to the selected schema-level structure and explicitly represents mapping distinctions of the core model. These criteria encompass: (1) checking the commutativity of the relevant diagrams during transformation and refinement operations and (2) validating the preservation of defined invariants, such as cardinalities and transition completeness, through each transformation step.

3. Ontologies as a Basis for Model Specification

Ontologies are formal specifications of shared conceptualizations, expressed in a language with well-defined syntax and semantics, intended to capture the structure of a domain in a machine-interpretable form. In the context of MDE and code generation, ontologies serve as a particularly suitable foundation for specifying platform-independent models: their formal syntax permits automated processing, their declarative semantics admits verification through reasoning, and their extensibility through import mechanisms supports modular construction of domain and platform descriptions. This section introduces the ontological concepts used throughout the paper. The presentation is kept to the minimum needed for the categorical formalism of subsequent sections; for a comprehensive treatment of the semantic web technology stack, we refer the reader to [35].

3.1. OWL 2 Ontologies basics

The Web Ontology Language (OWL 2) is a W3C standard for expressing ontologies, based on description logic. The language defines several computational profiles; in this work, we use OWL 2 DL, the profile corresponding to the description logic, which provides a balance between expressiveness and decidability of reasoning. The BAB Core ontology, the BABj ontology, and all domain ontologies constructed on the top of them conform to this profile.

Classes, Individuals, and Object Properties An OWL ontology consists of three primary kinds of entity:

- Classes represent types or categories of concept, for instance, `core:Entity`, `core:Process`, or `pms:Shipment`. Formally, a class is interpreted as a set of individuals sharing structural or semantic characteristics.
- Individuals represent specific instances of classes, for instance, a particular shipment `pms:shipment123` as an instance of `pms:Shipment`. In description-logic terminology, individuals are the elements of the universe of discourse.
- Properties represent binary relations, and are subdivided into object properties, which relate pairs of individuals (e.g., `core:hasAttribute` relating an entity to its attribute), and datatype properties, which relate an individual to a literal value (e.g., a string, integer, or date).

Each property declaration carries `rdfs:domain` and `rdfs:range` restrictions that specify, respectively, the class of admissible sources and the class or datatype of admissible targets. These declarations enable both syntactic validation of the ontology and inference of class membership from property usage.

Object Properties versus Datatype Properties The distinction between object properties and datatype properties is structurally significant for the methodology developed in this paper. An object property links two individuals, both of which are first-class entities in ontology and can themselves participate in further relations. A datatype property links an individual to a literal value (a string, number, or other XSD-typed datum) that has no further ontological structure. Consequently, only object properties generate navigable relations in the class diagram of the ontology, and only they contribute edges to the schema graph on which the categorical formalism of Section 5.4 is built. Datatype properties carry information at the instance level but do not participate in the schema-level structure on which functor construction depends. This distinction will be used in Section 5.1 to justify the design of the `ImplementationMapping` mechanism.

Subsumption and Subproperty Hierarchies Classes and properties in OWL form hierarchies through the `rdfs:subClassOf` and `rdfs:subPropertyOf` relations. A declaration asserts that every instance of `is` is also an instance of `is`; analogously, asserts that every pair related by `is` is also related by `is`. These hierarchies express taxonomic organization of the domain and serve as the basis for several mechanisms used in this work: domain specialization (a domain class becomes a subclass of an abstract BAB Core class), refinement of mapping strategies (a specialized mapping class becomes a subclass of a general one), and inheritance of properties from classes to their subclasses.

Reification A modeling decision frequently encountered in ontology engineering is whether to represent a concept as a property or as a class. A property, being a binary relation, can associate a subject with an object but cannot itself carry structured information about the nature of the association. A class, in contrast, admits instantiation into individuals and participation in its own hierarchy and relations. When a concept inherently carries structured information, multiple attributes, internal composition, or its own refinement structure, modeling it as a class rather than a property is the standard choice. This transformation is known as reification, and it is the modeling pattern underlying the `core:ImplementationMapping` construct introduced in Section 5.1.

The Open World Assumption OWL 2 operates under the Open World Assumption (OWA): a fact that is not asserted in the ontology is treated as unknown, not a false. This contrasts with the closed-world assumption common in relational databases, where absence from the database is interpreted as absence from the world. Under OWA, the absence of a `hasAttribute` declaration for a particular entity does not entail that the entity has no attributes; it entails only that the ontology does not record such information. Consequences for the presented work are modest but important: statements of the form “every entity has at least one attribute” cannot be enforced by mere silence but must be expressed through explicit cardinality restrictions, verified by the reasoner, to take effect.

Reasoning A reasoner is an automated procedure that derives implicit consequences from the explicit assertions of an ontology, using formal semantics based on underlying logical language. Standard reasoning tasks include consistency checking (verifying that the ontology admits a non-empty interpretation), classification (computing the full subsumption hierarchy of classes), and realization (inferring the most specific class memberships of individuals). In this work, we use Hermit [31], a complete OWL 2 DL reasoner, to validate domain ontologies constructed on top of the BAB Core: cardinality constraints, property restrictions, and disjoint declarations defined in BAB Core serve as validation criteria, and the reasoner’s verdict, consistent or inconsistent, determines whether a domain ontology is admitted as input to the transformation pipeline. The role of the reasoner in the overall methodology is discussed in Section 5.6.

SPARQL SPARQL is the W3C standard query language for RDF graphs, and by extension for OWL ontologies serialized as RDF. A SPARQL query consists of a pattern of triples with variables; the query engine returns all bindings of the variables that make the pattern a subgraph of the queried data. SPARQL is the mechanism by which the implementation of the proposed methodology extracts structural information from domain and platform ontologies during transformation: the functorial specifications developed in Section 5.4 are operationally realized by a collection of SPARQL queries that traverse the ontology graph along the declared object properties and collect the resulting structural data into an in-memory representation. The implementation details are presented in Section 6.

Ontology Import OWL 2 provides an `owl:imports` construct by which one ontology incorporates the class, property, and individual declarations of another ontology. Imports

are transitive and support modular construction of complex ontological landscapes: a domain ontology (such as the Postal Management System ontology used in later examples) imports BAB Core; the BABj platform ontology imports both BAB Core and the Web of Code (WoC) vocabulary; and domain ontologies targeting the Java stack import BABj. This layered structure is central to the methodology of this paper, allowing a separation between platform-independent abstract vocabulary (BAB Core), platform-specific structural vocabulary (BABj, based on WoC), and domain-specific elaboration.

Schema versus Data: T-box and A-box An OWL ontology is conventionally understood to comprise two strata. The T-box (terminological box) contains class and property declarations, subsumption hierarchies, and logical axioms, i.e., the schema of the domain. The A-box (assertional box) contains individuals and assertions of their class memberships and property values, the data populating the schema. The transformation formalized in this paper operates at the T-box level: it transforms one schema (BAB Core together with a domain ontology) into another (BABj with a generated class hierarchy and methods) and ultimately into source code. Assertions about individuals, which constitute the A-box, are not part of the transformation, and consequently the formal guarantees proved in Section 5.5 are guarantees about the schema structure, not about the runtime data flowing through the generated system. This distinction will be made explicit in the scope discussion accompanying the main preservation result.

4. Category Theory as a Basis for Model Transformations

Category Theory (CT) is a branch of mathematics that studies structures and relationships between them at a high level of abstraction. It is often described as “mathematics about mathematics” because it provides a unified language and formalism for describing various mathematical domains (such as set theory, algebra, topology). For software engineering, and especially for Model-Driven Development and model transformations, CT offers a rigorous foundation for precisely defining, composing, and verifying systems and processes. The basic idea is to shift the focus from elements (what is inside a set or class) to the relationships between them (morphisms or “arrows”). Roughly speaking, category theory expresses the view that an element has no internal structure; rather, all knowledge about an element is contained in its connections with other elements.

4.1. Basic Concepts of Category Theory

This section concisely presents the basic concepts of category theory used in the proposed methodology: category, functor, natural transformation, and bifunctor as fundamental concepts, as well as adjunction as a construct that enables the formal interpretation of bidirectional mappings between models and code.

Category A category $\mathcal{C}$ is formally defined by two sets of elements:

- A set of objects, denoted as $\text{Ob}(\mathcal{C})$.
- A set of morphisms (arrows), denoted as $\text{Mor}(\mathcal{C})$.

Each morphism f has exactly one source object X and one target object Y , written as $f : X \rightarrow Y$. The elements of a category must satisfy two conditions (axioms):

- **Composability:** For any two morphisms $f : X \rightarrow Y$ i $g : Y \rightarrow Z$, there must exist a third morphism $h = g \circ f$ (read: “g after f”) which maps $h : X \rightarrow Z$. This composition must be associative: $(h \circ g) \circ f = h \circ (g \circ f)$.
- **Identity:** For every object X in $\text{Ob}(\mathcal{C})$, there must exist a special identity morphism $\text{id}_X : X \rightarrow X$, which acts neutrally in composition. For every $f : A \rightarrow X$ and $g : X \rightarrow B$, it holds that: $\text{id}_X \circ f = f$ and $g \circ \text{id}_X = g$.

Examples of categories:

- **Set:** A category where objects are sets and morphisms are functions between sets.
- **Types:** In programming languages, objects can be data types (e.g., `int`, `String`), and morphisms can be functions (e.g., `toString: int → String`).
- **Ontology:** An ontology can be viewed as a category where objects are concepts and morphisms are relations between them.

Functor Functors are mappings between categories that preserve the structure. If there are two categories, $\mathcal{C}$ and $\mathcal{D}$, a functor $F : \mathcal{C} \rightarrow \mathcal{D}$ defines:

- A mapping of every object X from $\mathcal{C}$ to an object $F(X)$ in $\mathcal{D}$.
- A mapping of every morphism $f : X \rightarrow Y$ from $\mathcal{C}$ to a morphism $F(f) : F(X) \rightarrow F(Y)$ in $\mathcal{D}$.

For this mapping to be a functor, it must “respect” the structure of category $\mathcal{C}$:

- It must preserve identities: $F(\text{id}_X) = \text{id}_{F(X)}$ for every object X in $\mathcal{C}$.
- It must preserve composition: $F(g \circ f) = F(g) \circ F(f)$ for all morphisms f and g .

A functor can be interpreted as a structural translator, enabling different domains (e.g., an ontology and a database) to be connected through the same abstract structure.

Bifunctor A bifunctor is a functor of two arguments, formally, a functor from a product category $\mathcal{C} \times \mathcal{D}$ to a target category $\mathcal{E}$. Given two categories $\mathcal{C}$ and $\mathcal{D}$, their product category $\mathcal{C} \times \mathcal{D}$ has pairs (A, B) as objects and pairs of morphisms (f, g) composed component-wise. A bifunctor $F : \mathcal{C} \times \mathcal{D} \rightarrow \mathcal{E}$ maps each pair of objects to an object in $\mathcal{E}$ and each pair of morphisms to a morphism in $\mathcal{E}$, preserving identities and composition in both arguments simultaneously. Bifunctors arise naturally in settings where a construction depends on two independent inputs, such as the evaluation functor $\text{eval} : [\mathcal{C}, \mathcal{D}] \times \mathcal{C} \rightarrow \mathcal{D}$ that applies a functor to an argument.

Isomorphism in Categories An isomorphism constitutes a morphism within a category that is formally invertible. A morphism $f : A \rightarrow B$ is defined as an isomorphism if there exists an inverse morphism $g : B \rightarrow A$ satisfying the conditions $g \circ f = \text{id}_A$ and $f \circ g = \text{id}_B$, where id denotes the identity morphism on the respective objects. When such a relationship exists, the objects are considered isomorphic, implying they are structurally indistinguishable from the perspective of that category. In a programming context, this equivalence is exemplified by the mapping between a `String` and a `List<Char>`, where the transformation allows for lossless bidirectional conversion of the underlying data structure.

Natural Transformation Natural transformations are a central concept used in this paper. If functors are mappings between categories, natural transformations are mappings between functors, defined as follows.

Let F and G be two functors, both mapping from the same source category $\mathcal{C}$ to the same target category $\mathcal{D}$ ($F, G : \mathcal{C} \rightarrow \mathcal{D}$). A natural transformation $\tau : F \Rightarrow G$ (read as: “tau from F to G ”) is a family of morphisms in the target category $\mathcal{D}$. For every object X in $\mathcal{C}$, τ provides a morphism $\tau_X : F(X) \rightarrow G(X)$ in category $\mathcal{D}$.

Furthermore, this family of morphisms must satisfy the naturality condition (or “naturality square”) $G(m) \circ \tau_X = \tau_Y \circ F(m)$, implying that for every morphism $m : X \rightarrow Y$ in category $\mathcal{C}$, the following diagram (Figure 1) in category $\mathcal{D}$ must be commutative.

$$\begin{array}{ccc}
 X & \xrightarrow{m} & Y \\
 \\
 F(X) & \xrightarrow{\tau_X} & G(X) \\
 F(m) \downarrow & & \downarrow G(m) \\
 F(Y) & \xrightarrow{\tau_Y} & G(Y)
 \end{array}$$

Fig. 1. Naturality Square Diagram

Intuitively, the naturality condition signifies that the order of applying the functor and the transformation does not affect the result; both paths in the diagram yield the same outcome:

- Path 1 ($\tau_Y \circ F(m)$): First apply functor F to relation m (from $F(X)$ to $F(Y)$), and then the transformation component τ_Y (going from $F(Y)$ to $G(Y)$).
- Path 2 ($G(m) \circ \tau_X$): First apply the transformation component τ_X (from $F(X)$ to $G(X)$), and then functor G to relation m (from $G(X)$ to $G(Y)$).

If these two paths are always equal, the transformation τ is “natural”. This means that τ is not just a random collection of relations, but a coherent process that respects the structure defined by functors F and G .

Natural Isomorphism Distinct from an isomorphism between individual objects, a natural isomorphism establishes a structural equivalence between functors. Formally, let F and G be functors between categories $\mathcal{C}$ and $\mathcal{D}$. A natural transformation $\eta : F \Rightarrow G$ is defined as a natural isomorphism if, for every object X in $\mathcal{C}$, the component morphism

$\eta_X : F(X) \rightarrow G(X)$ constitutes an isomorphism within $\mathcal{D}$. This condition implies that the functors F and G are isomorphic as objects in the functor category $[C, D]$, meaning they represent C in D in the same way up to isomorphism. Consequently, constructions and morphisms expressed via F correspond uniquely and functorially to those expressed via G . As a computational analogy, a natural isomorphism may relate a functor that represents data as serialized JSON to one that represents the same data as an in-memory Map, provided that the conversion is bijective, structure-preserving, and compatible with all data transformations (i.e., commutes with the corresponding morphisms).

Adjunction An adjunction $I \dashv J$ is a pair of functors $I : \mathcal{C} \rightleftarrows \mathcal{D} : J$ with a natural isomorphism:

$$\text{Hom}_{\mathcal{D}}(I(X), Y) \cong \text{Hom}_{\mathcal{C}}(X, J(Y)). \quad (1)$$

The natural transformations $\eta : \text{Id}_{\mathcal{C}} \Rightarrow JI$ (unit) and $\varepsilon : IJ \Rightarrow \text{Id}_{\mathcal{D}}$ (counit) exist and satisfy the triangle identities as a direct consequence of the above natural isomorphism.

4.2. Research Hypothesis

Based on the ontological and categorical concepts introduced above, the fundamental hypothesis of this research is that ontological modeling and category theory provide a rigorous apparatus for formalizing the process of ontology-driven code generation in a way that (i) makes schema-level information preservation a verifiable property rather than a design claim, (ii) supports multi-target generation through parametrization rather than code duplication, and (iii) cleanly separates the domain model from its platform-specific realization.

To prove our hypothesis, we formalize the ODCG process as a sequence of three categories connected by two functors:

$$C_{\text{Core}} \xrightarrow{F_s} C_{\text{BAB}}^s \xrightarrow{H_s} C_{\text{code}}^s \quad (2)$$

where C_{Core} is the schema category, constructed from the BAB Core ontology, C_{BAB}^s is the platform-specific schema category for technology stack s , and C_{code}^s is the category of generated source code artifacts. For each stack s , functor F_s performs the model-to-model (M2M) transformation from platform-independent to platform-specific representation, while functor H_s performs the model-to-code (M2T) transformation from the platform-specific schema to concrete source code files.

