

Pan-European University, Faculty of Informatics

Interim Research Report

Translation of subset of SQL to a low-code language based on object-centric processes

of the project

Requirements and formal definition of a low-code language based on object-centric processes - LowcodeOCP

Authors:

Gabriel Juhás, Milan Mladoniczky, Juraj Mažári, Tomáš Kováčik and Michal Drobny

Contact:

gabriel.juhas@paneurouni.com, milan.mladoniczky@paneurouni.com

Funded by the EU NextGenerationEU through the Recovery and Resilience Plan for Slovakia
under the project No.

09I03-03-V04-00493

August 31, 2026

Version: v0.1

Abstract

Data-driven business process management requires a seamless integration between data structures and process behavior. Process-Centric Classes (PCC) in Petriflow language provide an object-oriented, Petri net based paradigm where data schemas and behavioral lifecycles are unified into executable object classes. However, designing PCCs from scratch for existing enterprise systems can be time consuming and error prone. In this paper, we present an approach for automatically deriving executable Process-Centric Classes directly from relational data models, including a subset of SQL DDL. By mapping relational tables, primary/foreign key relationships, and integrity constraints to PCC data structures, and generating baseline lifecycle Petri nets for entity states, our method transforms static database schemas into fully executable, process-centric object specifications. We demonstrate the applicability of our approach through a concrete implementation in the Petriflow framework and show how the derived PCCs preserve data consistency while enabling dynamic process execution.

Contents

Contents	i
List of Figures	iii
List of Tables	iv
1 Introduction	1
2 Related Work	3
2.1 Schema Transformation and Model-Driven Engineering	3
2.2 Data-Aware and Object-Centric Petri Nets	3
2.3 Process-Centric Classes and Petriflow	4
2.4 Summary and Positioning	4
2.5 Interplay between data lifecycles (CRUD operations) and relational constraints	4
3 Structural Mapping: From Entities and Attributes to Process-Centric Classes	6
3.1 Primitive Attribute Mapping	6
3.2 Abstract Data Schema Representation	7
3.3 Scope of Static Mapping	7
4 Deriving Relational Lifecycles and Event-Synchronized Protocols	8
4.1 DDL-Grounded Derivation of Relational Protocols	8
4.2 Architectural Paradigm: Direct Pointer Symmetry	9
4.3 Methodological Assumptions: Metadata Resolution and FK Ownership .	9
4.4 Operationalization via Invariant Universal Net Archetype	10
4.5 Terminology and Conceptual Equivalence	11
4.6 Working Data Variables	13
4.7 Arc Weight Variables	13
4.8 Evaluation Vectors for Concurrently Fired Branches	14
4.8.1 Active Branch Evaluation Vectors	14

4.8.2	Passive Branch Evaluation Vectors	14
4.9	Protocol Eligibility per Active Execution Vector	15
4.9.1	Protocol Eligibility per Passive Execution Vector	16
4.10	Orthogonality and Independent Scenario Analysis	17
4.11	Common Execution Prefix: Data Initialization, Read, and Update	17
4.12	Scenario A1: No Active Relations	19
4.13	Scenario A2: Active Optional Only	19
4.14	Data Visibility and Composite Full Case View	22
4.15	Scenario A3: Multiple Active Mandatory	22
4.16	Scenario A4: Single Active Mandatory	23
4.16.1	Read Consistency	24
4.16.2	Implementation of the Unbinding Interface	24
4.17	Scenario P1: No Passive Relations	25
4.18	Scenario P2: Passive Optional Only	26
4.18.1	Universal Reactive Reference Management via update_relation	26
4.19	Scenario P3: Passive Mandatory	26
4.19.1	Lifecycle State Transitions in P_3	28
4.20	Processing Incoming Requests on Removal on Active Endpoints	28
4.21	Symmetric Handling of M:N Relations	29
4.22	Scenario D: Entity Deletion and Cascade Propagation	30
5	Conclusion	31
5.1	Future Work	32

List of Figures

4.1	Universal Net Archetype (UNA) invariant Petri net structure.	12
-----	--	----

List of Tables

3.1	Mapping Rules from Relational/Object Entities to Process-Centric Classes (PCC).	7
4.1	Classification of Relational Protocols between Active and Passive Entities	10

Chapter 1

Introduction

Modern enterprise information systems rely heavily on relational database management systems (RDBMS) or structural conceptual models to persist and manage core business data. While these structural models - whether represented as SQL DDL scripts, Entity-Relationship (ER) diagrams, or UML Class diagrams - excel at enforcing structural integrity and domain constraints, they inherently lack an explicit representation of business process logic. Conversely, traditional process modeling paradigms focus primarily on control-flow constraints, often treating data as secondary or external to the process execution context.

To bridge this gap between data structures and process behavior, the concept of Process-Centric Classes (PCC) was introduced. PCCs offer an object-oriented paradigm where data schemas, state spaces, behavioral lifecycles and user interfaces are unified into executable object classes. Implemented in frameworks such as Petriflow, PCCs treat business objects as active entities driven by Petri net semantics. However, migrating existing systems or creating new process-centric applications often requires manual design of these classes from scratch — a process that is time consuming, prone to human error, and frequently duplicates information already captured in structural schemas.

In this paper, we address this challenge by proposing an automated transformation approach for deriving executable Process-Centric Classes directly from structural data specifications (such as SQL DDL, ER, or Class diagrams). While mapping the static data structures themselves is straightforward and follows established transformations, the core innovation of our work lies in the synthesis of relational behavior and lifecycle dynamics. Using event synchronization mechanisms in Petriflow, our approach enables the automatic generation of process lifecycles that dynamically handle the creation, modification, and deletion of referenced relationships, as well as the propagation of lifecycle state changes across linked objects.

The main contributions of this work are threefold:

- A unified mapping scheme translating structural data specifications (SQL DDL, ER, UML) into Process-Centric Class data attributes and object references.

- A pattern-based method leveraging event synchronization in Petriflow to synthesize executable lifecycles that manage relational dependencies and cross-object state transitions.
- A validation of the approach through a concrete implementation within the Petriflow framework.

The remainder of this paper is structured as follows: Section 2 reviews related work in schema transformation and artifact-centric process modeling. Section 3 formalizes the structural mapping rules from data models to PCCs. Section 4 presents the core lifecycle generation method and event synchronization semantics. Section 5 concludes the paper with directions for future work.

Chapter 2

Related Work

The challenge of bridging structural data specifications and process behavior has been actively studied across several areas of business process management (BPM) and software engineering. Our work sits at the intersection of schema transformation, data-aware process modeling, and object-centric process paradigms.

2.1 Schema Transformation and Model-Driven Engineering

Transforming conceptual data models, such as Entity-Relationship (ER) models, UML Class diagrams, or SQL DDL, into executable target structures is a cornerstone of Model-Driven Engineering (MDE) [BN12]. Traditional approaches focus primarily on generating static code artifacts, object-relational mapping (ORM) layers, or relational schemas. While these techniques preserve structural integrity and foreign key constraints, they leave the behavioral lifecycle of data entities unmodeled or embedded within external application code. Our approach leverages these established static mappings for attributes and references, but extends them by automatically synthesizing dynamic process lifecycles directly from structural relations.

2.2 Data-Aware and Object-Centric Petri Nets

To incorporate data into formal process models, various extended Petri net formalisms have been proposed, including Colored Petri Nets (CPN), Object-Centric Petri Nets (OCPN), Petri Nets with Identifiers (PNID), and Reference Nets (nets-within-nets). These high-level formalisms typically model a centralized run-time environment. In essence, they represent the entire system as a single, global Petri net where complex data structures, identifiers, or nested net instances flow as tokens through a shared control structure.

While these approaches excel at descriptive modeling, and are frequently utilized in process mining to capture complex, multi-entity system traces, they operate at a macro-level. Metaphorically, a High-Level Petri net models the entire ecosystem or forest, capturing how trees, animals, and resources dynamically interact within one overarching environment. However, constructing such unified, global models upfront for execution is complex and hard to maintain.

2.3 Process-Centric Classes and Petriflow

In contrast to global, system-wide Petri nets, the concept of Process-Centric Classes implemented within the Petriflow framework shifts the modeling paradigm from the global environment to the individual entity. Instead of modeling the entire forest within a single net, PCCs define the "DNA" of individual objects (the specific trees and animals), unifying their structural attributes, state spaces, user interfaces, and behavioral lifecycles into modular, reusable class definitions. Each object instance executes its own local Petri net lifecycle while dynamically interacting with other instances.

