

Pan-European University, Faculty of Informatics

Interim Research Report

Meta definition of notion of version and extension of the low-code language

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 and Tomáš Kováčik

Contact:

gabriel.juhás@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

Traditional process modelling often suffers from separation between data structures, business logic, and presentation layer. In this paper we introduce the Netgrif Platform - an application development platform designed for modelling and execution of object-centric processes using the Petriflow language, that integrates these three layers into a single human-readable and executable model. The Netgrif Platform enables rapid development of business-aligned processes and complex applications by providing this vertical integration of layers.

Contents

Contents	i
List of Figures	ii
List of Tables	iii
1 Introduction	1
2 Petriflow Language	3
3 Functional Specification of Order and Invoice Management	5
3.1 Discussion and Key Architectural Concepts	8
4 Netgrif Platform Architecture	10
4.1 Modeling Component: Netgrif Application Builder	10
4.2 Execution Component: Netgrif Application Engine	10
4.3 The Modeling Workflow: Netgrif Application Builder	11
4.4 Deployment and Runtime Interpretation	12
5 Related Work and Comparison	13
6 Conclusion	14

List of Figures

3.1	The Netgrif Application Builder interface showing the Petri net lifecycle for the Order class in editing mode and for Invoice class in simulation mode with transition (task) Register invoice being assigned to a user.	6
3.2	Detailed synchronization logic between <i>Order</i> and <i>Invoice</i> objects. The <i>Assign Event</i> in the parent process triggers a lookup of child references and dynamically composes the integrated user task.	8
4.1	The Netgrif Application Engine architecture: A cloud-native stack comprising Spring Boot services, MongoDB for persistence, and Elasticsearch for object-centric querying.	11
4.2	A dynamic form in eTask demonstrating Bulk Approval: parent process (Order) synchronizing events across multiple child objects (Invoices). . . .	12

List of Tables

Chapter 1

Introduction

Current advancements in process science have established Object-Centric Processes (OCP) [GGR25; DP24], represented by Object-Centric Petri Nets (OCPN) [AB20], as a specialized subclass of Colored Petri Nets (CPN) [JK09] where tokens are restricted to unique object identifiers. While OCPNs excel at capturing the 'forest' of concurrent interactions as a runtime execution model, they often result in complex, monolithic structures that are difficult to re-implement.

To understand the spectrum of object-aware process modeling, we categorize Petri net-based models into three categories based on whether they model the runtime of a system or the definition of object types:

Object-Centric Processes (The Observational Forest): Represented by OCPN, this is a top-down approach where the entire 'forest' of interactions is modeled in a single net. Here, objects are merely IDs (tokens) flowing through a shared environment, primarily used for descriptive analysis.

Embedded Object Processes (Trees Embedded in a Forest): Represented by the Nets-within-Nets paradigm [KWM04], where object nets act as tokens within a higher-level system net. While this approach introduces modularity through a hierarchical structure, the object 'trees' remain embedded within and dependent on a predefined global 'forest' (the system net) to orchestrate their lifecycle and cross-object interactions.

Process-Centric Objects (PCO) (The DNA of Trees): Represented by the Petriflow language, this category focuses on modeling object types (classes) by adding a lifecycle directly to the class definition. We recognize Procleets (Lightweight Nets) as an early, seminal attempt at this approach, modeling independent process fragments that interact via messages. Petriflow evolves this concept into autonomous, encapsulated blueprints where classes define the potential relationships and interactions of their objects. PCO approach aligns with the vision given in [KP09], where the authors extend Petri nets into modular components by formalizing interfaces and composition rules for net modules.

In PCO approach, the classes define the potential relationships and interactions of their objects. The 'forest', i.e. the runtime application is not a static model; it is an emergent

running program created by the interaction of objects from independent Process-Centric Classes composed like Lego bricks.

These classes can be utilized in two primary ways. First, as in object-oriented programming, to design an application from scratch. Second, as part of a two-step reverse engineering framework:

1. Step 1: Process Mining from raw object-centric event logs into an OCPN runtime model (The Forest).
2. Step 2: Synthesizing these observations into Process-Centric Classes (The DNA) from the OCPN runtime model.