The functor F_s is further represented through an evaluation bifunctor $\{F_s\}_{s \in S}$ parametrized by the target technology stack $s \in S$, with each F_s arising from a conformance functor Φ_s that interprets abstract mapping schema elements as platform-specific generation rules. This decomposition captures the fact that code generation depends functorially on two independent inputs, the domain model and the mapping class, and that multi-target generation is achieved by varying the stack index s without modifying the domain model or the generation architecture.

The correctness of the overall transformation is formalized through structural properties of the composite functor $T_s = H_s \circ F_s$: totality, structural coherence, faithfulness, and injectivity on objects. Together, these properties constitute a conditional schema-level

guarantee that, under the stated verification assumptions, the selected structural information and explicitly represented mapping distinctions are not lost or collapsed during transformation to generated code.

The categorical framework suggests a possible extension to round-trip engineering through the introduction of an adjunction $I_s \dashv J_s$ between a forward generation functor $I_s : C_{Core} \rightarrow C_{code}^s$ and a reverse abstraction functor $J_s : C_{code}^s \rightarrow C_{Core}$. Establishing such an adjunction, together with suitable semantic conditions, could provide a formal basis for consistent generation and recovery. This extension is outside the scope of the present work and is discussed in Section 8 as a direction for future research.

5. Formal Model of the ODCG Process

The proposed methodology formalizes the ODCG process as a multi-phase, deterministic transformation based on three levels of abstraction: a Platform Independent Model (PIM), a Platform Specific Model (PSM), and the final generation of executable code. The methodology encompasses three key artifacts: BAB Core ontology as the PIM, the BABj Java library and ontology as the target architecture (PSM), and the model transformer as the mechanism executing the transformations (Figure 2). This division ensures that each phase has clearly defined semantics and formal verification that models obtained by transformation preserve the structural properties of the source model.

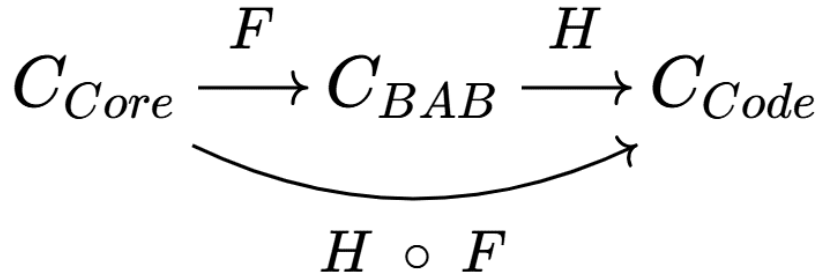


Fig. 2. Overview of the multi-phase transformation from ontology to code

The presentation of this section proceeds in two passes. The first three subsections describe the ontological layer of the methodology, the three artifacts as OWL 2 DL ontologies and as generated source files, without invoking any categorical apparatus. This pass is self-contained and readable by those interested in the ontology engineering aspects of the methodology. The subsequent subsection introduces the categorical layer, constructing schema categories from the above-mentioned ontologies and formalizing the transformations between them as functors. The categorical formalism is developed in one place, after the reader is familiar with all three ontological artifacts, rather than interleaved with the ontology descriptions. This separation is deliberate so that readers with different backgrounds can navigate the two layers independently.

5.1. Level 1: Platform Independent Model (PIM) – BAB Core Ontology

The first and highest level of abstraction is the PIM, realized as the BAB Core ontology, a semantic framework for modeling PAIS. Unlike standard domain ontologies that classify data, BAB Core is designed as an ontology of a software system structure: it does not describe a specific business domain but abstracts the fundamental building blocks of any information system, establishing a strict separation between business semantics and technological implementation.

BAB Core ontology (Figure 3) is a formal meta-model of an information system defined using OWL 2 DL (Web Ontology Language, Description Logic profile). It provides the semantic skeleton on which domain-specific applications are constructed through ontology import and specialization.

Core classes The root of the BAB Core taxonomy consists of the following classes, each modeling an essential aspect of application architecture:

- `core:Entity`: Represents the basic unit of persistence and data structure (equivalent to an entity in the ER model or a class in OOP).
- `core:Process` and `core:Task`: Model the dynamic behavior of the system, where processes are decomposed into atomic units of work (e.g., `core:UserTask` for human interactions and `core:ServiceTask` for automated operations).
- `core:UIElement`: Abstracts the presentation layer through concepts such as `core:View` and `core:Form`, enabling a declarative description of the user interface independent of a concrete frontend framework (e.g., Angular, React, or Vaadin).
- `core:Service` and `core:Module`: Define the logical organization of business logic and application granularity.

Core properties Object properties in BAB Core formalize structural and behavioral relations among these classes. System composition is defined by relations such as `core:hasAttribute` (entity structure), `core:usesEntity` (service-to-data dependency), and `core:hasTask` (process decomposition), while the workflow order is formalized by `core:precedes`. These relations impose architectural constraints on any domain ontology built on the top of the BAB Core, ensuring that every instantiated model possesses structural integrity before any transformation takes place.

The ImplementationMapping mechanism A key design decision in BAB Core is that code generation rules are not hidden in a generator algorithm but are explicitly modeled within the ontology itself. This is achieved by reifying generation rules as an OWL class `core:ImplementationMapping`, rather than as an OWL object property. The reification is deliberate and has modeling consequences:

- A property, by its nature, has only a subject and an object, it can relate a domain concept to a target element, but it can't itself carry structured information about how the relation is realized on a specific platform.

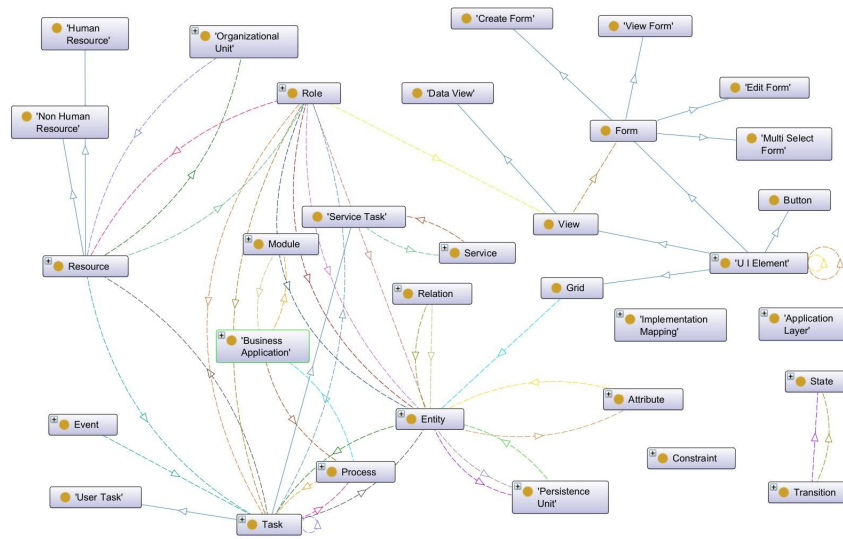


Fig. 3. BAB Core Ontology Diagram

- A class, on the other hand, admits a hierarchy of subclasses through `rdfs:subClassOf`, allowing progressive refinement of mapping strategies: specific mapping classes such as `core:EntityMapping`, `core:ViewMapping`, `core:ProcessMapping`, and `core:TaskMapping` are declared as subclasses of `core:ImplementationMapping`, and further refinements (e.g., `core:PersistentEntityMapping` extending `core:EntityMapping`, or `core:ValidatableEntityMapping` for entities requiring validation logic) may be introduced to specialize generation behavior for particular architectural patterns. Each of these classes can then be instantiated into named individuals in stack-specific ontologies (such as `babj:DefaultEntityMapping_J` in the BABj ontology), where each such individual carries the concrete platform-specific bindings, the target library class, default annotations, and auxiliary properties, needed for code generation.
- The association between a domain concept and its mapping is expressed through a separate object property `core:hasImplementationMapping`, which connects each domain concept to one or more `ImplementationMapping` individuals governing distinct aspects of its platform realization (e.g., persistence, DTO projection for a specific view, UI binding).

This reified design makes the mapping mechanism first-class ontological content: refinements are expressible as subclass relations among mapping classes, inspection and querying of mapping strategies are possible through standard SPARQL queries, and the reasoning layer can verify mapping consistency as a part of ontology validation. The essential role of the refinement structure is developed in the Section 5.4.3, where it is shown to generate natural transformations between stack-specific generation functors.

Business domain specification A platform-independent model of the specific business domain is built by creating a domain-specific ontology (e.g., for a Postal Management System, PMS) that imports BAB Core (`owl:imports`) and specializes its classes. In this process, specific domain terms (e.g., `pms:Shipment`) become subclasses of abstract BAB Core constructs (`pms:Shipment rdfs:subClassOf core:Entity`). BAB Core thus acts as a semantic skeleton imposing architectural patterns and constraints, while the domain ontology "fills" that skeleton with specific business logic.

Axiomatic layer BAB Core further declares necessary OWL 2 DL axioms comprising inverse property declarations (e.g., `core:hasAttribute owl:inverseOf core:isAttributeOf`), subproperty hierarchies, and domain/range restrictions, which govern instance-level reasoning performed by a reasoner (e.g., Hermit) during ontology validation. These axioms are used to verify consistency of domain ontologies built on the top of BAB Core but play no role in the schema-level categorical transformation developed in Section 5.4. The formal separation between instance-level reasoning and schema-level transformation is returned to in Section 5.4.1.

The formal categorical representation of this ontology and the mechanism that enables technology-neutral code generation are presented in Section 5.4.

5.2. Level 2: Platform Specific Model (PSM) – BABj Library and Ontology

The second level of abstraction, the PSM, is exemplified as the BABj Java library and the BABj ontology. Together they define the target architecture into which domain models are transformed. The BABj ontology is constructed on the top of the Web of Code (WoC) vocabulary [5], which provides a standard OWL 2 DL representation of object-oriented programming language constructs, and is extended with artifacts from the BABj library specific to the BAB framework.

The BABj Library The BABj library provides infrastructural scaffolding on which generated code depends. Its elements are divided into two kinds:

- Structural elements: base classes such as `AbstractEntity` and `AbstractHome`, which provide infrastructural mechanisms for persistence, session handling, and transaction management.
- Behavioral elements: interfaces defining business logic contracts, such as `CheckValidateEntityInterface`, `CheckArrivedEntityInterface`, and `CheckPayoutEntityInterface`, which are implemented by generated code to inject domain-specific logic like validation, arrival, and payout.

The generated code relies on these library elements through Java inheritance and interface realization. This division ensures that generated classes remain thin and focused on domain logic, while all boilerplate and infrastructure are delegated to the library.

The BABj Ontology The BABj ontology represents, in OWL 2 DL, the structural vocabulary that generated code will instantiate. It imports the WoC vocabulary and extends it with BABj-specific library classes and mapping declarations. Its content consists of:

- Imported WoC constructs, representing generic Java programming-language elements: `woc:Class`, `woc:Interface`, `woc:Field`, `woc:Method`, `woc:hasField`, `woc:hasMethod`, `woc:extends`, `woc:implements`, and so forth.
- BABj library classes, added as OWL classes referring to the concrete Java types in the BABj library (e.g., `AbstractEntity`, `AbstractHome`, the validation interfaces), enabling the ontology to express that a generated class extends or implements a specific library element.
- Mapping individuals, declared as instances of specific refinement classes from the `core:ImplementationMapping` hierarchy, which specify how each PIM concept is realized in the Java stack. Examples include `babj:DefaultEntityMapping_J` (an instance of `core:EntityMapping` or its appropriate refinement, specifying that `core:Entity` is realized as a `woc:Class` annotated with `@Entity` and extending `babj:AbstractEntity`), `babj:DefaultProcessMapping_J` (an instance of `core:ProcessMapping`), and so forth. Each individual provides concrete binding information via `core:mapsToTypeSpec` (pointing to the BABj class that realizes the mapping) and auxiliary annotation properties. The association between a mapping individual and its target stack is established by the PSM ontology in which the individual is declared (e.g., a mapping individual declared in BABj implicitly targets the Java stack).

These mapping individuals are the concrete, stack-specific realizations of the abstract mapping schema defined in BAB Core. Because they are declared as instances of `core:ImplementationMapping` and further refined through `rdfs:subClassOf` relationships, they inherit the structural discipline of the core mapping mechanism while providing platform-specific detail.

Axiomatic layer. Like BAB Core, the BABj ontology carries OWL 2 DL axioms, including WoC's inverse property declarations (`woc:hasField owl:inverseOf woc:isFieldOf`, etc.), functional property constraints, and subproperty hierarchies, which are used by reasoner for instance-level consistency checking. These axioms primarily belong to the reasoning layer and do not automatically enter the schema-level categorical apparatus of Section 5.4. Selected schema-level equations may nevertheless be introduced explicitly in the presented target category when required by the transformation semantics. This separation is discussed in detail in Section 5.4.1.

The same approach applies to other platforms by replacing constituents of the PSM with corresponding platform-specific ontologies/libraries.

Integration of Mapping Across the Two Ontologies A significant design decision in the methodology is that ontological mappings are not isolated as separate ontology but are distributed across BAB Core and BABj (i.e., platform-specific) components. BAB Core declares the abstract `ImplementationMapping` class and its core refinement hierarchy, while BABj instantiates this class with stack-specific individuals. This distributed design means that the reasoner can verify both halves, the abstract mapping schema and its concrete realization, as part of a single ontology consistency check.

In this integrated structure, BAB Core defines the meaning and identity of concepts, while BABj describes their realization through programming constructs and annotations. The reasoner verifies that the mapping individuals in BABj conform to the abstract schema in BAB Core, ensuring ontological consistency between the abstract schema and its

platform-specific realization. AI-assisted mapping may be employed at this level, where the transformer could propose likely implementations based on semantic similarity while a human (engineer) should confirm the proposal, maintaining formal control and process validity.

The formal representation of BABj as a category, and the formalization of the translation from BAB Core to BABj as a bifunctor, are developed in Section 5.4.

5.3. Level 3: Generated Code – Category C_{code}^s

The third and the final level of our MDA is a free category of generated code, C_{code}^s , constructed over directed graph that represents internal programming language structure. Its objects are generated source-artifact specifications, typically materialized as source files, while morphisms are structural dependency relations such as import, extends, implements, field-type dependencies, and method-signature dependencies. The generated code relies entirely on the corresponding (i.e., BABj) library, achieving architectural consistency and minimizing redundancy.

5.4. Transformation Process: $H_s \circ F_s$

This subsection presents the entire categorical formalism of the methodology. Having introduced the three ontological artifacts in previous sections, we now construct schema categories from these ontologies and formalize the transformations between them as functors. The subsection is organized as follows: first, we introduce the categorical constructions used in the formalism and record their well-definedness by construction; next, we define and verify the model-to-model functor F_s together with the ImplementationMapping mechanism that parametrizes it over technology stack s ; finally, we define the model-to-code functor H_s that completes the transformation pipeline.

The intermediate category C_{BAB}^s can be suppressed when the transformation is viewed only at the level of the composite mapping, because the model-to-model and model-to-code functors compose to a single functor $T_s = H_s \circ F_s$. Nevertheless, C_{BAB}^s remains essential to the chosen factorization, specification, and verification of that composite: it is at this level that the stack-parametrized bifunctor F_s , the presented-category relations E_{BAB}^s , and the refinement natural transformations are defined; and it is through C_{BAB}^s that T_s factors.

Construction of the three categories Before constructing the categories, we emphasize a distinction that is essential to the proposed formalism. The BAB Core and BABj/WoC ontologies contain two kinds of content relevant to the methodology. The structural schema layer consists of OWL classes, object properties, and the declarations used to extract the schema graph. This layer determines the objects and generating morphisms of the categories constructed below. The logical axiomatic layer consists of OWL 2 DL axioms, such as inverse-property declarations, functional and inverse-functional property declarations, transitivity, symmetry, and subproperty axioms. These axioms govern semantic constraints on ontology interpretations and are used by the OWL reasoner during validation.

The categorical formalism reflects these layers asymmetrically. The source category, representing the PIM layer, is constructed as the free category generated by its schema graph. This allows the model-to-model transformation to be defined on objects and generating edges, with no source-side equations to verify. For each target technology stack, the corresponding target category, representing the PSM layer for that stack, is constructed as a presented category [47]. Its generating graph is extracted from the stack-specific ontology and its imported code-structure vocabulary. Its relations form an explicitly selected set of schema-level path equations required by the transformation semantics of that stack. These equations may be motivated by the intended structural semantics of the target platform and by selected ontology declarations used as schema-normalization rules, but they should not be understood as automatic translations of arbitrary OWL 2 DL axioms into categorical equations. Consequently, the Java case is obtained as an instance based on the BABj ontology and the imported WoC vocabulary.