While Petriflow provides powerful event synchronization mechanisms, allowing actions in one PCC instance to trigger state changes or reference updates in linked instances, designing these synchronized lifecycles manually from scratch remains complex and error prone.

2.4 Summary and Positioning

Existing works either focus strictly on static data transformation (MDE/ORM) or rely on complex, global high-level Petri nets for multi-entity behaviors. The main contribution of our work is bridging this gap: we take established structural data specifications (SQL DDL, ER, Class diagrams) as input and automatically derive modular, executable Petriflow Process-Centric Classes. By using event synchronization, our approach natively enforces relational lifecycle dynamics and manages relationship lifecycles at the entity level without requiring a single, centralized system net.

2.5 Interplay between data lifecycles (CRUD operations) and relational constraints

The interplay between data lifecycles (CRUD operations) and relational constraints has been extensively studied in data-centric process management paradigms.

Artifact-Centric Business Processes [Hul08] pioneered the coupling of data state transitions with process routing. However, operationalizing fine-grained cardinality bounds

during relational updates often relies on custom scripting rather than standardized protocol patterns.

Frameworks such as *PHILharmonicFlows* [KR11] specifically target object-aware process management, analyzing how $1 : 1$, $1 : N$, and $N : M$ cardinalities enforce execution semantics. Our work extends these concepts by establishing a formal taxonomy of 11 relational protocols ($P^{\text{passive, active}}$) and introducing explicit unbinding mechanics (`unbind_relations`) to guarantee state consistency in mandatory single-target dependencies without race conditions.

Chapter 3

Structural Mapping: From Entities and Attributes to Process-Centric Classes

The first step of our transformation focuses on converting the static entity structures and scalar attributes from the source schema (such as SQL DDL, ER models, or UML Class diagrams) into Process-Centric Class data containers. Formally, we consider an input schema $S = (E, R)$, where E represents the set of entities (tables/classes) and R represents the set of (binary) relationships (foreign keys/associations) between entities.

In this section, we address the direct mapping of entities $e \in E$ and their primitive columns. The operational logic, relational attributes, and reference management associated with R are handled dynamically within the lifecycle synthesis in Section 4.

3.1 Primitive Attribute Mapping

For each entity $e \in E$, a corresponding Process-Centric Class PCC_e is instantiated. Primitive columns and scalar attributes defined in the source data model translate directly into data variables within the PCC specification:

- **Primitive Data Types:** Native database types (e.g., VARCHAR, INTEGER, BOOLEAN, TIMESTAMP) map directly to their corresponding Petriflow data types (`text`, `number`, `boolean`, `date`).
- **Nullability and Constraints:** Domain constraints like NOT NULL are mapped to required field validation rules in the PCC data schema.
- **Enumerations and Domain Sets:** Predefined enumeration domains translate into single-choice lookup variables, ensuring input validation at runtime.

3.2 Abstract Data Schema Representation

At the structural level, each derived PCC_e receives a set of primitive data fields $D_e = \{d_1, d_2, \dots, d_k\}$ corresponding to the non-relational columns of entity e . Table 3.1 summarizes this direct structural mapping.

Table 3.1: Mapping Rules from Relational/Object Entities to Process-Centric Classes (PCC).

Source Construct (SQL / UML)	PCC Data Field Target	Behavioral & Runtime Rule
Entity / Table ($e \in E$)	Process-Centric Class UNA_e	Parametrized Universal Net Archetype UNA
VARCHAR, TEXT, CHAR	text variable	Scalar text input field
INT, DECIMAL, FLOAT	number variable	Numeric input field with validation
BOOLEAN, BIT	boolean variable	Binary checkbox / toggle field
DATE, TIMESTAMP	date, dateTime variable	Temporal date picker
ENUM(...) / Domain Set	enumeration variable	Single-choice dropdown selection
NOT NULL constraint	<behavior> required </behavior>	Enforces mandatory input
Foreign Key / Association ($r \in R$)	Deferred to Section 4	Managed via case references and event triggering

3.3 Scope of Static Mapping

This preliminary mapping establishes a clean, static data schema for each object class. However, because relational schemas model connections via foreign key columns, treating relationships as simple static attributes is insufficient for process execution. In Petriflow, relationships are non-static: creating, modifying, or removing a link between entities requires updating object references dynamically across active process instances.

Therefore, rather than generating static reference fields in this structural phase, all foreign keys, association attributes, and relational dependencies (R) are mapped directly into the dynamic lifecycle nets and event synchronization mechanisms detailed in Section 4.

Chapter 4

Deriving Relational Lifecycles and Event-Synchronized Protocols

While structural schema transformations natively map primitive data attributes, operationalizing dynamic relationships requires formalizing how relational integrity constraints (min..max) translate into state-based process lifecycles.

In relational database design (SQL DDL), establishing binary links between tables yields a bounded set of physical schema representations. The directional placement of the **Foreign Key (FK)** or the introduction of an intermediary junction table determines which entity manages reference pointers and enforces operational constraints. Combining physical FK placement with lower-bound obligations ($\min \in \{0, 1\}$) across participating entities one can identify **11 fundamental relational protocols**.

4.1 DDL-Grounded Derivation of Relational Protocols

To formalize the relational space, we classify binary associations into single-cardinality ($1 : 1$), one-to-many ($1 : N$), and many-to-many ($N : M$) structural configurations:

1. **Single-Cardinality Patterns (1:1) - 4 Protocols:** Because a $1 : 1$ link requires physically placing the FK on one specific side (introducing directional asymmetry between the owning and target tables), identity permutation cannot be reduced. Combining physical FK placement with mandatory vs. optional lower bounds yields 4 distinct physical operational protocols:

- $0..1 \leftarrow 0..1$ (Optional FK, optional target)
- $1..1 \leftarrow 0..1$ (Mandatory FK, optional target)
- $0..1 \leftarrow 1..1$ (Optional FK, mandatory target)
- $1..1 \leftarrow 1..1$ (Mandatory FK, mandatory target)

2. **One-to-Many Patterns (1:N) - 4 Protocols:** Standard relational DDL dictates that the Foreign Key must reside strictly on the N -side (child entity). Evaluating optionality ($\min \in \{0, 1\}$) for both parent and child yields 4 distinct directional protocols:

- $0..1 \leftarrow 0..N$ (Optional parent, optional children)
- $1..1 \leftarrow 0..N$ (Mandatory parent, optional children)
- $0..1 \leftarrow 1..N$ (Optional parent, mandatory children)
- $1..1 \leftarrow 1..N$ (Mandatory parent, mandatory children)

3. **Many-to-Many Patterns (N:M) - 3 Protocols:** Relational DDL enforces relational symmetry by decomposing an $N : M$ link into an intermediary junction table containing two Foreign Keys. Evaluating the lower bounds across both sides yields 3 symmetrical constraint configurations:

- $0..N \leftrightarrow 0..N$ (Fully optional association)
- $1..N \leftrightarrow 0..N$ (Asymmetric mandatory association)
- $1..N \leftrightarrow 1..N$ (Symmetrically mandatory association)

4.2 Architectural Paradigm: Direct Pointer Symmetry

The proposed Process-Centric Class (PCC) framework preserves the exact operational semantics and asymmetry of the $4 + 4$ single and one-to-many SQL protocols. However, to eliminate runtime overhead and avoid generating auxiliary junction Petri nets, the architecture transforms the 3 junction-table $N : M$ patterns.

Rather than instantiating an intermediary junction net to achieve symmetry as in SQL DDL, the framework replaces junction-table symmetry with **direct vector symmetry**.

By leveraging native Petriflow array references (**caseRef** vector lists) directly within both participating process entities, $N : M$ relationships are managed symmetrically without middleman artifacts.

Consequently, all 11 SQL DDL relational protocols, alongside the baseline boundary case of an isolated entity possessing zero attached relations ($R = \emptyset$), are operationalized seamlessly on top of a single Universal Net Archetype.