This transition from a descriptive 'forest' to a prescriptive 'design model' enables the automated generation of scalable, executable software architectures directly from observed real-world behavior. Thus, Process-Centric Object-oriented languages like Petriflow are not competing with **object-centric process runtime models like OCPN**, but rather complementing them by providing a direct path to implementation.

Netgrif Platform was designed for modelling and execution of process-centric objects in language Petriflow [Juh+21]. Netgrif Platform is used to run many enterprise applications in different industry fields like healthcare, insurance, telecommunications, electric utility, and others. The platform is also used on multiple universities for teaching and research purposes and management of internal processes, namely on the Faculty of Informatics and Information Technologies of the Slovak University of Technology, the Faculty of Informatics of the Pan-European University in Bratislava, the Faculty of Mathematics and Computer Science of the FernUniversität in Hagen, and the Faculty of Applied Computer Science of the University of Augsburg.

Chapter 2

Petriflow Language

Inspired by object-oriented programming (OOP), the basic building blocks of the *Petriflow* language are **Petriflow classes**. A Petriflow class serves as a blueprint that encapsulates three integrated layers: *data attributes*, a *lifecycle* in the form of extended Petri nets, and *user interfaces* (UI) as forms associated with tasks.

Data attributes are XML-based objects defined by their type, identifier, title, and various validations. The lifecycle layer is represented by place/transition nets [DJ01; Des05; DR15], enhanced with inhibitor [AJ15], read [Vog02], and reset arcs [Aal+09]. Furthermore, Petriflow supports variable arc weights determined by data attribute values or place markings, a concept inspired by self-modifying nets [Val78]. The UI layer consists of forms, which are subsets of data attributes defining permissions such as required, editable, or read-only for each specific task.

The novelty of this paradigm lies in the vertical integration of these layers within a single model. The Petriflow interpreter automatically handles communication between the client-side device, the application server, and the persistent database. This automation results in compact, all-in-one, human-readable models that ensure business-aligned execution while enabling the realization of complex relationships between different objects.

Relations between **cases** (instances of Petriflow classes) are modeled at the data layer. Data attributes can store references to one or multiple related cases, mirroring concepts like foreign keys in SQL, object references in OOP, or 1:N relationships in class diagrams.

The Petriflow language defines events for classes, cases, lifecycle tasks, and data attributes. These events can be triggered manually by users, automatically by the system, or by *actions*—anonymous Groovy code snippets—as reactions to other events. For example, creating a new case triggers a *create case event*, which generates a unique instance (copy) of the underlying Petri net and data attributes defined in the Petriflow class.

Tasks, represented as transitions, follow semantics based on the *first-consume-then-produce* principle [Juh00]. Unlike standard Petri nets where firing is an atomic operation, here it is decomposed into three discrete logical events:

Assign (Start): Assigns a user to the task and consumes tokens from the preset of the transition.

Finish: Completes the task and produces tokens in the postset of the transition.

Cancel: Aborts the task, unassigns the user, and returns the tokens consumed during the assign event back to the preset.

Data attributes feature *set* events for value modifications and *get* events for data retrieval. Access to these events is governed by role-based access control (RBAC) [BDM11].

When an event is triggered, an action can be executed in either the *pre-phase* or the *post-phase*. Actions in the *pre-phase* operate on the state (data and markings) as it existed before the event, while *post-phase* actions work with the updated state. These actions can trigger a chain of subsequent events across different cases.

By combining data-level references with event synchronization, Petriflow enables sophisticated coordination. For instance, finishing a task in a parent case can programmatically trigger *finish* events for tasks in referenced child cases. Because data attributes can store task references, Petriflow allows the dynamic embedding of referenced task forms as sub-forms within a parent form.

Finally, similar to SQL in relational databases, the Petriflow language provides a powerful query language to create dynamic filters over cases and their tasks [JMP24; Juh+22; JJP23].

Chapter 3

Functional Specification of Order and Invoice Management

To illustrate a process-based application in the Petriflow language, created and executed on the Netgrif platform, we use a practical example of managing orders and their associated invoices. The system consists of two main process classes: *Order* and *Invoice*. The life-cycle of the classed is depicted in Figure 3.1.

The Order Process Lifecycle The *Order* class contains data attributes such as *Order ID*, *Subject*, and *Supplier*. It also includes an attribute for the *Approval Decision*, implemented as an **enumeration map** (with options [accept:Accept], [back:Back], and [reject:Reject]).