Category C_{Core} Let G_{Core} denote the directed multigraph underlying the schema layer of the BAB Core ontology. The nodes of G_{Core} are the OWL classes declared in BAB Core. The directed edges of G_{Core} are the object properties declared in BAB Core, each edge labeled by the property IRI, with the edge's source and target given by the property's domain and range declarations.

We define C_{Core} as the free category generated by G_{Core} :

- $Ob(C_{Core})$ is the set of nodes of G_{Core} .
- $Mor(C_{Core})$ consists of finite (possibly empty) sequences of composable edges of G_{Core} , where a sequence $(e_1, e_2, \dots, e_n)$ with $e_i : X_i \rightarrow X_{i+1}$ represents a morphism $X_1 \rightarrow X_{n+1}$. The empty sequence at X represents the identity morphism id_X .
- Composition of morphisms is concatenation of sequences, which is associative by construction of concatenation.
- Identities act neutrally: concatenation with an empty sequence leaves any sequence unchanged.

Construction 1. The BAB Core schema category C_{Core} .

The category C_{Core} is defined as the free category generated by the directed multigraph G_{Core} extracted from the schema-level layer of the BAB Core ontology. Hence, its objects are OWL classes, its generating morphisms are BAB Core object properties, and arbitrary morphisms are finite composable paths of such properties. Identities are empty paths and composition is given by path concatenation. Therefore, C_{Core} is a well-defined category by the standard free-category construction.

Decomposition of C_{Core} . The object set of C_{Core} splits into two disjoint subsets:

- $Ob(C_{Core}^{NonMapping}) = \{X \in Ob(C_{Core}) \mid X \text{ is not a subclass of } core : ImplementationMapping\}$ containing all domain concepts.
- $Ob(C_{Core}^{MappingOnly}) = \{X \in Ob(C_{Core}) \mid X \text{ is } core:ImplementationMapping \text{ or one of its subclasses}\}$ containing the reified mapping elements classes.

These object sets are disjoint by construction, and their union exhausts $Ob(C_{Core})$.

Let $C_{Core}^{NonMapping}$ and $C_{Core}^{MappingOnly}$ be the full subcategories of C_{Core} determined by the corresponding object sets. Each subcategory contains all morphisms of C_{Core} between objects in its own object set, and the category axioms are inherited automatically. Since the cross-category generating morphisms induced by `core:hasImplementationMapping` are oriented from NonMapping objects to MappingOnly objects, and no generating edge in G_{Core} is oriented from a MappingOnly object back to a NonMapping object, no path in C_{Core} between two NonMapping objects can pass through a MappingOnly object. Hence every such path lies entirely in the induced graph $G_{Core}^{NonMapping}$, and $C_{Core}^{NonMapping}$ is the free category generated by $G_{Core}^{NonMapping}$. The same argument applies to $C_{Core}^{MappingOnly}$: no path between two MappingOnly objects leaves the MappingOnly part of the graph. Therefore $C_{Core}^{MappingOnly}$ is also freely generated by its induced graph.

Observation 1. Decomposition of C_{Core} .

The object set of C_{Core} decomposes into domain-level concepts and reified mapping concepts. These determine two full subcategories, $C_{Core}^{NonMapping}$ and $C_{Core}^{MappingOnly}$. Morphisms generated by `core:hasImplementationMapping` connect objects from the former to objects from the latter; therefore, they belong to C_{Core} but not to either full subcategory.

The significance of Observation 1 is that it justifies decomposing C_{Core} 's object set into the disjoint union $Ob(C_{Core}^{NonMapping}) \sqcup Ob(C_{Core}^{MappingOnly})$ while still recognizing that the cross-category morphisms (generated by `core:hasImplementationMapping`) exist in C_{Core} . These cross-category morphisms do not belong to either full subcategory and therefore do not enter the product category $C_{Core}^{NonMapping} \times C_{Map}$ that serves as the domain of the bifunctor in Section 5.4.3. Their role is semantical: they specify which pairs (D, M) carry intended architectural meaning, as developed in Section 5.4.3. The schema-level edge in G_{Core} corresponding to `core:hasImplementationMapping` is extracted from its declared `rdfs:domain` and `rdfs:range`: domain classes on the left, the abstract `core:ImplementationMapping` class (or its declared subclasses) on the right. The instance-level assertions found in the operational ontology are ABox content that provides the data from which the schema-level pair (D, M) is recovered: D is the domain class subject of the assertion, and M is the `rdf:type` of the asserted mapping individual. The schema-level cross-edge thus exists structurally in C_{Core} , while the ABox assertions populate the set of selected pairs that carry intended architectural meaning.

Mapping-refinement category C_{Map} Although $C_{Core}^{MappingOnly}$ identifies the mapping-related objects inside C_{Core} , refinements between mapping strategies are governed by the `rdfs : subclassOf` hierarchy of `ImplementationMapping` classes rather than by ordinary BAB Core object properties. Therefore, we introduce a separate mapping-refinement category C_{Map} . Its objects are `core:ImplementationMapping` and its subclasses, and its morphisms are generated by the refinement order induced by the `rdfs:subclassOf` hierarchy. The arrows are oriented from a more general mapping class to a more specific one: a morphism $h : M \rightarrow M'$ in C_{Map} means that M' is a refinement, i.e., a subclass-specialization, of M . This category is used as the second argument of the conformance and evaluation constructions below. Formally, C_{Map} is the preorder category of mapping refinements. There is at most one morphism $M \rightarrow M'$, and such

a morphism exists exactly when M' is a subclass-specialization of M , that is, when $M' \text{ rdfs} : \text{subClassOf}^* M$. Identities and composition follow from reflexivity and transitivity respectively.

Category C_{BAB}^s Let s denote an arbitrary target technology stack. Let O_{BAB}^s denote the stack-specific ontology for s , and let V^s denote the code-structure vocabulary imported by that ontology. We define G_{BAB}^s as the directed multigraph extracted from the schema layer of O_{BAB}^s together with V^s . The graph G_{BAB}^s represents the stack-specific realization signature, not a concrete generated application model. Its nodes are stack-specific realization types and realization patterns. A realization pattern specifies how a BAB Core concept of a given kind, together with a selected mapping class, can be represented in the target stack. Such patterns include platform-level type specifications, metadata or annotations, inheritance or composition constraints, interface or contract declarations, template roles, imports, fields, methods, and other structural parameters used by the generator. The directed edges of G_{BAB}^s are selected schema-level relations used as structural navigation and refinement relations inside the target realization signature. In particular, G_{BAB}^s provides admissible platform-level relation labels and refinement relations. Concrete domain concepts are not part of the definition of G_{BAB}^s ; they are supplied later, when the conformance functor $\Phi_s(M)$ applies these stack-specific realization patterns to objects of $C_{Core}^{NonMapping}$.

Let $Free(G_{BAB}^s)$ be a free category generated by G_{BAB}^s . Let E_{BAB}^s be an explicitly specified set of pairs (u, v) of parallel paths in $Free(G_{BAB}^s)$, where $u, v : X \rightarrow Y$. They may include: (i) conformance equations, such as the naturality equations introduced in Section 5.4.3; (ii) refinement-coherence equations, such as equations identifying a selected direct refinement edge with the corresponding composite refinement path; and (iii) explicitly intended normalization equations, where the target stack deliberately treats two alternative structural paths as the same realization pattern.

The third case is optional and must be declared explicitly as part of the stack-specific transformation specification. Such normalization equations are not introduced automatically and do not express arbitrary semantic equivalence of target-language programs. They only state that, for the purposes of the selected schema-level transformation, two parallel structural paths are intentionally treated as the same target-side realization.

Let $\equiv_E$ be the smallest congruence on $Free(G_{BAB}^s)$ that contains E_{BAB}^s and is compatible with composition. We define $C_{BAB}^s = Free(G_{BAB}^s) / \equiv_E$. Then, the objects of C_{BAB}^s are the objects of G_{BAB}^s , while morphisms are equivalence classes $[u]$ of paths in $Free(G_{BAB}^s)$. Composition is defined by $[v] \circ [u] = [v \circ u]$, which is well-defined because $\equiv_E$ is a congruence.

Equivalently, C_{BAB}^s is the category presented by $\langle G_{BAB}^s \mid E_{BAB}^s \rangle$. The set E_{BAB}^s is not obtained by mechanically translating all OWL 2 DL axioms into categorical equations. Rather, it is an explicitly selected set of schema-level coherence equations required by the transformation. OWL declarations such as `owl:inverseOf`, `owl:FunctionalProperty`, `owl:InverseFunctionalProperty`, and `rdfs:subPropertyOf` remain primarily part of the ontology reasoning layer. They may motivate selected equations in E_{BAB}^s only when the transformation explicitly treats the corresponding structural paths as equivalent.

When explicitly intended normalization equations are included in E_{BAB}^s , the faithfulness statement in Proposition 2 is understood modulo those declared normalizations. If no

such normalization equations are included, the corresponding faithfulness claim is strict on the selected schema fragment.

Construction 2. The stack-specific schema category C_{BAB}^s .

For a fixed target stack s , let $Free(G_{BAB}^s)$ denote the free category generated by G_{BAB}^s . We define C_{BAB}^s as the quotient of $Free(G_{BAB}^s)$ by the smallest congruence generated by E_{BAB}^s . Its objects are stack-specific realization types and realization patterns, while its morphisms are equivalence classes of finite composable paths generated by the selected structural, navigation, and refinement relations of G_{BAB}^s . Since C_{BAB}^s is determined by the presentation $\langle G_{BAB}^s \mid E_{BAB}^s \rangle$, it is a well-defined category. The construction of presented categories from generators and relations is standard and verifies the category axioms automatically.

In the case of concrete Java instance ($s = j$), O_{BAB}^j is the BABj ontology and V^j is the imported WoC vocabulary. Therefore, G_{BAB}^j is a directed multigraph whose nodes are Java realization types and realization patterns constructed from the OWL classes declared in BABj or WoC, including generic code-structure concepts such as `woc:Class`, `woc:Interface`, `woc:Field`, and `woc:Method`, BABj-specific library classes, and the generation profiles attached to mapping individuals. Its selected structural edges include the code-structure relations used by the generator, such as `woc:hasField`, `woc:hasMethod`, `woc:extends`, and `woc:implements`. The relation `woc:extends` is retained as the ordinary object-oriented inheritance relation. Mapping refinement, however, is not identified with `woc:extends`; it is represented by explicitly selected generation-refinement edges, denoted here as `babj:refinedBy`, oriented from the more general realization specification to the more specific one. The category C_{BAB}^j is then the presented category $\langle G_{BAB}^j \mid E_{BAB}^j \rangle$. The remainder of this section develops the formalism for an arbitrary but fixed stack $s \in S$, while the Java instance $s = j$ is used as the running example. The defining equations of E_{BAB}^s are formulated as equations between parallel paths in the unquotiented free category $Free(G_{BAB}^s)$. The graph-level assignment of realization patterns and generating relation labels is specified before the quotient is formed. Hence the naturality equations are well-formed at the free-category level, and the corresponding functors and natural transformations descend to C_{BAB}^s by the universal property of the quotient. Therefore, the use of the presented category C_{BAB}^s is not circular: the quotient records exactly those schema-level path identifications that the transformation explicitly requires.

In the Java instance, the BABj and WoC vocabularies declare several OWL 2 DL axioms, including inverse-property, functional, inverse-functional, and subproperty declarations. These axioms are used by the ontology reasoning layer for validation and inference. They do not automatically generate equations in C_{BAB}^j . Only those schema-level path equations that are explicitly required by the Java transformation semantics, such as conformance and refinement equations introduced below, are included in E_{BAB}^j .

Category C_{code}^s For a fixed target stack s , let G_{code}^s denote the directed multigraph whose nodes are generated source-artifact specifications for that stack and whose edges are explicit structural dependencies induced by the generated code structure. We define C_{code}^s as the free category generated by G_{code}^s ; its morphisms are finite paths of structural dependencies. Because G_{code}^s depends on the output of the generator for a specific domain ontology and stack, C_{code}^s is not a fixed structure but varies with the input. An object of C_{code}^s is understood as a generated source-artifact specification, not merely as a physical

file path. Its primary operational identifier is the target source file or fully qualified name. However, in the formal category, the object is identified as a complete artifact specification: the artifact role, the fully qualified name, and the generated structural content, including superclass declarations, implemented interfaces, annotations, fields, methods, imports, and other code-level features introduced by the selected mapping strategy. Therefore, C_{Code}^s does not quotient generated code by target-language semantic equivalence. Two generated artifacts with different fully qualified names are distinct. If two realization paths attempt to produce the same fully qualified name with different structural content, this is not treated as an equality or normalization in C_{Code}^s , but as a verification error. Only a single consistent structural specification may be admitted for a given generated artifact identifier.

Model-to-Model Transformation F_s for stack s For each target technology stack s , the model-to-model (M2M) transformation is formalized as a bifunctor $F_s : C_{Core}^{NonMapping} \times C_{Map} \longrightarrow C_{BAB}^s$ that maps each pair (D, M) , consisting of a domain concept D and a mapping class M , to the stack-specific realization specification $F_s(D, M)$ in C_{BAB}^s . The two arguments are independent: D determines “what” is being transformed, while M determines “how” it is realized on the target platform. Different choices of M for the same D may yield structurally distinct artifacts. For the Java instance $s = j$, the bifunctor F_j is implemented as a SPARQL-driven extraction that reads the BAB Core ontology together with its core:hasImplementationMapping bindings and constructs an in-memory DataModel representation in the vocabulary of BABj.

The construction of F_s , including the conformance functor Φ_s that captures how mapping refinements act uniformly on the domain and the proof that F_s is a well-defined bifunctor, is presented in the following subsection.

Formalization of the ImplementationMapping mechanism The technological neutrality of BAB Core rests on the core:ImplementationMapping mechanism: the explicit ontological encoding of generation rules within the model itself. This subsection provides the formal categorical basis for this mechanism, presenting a family of evaluation bifunctors $\{F_s\}_{s \in S}$, one for each supported technology stack.

We want to express code generation as a function of two independent inputs: the domain model (an object of $C_{Core}^{NonMapping}$) and the mapping class (an object of C_{Map}). Each input may also change, the domain may be refined, and the mapping class may be refined, and we want these changes to compose functorially with the generation process. Specifically, we seek a bifunctor:

$$F_s : C_{Core}^{NonMapping} \times C_{Map} \longrightarrow C_{BAB}^s \quad (3)$$

The existence of such a bifunctor, and its well-definedness on morphisms of the product category, are the content of the proposition below. The construction proceeds by introducing an intermediate object, the conformance functor Φ_s , whose functoriality captures the essential idea that refinements of the mapping class act uniformly on the domain, independently of which specific domain element is being transformed.

The Conformance Functor Φ_s For each technology stack $s \in S$, the stack-specific ontology O_{BAB}^s supplies, for every mapping class $M \in Ob(C_{Map})$, a designated named individual $\mu_s(M) \in O_{BAB}^s$ that is asserted to be an instance of M and that carries the concrete platform-specific bindings on which code generation depends, including the target type specification (core:mapsToTypeSpec), annotations, implemented interfaces, superclass declarations, and auxiliary generation parameters. We refer to $\mu_s(M)$ as the realization individual of M for stack s . For the Java instance ($s = j$), the realization assignment is exemplified by

$$\mu_j(\text{core:EntityMapping}) = \text{bobj:DefaultEntityMapping_J} \quad (4)$$

$$\mu_j(\text{core:ValidatableEntityMapping}) = \text{bobj:DefaultValidatableEntityMapping_J} \quad (5)$$

and analogously for the remaining mapping classes. The assignment μ_s is part of O_{BAB}^s and is fixed once the target stack is chosen. The conformance functor Φ_s defined below reads its bindings from $\mu_s(M)$, never from M itself; the schema-level category C_{Map} is thereby kept stack-independent, while all stack-specific resolution is localized in μ_s and Φ_s .

With this realization assignment fixed, we define the conformance functor as

$$\Phi_s : C_{Map} \longrightarrow [C_{Core}^{NonMapping}, C_{BAB}^s] \quad (6)$$

where $[C_{Core}^{NonMapping}, C_{BAB}^s]$ denotes the functor category whose objects are functors $C_{Core}^{NonMapping} \rightarrow C_{BAB}^s$ and whose morphisms are natural transformations between such functors.