4.3 Methodological Assumptions: Metadata Resolution and FK Ownership

Translating conceptual domain models into operational process architectures relies on two fundamental assumptions regarding metadata resolution:

1. Model-Driven Mandatory Extraction:

Extracting lower-bound constraints ($\min = 1$) for many-to-many relationships ($1..N \leftrightarrow 0..N$ or $1..N \leftrightarrow 1..N$) directly from relational database DDL is non-trivial, as physical junction tables lack native constructs for collection constraints. The presented framework assumes that metadata is ingested from a conceptual domain model (e.g., UML Class Diagram or ER model) where lower bounds are explicitly declared.

2. Explicit Foreign Key Ownership Assignment:

In abstract domain models, single-cardinality links ($1 : 1$) frequently omit explicit ownership indicators designating which participant holds the reference pointer.

Just as SQL DDL strictly requires choosing which table physically receives the Foreign Key column during database schema design, the PCC framework requires designating which process entity acts as the active owner (role = active) holding UI assignment controls. If this ownership is not explicitly specified in the high-level domain model, the presented framework deterministically selects one entity as the owner (e.g., via structural conventions), resolving any ambiguity.

By satisfying these assumptions, every attached relationship r_i in the conceptual model maps deterministically onto exactly one of the 11 operational DDL protocols.

Table 4.1: Classification of Relational Protocols between Active and Passive Entities

Protocol ID	Formal Name	Bounds ($e2_{\text{passive}} \leftarrow e1_{\text{active}}$)
$P_{1:1}^{OO}$	Fully Optional 1:1	$0..1 \leftarrow 0..1$
$P_{1:1}^{OM}$	Active-Mandatory 1:1	$0..1 \leftarrow 1..1$
$P_{1:1}^{MO}$	Passive-Mandatory 1:1	$1..1 \leftarrow 0..1$
$P_{1:1}^{MM}$	Fully Mandatory 1:1	$1..1 \leftarrow 1..1$
$P_{1:N}^{OO}$	Fully Optional 1:N	$0..1 \leftarrow 0..N$
$P_{1:N}^{OM}$	Active-Mandatory 1:N	$0..1 \leftarrow 1..N$
$P_{1:N}^{MO}$	Passive-Mandatory 1:N	$1..1 \leftarrow 0..N$
$P_{1:N}^{MM}$	Fully Mandatory 1:N	$1..1 \leftarrow 1..N$
Protocol ID	Formal Name	Bounds ($e2_{\text{active2}} \leftrightarrow e1_{\text{active1}}$)
$P_{N:M}^{OO}$	Fully Optional N:M	$0..N \leftrightarrow 0..N$
$P_{N:M}^{OM}$	Half-Mandatory N:M	$0..N \leftrightarrow 1..N$
$P_{N:M}^{MM}$	Fully Mandatory N:M	$1..N \leftrightarrow 1..N$

4.4 Operationalization via Invariant Universal Net Archetype

Every domain **entity** operationalizes its relational lifecycle behavior by translating into a single, invariant **Universal Net Archetype (UNA)**. The underlying Petri net graph

topology remains strictly identical across all generated **Petriflow classes**; structural and operational adaptation is achieved exclusively by compiling relation-specific data variables and transition execution actions into the invariant net structure.

The translation algorithm aggregates all active and passive relations belonging to an entity into a single process class. Depending on the cardinality and foreign-key ownership of each attached protocol P , the translation algorithm generates the corresponding operational roles within the invariant net:

1. **Asymmetric Protocols** ($1 : 1$ and $1 : N$): For a given relation, the protocol establishes a directional role interaction between the active initiator ($e1_{\text{active}}$) and the passive target ($e2_{\text{passive}}$), expressed via the bounds notation $e2_{\text{passive}} \leftarrow e1_{\text{active}}$. The active role in the initiating process **case** is translated with relation-specific UI selection inputs and link creation actions. The passive role in the target process **case** contains no primary selection UI for this relation, executing reactive event actions to register incoming reference updates.
2. **Symmetric Protocols** ($N : M$): Because $N : M$ relations allow bidirectional reference management and symmetric modification, the protocol generates a peer-to-peer active interaction ($e1_{\text{active1}}$ and $e2_{\text{active2}}$), denoted by the bi-directional bound syntax $e2_{\text{active}} \leftrightarrow e1_{\text{active}}$. In this configuration, both participating process **cases** are translated with active UI selection variables and event actions for their respective local end, while simultaneously executing reactive event actions to respond to counterparty mutations.

Participating process **cases** communicate asynchronously via event synchronization mechanisms embedded in the transition actions, exchanging case identifiers (`caseRef` / foreign keys) to update their internal reference vectors while executing their respective local lifecycles.

4.5 Terminology and Conceptual Equivalence

To simplify the description of the transformation pipeline, we group equivalent concepts across design-time and runtime layers:

- **Model-Level Synonyms (Static Abstraction):** Terms **Entity**, **Database Table**, **Petriflow Class**, and **Process** are used interchangeably to describe entities.
- **Runtime-Level Synonyms (Dynamic Execution):** Terms **Case**, **Object**, **Instance**, and **Record** are used interchangeably to denote a specific instance executed within the workflow engine alongside its corresponding data values in memory or persistence.

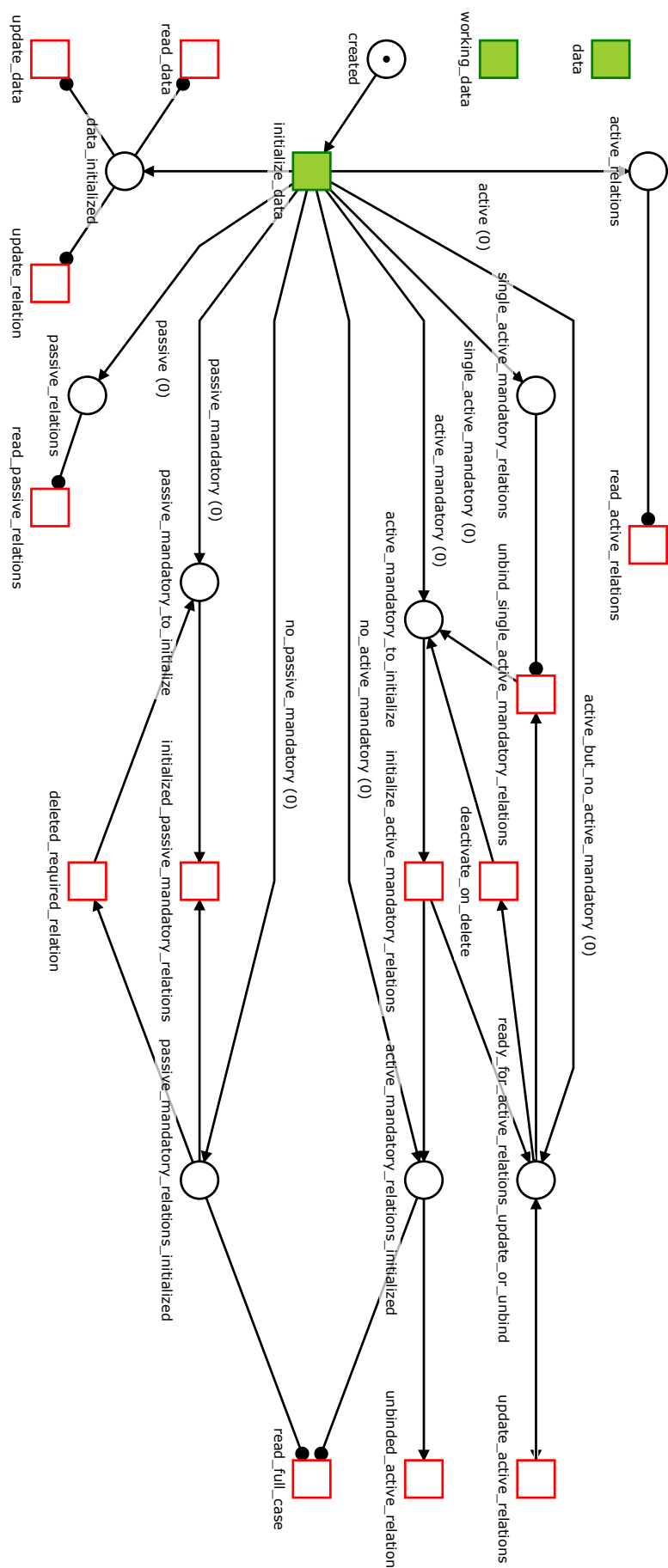


Figure 4.1: Universal Net Archetype (UNA) invariant Petri net structure.