When a new *Order* is created, the *Order ID* is automatically set. Following initialization, in form associated with the transition (task) *Request Order Approval* an employee fills out the order details. Subsequently, an approving manager executes the *Order Approval* task. Form of this task contains the enumeration map *Decision*.

The selection in the enumeration map determines the subsequent path of the order through the use of **variable arcs**. Upon finishing the *Order Approval* task an action updates the value of attributes *accept*, *finish* and *back* determining the weights of the corresponding variable arcs.

Based on the decision, the token is moved accordingly:

1. **Accept:** The token moves to the place *accepted* (p_4). Subsequently, the task *Send Order to Supplier* must be executed to move the token to the *sended* state (place p_5).
2. **Reject:** The token moves to the place *rejected*.
3. **Back:** The token is returned to the *start* place, allowing for modifications.

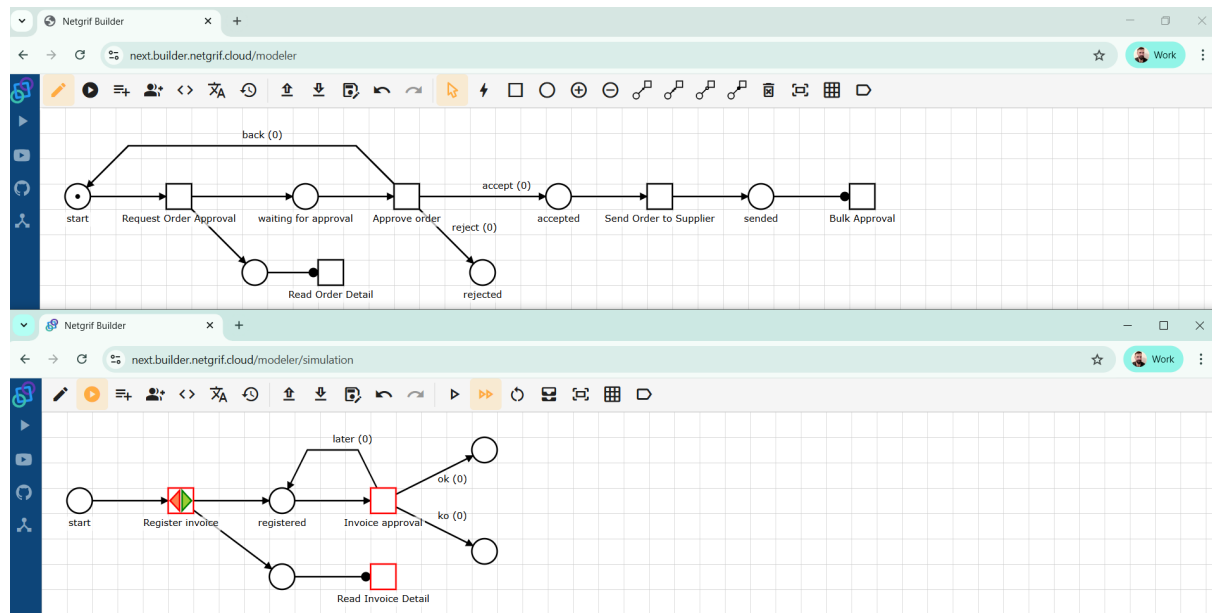


Figure 3.1: The Netgrif Application Builder interface showing the Petri net lifecycle for the Order class in editing mode and for Invoice class in simulation mode with transition (task) Register invoice being assigned to a user.

If accepted, the order moves to the *sended* state (represented by a token in place p_5) by assigning and finishing task *Send Order to Supplier*, making it available for *Bulk Approval* of associated invoices.

Invoice Registration and Linkage Similar to the *Order* process, the *Invoice ID* is automatically assigned by a script executed in response to the `createCase` event upon the instantiation of a new *Invoice* object. Subsequently, a person with the role *Invoice Processor* can assign themselves the *Register Invoice* task.