Before defining the conformance functor, we make explicit how the stack-specific ontology O_{BAB}^s enters the construction. For a fixed stack s , ontology O_{BAB}^s together with its imported code-structure vocabulary V_s , provides: (i) realization individuals $\mu_s(M)$ for mapping classes M , (ii) stack-specific realization profiles carried by these individuals, (iii) the assignment of selected BAB Core schema relations to admissible platform-level structural relation labels, and (iv) refinement morphisms between realization profiles induced by mapping refinements. These data define the target realization signature used by the transformation. The domain concepts themselves are not defined by O_{BAB}^s ; they enter only as arguments of the functors $\Phi_s(M)$.

The presented target category $C_{BAB}^s = Free(G_{BAB}^s) / \equiv_E$ is obtained by quotienting $Free(G_{BAB}^s)$ by the congruence generated by a relation set E_{BAB}^s . The equations in E_{BAB}^s are not arbitrary: among them are the path equalities required to make the component families assigned to mapping refinements natural transformations. Thus, the quotient construction of C_{BAB}^s is the categorical mechanism that enforces the naturality of the conformance transformation.

It is important to distinguish this formal role from empirical validation of the generator. The naturality equations are part of the categorical specification of the target presentation; they do not by themselves constitute an independent test of the implementation. The implementation must still be checked against this specification by the verification

conditions stated in Section 5.6 (mapping completeness, mapping injectivity, template determinism, quotient compatibility, no unintended collapse).

On objects of C_{Map} , for each mapping class M , the conformance functor assigns a functor

$$\Phi_s(M) : C_{Core}^{NonMapping} \rightarrow C_{BAB}^s.$$

For a fixed stack s and mapping class M , let $\mu_s(M)$ denote the realization individual declared in O_{BAB}^s . This individual does not denote a domain concept. Rather, it provides the stack-specific realization profile for M . This profile contains the generation-relevant bindings needed by the target stack, such as target type specifications, structural roles, metadata or annotations, inheritance or composition constraints, interface or contract declarations, template roles, and auxiliary generation parameters.

For each domain concept $D \in Ob(C_{Core}^{NonMapping})$, the value $\Phi_s(M)(D)$ is the stack-specific realization specification obtained by applying the realization profile $\mu_s(M)$ to the domain concept D . This specification combines the identity of the domain concept D with the stack-specific bindings read from $\mu_s(M)$. Different choices of M may yield different realization specifications for the same domain concept D , which is what makes $\Phi_s(M)$ genuinely depend on M .

It remains to define $\Phi_s(M)$ on generating morphisms. Since $C_{Core}^{NonMapping}$ is freely generated by $G_{Core}^{NonMapping}$, it is sufficient to define the image of each generating edge. The stack-specific ontology O_{BAB}^s , together with its imported code-structure vocabulary V_s , determines a relation-label assignment

$\lambda_s : Edge(G_{Core}^{NonMapping}) \rightarrow Lab(G_{BAB}^s)$, where $Lab(G_{BAB}^s)$ denotes the set of admissible platform-level structural relation labels. The value $\lambda_s(e)$ is not itself a concrete edge; it is the platform-level relation label used in the target realization signature. Let $e : D \rightarrow D'$ be a generating edge of $G_{Core}^{NonMapping}$. We define $\Phi_s(M)(e) : \Phi_s(M)(D) \rightarrow \Phi_s(M)(D')$ as the morphism in C_{BAB}^s represented by the platform-level structural relation whose label is $\lambda_s(e)$ with source $\Phi_s(M)(D)$ and target $\Phi_s(M)(D')$. The assignment λ_s is independent of M , while the source and target realization specifications depend on M through $\mu_s(M)$.

For an arbitrary morphism $f = e_n \circ \dots \circ e_1$ in $C_{Core}^{NonMapping}$, we define

$$\Phi_s(M)(f) = \Phi_s(M)(e_n) \circ \dots \circ \Phi_s(M)(e_1),$$

and identities are sent to identities. Functoriality of $\Phi_s(M)$ follows from the universal property of the free category $C_{Core}^{NonMapping}$. The object assignment is given by $D \mapsto \Phi_s(M)(D)$, while the edge assignment is determined by λ_s on selected generating edges, with sources and targets fixed by object assignment above. Since λ_s is assumed to be defined and domain/range-compatible on the selected generating edges, this assignment extends uniquely to a functor $\Phi_s(M) : C_{Core}^{NonMapping} \rightarrow C_{BAB}^s$.

On morphisms of C_{Map} , we work with the morphisms generated by the `rdfs:subClassOf` relations declared among mapping classes in BAB Core. We refer to each such declaration as refinement morphism. For each refinement $h : M \rightarrow M'$, the conformance functor assigns a natural transformation

$$\Phi_s(h) : \Phi_s(M) \Rightarrow \Phi_s(M') \quad (7)$$

whose component at each domain concept D from $Ob(C_{Core}^{NonMapping})$ is a morphism

$$\Phi_s(h)_D = \rho_s(D, h) : \Phi_s(M)(D) \rightarrow \Phi_s(M')(D) \quad (8)$$

in C_{BAB}^s . The morphism $\rho_s(D, h)$ represents the stack-level refinement from the realization specification obtained from M to the realization specification obtained from M' , both applied to the same domain concept D . At the graph level, it is either a selected generating refinement edge of G_{BAB}^s or a finite path of selected refinement edges; in C_{BAB}^s , it is taken as the corresponding equivalence class of paths.

For every refinement $h : M \rightarrow M'$, the family $\Phi_s(h) = (\rho_s(D, h))_{D \in Ob(C_{Core}^{NonMapping})}$ is the candidate component family of the natural transformation $\Phi_s(h)$.

To be a natural transformation, this family must satisfy the naturality condition. For every generating edge

$e : D \rightarrow D'$ of $G_{Core}^{NonMapping}$, the relation set E_{BAB}^s contains the naturality equation

$$C_{BAB}^s : \rho_s(D', h) \circ \Phi_s(M)(e) = \Phi_s(M')(e) \circ \rho_s(D, h) \quad (9)$$

whenever both sides are defined as parallel paths in $Free(G_{BAB}^s)$. After quotienting by $\equiv_E$, these two paths become equal in C_{BAB}^s . Since $C_{Core}^{NonMapping}$ is freely generated by $G_{Core}^{NonMapping}$, imposing these equations on generating edges is sufficient. Naturality for arbitrary morphisms follows by induction on path length and by compatibility of the congruence $\equiv_E$ with composition.

In implementation terms, the morphism $\rho_s(D, h)$ records the generation-relevant refinement from the realization profile of M to the realization profile of M' . Such a refinement may correspond to changes in metadata, structural roles, inheritance or composition constraints, interface or contract declarations, template roles, or other generation parameters. However, categorically, only the existence of the morphism $\rho_s(D, h)$ and the commutativity of the naturality square are used.

Figure 4 illustrates this naturality condition for a morphism $f : D \rightarrow D'$ in $C_{Core}^{NonMapping}$.

$$\Phi_s(h)_{D'} \circ \Phi_s(M)(f) = \Phi_s(M')(f) \circ \Phi_s(h)_D \quad (10)$$

$$\begin{array}{ccc} \Phi_s(M)(D) & \xrightarrow{\Phi_s(M)(f)} & \Phi_s(M)(D') \\ \Phi_s(h)_D \downarrow & & \downarrow \Phi_s(h)_{D'} \\ \Phi_s(M')(D) & \xrightarrow{\Phi_s(M')(f)} & \Phi_s(M')(D') \end{array}$$

Fig. 4. Naturality square for $\Phi_s(h) : \Phi_s(M) \Rightarrow \Phi_s(M')$ at the morphism $f : D \rightarrow D'$

Consider the refinement $h : \text{EntityMapping} \rightarrow \text{PersistentEntityMapping}$ in C_{Map} and the structural morphism $\text{hasAttribute} : \text{Entity} \rightarrow \text{Attribute}$ in $C_{Core}^{NonMapping}$. Under

both mapping strategies, `hasAttribute` is realized as `woc:hasField` in C_{BAB}^j , oriented from the realized class to the realized field, because field declarations are not altered by the addition of the `@Entity` annotation. The naturality square becomes:

$$\text{woc:hasField}_{\text{persistent}} \circ \text{refinedBy}_{\text{Entity}} = \text{refinedBy}_{\text{Attribute}} \circ \text{woc:hasField}_{\text{plain}} \quad (11)$$

The two sides are parallel paths in $\text{Free}(G_{BAB}^j)$ and are identified in C_{BAB}^j by the corresponding equation included in E_{BAB}^j . In the Java example, both paths correspond to a field declaration belonging to the generated JPA-annotated class.

Since $C_{\text{Core}}^{\text{NonMapping}}$ is freely generated by $G_{\text{Core}}^{\text{NonMapping}}$, it is sufficient to impose these equations on generating edges. Naturality for arbitrary morphisms follows by induction on path length and by compatibility of the congruence $\equiv_E$ with composition.

Two roles of the naturality square are worth noting explicitly, because they are both essential to the argument below:

- Role 1: Well-definedness of Φ_s on morphisms. The naturality square is precisely the condition that $\Phi_s(h)$ is a natural transformation in $[C_{\text{Core}}^{\text{NonMapping}}, C_{BAB}^s]$, which is what allows Φ_s to be defined as a functor from C_{Map} into the functor category.
- Role 2: Well-definedness of F_s on morphism pairs. A morphism pair $(f, h) : (D, M) \rightarrow (D', M')$ in the product category $C_{\text{Core}}^{\text{NonMapping}} \times C_{\text{Map}}$ admits two canonical factorizations: $(f, h) = (f, \text{id}_{M'}) \circ (\text{id}_D, h) = (\text{id}_{D'}, h) \circ (f, \text{id}_M)$. For the bifunctor F_s below to be well-defined on such morphism pairs, the two factorizations must yield the same morphism in C_{BAB}^s , which is exactly the naturality equation above.

Since C_{Map} is a preorder category, a refinement morphism $h : M \rightarrow M'$ is uniquely determined by its source and target mapping classes. Hence, to define Φ_s on morphisms of C_{Map} , it is enough to specify, for each comparable pair $M \leq M'$, the component family $\rho_s(D; M, M') : \Phi_s(M)(D) \rightarrow \Phi_s(M')(D)$ in C_{BAB}^s , natural in D . We write this family as $\Phi_s(h)_D = \rho_s(D; M, M')$. To make Φ_s a functor, these component families must satisfy the following coherence conditions:

- Identity: For $M = M'$, the refinement is the identity morphism $\text{id}_M : M \rightarrow M$. Its component is defined as $\rho_s(D; M, M) = \text{id}_{\Phi_s(M)(D)}$ for every D . Hence $\Phi_s(\text{id}_M) = \text{id}_{\Phi_s(M)}$.
- Composition: For refinements $h : M \rightarrow M'$ and $k : M' \rightarrow M''$, the composite $k \circ h$ is the unique refinement morphism $M \rightarrow M''$ in the preorder category C_{Map} . Its component is required to satisfy $\rho_s(D; M, M'') = \rho_s(D; M', M'') \circ \rho_s(D; M, M')$ in C_{BAB}^s ; then $\Phi_s(k \circ h)_D = \Phi_s(k)_D \circ \Phi_s(h)_D$ and $\Phi_s(k \circ h) = \Phi_s(k) \circ \Phi_s(h)$.

Hence Φ_s is a functor $C_{\text{Map}} \rightarrow [C_{\text{Core}}^{\text{NonMapping}}, C_{BAB}^s]$.

The evaluation bifunctor F_s We now define the family of evaluation bifunctors $\{F_s\}_{s \in S}$, with each member:

$$F_s : C_{\text{Core}}^{\text{NonMapping}} \times C_{\text{Map}} \longrightarrow C_{BAB}^s \quad (12)$$

On objects, for a pair (D, M) :

$$F_s(D, M) = \Phi_s(M)(D) \quad (13)$$

On morphism pairs $(f, h) : (D, M) \rightarrow (D', M')$:

$$F_s(f, h) = \Phi_s(h)_{D'} \circ \Phi_s(M)(f) \quad (14)$$

The cross-subcategory morphism `core:hasImplementationMapping`, which exists in C_{Core} but in neither $C_{Core}^{NonMapping}$ nor C_{Map} , does not belong to the domain of F_s . It plays the semantic role of specifying which pairs $(D, M) \in Ob(C_{Core}^{NonMapping} \times C_{Map})$ carry intended architectural meaning: each domain element D is linked to one or more `ImplementationMapping` elements $M_1, M_2, \dots$ via `core:hasImplementationMapping`, each governing a distinct aspect of D 's platform realization (persistence, DTO projection for a specific view, UI binding, etc.). The bifunctor F_s is evaluated independently for each such pair (D, M_i) , producing the corresponding platform-specific artifact. Formally, F_s is total on the full product category; architecturally, its semantic content is concentrated on pairs linked by `hasImplementationMapping`.

Proposition 1. For each $s \in S$, the mapping F_s defined above is a functor, hence a bifunctor in its two arguments.

Proof. Since $C_{Core}^{NonMapping}$, C_{Map} , and C_{BAB}^s are small categories (in the implemented transformation, they are generated from finite ontology fragments selected for generation: finite sets of OWL classes, object properties, mapping classes, mapping individuals, and stack-specific structural relations) the construction takes place in **Cat**, the category of small categories and functors between them. The category **Cat** is Cartesian closed [47], with exponentials given by functor categories. Hence, for small categories C , M , and B , morphisms $C \times M \rightarrow B$ correspond naturally to morphisms $M \rightarrow [C, B]$, where $[C, B]$ denotes the functor category from C to B .

In the present setting, this correspondence is realized by the evaluation functor

$$ev_s : [C_{Core}^{NonMapping}, C_{BAB}^s] \times C_{Core}^{NonMapping} \longrightarrow C_{BAB}^s. \quad (15)$$

On objects, it is defined by $ev_s(G, D) = G(D)$, where $G : C_{Core}^{NonMapping} \rightarrow C_{BAB}^s$ is a functor and D is an object of $C_{Core}^{NonMapping}$. On morphisms, for a natural transformation $\eta : G \Rightarrow G'$ and a morphism $f : D \rightarrow D'$ in $C_{Core}^{NonMapping}$, the evaluation functor is defined by $ev_s(\eta, f) = \eta_{D'} \circ G(f)$.

Equivalently, by the naturality of η , we have $\eta_{D'} \circ G(f) = G'(f) \circ \eta_D$. Therefore, $ev_s(\eta, f)$ is a well-defined morphism $G(D) \rightarrow G'(D')$ in C_{BAB}^s .

We now express the bifunctor F_s through the conformance functor $\Phi_s : C_{Map} \rightarrow [C_{Core}^{NonMapping}, C_{BAB}^s]$. Let $\pi_1 : C_{Core}^{NonMapping} \times C_{Map} \rightarrow C_{Core}^{NonMapping}$ and $\pi_2 : C_{Core}^{NonMapping} \times C_{Map} \rightarrow C_{Map}$ be the two projection functors. Then F_s factorizes as $F_s = ev_s \circ \langle \Phi_s \circ \pi_2, \pi_1 \rangle$.

Indeed, on objects (D, M) , we obtain

$$F_s(D, M) = ev_s((\Phi_s \circ \pi_2)(D, M), \pi_1(D, M)) = ev_s(\Phi_s(M), D) = \Phi_s(M)(D). \quad (16)$$

Thus, $F_s(D, M)$ is precisely the platform-specific realization of the domain concept D under the mapping strategy M .

On morphism pairs $(f, h) : (D, M) \rightarrow (D', M')$ in the product category $C_{Core}^{NonMapping} \times C_{Map}$ where $f : D \rightarrow D'$ is a morphism in $C_{Core}^{NonMapping}$, and $h : M \rightarrow M'$ is a refinement morphism in C_{Map} , we have $(\Phi_s \circ \pi_2)(f, h) = \Phi_s(h)$ and $\pi_1(f, h) = f$. Therefore, $F_s(f, h) = ev_s(\Phi_s(h), f)$.