- **Behavioral Synonyms (Activity Units):** Terms **Task** and **Transition** are used interchangeably to represent individual executable activities within a life-cycle process net.

4.6 Working Data Variables

To ensure data integrity during dynamic updates, each scalar attribute $d \in D$ is mapped alongside a corresponding working variable `working_d`. These working variables serve as temporary input buffers during user interaction in lifecycle tasks such as `initialize_data` or `update_data`. Upon successful task completion, the updated values from `working_d` are atomically committed to their corresponding target fields d .

4.7 Arc Weight Variables

To generate the execution logic within the translated **Petriflow class**, the generator derives 8 boolean structural variables for each domain **entity**. These variables quantify the existence and cardinality bounds of outgoing (active) and incoming (passive) relations. Because Petriflow arc conditions accept only single variables rather than evaluated expressions, these variables are not mutually independent. Consequently, explicitly declared redundant variables - including direct logical negations - are introduced to encode structural constraints onto arcs.

Active Subnet Arc Weight Variables

1. **active:** Evaluates to 1 if the domain entity owns at least one outgoing active relation ($|R_{\text{active}}| > 0$); otherwise 0.
2. **no_active_mandatory:** Evaluates to 1 if the domain entity has no mandatory active relations (i.e., all outgoing relations are optional $0..1$ or $0..N$, or $|R_{\text{active}}| = 0$); otherwise 0.
3. **active_but_no_active_mandatory:** Evaluates to 1 if active relations exist ($|R_{\text{active}}| > 0$), but none of them impose a mandatory constraint; otherwise 0.
4. **active_mandatory:** Negation of **no_active_mandatory**.
5. **single_active_mandatory:** Evaluates to 1 if at least one mandatory outgoing active relation follows an exclusive 1 : 1 cardinality constraint (Active-Mandatory 1:1 - $P_{1:1}^{OM}$ or Fully Mandatory 1:1 - $P_{1:1}^{MM}$); otherwise 0.

Passive Subnet Arc Weight Variables

1. **passive:** Evaluates to 1 if the domain entity is referenced by at least one incoming passive relation ($|R_{\text{passive}}| > 0$); otherwise 0.
2. **no_passive_mandatory:** Evaluates to 1 if the domain entity has no mandatory incoming passive relations (i.e., all incoming references are optional 0..1 or 0..N, or $|R_{\text{passive}}| = 0$); otherwise 0.
3. **passive_mandatory:** Negation of **no_passive_mandatory**.

4.8 Evaluation Vectors for Concurrently Fired Branches

Based on the 8 derived structural variables, the `initialize_data` transition evaluates two independent sets of mutually exclusive truth vectors (4 for the active subnet and 3 for the passive subnet), ordered by structural complexity:

4.8.1 Active Branch Evaluation Vectors

Defined over the tuple

(`active`, `no_active_mandatory`, `active_but_no_active_mandatory`, `active_mandatory`, `single_active_mandatory`):

1. **A1 [0, 1, 0, 0, 0] (No Active):** The domain entity defines no active relations; the process case bypasses active mandatory initialization.
2. **A2 [1, 1, 1, 0, 0] (Active Optional Only):** Active relations exist in the domain model, but all are optional; foreign-key assignment remains optional for the active object.
3. **A3 [1, 0, 0, 1, 0] (Multiple Active Mandatory):** Mandatory active relations exist following 1 : N or M : N cardinalities (the target case/object can be referenced non-exclusively by multiple active cases/objects).
4. **A4 [1, 0, 0, 1, 1] (Single Active Mandatory):** A mandatory 1 : 1 active relation exists (the target foreign-key field in the active object is required, and a single target case/object cannot be referenced by multiple active cases/objects).

4.8.2 Passive Branch Evaluation Vectors

Defined over the tuple

(`passive`, `no_passive_mandatory`, `passive_mandatory`):

1. **P1 [0, 1, 0] (No Passive):** The domain entity defines no incoming passive relations; the process case bypasses passive mandatory initialization.
2. **P2 [1, 1, 0] (Passive Optional Only):** Incoming passive relations exist in the domain model, but all are optional; the passive process case opens non-blocking transition paths for incoming references from active cases/objects.
3. **P3 [1, 0, 1] (Passive Mandatory):** At least one mandatory incoming reference exists (1..1 or 1..N bounds); the passive process case blocks initialization completion until it receives a valid binding notification signal from at least one initiating active case/object.

4.9 Protocol Eligibility per Active Execution Vector

To clearly relate the operational evaluation vectors (A_1 – A_4) to the relational protocols defined in Table 2, the presence of a specific vector dictates which active protocol configurations an entity is eligible to participate in:

- **Vector A1 [0, 1, 0, 0, 0] (No Active):** The entity owns no outgoing relations where it acts as the active initiator.
- **Vector A2 [1, 1, 1, 0, 0] (Active Optional Only):** The entity participates exclusively in relations with optional active bounds (0..1 or 0..N). It may participate as the active party in the following protocols:
 - $P_{1:1}^{OO}$ (Fully Optional 1:1)
 - $P_{1:1}^{MO}$ (Passive-Mandatory 1:1, active side optional 0..1)
 - $P_{1:N}^{OO}$ (Fully Optional 1:N)
 - $P_{1:N}^{MO}$ (Passive-Mandatory 1:N, active side optional 0..1)
 - $P_{N:M}^{OO}$ (Fully Optional N:M)
 - $P_{N:M}^{OM}$ (Half-Mandatory N:M, the entity is optional 0..N side in the relation)
- **Vector A3 [1, 0, 0, 1, 0] (Multiple Active Mandatory):** The entity contains at least one mandatory active relation without exclusive 1 : 1 constraints. It acts as the mandatory active party in:
 - $P_{1:N}^{OM}$ (Active-Mandatory 1:N)
 - $P_{1:N}^{MM}$ (Fully Mandatory 1:N)
 - $P_{N:M}^{OM}$ (Half-Mandatory N:M, the entity is mandatory 1..N side in the relation)
 - $P_{N:M}^{MM}$ (Fully Mandatory N:M)

- **Vector A4 [1, 0, 0, 1, 1] (Single Active Mandatory):** The entity contains at least one mandatory active relation requiring an exclusive target reference. It participates as the mandatory active party in:

- $P_{1:1}^{OM}$ (Active-Mandatory 1:1)
- $P_{1:1}^{MM}$ (Fully Mandatory 1:1)

4.9.1 Protocol Eligibility per Passive Execution Vector

Similarly, the derived passive evaluation vector (P_1-P_3) dictates which passive target roles an entity can fulfill based on incoming relational bounds:

- **Vector P1 [0, 1, 0] (No Passive):** The entity is not referenced by any external relations ($|R_{\text{passive}}| = 0$).
- **Vector P2 [1, 1, 0] (Passive Optional Only):** The entity acts as a passive target exclusively in relations with optional incoming bounds (0..1 or 0..N). It may participate as the passive counterparty in:

- $P_{1:1}^{OO}$ (Fully Optional 1:1)
- $P_{1:1}^{OM}$ (Active-Mandatory 1:1, passive side is optional 0..1)
- $P_{1:N}^{OO}$ (Fully Optional 1:N)
- $P_{1:N}^{OM}$ (Active-Mandatory 1:N, passive side is optional 0..1)
- $P_{N:M}^{OO}$ (Fully Optional N:M)
- $P_{N:M}^{OM}$ (Half-Mandatory N:M, the entity is optional 0..N side in the relation)

- **Vector P3 [1, 0, 1] (Passive Mandatory):** The entity requires at least one incoming reference to satisfy mandatory initialization bounds (1..1 or 1..N). It acts as the mandatory passive target in:

- $P_{1:1}^{MO}$ (Passive-Mandatory 1:1)
- $P_{1:1}^{MM}$ (Fully Mandatory 1:1)
- $P_{1:N}^{MO}$ (Passive-Mandatory 1:N)
- $P_{1:N}^{MM}$ (Fully Mandatory 1:N)
- $P_{N:M}^{MO}$ (Half-Mandatory N:M, the entity is mandatory 1..N side in the relation)
- $P_{N:M}^{MM}$ (Fully Mandatory N:M)

4.10 Orthogonality and Independent Scenario Analysis

A fundamental architectural property of the generated Petri net lifecycle is the structural orthogonality between the active and passive subnets. Because the `initialize_data` transition branches concurrently into two independent execution paths, the state space does not suffer from combinatorial explosion:

$$\mathcal{S}_{\text{total}} = \{A_1, A_2, A_3, A_4\} \times \{P_1, P_2, P_3\} \quad (4.1)$$

Instead of verifying all 12 potential combined states, the structural independence allows us to analyze each branch scenario in isolation. The complete runtime behavior of any entity is guaranteed to be a deterministic superposition of:

1. Exactly one Active Scenario (A_1, A_2, A_3 , or A_4), and
2. Exactly one Passive Scenario (P_1, P_2 , or P_3).

Consequently, the following detailed behavioral verification focuses on independently analyzing the execution transitions, event signals, and lifecycle guarantees for the 4 Active Scenarios (A_1 – A_4) and the 3 Passive Scenarios (P_1 – P_3).