In response to the `assign` event of this task, an action is executed to perform a lookup of all active *Order* instances that currently hold a token in the place p_5 (labeled *sended*). This filter ensures that only orders already dispatched to a supplier are available for selection. The *Order ID* values of these retrieved instances are then populated as keys into the *Parent Order ID* enumeration map of the invoice:

```
// Triggered on 'assign' event of Register Invoice
def orders = findCases {
  it.processIdentifier.eq(workspace + "order")
  .and(it.activePlaces.get("p5").eq(1))
}.collectEntries { [(it.stringId): "Order: " + it.stringId] }
change parent_order_id options { orders }
```

The transmission of the *Invoice ID* to the parent *Order* occurs when the task is finished:

```
// Triggered on 'finish' event of Register Invoice
def parent_order_case=findCase({it._id.eq(parent_order_id.value)});
setData("t1", parent_order_case,
["new_invoice_id": ["value": invoice_id.value, "type": "text"]]);
```

In the *Order* class, a reactive action on the **set** event of the *new_invoice_id* attribute ensures the ID is added to the collection of child cases:

```
// Triggered on 'set' event of new_invoice_id in the Order class
if (new_invoice_id.value !in children_invoice_cases.value) {
  change children_invoice_cases value {
    children_invoice_cases.value + new_invoice_id.value } }
```

Continuation of the Invoice Lifecycle Following registration, the *Invoice* enters its approval phase. In the form associated with the task *Invoice approval* the approver interacts with an *enumeration map* (with options [OK:Accept], [later:Later], and [KO:Reject]). Similarly to the *Order* approval, a **finish** event action updates the weights of variable arcs (*ok*, *ko*, and *later*) based on the selected decision.

This logic directs the token to the corresponding state: **OK** moves the token to the *accepted* place, **KO** to the *rejected* place, and **later** returns it to the *registered* place for future reconsideration.

Bulk Approval and Sub-form Integration The final stage is the *Bulk Approval* task in the *Order* process. On the **assign** event, the system identifies and assigns all unassigned *Invoice Approval* (transition t2) tasks:

```
// Triggered on 'assign' event of Bulk Approval
change invoice_approvals value { findTasks{
  (it.caseId.in(children_invoice_cases.value))
  .and(it.transitionId.eq("t2"))
  .and(it.userId.isNull()) }?.collect{it.stringId} }
invoice_approvals.value.each{ id ->
  def task = findTask({it._id.eq(id)}); assignTask(task) }
```

The situation is illustrated in Figure 3.2.

The manager interacts with embedded sub-forms. Upon finishing the *Bulk Approval*, the system programmatically completes all child tasks, triggering their individual variable arc logic:

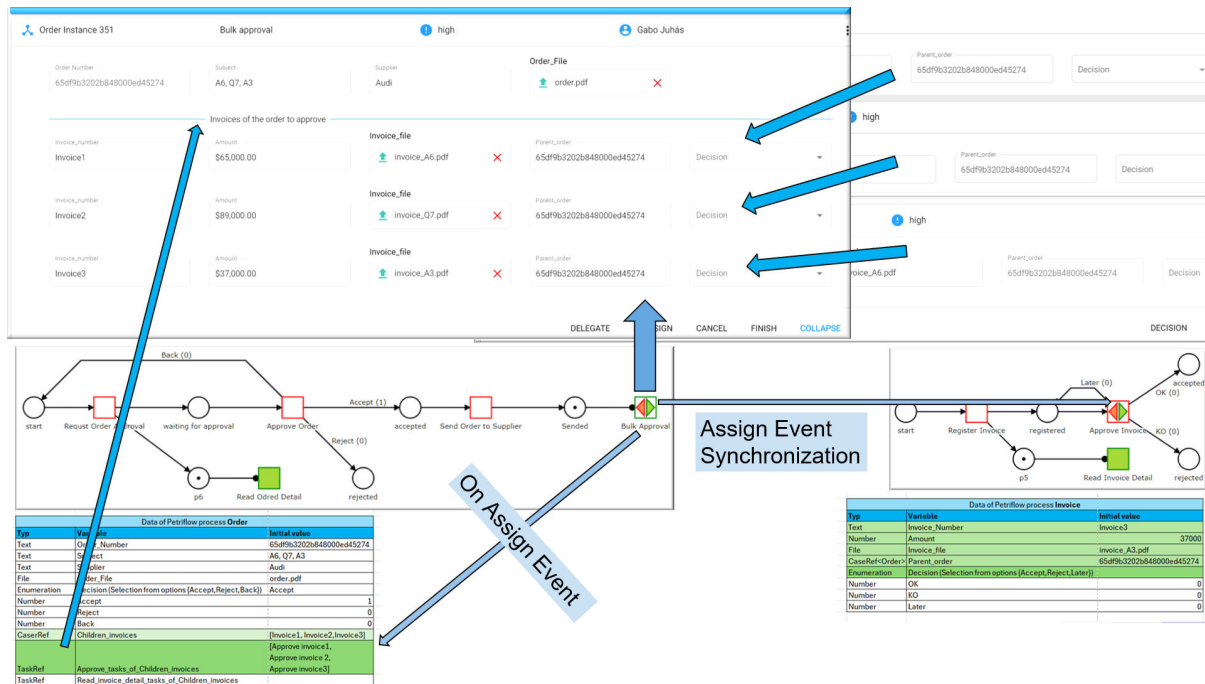


Figure 3.2: Detailed synchronization logic between *Order* and *Invoice* objects. The *Assign Event* in the parent process triggers a lookup of child references and dynamically composes the integrated user task.

```
// Triggered on 'finish' event of Bulk Approval
def list_of_tasks_ids = invoice_approvals.value;
change invoice_approvals value { []; }
list_of_tasks_ids.each{ id ->
  def task = findTask({it._id.eq(id)}); finishTask(task) }
```

3.1 Discussion and Key Architectural Concepts

The presented example illustrates how complex relationships between independent process instances are managed at the data and event levels within the Petriflow framework. Several key architectural patterns can be identified:

Data-Level Relationships: The system maintains a bidirectional link where the parent *Order* stores a list of child *Invoice* IDs, and each *Invoice* maintains a reference to its parent. This allows for efficient traversal of the object hierarchy.

Dynamic Object and Task Discovery: Using powerful query functions like `findCases` and `findTasks`, the system can dynamically locate objects and their current states (active tasks) based on attributes or place markings.

Event-Based Synchronization: By chaining event triggers (e.g., a `finish` event in one process invoking a `finish` in another), the system achieves seamless synchronization between distinct Petri net instances. Task pointers (Task References) serve as the

fundamental mechanism for this inter-process communication.

Hierarchical UI Composition: The ability to store task pointers as data attributes, combined with the platform's rendering engine, allows for the creation of composite forms. Sub-forms of child objects are dynamically embedded into the parent's context, providing a unified user experience for complex operations like bulk approvals.

In conclusion, this approach demonstrates that Petriflow moves beyond traditional monolithic Petri nets by treating process instances as reactive objects that can be dynamically linked, queried, and synchronized through a robust event-driven architecture.

Chapter 4

Netgrif Platform Architecture

The *Netgrif Platform* provides a model-driven environment for process-based applications. As illustrated in Figure 4.1, the architecture is logically divided into the design environment and the execution engine.

4.1 Modeling Component: Netgrif Application Builder

The *Application Builder* is a specialized web-based IDE for visual modeling of *Petriflow* classes. It serves as a standalone environment where designers define the structural aspects of a process: data attributes, Petri net lifecycles, and form layouts. The output is a *Petriflow* XML file—a transportable, technology-agnostic artifact that encapsulates the entire application logic.

The newest version of the *Application Builder* is freely available for public use at <https://next.builder.netgrif.cloud>.

4.2 Execution Component: Netgrif Application Engine

The core of the execution environment is the *Netgrif Application Engine* (NAE), which acts as a dynamic interpreter for Petriflow models uploaded via the *eTask* interface. The NAE follows a robust three-tier technical stack:

Presentation Layer: An **Angular** client that dynamically renders the UI based on the process marking and form definitions.

Application Layer: A **Java (Spring Boot)** backend that manages the process lifecycle, integrates **Drools** for rule evaluation, and executes Groovy scripts.

Data Layer: A hybrid persistence strategy using **MongoDB** for case states, **Elasticsearch** for advanced querying, and **Redis** for session caching.