Since $\Phi_s(h) : \Phi_s(M) \Rightarrow \Phi_s(M')$ is a natural transformation, the evaluation functor gives:

$$F_s(f, h) = (\Phi_s(h))_{D'} \circ \Phi_s(M)(f). \quad (17)$$

By the naturality of $\Phi_s(h)$, the following equality holds: $(\Phi_s(h))_{D'} \circ \Phi_s(M)(f) = \Phi_s(M')(f) \circ (\Phi_s(h))_D$. Hence, the two canonical ways of evaluating the morphism pair (f, h) give the same morphism in C_{BAB}^s . This is precisely the naturality condition established in the construction of the conformance functor.

Consequently, $F_s(f, h)$ is a well-defined morphism $F_s(D, M) \rightarrow F_s(D', M')$ in C_{BAB}^s .

Since Φ_s is a functor, the pairing $\langle \Phi_s \circ \pi_2, \pi_1 \rangle$ is a functor from $C_{Core}^{NonMapping} \times C_{Map}$ to $[C_{Core}^{NonMapping}, C_{BAB}^s] \times C_{Core}^{NonMapping}$.

The evaluation functor ev_s is a functor by the Cartesian closed structure of **Cat**. Therefore, their composition $F_s = ev_s \circ \langle \Phi_s \circ \pi_2, \pi_1 \rangle$ is a functor, where π_1 and π_2 denote the canonical projection functors from the product category, and $\langle \cdot, \cdot \rangle$ denotes the pairing (universal property of the product). Since the domain of F_s is the product category $C_{Core}^{NonMapping} \times C_{Map}$, the functor F_s is a bifunctor in its two arguments. This proves that the parametrized model-to-model transformation is functorial both in the domain model and in the mapping class. ■

Architectural implication This formalization ensures that multi-target code generation is achieved by varying only the stack index $s \in S$: the source domain category $C_{Core}^{NonMapping}$ and the mapping-refinement category C_{Map} remain unchanged, while the conformance functor Φ_s handles all stack-specific resolution. In the notation of Section 5.4.2, the functor F_s is induced by the Φ_s through evaluation functor ev_s for a fixed stack s , restricted to the mapping pairs selected by `core:hasImplementationMapping`. In the Java instance, this functor is denoted F_j . Adding support for a new platform requires only the definition of a new conformance functor $\Phi_{s'}$ and its associated target category $C_{BAB}^{s'}$, without any modification to the domain model or the generation architecture.

Functor $H_s : C_{BAB}^s \rightarrow C_{code}^s$ (Model-to-Code Transformation for stack s) For each target stack s , the second functor, H_s , represents the model-to-code (M2T) transformation: it takes the platform-specific schema representation in C_{BAB}^s and produces source files in C_{code}^s .

On objects, H_s sends each stack-specific realization specification produced by F_s to the source file or file fragment that realizes it. For example, H_s sends the representation of a class to a source file whose content is the textual rendering of that class by the corresponding FreeMarker template. The input object of H_s is a stack-specific realization specification, not merely an undecorated platform class. Therefore, H_s has access to all generation-relevant information carried by the selected mapping strategy: annotations,

implemented interfaces, superclass declarations, imports, fields, methods, template roles, and other generation parameters. This prevents mapping-specific information from being lost between the model-to-model and model-to-code phases.

On morphisms, H_s sends each structural relation in C_{BAB}^s to the corresponding textual relation in C_{code}^s .

Since C_{BAB}^s is a presented category, H_s is defined in two steps. First, a graph-level assignment is specified on the generators of G_{BAB}^s , mapping each platform-level structural relation to the corresponding code-level structural dependency. By the universal property of the free category, this assignment extends uniquely to a functor

$$\widetilde{H}_s : Free(G_{BAB}^s) \rightarrow C_{code}^s. \quad (18)$$

Second, because $C_{BAB}^s = Free(G_{BAB}^s) / \equiv_E$, the functor $\widetilde{H}_s$ induces a well-defined functor on the quotient only if it is constant on $\equiv_E$ -equivalence classes. Therefore, whenever $u \equiv_E v$, it must hold that $\widetilde{H}_s(u) = \widetilde{H}_s(v)$ as the same structural dependency path in the free category C_{code}^s . This means that platform-level paths identified by the defining equations of C_{BAB}^s must be rendered by the generator as the same code-level structural dependency path. This quotient-compatibility condition is required as part of transformation verification. Under this condition, $\widetilde{H}_s$ factors uniquely through the quotient, yielding a well-defined functor

$$H_s : C_{BAB}^s \rightarrow C_{code}^s \text{ defined by } H_s([u]_{\equiv_E}) = \widetilde{H}_s(u). \quad (19)$$

In the Java implementation, H_j is realized by the FreeMarker template mechanism, which traverses the in-memory BABj model and emits source-code artifacts guided by text templates specific to Java target language construct.

5.5. Structural Preservation and Faithfulness of the Transformation Pipeline

Having defined the full transformation pipeline $T_s = H_s \circ F_s$, we now state the central information-preservation result of the methodology. The proposition below captures the formal sense in which the transformation pipeline preserves the structural content of the domain model. It should be read as a conditional schema-level preservation theorem. It states what follows if the transformation artifacts satisfy the stated verification assumptions; it is not a claim that the categorical presentation alone proves correctness of an arbitrary generator implementation.

Proposition 2. (Conditional Structural Preservation). Let F_s and H_s be the model-to-model bifunctor and model-to-code functor, defined in Sections 5.4.3 and 5.4.4 respectively, for a fixed target stack s , and let $T_s = H_s \circ F_s$ denote their composition, evaluated on the mapping pairs selected by `core:hasImplementationMapping`. Assume that the following verification conditions hold: (i) the graph-level assignment underlying F_s is injective on the selected source morphism pairs, except for identifications induced by explicitly declared normalization equations in E_{BAB}^s , and the congruence $\equiv_E$ introduces no further identifications among their F_s -images; (ii) the graph-level assignment underlying H_s is constant on $\equiv_E$ equivalence classes, so that H_s is well-defined on C_{BAB}^s , and H_s is injective on the relevant target equivalence classes; (iii) the naming scheme is injective on the considered domain classes; and (iv) distinct selected mapping strategies

induce distinct generation profiles, so that the code-artifact identity distinguishes them by their generated structural content. Then T_s satisfies the following properties:

- a) Totality. For every selected object pair $(D, M) \in \text{Ob}(C_{Core}^{NonMapping} \times C_{Map})$, there exists a unique generated source-artifact specification $T_s(D, M) \in \text{Ob}(C_{code}^s)$.
- b) Structural coherence. For every selected morphism pair $(f, h) : (D, M) \rightarrow (D', M')$ in $C_{Core}^{NonMapping} \times C_{Map}$, there exists a unique morphism $T_s(f, h) : T_s(D, M) \rightarrow T_s(D', M')$ in C_{code}^s . Moreover, the composite functor preserves identities and composition: $T_s(\text{id}_D, \text{id}_M) = \text{id}_{T_s(D, M)}$ and, for composable morphism pairs $(f, h) : (D, M) \rightarrow (D', M')$ and $(f', h') : (D', M') \rightarrow (D'', M'')$ it holds that $T_s((f', h') \circ (f, h)) = T_s(f', h') \circ T_s(f, h)$. Equivalently, since composition in the product category is componentwise, $T_s(f' \circ f, h' \circ h) = T_s(f', h') \circ T_s(f, h)$, where equality is understood in the corresponding quotient/presented target category.
- c) Faithfulness modulo explicitly intended normalizations. If the morphism pairs

$$(f, h), (g, k) \in \text{Hom}_{C_{Core}^{NonMapping} \times C_{Map}}((D, M), (D', M')) \quad (20)$$

have F_s -images that are not identified by the explicitly intended normalization equations included in E_{BAB}^s , then

$$(T_s(f, h) \neq T_s(g, k) \text{ in } \text{Hom}_{C_{code}^s}(T_s(D, M), T_s(D', M'))). \quad (21)$$

If no explicitly intended normalization equations are included, this reduces to strict faithfulness on the selected schema fragment.

- d) Injectivity on objects. If $(D, M) \neq (D', M')$ in $\text{Ob}(C_{Core}^{NonMapping} \times C_{Map})$, then $T_s(D, M) \neq T_s(D', M')$ in $\text{Ob}(C_{code}^s)$.

Proof. (a) and (b) follow directly from the definition of functor composition: the composition of two functors is again a functor, and functors are by definition total on objects and morphisms and preserve the category axioms. This establishes totality and structural coherence without further assumption.

(c) Faithfulness requires that distinct morphism pairs in $C_{Core}^{NonMapping} \times C_{Map}$ remain distinct after applying T_s . This does not follow merely from the construction of C_{BAB}^s , because C_{BAB}^s is a quotient of $\text{Free}(G_{BAB}^s)$ and the congruence $\equiv_E$ may identify distinct target paths. Therefore, faithfulness is obtained under the stated separation assumptions: F_s must map distinct source morphism pairs to distinct equivalence classes in C_{BAB}^s , except for explicitly intended normalization, and H_s must preserve the distinction between the relevant equivalence classes at the code level. Under these assumptions, if $(f, h) \neq (g, k)$ in $\text{Hom}_{C_{Core}^{NonMapping} \times C_{Map}}((D, M), (D', M'))$, then $F_s(f, h) \neq F_s(g, k)$ in C_{BAB}^s . Since H_s is injective on the relevant target equivalence classes, distinct F_s -images remain distinct after applying H_s . Consequently, if two selected morphisms are distinct in the considered finite schema fragment and are not identified by an explicitly intended normalization equation $H_s(F_s(f, h)) \neq H_s(F_s(g, k))$, their T_s -images are distinct $T_s(f, h) \neq T_s(g, k)$ in C_{code}^s . Hence T_s is faithful on the selected schema fragment.

(d) Injectivity on objects follows from the namespace-aware IRI-based naming scheme together with the mapping-sensitive definition of code-artifact identity. The object $T_s(D, M)$ is not identified merely by the physical file path, but by the generated source-artifact specification, including the target artifact role, generated structural content, superclass declarations, implemented interfaces, annotations, fields, methods, imports, and template roles. Therefore, two selected mapping strategies that induce distinct generation profiles yield distinct generated source-artifact specifications. If two mapping declarations induce the same generation profile, they are treated as equivalent for the purposes of this proposition rather than as distinct preserved inputs. If $D \neq D'$, the generated identifiers are distinct by the naming scheme. If $D = D'$ but $M \neq M'$, then the two mapping strategies determine different generated structural content, such as different implemented interfaces, annotations, superclass declarations, fields, methods, or template roles. Hence, they define distinct generated source-artifact specifications in C_{code}^s . Therefore, if the selected object pairs (D, M) and (D', M') are distinct and are not identified by generation-profile normalization, then their images $T_s(D, M)$ and $T_s(D', M')$ are distinct objects in C_{code}^s . ■

Corollary (Schema-level information preservation). *Under the assumptions of Proposition 2, every mapped domain class and every mapped structural relation in the considered schema fragment, together with its selected mapping strategy, has a corresponding schema-level generated source-artifact specification in C_{code}^s . Moreover, the correspondence is injective on the selected schema fragment: distinct object pairs (D, M) in $C_{Core}^{NonMapping} \times C_{Map}$ that are not identified by generation-profile normalization produce distinct code-level artifact specifications, and distinct morphism pairs (f, h) not identified by explicitly intended normalization produce distinct morphisms in C_{code}^s . Consequently, under the stated verification assumptions, the schema-level distinctions present in the mapped domain model and in its selected mapping refinements are neither lost nor conflated during transformation.*

Corollary (Image subcategory). *Under the assumptions of Proposition 2, image of T_s forms a subcategory $Im(T_s) \subseteq C_{code}^s$. After quotienting the selected object and morphism pairs determined by $core : hasImplementationMapping$ by the explicitly intended normalization equations, the induced restriction of T_s is an isomorphism onto this image. Hence, the generated code category contains a subcategory isomorphic to the selected mapped schema fragment, not necessarily to the entire $C_{Core}^{NonMapping} \times C_{Map}$.*

The proposition above asserts conditional structural preservation: under the assumptions stated in Proposition 2, distinctions between object pairs and morphism pairs in $C_{Core}^{NonMapping} \times C_{Map}$ are preserved by the composite transformation T_s into C_{code}^s . It does not assert:

- *Instance-level preservation.* We do not formally verify that data populating the domain model (A-box individuals) is preserved in any operational sense by the generated code's runtime behavior. The methodology is a schema-level (T-box) transformation; instance-level consistency and completeness are separate concerns handled by the ontology validation layer, combining OWL reasoning with structural SPARQL/SHACL-style checks before transformation.
- *Semantic equivalence.* We do not assert that the generated code "means the same thing" as the domain ontology in an operational or denotational sense. Such a claim would require a formal operational semantics of both the ontology and the target

language, together with a semantics-preserving morphism between them, which is beyond the scope of the present work.

- *Round-trip equivalence.* We do not assert that the domain ontology can be recovered from the generated code. The composite functor $T_s = H_s \circ F_s$ is not claimed to have a left or right adjoint. Such a construction, which would underpin formal round-trip engineering, is identified as a direction for future work.

The scope of Proposition 2 is precisely the schema-level guarantee that a rigorous model-driven transformation can offer: under the stated verification conditions, the structure of the mapped domain model and its selected mapping refinements is neither lost nor conflated as it passes through the categorical pipeline to generated code.

5.6. Model validation and transformation verification

The methodology distinguishes two complementary correctness mechanisms, operating at different stages of the code generation pipeline and relying on different formal bases.

Model validation takes place before the transformation begins and is the responsibility of the ontology reasoning layer. The validation layer, combining OWL reasoning with structural SPARQL/SHACL-style checks, verifies that a domain ontology, constructed by importing and specializing BAB Core, is a well-formed description of the intended PAIS system. This includes logical checks handled by the OWL reasoner, as well as completeness checks handled by structural SPARQL/SHACL-style constraints, such as verifying cardinality constraints (e.g., that each class of type `core:Entity` declares at least one `core:hasAttribute` relation), enforcing structural obligations (e.g., that every `core:Process` contains both a `core:StartTask` and an `core:EndTask`), and ensuring that disjointness constraints are respected (e.g., that `pms:Shipment` and `pms:Customer` are not inadvertently identified). The axioms declared in BAB Core, including inverse property declarations, functional property constraints, and subclass hierarchies, constitute the logical part of the validation criteria. Completeness conditions that depend on the explicit presence of modeling elements are checked by SPARQL/SHACL-style structural constraints.

Model validation, as described above, is an ontological property: it asserts that the domain ontology is a structurally and semantically coherent model of the target system. It is not, and does not need to be, a categorical property. The categorical formalism developed in this section operates at the schema level and assumes that the input ontology has already passed reasoner validation. In this sense, the reasoning layer and the categorical layer are cleanly separated: the reasoner ensures that the input to the transformation is well-formed, while the categorical formalism ensures that the transformation itself preserves structure.

Transformation verification concerns the behavior of the generator, not the input ontology. It asks whether the bifunctor $F_s : C_{Core}^{NonMapping} \times C_{Map} \rightarrow C_{BAB}^s$ and functor $H_s : C_{BAB}^s \rightarrow C_{code}^s$, and in particular their Java instances F_j and H_j , as implemented by the SPARQL-driven extraction and the FreeMarker template mechanism, truly satisfy the properties claimed of them in Sections 5.4.2 and 5.4.4. The bifunctor F_s is obtained from the conformance functor construction of Section 5.4.3: each $\Phi_s(M)$ is determined by the universal property of the free source category $C_{Core}^{NonMapping}$, and F_s is then obtained by evaluation. The functor H_s is obtained by first extending a graph-level assignment from G_{BAB}^s to $Free(G_{BAB}^s)$, and then verifying that this extension $\tilde{H}_s$ respects the congruence $\equiv_E$ so that it factors through the quotient C_{BAB}^s . The functoriality of

F_s follows from the functoriality of Φ_s and the evaluation functor, as proved in Proposition 1. The functoriality of H_s follows after the graph-level extension $\widetilde{H}_s$ is verified to respect the quotient congruence $\equiv_E$. Thus, preservation of identities and composition is guaranteed by construction once these specification-level conditions are satisfied. The structural preservation properties proved in Proposition 2 (totality, structural coherence, faithfulness, and injectivity on objects) similarly follow formally, conditional on the implementation respecting the specified mapping rules.