4.11 Common Execution Prefix: Data Initialization, Read, and Update

Before we start to describe the scenarios A_1 – A_4 and P_1 – P_3 , let us start with a description of the common prefix. All scenarios start with the execution of the task `initialize_data`. The form of the task `initialize_data` has just one `dataRef`, namely a variable of type `taskRef` named `my_working_data` with an initial value equal to a reference on the task `working_data`. The task `working_data` has the system role, i.e., it is not directly accessible by a user and serves just as a reusable form.

The form of the task `working_data` contains a working twin `working_d` for every variable d_i of entity $e \in E$, as well as read only metadata of the case, such as case title and ID. The data `working_d` is editable and, if defined as not-null in the original definition of the entity, it is also `required`. Thus, after assigning the task `initialize_data` to a user, the form of the `working_data` task appears and the user can fill in the data attributes of the case of the entity. On the pre-phase of the finish event of the task `initialize_data`, an action copies `working_d` to d for each attribute d of the case of the entity e .

After the finish event of `initialize_data` is completed, a token appears in the place `data_initialized`, which makes tasks `read_data` and `update_data` executable. The `read_data` task contains again just one `dataRef` of type `taskRef`, named `my_data`, initialized to a reference on the task `data`. Similar to the task `working_data`, the

system task `data` contains necessary metadata of the case - such as title and ID - and a `dataRef` to each attribute d of the case. The task `read_data` is connected with the place `data_initialized` via a read arc. Recall that the read arc means that task `read_data` can be viewed whenever a token is present in the place `data_initialized`. Thus, after successful finish of the task `initialize_data`, task `read_data` and task `update_data` are accessible until the case is deleted.

The task `update_data` is almost a copy of `initialize_data`, with the only difference being that in the pre-phase of the assign event of the `update_data` task, the value of attribute d_i is copied to its working twin `working_di` for each d_i of the case of the entity e . The meaning of the `update_data` task is obvious: it allows a user to update values of the data attributes of the entity.

Furthermore, a successful finish event of the task `initialize_data` concurrently moves a token to the respective post-places of the task (i.e., places connected to the transition by a directed outgoing arc). These outgoing arcs are weighted by the 8 derived structural variables (`active`, `no_active_mandatory`, `active_but_no_active_mandatory`, `active_mandatory`, `single_active_mandatory`, `passive`, `no_passive_mandatory`, and `passive_mandatory`), which are initialized during the domain model translation. As a result, the token automatically routes the process execution into exactly one of the active scenarios (A_1 – A_4) and concurrently into exactly one of the passive scenarios (P_1 – P_3).

Further, we examine each scenario in detail. For tasks `read_data`, `read_active_relations`, `read_passive_relations`, and `read_full_case`, it holds that they cannot be executed; only the view event on these tasks is allowed.

Tasks `unbound_active_relation`, `deactivate_on_delete`, `update_passive_relations`, `initialized_passive_mandatory_relations`, and `deleted_required_relation` are all system tasks, i.e., they cannot be performed directly by a user.

Notice that the task `read_full_case` can be viewed if and only if a token is present in the place `active_mandatory_relations_initialized` and a token is present in the place `passive_mandatory_relations_initialized`.

Therefore, by examining active scenarios A_1 – A_4 , we will show how a token reaches the place `active_mandatory_relations_initialized`, and through passive scenarios P_1 – P_3 , we will show how a token reaches the place `passive_mandatory_relations_initialized`. In addition to these scenarios, we will discuss the dynamics when a case is deleted, denoting this situation by D .

4.12 Scenario A1: No Active Relations

We begin by examining the active branch evaluation vector corresponding to Scenario A_1 , defined over the structural tuple:

```
(active, no_active_mandatory, active_but_no_active_mandatory,
  active_mandatory, single_active_mandatory)
```

For Scenario A_1 , this tuple evaluates to the vector $[0, 1, 0, 0, 0]$, representing a process entity with zero outgoing active relations ($|R_{\text{active}}| = 0$).

When a user completes the `initialize_data` task, outgoing arc weights are evaluated using the derived structural variables. Because `no_active_mandatory` equals 1 and all other arc variables in the active subnet evaluate to 0, the task immediately routes a token through the corresponding branch directly into the place `active_mandatory_relations_initialized`.

Consequently, the active subnet completely bypasses the task `initialize_active_mandatory_relations` and transitions straight to the initialized state. The token remains stably in `active_mandatory_relations_initialized` until case deletion, satisfying the active prerequisite for viewing the `read_full_case` task.

4.13 Scenario A2: Active Optional Only

In Scenario A_2 , active relations exist in the domain model, but all of them are optional ($|R_{\text{active}}| > 0 \wedge \text{active_mandatory} = 0$). This scenario corresponds to the active branch evaluation vector $[1, 1, 1, 0, 0]$, meaning that foreign-key assignment remains entirely optional for the active object.

Upon completing the `initialize_data` task, tokens are concurrently routed to three places: `active_relations`, `ready_for_active_relations_update_or_unbind`, and `active_mandatory_relations_initialized`. Because no mandatory active dependencies exist, the active subnet immediately satisfies its initialization condition upon data setup, placing a token directly in `active_mandatory_relations_initialized`.

The token in `active_relations` enables the viewing of the `read_active_relations` task. For each relation $r_i \in R$ originating from the current process class $e_{i,1} \in E$ (which means that $e_{i,1}$ is active in r_i) towards a target entity class $e_{i,2} \in E$ (represented by `target_process_i`), a `taskRef` variable named `my_relation_data_i` is declared. This variable collects references to the `read_data` forms of all target cases from $e_{i,2}$ currently bound to the active case, rendering them viewable inside `read_active_relations`. In other words, `read_active_relations` allows a case to view data of all cases to which the case is an active part in relation.

Concurrently, the token in place `ready_for_active_relations_update_or_unbind` enables the execution of the `update_active_relations` task. Denoting the entity of the current case c by e_c , in the pre-phase of `assign` event of `update_active_relations` task, it first copies all current data attributes $d \in D_{e_c}$ to their corresponding `working_d` variables. Given a relation $r_i = e_{i,2} \leftarrow e_{i,1}$ with $e_{i,1} = e_c$, to allow users to update links from the current case to cases of $e_{i,2}$, the translation defines a `caseRef` variable `referenced_cases_i` holding the list of bound target case IDs, alongside a user-interface selection variable `working_referenced_cases_i`. Depending on relationship cardinality, this field is typed as `enumeration_map` (for $1 : N$ bounds) or `multichoice` (for $N : M$ bounds). Since all active relations in A_2 are optional, these input fields are marked as non-required.

Listing 1 illustrates the exact action executed on the pre-phase of the `assign` event for `update_active_relations`, where available target cases with initialized data are dynamically queried and populated as selection options for `multichoice`.

```

1 change working_d value { d.value }
2 def possible_references_i = findCases {
3     it.processIdentifier.eq(target_process_i)
4     .and(it.activePlaces.get("data_initialized").eq(1))
5 }.collectEntries {
6     [(it.stringId): it.title + " " + it.stringId]
7 }
8 change working_referenced_cases_i options {
9     possible_references_i }
10 change working_referenced_cases_i value {
11     referenced_cases_i.value as HashSet }

```

Listing 1: Pre-assign action populating available target cases for active $N : M$ relation updates in Scenario A_2 .