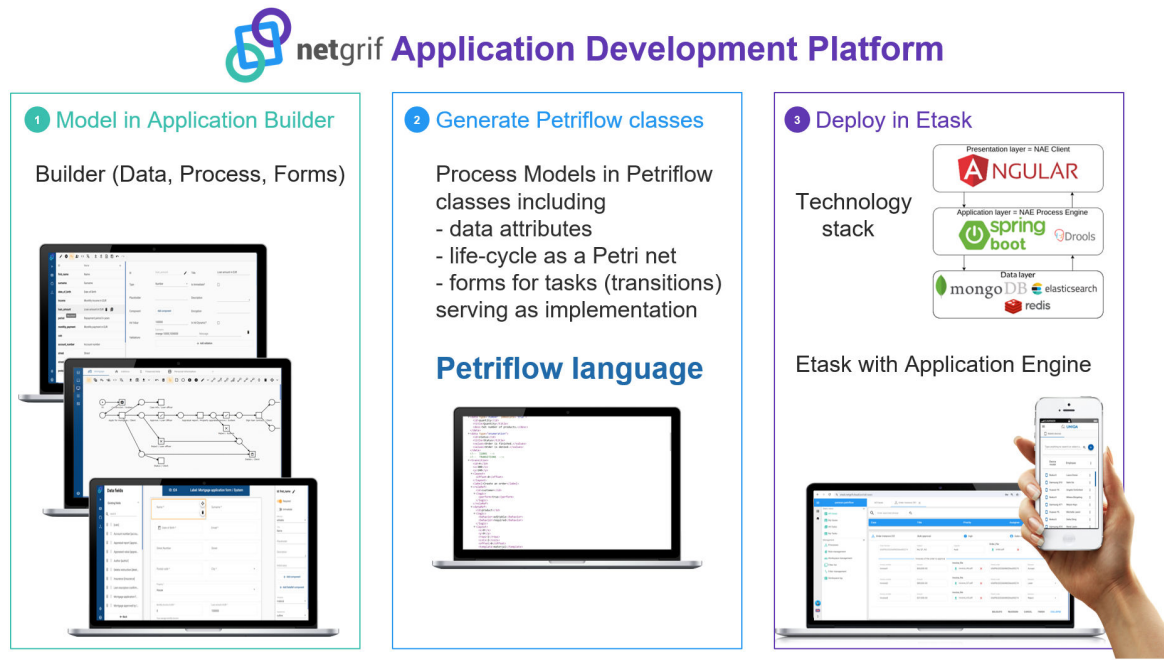


Figure 4.1: The Netgrif Application Engine architecture: A cloud-native stack comprising Spring Boot services, MongoDB for persistence, and Elasticsearch for object-centric querying.

The execution environment is accessible after registration at <https://etask.netgrif.cloud>, where users can create their own workspaces and deploy Petriflow models.

4.3 The Modeling Workflow: Netgrif Application Builder

The *Netgrif Application Builder* (NAB) serves as a specialized IDE for the visual design of *Petriflow classes*. The core functional workflow is centered around three integrated phases:

Data and Lifecycle Definition: In the first phase, the designer defines two independent layers of the class. First, the **data attributes** (e.g., *Order ID*, *Price*) are specified to form the object's data structure. Second, the **lifecycle** is modeled as an extended Petri net (see Figure 3.1), defining abstract states given by markings and tasks given by transitions.

Binding, Roles, and UI Composition: The integration occurs by binding data and permissions to the process. Forms are created as views on data attributes and bound to transitions. NAB allows designers to drag-and-drop attributes into forms, automatically synchronizing them with the data model. Crucially, this phase also includes **Role Definition**, where user roles (e.g., *Manager*, *Accountant*) are created and assigned to specific transitions. Designers can define fine-grained permissions for each task's events (e.g., who can *assign*, *cancel*, or *finish* a task), ensuring a robust RBAC model within the

The screenshot shows a web application interface for 'etask.netgrif.cloud/portal/cases'. The user is logged in as 'Gabo Juhás'. The main content area displays a 'Bulk approval' form for 'Order Instance 351'. The form has a header with 'Case', 'Title', 'Priority', and 'Assignee'. The main form area contains several input fields and a table of invoices.

Case	Title	Priority	Assignee
Order Instance 351	Bulk approval	high	Gabo Juhás

Below the header, there are input fields for 'Order Number', 'Subject', 'Supplier', and 'Order File'. The 'Order File' field has a red 'X' icon, indicating an error.