In practice, engineering verification consists of checking that the implementation does not deviate from the specification. Specifically, five checks are performed:

- Mapping completeness: every selected generating edge of $G_{Core}^{NonMapping}$ that appears in the domain ontology must be in the domain of the property-label assignment λ_s . Equivalently, every selected BAB Core schema relation used by the transformation has a corresponding platform-level structural relation label in G_{BAB}^s . A SPARQL query over BAB Core and the stack-specific BABj ontology reports any unmapped properties before generation begins.
- Mapping injectivity: no two distinct selected BAB Core schema relations are mapped by λ_s to the same platform-level structural relation label unless this identification is explicitly intended by the normalization equations in E_{BAB}^s . This is a precondition for Proposition 2's faithfulness claim and is verified statically by inspection of the stack-specific ontology and transformation specification.
- Template determinism: each FreeMarker template produces a deterministic output for a given BABj element, with no nondeterministic choices or dependency on external state. This is verified by code review of the template library and by snapshot testing against a fixed corpus of domain ontologies.
- Quotient compatibility: every equation included in E_{BAB}^s must be preserved by the model-to-code functor H_s . Concretely, if two parallel paths are identified by the congruence $\equiv_E$, then their template-level interpretations must produce the same code-level structural dependency path in C_{Code}^s . Operationally, this means that the FreeMarker templates must not distinguish between two paths that the presented category C_{BAB}^s identifies. If two different realization paths attempt to generate the same artifact identifier with different structural content, this is treated as a naming or generation-profile conflict and the verification fails. It is not treated as a code-level normalization. This condition ensures that H_s is well-defined on equivalence classes of paths in C_{BAB}^s .
- No unintended collapse: the congruence $\equiv_E$ must not identify the images under F_s of distinct selected source morphisms, except where the identification is explicitly declared as an intended normalization equation in E_{BAB}^s . This condition is required for the faithfulness-modulo normalization part of the structural preservation result.

When all five checks pass, the engineering implementation conforms to the categorical specification, and the guarantees established in Proposition 2 apply to the generated code.

The entire transformation can be summarized as the following pipeline:

$$C_{Core}^{NonMapping} \times C_{Map} \rightarrow C_{BAB}^s \xrightarrow{H_s} C_{code}^s \quad (22)$$

where each arrow is a functor with a precise categorical interpretation, and the structural preservation of the composition T_s is established by Proposition 2. In this sense, the

methodology transforms ontology-driven code generation from a heuristic engineering process into a formally specified transformation with provable schema-level correctness.

6. Software implementation

The transformer prototype is realized in the Java programming language and represents a concrete implementation of the transformation chain $H_j \circ F_j$ for the Java technology stack (denoted $s = j$ throughout this section). It integrates libraries for working with ontologies (Apache Jena) and a text generation mechanism (FreeMarker).

The transformation process unfolds through clearly defined phases corresponding to functors F_j and H_j (Figure 5).

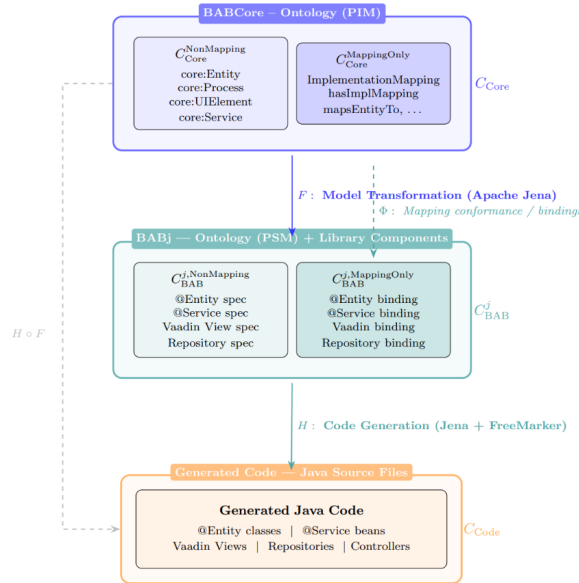


Fig. 5. Architecture of the implementation solution

6.1. Implementation of Bifunctor F_j – Model-to-Model (M2M) Transformation

The first phase, corresponding to the bifunctor $F_j : C_{Core}^{NonMapping} \times C_{Map} \rightarrow C_{BAB}^j$, is the Model-to-Model transformation, and this logic is executed by the model transformer.

The domain engineer defines a concrete system model as a specific instance of the ontology. Practically, this means describing the specific application by adding new classes or individuals to the ontology representing entities, processes, modules, etc. For example, for a Postal Management System (PMS), the engineer would add a `pms:Shipment` class as a subclass of `core:Entity`, a `pms:Courier` class also as a subclass of `core:Entity`, a `pms:PackageDeliveryProcess` process as an instance of

core:Process with appropriate states (e.g., 'In Transit', 'Delivered') and tasks (e.g., 'ScanPackage', 'DeliverPackage'). All these added elements and their connections constitute one application model at the PIM level.

It is assumed that a target stack (e.g., Java) is selected as a configuration input to the transformer, so the corresponding PSM ontology (e.g., BABj) is loaded alongside the domain model. The BAB Core ontology enables a `hasImplementationMapping` link between every model element and an object carrying implementation details. In practice, an instance of `ImplementationMapping` is defined in the stack-specific ontology for every relevant domain concept or set of concepts. For example, for the Java stack, the BABj ontology declares generic mapping individuals, one per architectural aspect, that collectively specify how domain elements are realized. A typical setup includes `babj:DefaultEntityMapping_J`, an instance of `core:EntityMapping`, which declares via `core:mapsToTypeSpec` that domain entities are realized as Java classes extending `babj:AbstractEntity` with the `@Entity` annotation; `babj:DefaultAttributeMapping_J`, an instance of `core:AttributeMapping`, specifying that domain attributes are realized as Java fields with `@Column` annotations; and similar default mapping individuals for processes, tasks, relations, and UI elements. A domain class such as `pms:Shipment` connects to its mapping individuals through `core:hasImplementationMapping`, which is a multi-valued relation: `Shipment` is linked simultaneously to `DefaultEntityMapping_J` (governing its class-level realization) and to the relevant `DefaultAttributeMapping_J` (governing its field realizations), and potentially to additional mapping individuals governing other aspects of its platform presence.

More refined mapping strategies are expressed through subclasses of the generic mapping classes. For instance, `core:ValidatableEntityMapping` may be declared as a subclass of `core:EntityMapping`, and a domain class that requires validation logic can reference an individual of this refined class instead of the default one. Refinement thus acts pointwise on individual aspects rather than on entire domain elements, and multiple aspects of the same domain class may use independently chosen refinement levels. For simplicity, the prototype defaults to using the generic mapping individuals for all non-refined domain elements, applying class-specific refinement only where the ontology explicitly declares it.

The implementation operates at the instance level by traversing ABox assertions of `core:hasImplementationMapping`, retrieving named mapping individuals such as `babj:DefaultValidatableEntityMapping_J`. Each retrieved individual realizes a mapping class $M \in \text{Ob}(C_{Map})$ in the sense of the realization assignment μ_j introduced in Section 5.4.3: the mapping class is recovered as the asserted `rdf:type` of the retrieved individual. The formal pair (D, M) on which the bifunctor F_j is defined therefore corresponds operationally to the pair $(D, \mu_j(M))$, with M recovered from the individual's type. The remainder of this section uses individual names (e.g., `DefaultEntityMapping_J`) when describing operational behavior and the corresponding mapping classes (e.g., `core:EntityMapping`) when referring to the formal indexing.

The SPARQL-driven extraction mechanism, defined in the `SparqlQueries.java` class, implements functor F_j by executing a series of queries over the loaded do-

main and BABj ontologies. The queries first collect all domain classes. For each such class D , the queries then retrieve every mapping individual linked to D via `core:hasImplementationMapping`, following the multi-valued relation so that all aspect-specific mapping individuals (class-level, attribute-level, and any further ones) are gathered in one pass. For each retrieved mapping individual, the queries read its structural bindings, specifically `core:mapsToTypeSpec` (pointing to the BABj library class that realizes the mapping) and any auxiliary annotation or interface-realization properties declared in the BABj ontology and accumulate this information into the DataModel representation used in the next phase.

The following simplified Java fragment (Listing 1) illustrates the operational core of this extraction step. Prefixes are resolved from the ontology vocabulary constants used by the transformer.

```

public DataModel extractClassModel(Resource domainClass) {
    DataModel model = new DataModel();
    model.setClassName(localName(domainClass));

    String sparql = """
        SELECT ?mapping ?mappingClass ?targetType
        WHERE {
            ?domain core:hasImplementationMapping ?mapping .
            ?mapping rdf:type ?mappingClass .
            ?mappingClass rdfs:subClassOf* core:ImplementationMapping .
            OPTIONAL { ?mapping core:mapsToTypeSpec ?targetType . }
        }
        """;

    ParameterizedSparqlString q = new ParameterizedSparqlString(sparql);
    q.setNsPrefix("core", CoreVocabulary.NS);
    q.setNsPrefix("rdf", RDF.uri);
    q.setNsPrefix("rdfs", RDFS.uri);
    q.setParam("domain", domainClass);

    try (QueryExecution qe =
        QueryExecutionFactory.create(q.asQuery(), ontologyModel)) {
        ResultSet rs = qe.execSelect();
        while (rs.hasNext()) {
            QuerySolution row = rs.next();
            Resource mappingClass = row.getResource("mappingClass");
            model.addMappingClass(localName(mappingClass));
            if (row.contains("targetType")) {
                model.setExtendsClass(
                    resolveTypeSpec(row.getResource("targetType")));
            }
        }
    }

    readAttributes(domainClass).forEach(model::addAttribute);
    return model;
}

```

Listing 1 - Simplified Java/SPARQL fragment implementing the M2M extraction step for the Java stack

The fragment is intentionally simplified. The actual transformer executes additional queries for annotations, interfaces, attributes, relations, and behavioral elements. Nevertheless, it shows the essential mechanism of F_j : the domain class determines what is transformed, while the selected mapping individual and its recovered mapping class determine how the element is realized in the Java stack.

The results of the SPARQL queries are parsed and used to populate internal memory objects, defined by the `DataModel.java` class. The `DataModel` object is an in-memory representation of the realization specification constructed for a domain class D from the mapping individuals linked to D via `core:hasImplementationMapping`. Operationally, each such mapping individual contributes one generation-relevant aspect of the realization specification: class-level annotations and superclass from `core:EntityMapping`, field-level annotations from `core:AttributeMapping`, interface bindings from refinements such as `core:ValidatableEntityMapping`, and so on. The `DataModel` records the union of these implementation-level aspects: the target library class, annotations, implemented interfaces, fields, and structural bindings. This aggregation is an implementation-level construction used to form a single realization specification for D ; it is not used as a categorical coproduct in C_{BAB}^j .

This step constitutes the essence of the M2M transformation. Operationally, it transforms the schema-level graph G_{Core} , extracted from the BAB Core RDF/OWL ontology, into a stack-specific, yet still textually neutral, PSM `DataModel` object. Categorically, this object represents the corresponding fragment of C_{BAB}^j .

6.2. Implementation of Functor H_j – Model-to-Code (M2T) Transformation

The second phase, corresponding to functor $H_j : C_{BAB}^j \rightarrow C_{code}^j$ is the Model-to-Code transformation. `CodeGeneration.java` passes the populated `DataModel` object (PSM) to the FreeMarker template engine. Templates such as `EntityTemplate.ftl` and `SessionTemplate.ftl` implement functor H_j . These templates iterate through the `DataModel` and generate textual output. For example, `EntityTemplate.ftl` contains logic generating Java code, Listing 2.

```
package ${dataModel.packageName};

<#- Imports generated from class-level annotations ->
<#list dataModel.annotations as annotation>
    import ${annotation.fullName};
</#list>

<#- Imports generated from attribute-level annotations ->
<#list dataModel.attributes as attr>
<#if attr.annotation?? && attr.annotation.fullName??>
    import ${attr.annotation.fullName};
</#if>
</#list>
    import ${dataModel.extendsClass.fullName};

<#- Loop adding imports for all interfaces ->
<#list dataModel.interfaces as interface>
    import ${interface.fullName};
</#list>

<#- Class-level annotations generated from the selected mapping profile ->
<#list dataModel.annotations as annotation>
    ${annotation.expression}
</#list>
public class ${dataModel.className} extends ${dataModel.extendsClass.name}
<#- Logic for adding 'implements' clause ->
<#if dataModel.interfaces?has_content>
    implements <#list dataModel.interfaces as interface>${interface.name}<#sep>, </#list>
</#if>
{
    <#- Loop through attributes ->
    <#list dataModel.attributes as attr>
    @${attr.annotation.name}<#if attr.annotation.parameters?? &&
    attr.annotation.parameters?has_content>${attr.annotation.parameters}</#if>
    private ${attr.type} ${attr.name};

    // ... generated getters and setters ...
</#list>

    <#- Loop generating 'hook' methods for each behavior ->
    <#if dataModel.beforeValidate >
```

```

@Override
public void beforeValidate() {
    // TODO: Programmer adds logic
}
</#if>
}

```

Listing 2 - Fragment of FreeMarker template (*EntityTemplate.ftl*) implementing functor H_j

The output of this M2T transformation is a Java file, a specific object in category C_{code}^j .

6.3. Transformation Example: From BAB Core to Shipment.java

To illustrate the entire process, let us consider the transformation of the Shipment entity from the Postal Management System (PMS) domain

PIM Level (C_{Core}) – Definition in Ontology The domain expert defines Shipment together with the mapping strategy governing its platform realization (Listing 3). The PIM ontology does not directly contain Java annotations; instead, it selects mapping individuals whose stack-specific generation profiles later provide annotations, imports, superclass bindings, implemented interfaces, and template hooks. The BAB Core ontology declares `core:ValidatableEntityMapping` as a subclass of `core:EntityMapping`, specializing the generic entity mapping strategy for entities that require validation logic. The BAB_j ontology provides `babj:DefaultEntityMapping_J` as a default instance of `core:EntityMapping` for non-validatable entities, and `babj:DefaultValidatableEntityMapping_J` as an instance of `core:ValidatableEntityMapping` for the validatable case; the latter carries the binding `core:mapsToTypeSpec` pointing to `babj:AbstractEntity`, together with a specification of the `CheckValidateEntityInterface` realization. Attribute-level realization is governed separately by `babj:DefaultAttributeMapping_J`, an instance of `core:AttributeMapping`. The listing uses the class IRI as the attachment point for generation metadata; operationally, the transformer treats these triples as mapping-selection assertions used by the SPARQL extraction layer.

```

<owl:Class rdf:about="&pms;Shipment">
  <rdfs:subClassOf rdf:resource="&core;Entity"/>
  <rdfs:label>Shipment</rdfs:label>
  <core:hasImplementationMapping
    rdf:resource="&babj;DefaultValidatableEntityMapping_J"/>
  <core:hasImplementationMapping
    rdf:resource="&babj;DefaultAttributeMapping_J"/>
</owl:Class>

<owl:DatatypeProperty rdf:about="&pms;trackingId">
  <rdfs:domain rdf:resource="&pms;Shipment"/>
  <rdfs:range rdf:resource="&xsd:string"/>
</owl:DatatypeProperty>

```

Listing 3 - RDF/XML definition for *pms:Shipment*

PSM Level (C_{BAB}^j) – Internal DataModel The SPARQL-driven extraction (implementing bifunctor F_j) reads the RDF graph and creates the following DataModel object in the transformer’s memory:

```

className: "Shipment"
annotations: [
  (name: "Entity", fullName: "javax.persistence.Entity",
   expression: "@Entity"),
  (name: "Table", fullName: "javax.persistence.Table",
   expression: "@Table(name = \"Shipment\")")
]
extendsClass: (name: "AbstractEntity", fullName:
  "rs.co.bora5.programs.bab.model.AbstractEntity")
interfaces: [
  (name: "CheckValidateEntityInterface", fullName:
    "rs.co.bora5.programs.bab.model.interfaceCheck.CheckValidateEntityInterface")
]
attributes: [
  {
    name: "trackingId",
    type: "String",
    isMandatory: false,
    annotation: (name: "Column",
      fullName: "javax.persistence.Column", parameters: "")
  }
]

```

Code Level (C_{code}^j) – Generated Java File The FreeMarker template engine (implementing functor H_j , using `EntityTemplate.ftl`) takes the DataModel and produces the final Java file, Listing 4.

```

package rs.co.bora5.programs.pms.model;

import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Column;
import rs.co.bora5.programs.bab.model.AbstractEntity;
import rs.co.bora5.programs.bab.model.interfaceCheck.CheckValidateEntityInterface;

@Entity
@Table(name = "Shipment")
public class Shipment extends AbstractEntity
    implements CheckValidateEntityInterface {

    @Column
    private String trackingId;

    public String getTrackingId() { return trackingId; }
    public void setTrackingId(String trackingId) { this.trackingId = trackingId; }

    @Override
    public void beforeValidate() {
        // TODO: Programmer adds shipment validation logic
    }
}

```

Listing 4 - Generated class Shipment.java (simplified)

This example illustrates how the semantic definition from the PIM (including both structure and abstract “Validatable” behavior) is deterministically transformed into concrete, architecturally correct Java code directly tied to the BABj library.