On the finish event of the `update_active_relations` task, the system commits all changes. First, updated scalar values are transferred from their working twins `working_d` to target attributes $d \in D_{ec}$. To update active relationships dynamically, the script identifies added and removed references by comparing `working_referenced_cases_i` with the current state `referenced_cases_i`.

For every newly added reference ID, an asynchronous `setData` call triggers the `update_relation` system task on the target case to register the new link. Conversely, removed references trigger a disassociation event. Finally, the newly committed target case IDs are saved into `referenced_cases_i`, and the `read_data` task IDs of all bound target cases are collected to refresh the `my_relation_data_i` view variable.

Listing 2 shows the complete Petriflow execution script for this finish phase for $N : M$ relations.

```

1  change d value { working_d.value }
2
3  def added_i = working_referenced_cases_i.value -
4    referenced_cases_i.value
5  added_i.each { reference_id ->
6    def reference_case = findCase({ it._id.eq(reference_id) })
7    setData("update_relation", reference_case, [
8      "reference_to_add_i": [
9        "value": useCase.stringId, "type": "text"
10     ]
11  })
12
13  def removed_i = referenced_cases_i.value -
14    working_referenced_cases_i.value
15  removed_i.each { reference_id ->
16    def reference_case = findCase({ it._id.eq(reference_id) })
17    setData("update_relation", reference_case, [
18      "reference_to_remove_i": [
19        "value": useCase.stringId, "type": "text"
20     ]
21  })
22
23  change referenced_cases_i value {
24    working_referenced_cases_i.value as List
25  }
26
27  def reference_data_task_ids_i = findTasks {
28    it.caseId.in(referenced_cases_i.value)
29    .and(it.transitionId.eq("read_data"))
30  }.collect { it.stringId }
31
32  change my_relation_data_i value { reference_data_task_ids_i }

```

Listing 2: Finish action committing active $N : M$ relation changes in Scenario A_2 .

4.14 Data Visibility and Composite Full Case View

A key operational property of the Universal Net Archetype (UNA) is the continuous availability of the `read_full_case` view once relational initialization completes. Notably, during subsequent update transactions (e.g., executing `update_active_relations`), target case details remain fully accessible without locking the reading interface.

Architecturally, `read_full_case` functions as a composite view dynamically aggregated from three core read-only tasks: `read_data`, `read_active_relations`, and `read_passive_relations`. This composition guarantees that:

- All committed native data variables $d \in D_{e_1}$ of the current case instance are displayed.
- All `read_data` task interfaces belonging to counterparty cases currently bound in active (R_{active}) and passive (R_{passive}) relations are dynamically rendered within the layout via collected task IDs (`my_relation_data_i`).
- Isolation of uncommitted data is strictly maintained; the view exclusively renders committed domain data d , explicitly omitting temporary working variables (D_{working}) modified during active update sessions.

By defining this composite visibility pattern globally, the same data access rules apply consistently across all scenarios.

4.15 Scenario A3: Multiple Active Mandatory

Scenario A_3 formalizes execution paths where an entity owns mandatory active relations that lack exclusive 1 : 1 constraints ($|R_{\text{active}}| > 0 \wedge \text{active_mandatory} = 1 \wedge \text{single_active_mandatory} = 0$). This configuration evaluates to the active branch structural vector $[1, 0, 0, 1, 0]$. Eligible relational configurations include $P_{1:N}^{OM}$, $P_{1:N}^{MM}$, $P_{N:M}^{OM}$, and $P_{N:M}^{MM}$.

Upon completing the `initialize_data` transition, tokens are concurrently distributed along three active branch arcs based on the derived structural variables:

1. A token is placed into `active_relations` (`active = 1`), enabling the `read_active_relations` view task.
2. A token is placed into `active_mandatory_to_initialize` (`active_mandatory = 1`).
3. Since `no_active_mandatory = 0`, `active_but_no_active_mandatory = 0`, and `single_active_mandatory = 0`, no tokens are routed directly to `active_mandatory`.

strictly filtered to present only unallocated (free) target cases (e.g., unoccupied exam seats). In subsequent modifications via `update_active_relations`, the option list expands to include free cases alongside the currently bound counterparty case (allowing the student to keep their current seat or swap to a free one).

- **Explicit Unbinding** (`unbind_single_active_mandatory_relations`): To prevent pool exhaustion and allow an entity to release its assigned target without immediately picking a replacement (e.g., a student releasing their exam seat and returning to an unassigned state), the archetype introduces the `unbind_single_active_mandatory_relations` task. This acts as an alternative branch to `update_active_relations`, resetting the entity back to an uninitialized mandatory state with a token in place `active_mandatory_to_initialize`.

4.16.1 Read Consistency

To ensure continuous access to the `read_full_case` view during editing, the unbinding task `unbind_single_active_mandatory_relations` cannot remove token from place `active_mandatory_relations_initialized` by assign event:

- Upon assign of `unbind_single_active_mandatory_relations`, the token in `active_mandatory_relations_initialized` remains untouched. This preserves full read access via `read_full_case` throughout user form editing and guarantees that if the user cancels the task, read access is never interrupted.
- The finish event of `unbind_single_active_mandatory_relations` produces a token in place `active_mandatory_to_initialize` while executing the `unbound_active_relation` task that removes the token from place `active_mandatory_relations_initialized`.

4.16.2 Implementation of the Unbinding Interface

The form interface for `unbind_single_active_mandatory_relations` contains a single selection variable, `single_active_mandatory_processes` (type `multichoice_map`), with options formed by the IDs of those **Processes** which:

- hold a 1 : 1 relation with the **Process** of the active case, and
- the relation is mandatory from the perspective of the active **Process**.

The options are only the target relation **Process** IDs rather than explicit target **Case** IDs (e.g., simply instructing the student to release their seat, without requiring them to specify which seat **Case** ID they hold).

Upon **finish**, the relation is disassociated for each selected target **Process**: because the relation is strictly 1 : 1, the single bound target **Case ID** is directly referenced via `referenced_cases_i.value[0]`. It receives a signal containing the current case ID in `reference_to_remove_i`, and is removed from the caseRef `referenced_cases_i` of the current case.

Listing 3 demonstrates the **finish** logic for disassociating the selected mandatory 1 : 1 relation **Processes** and revoking the initialization token.

```

1 // Iterate over single active mandatory relation processes and unbind if
  ↳ selected
2 if (target_process_i in single_active_mandatory_processes.value) {
3   def ref_id = referenced_cases_i.value[0]
4   if (ref_id) {
5     def reference_case = findCase({ it._id.eq(ref_id) })
6     setData("update_relation", reference_case, [
7       "reference_to_remove_i": [
8         "value": useCase.stringId,
9         "type": "text"
10      ]
11    ])
12  }
13
14  change working_referenced_cases_i value { [] }
15  change referenced_cases_i value { [] }
16 }
17
18 // Revoke initialization token from place
  ↳ active_mandatory_relations_initialized
19 async.run {
20   assignTask("unbounded_active_relation")
21   finishTask("unbounded_active_relation")
22 }

```

Listing 3: Atomic unbinding execution and token management in Scenario A_4 .

4.17 Scenario P1: No Passive Relations

Scenario P_1 represents cases where the entity defines zero incoming passive relations ($|R_{\text{passive}}| = 0$). This configuration corresponds to the passive structural evaluation vector $[0, 1, 0]$.

Upon completing `initialize_data`, the transition evaluates the outgoing arcs using the derived structural variables. Because `no_passive_mandatory` = 1, the transition immediately places a token into `passive_mandatory_relations_initialized`. The passive subnet completely bypasses passive mandatory initialization, leaving the token

stably in place to satisfy the passive prerequisite for the `read_full_case` view task.

4.18 Scenario P2: Passive Optional Only

In Scenario P_2 , incoming passive relations exist, but all are optional ($0..1$ or $0..N$). This scenario corresponds to the passive structural evaluation vector $[1, 1, 0]$.

Upon completing `initialize_data`, tokens are concurrently routed along two arcs:

- A token is placed into `passive_mandatory_relations_initialized` (`no_passive_mandatory = 1`), immediately fulfilling the passive requirement for `read_full_case`.
- A token is placed into `passive_relations` (`passive = 1`), enabling the `read_passive_relations` view task.