The main section is titled 'Invoices of the order to approve'. It contains a table with three rows of invoice data:

Invoice number	Amount	Invoice file	Parent_order	Decision
Invoice1	\$65,000.00	invoice_A6.pdf	65df9b3202b84800ed45274	Accept
Invoice2	\$89,000.00	invoice_Q7.pdf	65df9b3202b84800ed45274	Later
Invoice3	\$37,000.00	invoice_A3.pdf	65df9b3202b84800ed45274	Reject

At the bottom of the form, there are buttons for 'DELEGATE', 'REASSIGN', 'CANCEL', 'FINISH', and 'COLLAPSE'.

Figure 4.2: A dynamic form in eTask demonstrating Bulk Approval: parent process (Order) synchronizing events across multiple child objects (Invoices).

application.

Logic Implementation and Interactive Simulation: Complex business rules are scripted via Groovy actions attached to events. To verify the integrated model, NAB provides an **interactive simulation** mode. This allows for manual triggering of assign, finish and cancel events of transitions to play token game to manually check the correctness of the life-cycle.

4.4 Deployment and Runtime Interpretation

Once the XML artifacts are uploaded to *eTask*, the platform immediately instantiates the application without recompilation.

Instance and Task Management The engine manages concurrent instances (cases) within a unified workspace. It dynamically calculates enabled transitions, i.e. available tasks for each user based on the Petri net marking and RBAC permissions.

Cross-Process Coordination The runtime's unique capability is demonstrated during the **Bulk Approval** of invoices (see Figure 4.2). When a finish event is triggered in the *Order* case, the engine automatically synchronizes this event with all referenced *Invoice* cases, ensuring atomic and consistent state updates across the process hierarchy.

Chapter 5

Related Work and Comparison

The *Netgrif Platform* occupies a unique niche by integrating formal modeling with rapid application development in *NAB* and execution in *eTask*.

Petri Net Tools: Unlike *CPN Tools* [Wes13], *Renew* [Cab+09], or *ProM* [Aal+05], which prioritize formal analysis and model checking, the *NAB* is a **generative** environment. Its primary goal is the visual composition of executable *Petriflow classes*, avoiding manual XML coding and bridging the gap between theory and runtime.

BPMN and Process Engines: Compared to business-oriented modelers (*Signavio*, *Bizagi*) or engines like *Camunda*, Netgrif provides tighter vertical integration. While *Camunda* often requires external services for persistence and frontend, the *eTask* powered by *NAE* natively manages the entire state, including MongoDB persistence and Elasticsearch indexing within a single artifact.

Low-Code Platforms: Unlike *Microsoft Power Apps* or *Mendix*, which often rely on fragmented logic (e.g., combining with Power Automate and external DBs), Netgrif offers a **process-centric object-oriented** paradigm. It treats the Petri net not as documentation, but as the "DNA" of the application, where data, logic, and UI are inseparable. In conclusion, Netgrif is a platform designed for seamless object-lifecycle coordination through a formal foundation.

Chapter 6

Conclusion

We presented the *Netgrif Platform* as a model-driven environment for object-centric processes using *Petriflow*. Unlike traditional BPM, it provides vertical integration of lifecycles, data, and UIs in executable artifacts. The platform is proven in **academia** for teaching process-driven application development and in **industry** for scalable applications in the healthcare, insurance, utility, telecom, and public sectors.

Evolution and Future Roadmap. The platform's development continues toward a fully integrated ecosystem. While the *Builder* and *eTask* currently require manual XML synchronization, our immediate roadmap includes a unified API-based integration to enable seamless, direct deployment from the modeling to the execution environment. Furthermore, we aim to simplify the query language and introduce graphical modeling for class inheritance, interfaces and cross-class synchronization. By bridging the gap between descriptive object-centric models and prescriptive executable applications, the Netgrif Platform offers a path toward a next-generation paradigm of process-centric software engineering.