In the example above, it is evident that the Shipment class is annotated with `@Entity` (originating from the `entityMapping` rule defined in the ontology). It inherits `AbstractEntity` from the BABj library (this base class contains generic implementations such as an id field with an auto-generated key, metadata retrieval methods,

etc.) ensuring that the generated `Shipment` class is minimalistic and focused only on specific attributes. The `trackingId` field is generated based on ontological attributes. Had the ontology defined constraints like uniqueness for `trackingId`, this could have been reflected by generating an additional annotation (e.g., `@Column(unique=true)` or logical checks).

By contrast, a transient helper class such as `pms:TempFormData`, also modeled at the core level but linked to a separate `babj:DefaultTransientModelMapping_J`, would produce a class extending the appropriate `BABj` base type without the `@Entity` annotation and without `CheckValidateEntityInterface`, suitable for form validation or object representation of external query results. This contrast illustrates the bifunctor's genuine dependence on the mapping argument: the same domain type (`core:Entity`) yields structurally distinct artifacts depending on the chosen mapping strategy.

A similar procedure applies to other model elements. For example, if there is a business process `PackageDeliveryProcess` with two states (`In_Transit` and `Delivered`) and several tasks, the transformer would create a class (or a set of classes) for that process. It is possible to generate an enum with process state definitions, followed by a service class containing methods representing tasks (e.g., `onScan()` for the initial task, `onDelivered()` for the final one, etc.), where implementation details (repository calls, sending notifications) are delegated to the `BABj` library or left empty for the developer to fill in. For UI elements (forms and views), in the case of the Vaadin framework, the transformer creates corresponding Java classes extending Vaadin components and binding them to entities/processes. For instance, for a `Shipment` entity, an `EditShipmentWindow` class could be generated inheriting the `BABj` library base class for forms, `GenericWindow`, containing fields for name and email entry, as well as save buttons, where button logic is connected to service methods (again, following a library pattern).

It is important to emphasize that the transformer does not attempt to solve all domain-specific code; its role is to create an application skeleton that is immediately executable and contains all standard system parts. The possibility is left for the engineer to supplement specific business logic methods where necessary. For example, the generated `deliverPackage()` method in the service might initially be empty (or throw an `UnsupportedOperationException`) until the engineer adds concrete steps (inventory check, database update, sending email notifications, etc.). However, even these additions can be made platform-independent using ontological annotations or rules, e.g., a specific task could be marked as “email notification”, allowing the transformer to create an email sending skeleton using the appropriate target framework.

In this way, the prototype confirms that the formal functorial specification can be successfully implemented in a practical tool that deterministically produces code from ontologically defined models.

7. Analysis and Discussion of Results

This section presents a detailed analysis and discussion of the results obtained by applying the proposed Ontology-Driven Code Generation (ODCG) system. Through a case study of a prototype project management application, the benefits of the approach regarding the

acceleration of initial development, reduction of manual work, and assurance of architectural consistency are quantified. In addition to the empirical evaluation of the quality of the generated code and the analysis of its sustainability through technological changes, the chapter clearly positions the achieved contribution in relation to existing paradigms, such as Model-Driven Engineering (MDE) and Ontology-Based Data Access (OBDA). Finally, a critical review of the methodology's limitations is provided, including the challenges of round-trip engineering and dynamic system aspects modeling, thereby defining the scope of applicability and opening directions for future research. Application of ODCG showed significant acceleration of the initial development cycle and increased architectural consistency. To assess the gains, a prototype business application of medium size was created using the proposed approach. For demonstration purposes, a project management system was modeled, including ~ 10 entities (Project, Task, Employee, Client, Invoice, etc.), 5 business processes (e.g., project approval process, invoicing), and 3 modules (Finance, Operations, User Management). The domain description in the BAB Core ontology itself encompassed about 50 concepts (classes and relations). From that model, the transformer produced a complete multi-tier application architecture for the Java stack. For each defined entity, a set of five coupled components was generated (JPA entity, DTO object, service class, Vaadin view, and input/update form), resulting in the generation of over 80 Java classes with a total of approximately 5000 lines of code. This amplification factor shows that a single concept in the ontology results in the generation of multiple implementation artifacts. For an experienced programmer, writing a similar application foundation manually would require weeks of work, while the transformer produced all the code automatically in a few seconds. Naturally, the generated code required minimal refinements (mainly populating concrete logic in specified methods). This result illustrates a measurable time saving and a reduction in manual work of approximately 60–70% at the start of the project, compared to complete manual setup. This figure should be interpreted as an exploratory engineering observation based on the authors' experience with comparable manually developed application foundations rather than as a rigorously validated metric. A controlled baseline comparison can be established in future work by implementing the same application foundation manually and through the ODCG process under comparable conditions, measuring development effort (person-hours), number of produced artifacts, defect counts, and maintenance effort across multiple independent projects. Since the basic code skeleton is generated automatically, frequent errors caused by the human factor such as typographical errors in class and field names, inconsistent application of annotations, or omissions in connecting application layers are removed from the process. The generated code possesses a consistent, template-defined structure and passes automatic validation according to ontology rules. For example, if the ontology prescribes that every entity class must have at least one key attribute, the transformer automatically ensures this or generates an appropriate warning. The quality of the generated code was assessed using static analysis methods. The analysis of the generated code was performed using the SonarQube tool (Community Edition version), using the standard Sonar way profile for the Java programming language. Given that the code was generated automatically, the goal of the analysis was verification of structural correctness and the absence of "code smell" patterns characteristic of manual coding. The aggregated analysis results for the application module are shown in Table 1. The evaluation has two distinct roles. The categorical preservation result is assessed through the verification obligations defined in Sec-

tion 5.6: mapping completeness, mapping injectivity, quotient compatibility, template determinism, and absence of unintended collapse. Static analysis of the generated code has a different role: it evaluates engineering quality of the generated artifacts and templates, but it should not be read as an empirical proof of the categorical preservation theorem.

Table 1. Overview of Code Quality Metrics (SonarQube)

Metric	Result	Grade	Comment
Bugs	0	A	No logical errors detected (e.g., <code>NullPointerException</code> risks).
Vulnerabilities	0	A	No security flaws (e.g., SQL Injection), thanks to the use of parameterized queries in generated repositories.
Code Smells	< 10	A	Negligible number, mostly related to specific domain class names not recognized by the tool.
Technical Debt	< 1 h	A	Technical Debt Ratio is below 0.1%, which is drastically better than the industry standard (5%).
Duplication	~ 3.5%	–	Duplication is present only at the structural level (“boilerplate” code), but not in business logic.
Cyclomatic Complexity	1.2 / method	–	Average complexity is extremely low, as generated methods are short, atomic, and without deep branching.

The generated code achieved an “A” grade in Reliability, Security, and Maintainability. In the analyzed case, SonarQube detected no bugs, vulnerabilities, security hotspots, or naming-convention violations, while the templates produced consistent use of Java annotations and resource-management patterns. The low cyclomatic complexity indicates that the generated methods are structurally simple. These results provide engineering evidence about the quality of the current Java templates and generated artifacts in the analyzed case; they do not establish that ODCG eliminates all syntax, style, or security errors in every generated system, nor do they constitute an independent proof of the categorical preservation theorem. One of the key advantages of the approach is easier system maintenance and evolution. If requirements change, say new entities or fields are added, it is sufficient to update the domain ontology and re-run code generation. Changes will propagate through the entire system consistently. Experience with the prototype showed that even more complex refactorings (e.g., renaming entities or splitting one process into two) can be carried out practically automatically, without the need for manual searching and correcting in multiple places in the code. Moreover, through a centralized ontology, it is easier to enforce best practices. For example, if an organization decides that all entities must have `createdAt` and `updatedAt` fields (creation and modification time) for tracking, this is simply added to the BAB Core ontology as a mandatory property for every class that is a subclass of `Entity`. The transformer will then add appropriate fields and annotations (e.g., `JPA @PrePersist/@PreUpdate` methods) to all generated entities. Such a mechanism ensures standardization of architecture and design across different projects.

The stack-parametrized categorical construction also provides a formal basis for possible future support of additional technological environments. Extending the methodology to another target stack would require defining a corresponding stack-specific ontology and library, a conformance functor, a target category, and the associated generation templates, while leaving the BAB Core domain model and the mapping schema unchanged. Under the same verification conditions, such an implementation could be checked against the structural-preservation properties established in Section 5.5. However, the present work implements and evaluates only the Java-based BABj stack; no additional stack-specific ontology, library, or generator is claimed. The ontology and transformer do not participate in normal production execution; the generated application runs on the standard target libraries. Consequently, the methodology introduces no additional ontology-reasoning or transformation component at runtime. Runtime performance relative to an equivalent manually written application was not independently benchmarked in this study. The proposed approach differs from conventional MDE and OBDA approaches in both its ontological foundation and its categorical formalization. OWL 2 DL reasoning is used to validate the input model before generation, while the complete generation pipeline is formalized as the composite $T_s = H_s \circ F_s$. Natural transformations do not represent the entire model-to-code process; they express coherent refinements between mapping strategies. Under the verification conditions stated in Section 5.6, the composite pipeline preserves the selected schema-level structure and explicitly represented mapping distinctions, without claiming operational or denotational semantic equivalence between the ontology and the generated code. Compared to OBDA approaches (e.g., Ontop), which use ontology as an interface for executing queries over databases without generating code [16], this work generates static code in advance. Ontological knowledge is not used just as an abstraction for queries, but is directly implemented in software, enabling deeper integration of the conceptual layer and better control over system performance and behavior in specific contexts. It is useful to separate two kinds of benefit. Benefits shared with model-driven and low-code approaches in general include acceleration of initial development, architectural consistency, elimination of a class of manual errors, and easier maintenance; these follow from the model-driven nature of the method and are not, by themselves, distinctive to this work. Benefits distinctive to the proposed method are: (1) a formal, verifiable guarantee of schema-level structural preservation, expressed as the conditional theorem of Section 5.5 and reduced to the five verification conditions of Section 5.6; (2) formalization of a stack-parametrized generation mechanism as a family of evaluation bifunctors with provable functoriality, providing a formal extension point through which support for another platform would require a new conformance functor and target category while leaving the domain model unchanged; the present operational implementation of this mechanism is limited to BABj; and (3) the ability to encode selected semantic constraints, such as cardinalities and disjointness, into the validation that gates generation. The first two, in particular, are not provided with formal guarantees by typical MDD or low-code frameworks. Although the proposed approach offers a number of advantages, there are important limitations that should be explicitly stated. First, complete generation of all application aspects is not always achievable. Certain business rules and procedural logic often exceed the expressive power of ontologies based on description logic and require additional interventions. This challenge is solved by introducing clearly defined "hook" points in the generated code, intended for manual extension. Approaches such

as [59] show that knowledge-representation models can drive the generation of domain logic; integrating such behavioral synthesis into the hook points of our pipeline is a concrete direction for future work. Second, the success of the entire process largely depends on the quality of the input ontology, which requires expertise in domain knowledge and formal modeling and significant initial effort in creating the input ontology. However, that effort pays off in the long term through reusability and automated maintenance and opens space for further research aiming to delegate at least part of the tasks occurring in the generation process to machines. Third, synchronization between ontology and code in the case of manual code changes is a significant engineering challenge that exceeds the scope of classic one-way transformation. Since the proposed methodology does not offer an appropriate mechanism so far, recommended practice implies retaining primary logic within the ontology, with strict isolation of necessary manual interventions in the code, thereby ensuring the predictability and safety of the regeneration process. Fourth, the question of dynamic system behavior is treated partially. Ontologies are currently used dominantly for modeling structure and do not in themselves support modeling temporal changes, object lifecycles, or states. Therefore, within the proposed system, such aspects are expressed via specific software interfaces. These interfaces allow linking the declarative model with procedural conditions depending on the current system state, enabling the processing of dynamic aspects without directly changing the ontological model. In this way, a balance between formal semantics and execution flexibility is achieved. Fifth, unlike OBDA approaches that allow flexible execution of queries over heterogeneous sources in real-time, here static code adapted to a predefined context is generated. Although this means less flexibility at runtime, advantages include better performance, easier integration into existing applications, and elimination of the need for reasoners in the production environment. The research combines ontological semantic modeling, categorical formalization, and software engineering. The code-generation pipeline is formalized as the composition $T_s = H_s \circ F_s$, while natural transformations express coherent refinements between mapping strategies. In this way, ad hoc model translation is replaced by a formally specified and conditionally verified transformation framework.

8. Conclusion

This paper presents a methodology for program code generation in which an ontology serves as a formal domain model and categorical transformations project that model into a platform-specific realization. A central formal contribution is the demonstration that the ImplementationMapping mechanism, which ontologically encodes generation rules within the model, gives rise to a family of evaluation bifunctors parametrized by a target technology stack. This result, grounded in the product-exponential adjunction in Cat, provides a formal stack-parametrized construction under which each instantiated target stack satisfying the stated verification conditions preserves the selected schema-level structure. The operational implementation and evaluation presented in this paper instantiate this construction only for the Java-based BABj stack. Through the multi-phase transformation from BAB Core, through BABj, to concrete Java source code, the paper demonstrates how a high-level description of a business system can be translated into working software. The key idea is to formalize the mappings between model levels as functors, with natural transformations ensuring coherence between mapping refinements; under the verification

conditions stated in Section 5.6, the generated code preserves the selected schema-level structure and explicitly represented mapping distinctions of the source ontology. The proposed solution is a step forward towards automation of software development, combining the power of semantic modeling and category theory with the practical requirements of software engineering. The result is a methodology that is both formal and practical: formal because it makes schema-level structural preservation a verifiable, conditional property, and practical because its BABj implementation supports the development of complex Java-based information systems. Development speed is increased by eliminating repetitive coding tasks; engineers can focus on domain design, while the transformer handles routine implementation. Automatic adherence to proven patterns and architectural principles contributes to improving code quality, which reduces technical debt and facilitates maintenance. The stack-parametrized construction applies uniformly to every target stack for which the corresponding conformance functor, target category, generation functor, and verification conditions are provided. Consequently, the same domain ontology can be reused across different current or future target stacks without modifying its platform-independent structure. BABj is the presently implemented and empirically evaluated instantiation of this general construction. The paper states clearly differentiation of the proposed methodology from existing MDE directions and OBDA approaches. Furthermore, limitations of the proposed solution have been identified and explicitly stated. Future research plans include research tasks aimed at mitigating and, where possible, eliminating the identified shortcomings. In this regard, short-term tasks encompass improvements aimed at enhancing semantic checks and optimizations, as well as improvements related to model and code synchronization. More specifically, one task is the use of a reasoner over the ontology in such a way that, in addition to checking model consistency before code generation, redundant entities or relations can be detected, inheritance automatically introduced when meaningful, etc. Another important research task that is currently in our focus is a two-way synchronization of a model and code: possibilities will be explored for reflecting code changes (made by programmers during fine-tuning) back into the ontology, thereby maintaining the source of truth in the model. In fact, research is actively being conducted on operationalizing the theoretical concept of adjunction between functors I_s and J_s , with an aim to establish a formally verified feedback loop whereby changes in code would consistently update the ontological model. Two other long-term research tasks that we see are exploring the possibilities of richer behavioral modeling and exploring the possibilities of automating ontology generation, which could potentially pave the way for applying the methodology to refactoring existing code. Finally, continuous and comprehensive performance evaluation is planned through the application of the approach to different domains and larger systems. We expect scalability to prove good, as ontologies can model even very complex domains, while adding new patterns to the transformer does not complicate existing ones. Availability of artifacts. The artifacts supporting this work (the BAB Core and BABj ontologies, the BABj library, the FreeMarker templates, and the BABGen prototype) are maintained in a GitHub repository. As the work is part of ongoing doctoral research and is subject to commercial exploitation, the repository is not publicly released; access is granted by the authors on reasonable request for academic evaluation and verification.