4.18.1 Universal Reactive Reference Management via `update_relation`

To maintain relational integrity across both active and passive endpoints without structural duplication, every relation in the process archetype embeds the system task `update_relation`. This task exposes two system data references: `reference_to_add_i` and `reference_to_remove_i`. These fields execute reactive script actions on their respective `setData` events:

- **Link Registration** (`reference_to_add_i`): Triggered asynchronously by an initiating case to attach a link. The script appends the case ID to `referenced_cases_i` and dynamically fetches the `read_data` task ID of the counterparty to update the view variable `my_relation_data_i` (Listing 4).
- **Link Revocation** (`reference_to_remove_i`): Triggered when a counterparty disassociates or is deleted. The script removes the ID from `referenced_cases_i` and updates the view variable accordingly (Listing 5).

Because all passive relations in P_2 are strictly optional, no further state transitions or initialization tasks are required beyond these universal reactive scripts.

4.19 Scenario P3: Passive Mandatory

Scenario P_3 formalizes execution paths where the entity requires at least one incoming passive relation to satisfy mandatory lifecycle obligations ($1..1$ or $1..N$ bounds). This configuration corresponds to the passive structural evaluation vector $[1, 0, 1]$.

```

1 // Reactive link registration on setData event of reference_to_add_i
2 if (reference_to_add_i.value !in referenced_cases_i.value) {
3   change referenced_cases_i value { referenced_cases_i.value +
4     ↳ reference_to_add_i.value }
5 }
6 def reference_to_add_id_i = findTask {
7   it.caseId.eq(reference_to_add_i.value).
8   and(it.transitionId.eq("read_data"))
9 }.collect { it.stringId }
10
11 change my_relation_data_i value { my_relation_data_i.value +
    ↳ reference_to_add_id_i }

```

Listing 4: Universal link registration logic executed on setData of reference_to_add_i.

```

1 // Reactive link revocation on setData event of reference_to_remove_i
2 if (reference_to_remove_i.value in referenced_cases_i.value) {
3   change referenced_cases_i value { referenced_cases_i.value -
4     ↳ reference_to_remove_i.value }
5 }
6 def reference_to_remove_id_i = findTask {
7   it.caseId.eq(reference_to_remove_i.value).
8   and(it.transitionId.eq("read_data"))
9 }.collect { it.stringId }
10
11 change my_relation_data_i value { my_relation_data_i.value -
    ↳ reference_to_remove_id_i }

```

Listing 5: Universal link revocation logic executed on setData of reference_to_remove_i.

Upon completing `initialize_data`, tokens are routed to `passive_relations` and `passive_mandatory_to_initialize`. Crucially, no token is placed into `passive_mandatory_relations_initialized`, blocking access to `read_full_case` until mandatory binding succeeds.

4.19.1 Lifecycle State Transitions in P_3

The lifecycle state in P_3 is dynamically managed through external signals evaluated within the universal `update_relation` event handlers:

- **Initialization Completion:** When `setData` of `reference_to_add_i` is executed for a mandatory passive relation r_i , the system checks if all other mandatory passive relations have also received their required counterparties. If all mandatory constraints are satisfied, the script programmatically triggers the system task `initialized_passive_mandatory_relations`. This transition consumes the token from `passive_mandatory_to_initialize` and deposits a token into `passive_mandatory_relations_initialized`, unblocking `read_full_case`.
- **Mandatory Relation Violation:** When `setData` of `reference_to_remove_i` is executed for a mandatory relation r_i , the script checks whether `referenced_cases_i` becomes empty. If no counterparties remain and a token is currently present in `passive_mandatory_relations_initialized`, the system executes the system task `deleted_required_relation`. This transition revokes the initialization token from `passive_mandatory_relations_initialized` and returns a token to `passive_mandatory_to_initialize`, safely re-locking `read_full_case`.

4.20 Processing Incoming Requests on Removal on Active Endpoints

This scenario addresses the reactive behavior of a process instance operating in an *active role* within a specific relation r_i . While the active entity manages outgoing links via selection workflows (`update_active_relations`), it must also reactively process incoming removal requests dispatched from the counterparty. When the target instance undergoes deletion or disassociation in case of $M : N$ relationships where both sides are active and passive, it invokes the active entity's system task `update_relation` by passing its ID into `reference_to_remove_i`.

Upon receiving a removal request via the `setData` action of `reference_to_remove_i`, the active entity executes the following precise execution logic:

1. **Reference Purging:** The calling case ID is purged from `referenced_cases_i`, and its corresponding references are removed from `my_relation_data_i`.

2. **Reactive Unbinding Execution:** If the place `active_mandatory_relations_initialized` currently holds a token, the `setData` action programmatically executes the system transition `unbinded_active_relation`.
3. **Conditional State Revocation:** If, simultaneously, the place `ready_for_active_relations_update_or_unbind` also holds a token, the action additionally executes the system transition `deactivate_on_delete`. This returns the token to `active_mandatory_to_initialize`, effectively re-locking `read_full_case` until mandatory bounds are re-established.

Furthermore, to handle user-initiated cancellations correctly, action is added to the `post` phase of the `cancel` event for both `unbind_single_active_mandatory_relations` and `update_active_relations` tasks. This action executes exclusively when a user triggers the `Cancel` event: the action checks whether the place `active_mandatory_relations_initialized` is empty ($\emptyset$). If it is empty, the script programmatically fires `deactivate_on_delete` to restore the system to a valid state and preserve execution invariants.

Implementation Note and Transactional Optimistic Locking In the legacy production runtime of the Petriflow engine, programmatic execution of system transitions within actions of the same case required asynchronous execution wrappers (e.g., `async.run`), as illustrated in Listing 3, due to platform limitations regarding intra-case synchronous event chaining. Under high concurrency, this introduced a potential race condition where an asynchronous transition (such as `unbinded_active_relation`) might not complete its token movement before a concurrent `cancel` event evaluated the state of `active_mandatory_relations_initialized`.

This limitation has been fully resolved in the prototype of the new engine version. The updated runtime natively supports wrapping operations into unified, atomic transactions—both within a single case context and across multiple linked cases—governed by transactional optimistic locking with configurable retry capabilities. Triggering an action, such as executing `setData`, runs the entire reactive cascade synchronously within a single transaction boundary. If a concurrent operation (such as a `cancel` event) triggers an optimistic lock conflict, the engine automatically re-executes the transaction according to its configured retry strategy. This ensures that retried actions evaluate deterministically against the newly committed state, thus enabling elimination of race conditions while preserving non-blocking read access.

4.21 Symmetric Handling of M:N Relations

In $M : N$ relational topologies, the operational boundary between strictly active and strictly passive endpoints dissolves. Because $M : N$ relations are inherently symmetric, ev-

ery participating entity maintains active selection workflows (`update_active_relations`) alongside passive reactive capabilities (`update_relation`) concurrently.

Consequently, receiving a disassociation signal via `reference_to_remove_i` impacts both relational subnets simultaneously depending on local constraints:

- **Active Subnet Handling:** If the disassociated target leaves an active mandatory relation empty, the reactive script executes `deactivate_on_delete`, returning the token from `active_mandatory_relations_initialized` to `active_mandatory_to_initialize`.
- **Passive Subnet Handling:** If the disassociated target leaves a passive mandatory relation empty, the reactive script executes `deleted_required_relation`, returning the token from `passive_mandatory_relations_initialized` to `passive_mandatory_to_initialize`.

This dual-subnet state synchronization guarantees absolute integrity: if a mandatory counterparty is removed from either direction, the process instance instantly revokes its initialized state, blocking `read_full_case` until the constraint is satisfied.

4.22 Scenario D: Entity Deletion and Cascade Propagation

Scenario *D* formalizes the terminal lifecycle behavior when an entity instance itself undergoes deletion. Deletion is not a localized operation; it actively broadcasts disassociation signals across the entire relational network.

Upon triggering the deletion action, the process instance executes a cascade broadcast across all its active and passive relations:

1. It iterates over every referenced case ID stored across all local relation variables (`referenced_cases_i`).
2. For each connected target, it invokes a remote `setData` call on the target's universal `update_relation` task, passing its own identifier (`useCase.stringId`) into `reference_to_remove_i`.
3. The receiving target entities process the removal reactively as defined in Section 4.20 (for active cases) or Scenario *P*₃ (for passive cases), executing mandatory deactivations (`unbounded_active_relation` and `deactivate_on_delete` for the active case, and `deleted_required_relation` for passive cases) if mandatory bounds are violated.