Acknowledgments. This research has been supported by the Ministry of Science, Technological Development and Innovation of the Republic of Serbia (Contract No. 451-

03-34/2026-03/200156) and the Faculty of Technical Sciences, University of Novi Sad through project “Scientific and Artistic Research Work of Researchers in Teaching and Associate Positions at the Faculty of Technical Sciences, University of Novi Sad 2026” (No. 01-3609/1).

References

1. Abouzahra, A., Sabraoui, A., Afdel, K.: Model composition in model driven engineering: A systematic literature review. *Information and Software Technology* 125, 106316 (2020)
2. Amr, M.F., Benmoussa, N., Mansouri, K., Qbadou, M.: Transformation of the CIM model into a PIM model according to the MDA approach for application interoperability: Case of the “COVID-19 patient management” business process. *International Journal of Online and Biomedical Engineering* 17(5), 49–68 (2021)
3. Asim, M.N., Wasim, M., Khan, M.U.G., Mahmood, W., Abbasi, H.M.: A survey of ontology learning techniques and applications. *Database* 2018, bay101 (Oct 2018)
4. Atkinson, C., Kühne, T.: Model-driven development: a metamodeling foundation. *IEEE Software* 20(5), 36–41 (2003)
5. Atzeni, M., Atzori, M.: CodeOntology: RDF-ization of source code. In: *The Semantic Web – ISWC 2017*. Cham (2017)
6. Berners-Lee, T.: The Semantic Web road map. W3C (Oct 1998), <https://www.w3.org/DesignIssues/Semantic.html>, accessed: 29 November 2025
7. Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H.F., Secret, A.: The World-Wide Web. *Communications of the ACM* 37(8), 76–82 (1994)
8. Bhogal, J., Macfarlane, A., Smith, P.W.H.: A review of ontology based query expansion. *Information Processing & Management* 43(4), 866–886 (Jul 2007)
9. Bogdanović, B., Konjović, Z., Obradović, Đ.: BAB (Business Application Builder) framework for rapid development of business information systems. In: *Proceedings of the International Scientific Conference – Sinteza 2023*. Belgrade (2023)
10. Bogdanović, B., Marković, V., Segedinac, M., Konjović, Z.: Ontological modelling and reasoning for the abstractHome class of the BAB framework for PAIS systems. In: *Proceedings of the International Scientific Conference – Sinteza 2025*. Belgrade (2025)
11. Bogdanović, B., Obradović, Đ., Segedinac, M., Konjović, Z.: BAB framework – towards an extensible software platform for AI-augmented process aware business information systems. In: *Disruptive Information Technologies for a Smart Society. ICIST 2024. Lecture Notes in Networks and Systems* (2024)
12. Boström, P., Neovius, M., Oliver, I., Waldén, M.: Formal transformation of platform independent models into platform specific models. In: Julliand, J., Kouchnarenko, O. (eds.) *Formal Specification and Development in B. B 2007. Lecture Notes in Computer Science*, vol. 4355, pp. 186–200. Springer, Berlin, Heidelberg (2006)
13. Brookes, B.C.: The foundations of information science. Part I. philosophical aspects. *Journal of Information Science* 2(3–4) (1980)
14. Calegari, D., Szasz, N.: Verification of model transformations: A survey of the state-of-the-art. *Electronic Notes in Theoretical Computer Science* 292, 5–25 (Mar 2013)
15. Calero, C., Ruiz, F., Piattini, M. (eds.): *Ontologies for Software Engineering and Software Technology*. Springer, Berlin, Heidelberg (2006)
16. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rodriguez-Muro, M., Rosati, R.: Ontologies and databases: The DL-Lite approach. In: Tessaris, S., Franconi, E., Eiter, T., Gutierrez, C., Handschuh, S., Rousset, M., Schmidt, R.A. (eds.) *Reasoning Web. Semantic Technologies for Information Systems. Reasoning Web 2009, Lecture Notes in Computer Science*, vol. 5689, pp. 255–356. Springer, Berlin, Heidelberg (2009)

17. Chen, J., Mashkova, O., Zhapa-Camacho, F., Hoehndorf, R., He, Y., Horrocks, I.: Ontology embedding: A survey of methods, applications and resources. *IEEE Transactions on Knowledge and Data Engineering* 37(7), 4193–4212 (Apr 2025)
18. Corcho, O., Priyatna, F., Chaves-Fraga, D.: Towards a new generation of ontology based data access. *Semantic Web* 11(1), 153–160 (2020)
19. Cranefield, S., Pan, J.: Bridging the gap between the model-driven architecture and ontology engineering. *International Journal of Human-Computer Studies* 65(7), 595–609 (2007)
20. Dávid, I., Aslam, K., Malavolta, I., Lago, P.: Collaborative model-driven software engineering – a systematic survey of practices and needs in industry. *Journal of Systems and Software* 199(C), 111626 (2023)
21. Decker, S., Melnik, S., van Harmelen, F., Fensel, D., Klein, M.C.A., Broekstra, J., Erdmann, M., Horrocks, I.: The Semantic Web: the roles of XML and RDF. *IEEE Internet Computing* 4(5), 63–73 (2000)
22. Diskin, Z., Maibaum, T.: Category theory and model-driven engineering: From formal semantics to design patterns and beyond. In: *Proceedings of ACCAT 2012* (2012)
23. Dudáš, M., Lohmann, S., Svátek, V., Pavlov, D.: Ontology visualization methods and tools: A survey of the state of the art. *The Knowledge Engineering Review* 33, e10 (2018)
24. Dumas, M., van der Aalst, W.M., ter Hofstede, A.H. (eds.): *Process-Aware Information Systems: Bridging People and Software through Process Technology*. John Wiley & Sons, Inc. (2005)
25. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: *Fundamentals of Business Process Management*. Springer, Berlin, Heidelberg (2018)
26. Espinoza-Arias, P., Garijo, D., Corcho, O.: Crossing the chasm between ontology engineering and application development: A survey. *Journal of Web Semantics* 100655 (Jul 2021)
27. Evermann, J., Wand, Y.: Towards ontologically based semantics for UML constructs. In: *Conceptual Modeling – ER 2001*. *Lecture Notes in Computer Science* (2001)
28. Flahive, A., Taniar, D., Rahayu, W.: Ontology as a service (OaaS): extracting and replacing sub-ontologies on the cloud. *Cluster Computing* 16(4), 947–960 (Dec 2013)
29. Fonseca, C.M., Sales, T.P., Viola, V., da Fonseca, L.B.R., Guizzardi, G., Almeida, J.P.A.: Ontology-driven conceptual modeling as a service. In: *FOMI 2021: 11th International Workshop on Formal Ontologies meet Industry*, held at JOWO 2021: Episode VII The Bolzano Summer of Knowledge, September 11–18. Bolzano, Italy (2021)
30. Gangemi, A., Catenacci, C., Ciaramita, M., Lehmann, J.: Modelling ontology evaluation and validation. In: *The Semantic Web: Research and Applications*. *ESWC 2006*. *Lecture Notes in Computer Science* (2006)
31. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: HermiT: An OWL 2 reasoner. *Journal of Automated Reasoning* 53(3), 245–269 (2014)
32. Goseva-Popstojanova, K., Kahsai, T., Knudson, M., Kyanko, T., Nkwocha, N., Schumann, J.: Report: Survey on model-based software engineering and auto-generated code (2016)
33. Grambow, G., Oberhauser, R., Reichert, M. (eds.): *Advances in Intelligent Process-Aware Information Systems – Concepts, Methods, and Technologies*. Springer, Cham (2017)
34. Henderson-Sellers, B.: Bridging metamodels and ontologies in software engineering. *Journal of Systems and Software* 84(2), 301–313 (Feb 2011)
35. Hitzler, P., Krötzsch, M., Rudolph, S.: *Foundations of Semantic Web Technologies*. Chapman and Hall/CRC (2009)
36. Hoseini, S., Theissen-Lipp, J., Quix, C.: A survey on semantic data management as intersection of ontology-based data access, semantic modeling and data lakes. *Journal of Web Semantics* 100819 (Jul 2024)
37. Jones, C.B.: *Software Development: A Rigorous Approach*. Prentice/Hall International (1980)
38. Kahani, N., Bagherzadeh, M., Cordy, J.R., Dingel, J., Varró, D.: Survey and classification of model transformation tools. *Software & Systems Modeling* 18, 2361–2397 (2019)

39. Kherraf, S., Lefebvre, É., Suryn, W.: Transformation from CIM to PIM using patterns and archetypes. In: ASWEC '08: Proceedings of the 19th Australian Conference on Software Engineering (2008)
40. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley (2003)
41. KOMMA Project: KOMMA – RDF object mapper and editing framework for Java. Fraunhofer IWU, <https://komma.enilink.net/>, accessed: 23 November 2025
42. Kovalyov, S.P.: Category-theoretic approach to software systems design. *Journal of Mathematical Sciences* 214(6), 814–853 (2016)
43. Lano, K., Kolaoudou-Rahimi, S., Yassipour-Tehrani, S., Sharbaf, M.: A survey of model transformation design patterns in practice. *Journal of Systems and Software* 140, 48–73 (2018)
44. Lassila, O., van Harmelen, F., Horrocks, I., Hendler, J., McGuinness, D.L.: The Semantic Web and its languages. *IEEE Intelligent Systems and their Applications* 15(6), 67–73 (2000)
45. Lehner, D., Zhang, J., Pfeiffer, J., Sint, S., Splettstößer, A.K., Wimmer, M., Wortmann, A.: Model-driven engineering for digital twins: a systematic mapping study. *Software and Systems Modeling* 24, 1339–1377 (Oct 2025)
46. Li, J., Garijo, D., Poveda-Villalón, M.: Large language models for ontology engineering: A systematic literature review (Apr 2025), <https://www.semantic-web-journal.net/system/files/swj3864.pdf>, accessed: 9 January 2026
47. Mac Lane, S.: *Categories for the Working Mathematician*. Springer, New York (1978)
48. Marcén, A.C., Iglesias, A., Lapeña, R., Pérez, F., Cetina, C.: A systematic literature review of model-driven engineering using machine learning. *IEEE Transactions on Software Engineering* 50(9), 2269–2293 (Jul 2024)
49. Michael, J., Cleophas, L., Zschaler, S., Clark, T., Combemale, B., Godfrey, T., Khelladi, D.E., Kulkarni, V., Lehner, D., Rumpe, B., Wimmer, M., Wortmann, A., Ali, S., Barn, B., Barosan, I., Bencomo, N., Bordeleau, F.: Model-driven engineering for digital twins: Opportunities and challenges. *Systems Engineering* 28(5), 659–670 (Apr 2025)
50. Miller, J., Mukerji, J.: MDA guide version 1.0.1. Tech. rep., Object Management Group (May 2003), <https://www.semanticscholar.org/paper/MDA-Guide-Version-1.0.1-Miller-/>, accessed: 29 November 2025
51. Mkhinini, M.M., Narsis, O.L., Nicolle, C.: Combining UML and ontology: An exploratory survey. *Computer Science Review* 35(1), 100223 (2020)
52. Mordecai, Y., Fairbanks, J.P., Crawley, E.F.: Category-theoretic formulation of the model-based systems architecting cognitive-computational cycle. *Applied Sciences* 11(4), 1945 (2021)
53. Noguera, M., Hurtado, M.V., Rodríguez, M.L., Chung, L., Garrido, J.L.: Ontology-driven analysis of UML-based collaborative processes using OWL-DL and CPN. *Science of Computer Programming* 75(8), 726–760 (Aug 2010)
54. Oberle, D., Grimm, S., Staab, S.: An ontology for software. In: *Handbook on Ontologies*, pp. 383–402. Springer, Berlin, Heidelberg (2009)
55. Olszewska, J.I., Allison, I.K.: ODYSSEY: Software development life cycle ontology. In: *Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2018)* (2018)
56. OpenRDF: AliBaba. Bitbucket, <https://bitbucket.org/openrdf/alibaba/src/master/>, accessed: 23 November 2025
57. Pan, J.Z., Staab, S., Aßmann, U., Ebert, J., Zhao, Y., Oberle, D.: Conclusion and outlooks. In: Aßmann, U., Ebert, J., Pan, J.Z., Staab, S., Zhao, Y. (eds.) *Ontology-Driven Software Development*, pp. 319–322. Springer, Berlin, Heidelberg (2013)
58. Ruiz, F., Hilera, J.R.: Using ontologies in software engineering and technology. In: Calero, C., Ruiz, F., Piattini, M. (eds.) *Ontologies for Software Engineering and Software Technology*, pp. 49–102. Springer, Berlin, Heidelberg (2006)
59. Rybiński, K., Śmiałek, M.: Low-code development using requirements and knowledge representation models. *Computer Science and Information Systems* 21(3), 685–724 (2024)

60. Sapel, P., Comet, L.M., Dimitriadis, I., Hopmann, C., Decker, S.: A review and classification of manufacturing ontologies. *Journal of Intelligent Manufacturing* 36, 3669–3693 (2025)
61. Shimizu, C., Hitzler, P.A.: Accelerating knowledge graph and ontology engineering with large language models. *Journal of Web Semantics* 85, 100862 (May 2025)
62. Silega, N., Noguera, M., Macias, D.: Ontology-based transformation from CIM to PIM. *IEEE Latin America Transactions* 14(9), 4156–4165 (Dec 2016)
63. Silega, N., Noguera, M., Rogozov, Y.I., Lapshin, V.S., González, T.: Transformation from CIM to PIM: A systematic mapping. *IEEE Access* 10, 90857–90872 (Aug 2022)
64. Singh, Y., Sood, M.: Transformation of platform-independent model into platform-specific model in model-driven architecture (2013)
65. Staab, S., Walter, T., Gröner, G., Parreiras, F.S.: Model driven engineering with ontology technologies. In: *Reasoning Web. Semantic Technologies for Software Engineering*, pp. 62–98. Springer, Berlin, Heidelberg (2010)
66. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley Professional, 2 edn. (2008)
67. Syriani, E., Luhunu, L., Sahraoui, H.: Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures* 52, 43–62 (Jun 2018)
68. Trojahn, C., Vieira, R., Schmidt, D., Pease, A., Guizzardi, G.: Foundational ontologies meet ontology matching: A survey. *Semantic Web* 13(4), 685–704 (2022)
69. Vickery, B.C.: Ontologies. *Journal of Information Science* 23(4), 277–286 (1997)
70. Völkel, M., Sure, Y.: RDFReactor – from ontologies to programmatic data access. In: *4th International Semantic Web Conference (ISWC)*. Galway, Ireland (2005)
71. Zhu, M., Grogono, P., Ormandjieva, O.: Using category theory to verify implementation against design in concurrent systems. *Procedia Computer Science* 52, 530–537 (2015)

Borivoj Bogdanović holds an M.Sc. degree and is currently a Ph.D. student in Computer Science at Singidunum University. He is also a senior software engineer with extensive experience in the development of business information systems. His research interests include software engineering, semantic technologies, ontology-driven and model-driven development, process-aware information systems, and AI-augmented business systems.

Vidan Marković is an Associate Professor in the field of Computer Science at Singidunum University. He holds a PhD in Computer Science from the University of Novi Sad. His research and professional interests include software engineering, software quality management and improvement, information systems interoperability, machine learning, the application of information technologies, and the management of complex software projects. He is a professional project manager with extensive experience in leading complex IT and digital transformation projects. He is the author and co-author of numerous scientific papers published in international journals and presented at scientific conferences.

Milan Segedinac received his M.Sc. degree in 2008 and Ph.D. in 2014 in Computer Science from the University of Novi Sad, Faculty of Technical Sciences. He holds a full professor position at the same faculty. He has authored papers in international and national journals and conferences in the field of Artificial Intelligence.

Zora Konjović, PhD, is a graduate mathematician with a PhD in robotics obtained at the Faculty of Technical Sciences of the University of Novi Sad. She is a retired full professor in Applied Computer Science and Informatics at the Faculty of Technical Sciences,

University of Novi Sad, as well as in Informatics and Computer Science at Singidunum University in Belgrade. Her primary areas of scientific and professional interest include software engineering, artificial intelligence, semantic technologies, technology-enhanced learning, and mathematical modeling. Throughout her long academic and research career, Professor Konjović has published more than 200 scientific papers and has successfully managed dozens of national and international scientific projects.

Received: February 4, 2026; Accepted: August 15, 2026.