This broadcast mechanism guarantees relational integrity across the domain model without leaving orphaned references in any connected process instance.

Chapter 5

Conclusion

In this paper, we presented a comprehensive method for automatically deriving executable Process-Centric Classes (PCC) in Petriflow directly from relational data models and structural schema specifications (such as SQL DDL, ER, and UML Class diagrams)[cite: 1]. The core objective of our approach is to systematically map each domain entity to an instance of a single, invariant **Universal Net Archetype (UNA)**[cite: 1]. This transformation enables full CRUD execution capabilities (Create, Read, Update, Delete) for every entity while fully preserving relational integrity and complex cardinality constraints across the system[cite: 1].

Through this unified architectural foundation, a single UNA instance can simultaneously act as either an active initiator or a passive target across any combination of the 11 identified relational protocol types ($P_{\text{passive, active}}$ across 1:1, 1:N, and N:M cardinalities)[cite: 1]. Specifically, our contributions and technical highlights include:

- **Entity-to-UNA Paradigm:** Each relational entity is instantiated using an identical Petri net graph topology[cite: 1]. Operational flexibility is achieved by compiling relation-specific data variables and event actions into the invariant net without requiring custom structural modifications per class[cite: 1].
- **Direct Vector Symmetry for N:M Links:** To streamline N:M relationships without generating intermediate junction-table nets, the archetype leverages native `caseRef` array references directly within participating process entities to achieve direct vector symmetry[cite: 1].
- **Orthogonal Subnet Execution and Lifecycle Integrity:** Decoupling lifecycle management into independent active (A_1-A_4) and passive (P_1-P_3) execution subnets prevents combinatorial state explosion[cite: 1]. Depending on the specific relational protocols in which a domain entity participates and whether it acts as the active initiator or passive target, the UNA dynamically routes execution into the corresponding active and passive sub-scenarios[cite: 1]. This structural orthogonality

ensures non-blocking read access (`read_full_case`) during updates while strictly enforcing mandatory binding requirements before case initialization completes[cite: 1].

- **Explicit Unbinding and Transactional Invariants:** We introduced explicit unbinding mechanics (`unbind_single_active_mandatory_relations`) and reactive signal propagation (`reference_to_add` / `reference_to_remove`)[cite: 1]. Supported by multi-case optimistic locking in the engine, these mechanisms eliminate race conditions and guarantee lifecycle state consistency during relation updates or cascade deletions[cite: 1].
- **Data Manipulation and Query Translation (SQL DML to Petriflow/PFQL):** In addition to schema transformation (SQL DDL), our translation pipeline can be straightforwardly extended to handle data manipulation operations (DML). For instance, translating record creation (`INSERT INTO`) into executable process instances can be achieved by invoking the corresponding Universal Net Archetype (UNA) entity lifecycle. Depending on the scenario (A1–A4) and relationship topology (P1–P3), a new case creation or binding action is executed directly via task actions in the generated Petriflow model. Furthermore, declarative database queries (`SELECT`) can be automatically translated into the Petriflow Query Language (PFQL) [Juh+26]. This allows querying active process instances and historical data while maintaining the process-centric boundaries and execution context of UNA objects.

5.1 Future Work

While our current approach successfully derives executable process-centric classes (PCC) and Universal Net Archetypes (UNA) from static relational schemas, extending this paradigm requires addressing the dynamic and temporal nature of real-world domain models. Based on the foundational taxonomy of temporal entity-relationship models surveyed by Gregersen and Jensen [GJ99], we identify three primary directions for future research:

- **Temporal Domain Models and Lifetime Cardinalities:** Current database-to-lifecycle mappings treat relational constraints as static snapshots. However, real-world domain models often require distinguishing between instantaneous state limits and cumulative historical bounds. Following the concepts introduced in the TER model [Tau91], our future compiler will formalize both *Snapshot Cardinalities* ($S[\min, \max]$) and *Lifetime Cardinalities* ($L[\min, \max]$) within UNA. This will allow the generated execution engines to enforce strict instantaneous rules (e.g., a student

occupying at most one exam seat at a given time t) alongside cumulative temporal tracking (e.g., a student completing multiple exams over their academic lifespan).

- **Behaviorally-Oriented Lifecycle Management:** Moving beyond static structure-driven mappings, future iterations will integrate behavioral operational context directly into process compilation. Drawing from state-transition formalisms in TERC+ [Zim+97] and temporal role modeling in STEER [EKF93], we plan to dynamically adjust object lifecycles based on execution history, active entity roles, and lifecycle states (e.g., active, suspended, terminated). Rather than relying solely on foreign key constraints, object permissions and available actions will evolve dynamically as entities transition through different operational phases.
- **Transitive and Context-Aware Relational Constraints:** Complex enterprise domains frequently involve multi-hop relationships (e.g., Student $\rightarrow$ Degree Program $\rightarrow$ Course). To prevent unauthorized or invalid cross-entity operations, UNA execution nets must automatically derive and enforce transitive scoping rules [GJ99]. By leveraging historical perspective modeling from the ERT model [The+92] and temporal relationship mapping techniques from RAKE [Fer85], we will implement automated context propagation. This ensures that target case queries and operational choices are dynamically filtered based on active temporal validity across chained relational dependencies.

By integrating these temporal, behavioral, and transitive dimensions [GJ99], the UNA framework will evolve from a static schema transformer into a fully context-aware domain compiler.

Bibliography

- [BN12] Dmitry Buzdin and Oksana Nikiforova. “Transformation of UML class diagram to internal java domain-specific language”. In: *Applied Computer Systems* 13.1 (2012), pp. 61–67.
- [EKF93] Ramez Elmasri, Vram Kouramajian, and Shian Fernando. “Temporal database modeling: an object-oriented approach”. In: *Proceedings of the Second International Conference on Information and Knowledge Management*. CIKM ’93. Washington, D.C., USA: Association for Computing Machinery, 1993, pp. 574–585. ISBN: 0897916263. DOI: 10.1145/170088.170429. URL: <https://doi.org/10.1145/170088.170429>.
- [Fer85] Stephen Ferg. “Modelling the Time Dimension in an Entity-Relationship Diagram”. In: *Proceedings of the Fourth International Conference on Entity-Relationship Approach*. USA: IEEE Computer Society, 1985, pp. 280–286. ISBN: 044487951X.
- [GJ99] H. Gregersen and C.S. Jensen. “Temporal entity-relationship models-a survey”. In: *IEEE Transactions on Knowledge and Data Engineering* 11.3 (1999), pp. 464–497. DOI: 10.1109/69.774104.
- [Hul08] Richard Hull. “Artifact-Centric Business Process Models: Brief Survey of Research Results and Challenges”. In: *On the Move to Meaningful Internet Systems: OTM 2008*. Ed. by Robert Meersman and Zahir Tari. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1152–1163. ISBN: 978-3-540-88873-4.
- [Juh+26] Gabriel Juhás et al. “Petriflow Query Language”. In: *Joint Workshop Proceedings of PNSE’26, PNAS’26, ATAED’26, and PHOCON’26 co-located with the 47th International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2026)*. Vol. 4236. CEUR Workshop Proceedings. CEUR-WS.org, 2026. URL: <https://ceur-ws.org/Vol-4236/>.
- [KR11] Vera Künzle and Manfred Reichert. “PHILharmonicFlows: towards a framework for object-aware process management”. In: *Journal of Software Maintenance and Evolution: Research and Practice* 23 (Jan. 2011), pp. 205–244. DOI: 10.1002/smr.524.

- [Tau91] Branka Tauzovich. “Towards Temporal Extensions to the Entity-Relationship Model”. In: *Proceedings of the 10th International Conference on Entity-Relationship Approach (ER’91), 23-25 October, 1991, San Mateo, California, USA*. Ed. by Toby J. Teorey. ER Institute, 1991, pp. 163–179.
- [The+92] Charalampos I. Theodoulidis et al. “The Entity-Relationship-Time Model”. In: 1992. URL: <https://api.semanticscholar.org/CorpusID:60592681>.
- [Zim+97] Esteban Zimanyi et al. “TERC+: a temporal conceptual model”. In: (1997).