Bibliography

- [AB20] Wil M. P. van der Aalst and Alessandro Berti. “Discovering Object-Centric Petri Nets”. In: *Fundamenta Informaticae* (2020). DOI: 10.48550/arXiv.2010.02047.
- [Aal+09] Wil M. P. van der Aalst et al. “Soundness of Workflow Nets with Reset Arcs”. In: *Trans. Petri Nets Other Model. Concurr.* 3 (2009), pp. 50–70. DOI: 10.1007/978-3-642-04856-2_3.
- [Aal+05] Wil M.P. van der Aalst et al. “The ProM Framework: A Combined Tool for Process Mining and Analysis”. In: *International Conference on Application and Theory of Petri Nets*. Springer, 2005, pp. 439–448. DOI: 10.1007/11494744_25.
- [AJ15] Mohammed A. Alqarni and Ryszard Janicki. “On Interval Process Semantics of Petri Nets with Inhibitor Arcs”. In: *Petri Nets*. Vol. 9115. Lecture Notes in Computer Science. Springer, 2015, pp. 77–97. DOI: 10.1007/978-3-319-19488-2_4.
- [BDM11] Robin Bergenthum, Jörg Desel, and Sebastian Mauser. “Workflow Nets with Roles”. In: *EMISA*. Vol. P-190. LNI. GI, 2011, pp. 65–78. URL: <https://www.fernuni-hagen.de/sttp/docs/2011workflownetswithrolesemisasubmitted.pdf>.
- [Cab+09] Lawrence Cabac et al. “Renew: The Reference Net Workshop”. In: *Transactions on Petri Nets and Other Models of Concurrency II* (2009), pp. 1–25. URL: <https://ceur-ws.org/Vol-1372/paper18.pdf>.
- [DP24] Jochen De Weerd and Luise Pufahl, eds. *Business process management workshops*. en. Lecture Notes in Business Information Processing. Cham: Springer Nature Switzerland, 2024. DOI: 10.1007/978-3-031-50974-2.
- [Des05] Jörg Desel. “Process Modeling Using Petri Nets”. In: *Process-Aware Information Systems*. Wiley, 2005, pp. 147–177. DOI: 10.1002/0471741442.ch7.

- [DJ01] Jörg Desel and Gabriel Juhás. ““What Is a Petri Net?” Informal Answers for the Informed Reader”. In: *Unifying Petri Nets: Advances in Petri Nets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–25. ISBN: 978-3-540-45541-7. DOI: 10.1007/3-540-45541-8_1.
- [DR15] Jörg Desel and Wolfgang Reisig. “The concepts of Petri nets”. In: *Softw. Syst. Model.* 14.2 (2015), pp. 669–683. DOI: 10.1007/s10270-014-0423-3.
- [GGR25] Katarzyna Gdowska, María Teresa Gómez-López, and Jana-Rebecca Rehse, eds. *Business process management workshops*. en. Lecture Notes in Business Information Processing. Cham: Springer Nature Switzerland, 2025. DOI: 10.1007/978-3-031-78666-2.
- [JK09] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer Science & Business Media, 2009. DOI: 10.1007/b95112.
- [JJP23] G. Juhás, A. Juhásová, and L. Petrovič. “Low-Code Languages in IT Education: Integrating Theory and Practice”. In: *Prof. ICETA 2023 - 21st Year of International Conference on Emerging eLearning Technologies and Applications*. IEEE, 2023, pp. 249–257. DOI: 10.1109/ICETA61311.2023.10343807.
- [Juh+21] G. Juhás et al. “Petriflow language and Netgrif Application Builder”. In: *BPM (PhD/Demos)*. Vol. 2973. CEUR Workshop Proceedings. CEUR-WS.org, 2021, pp. 171–175. URL: https://ceur-ws.org/Vol-2973/paper_284.pdf.
- [Juh+22] G. Juhás et al. “Low-code platforms and languages: the future of software development”. In: *Proc. 20th Anniversary of IEEE International Conference on Emerging eLearning Technologies and Applications, ICETA 2022*. IEEE, 2022, pp. 286–293. DOI: 10.1109/ICETA57911.2022.9974697.
- [Juh00] Gabriel Juhás. “On Semantics of Petri Nets Over Partial Algebra”. In: *26th Conference on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic, November 27 - December 4, 1999 Proceedings*. Ed. by Miroslav Bartošek Jan Pavelka Gerard Tel. Vol. 1725. Lecture Notes in Computer Science. Springer, 2000, pp. 414–422. DOI: https://doi.org/10.1007/3-540-47849-3_29.
- [JMP24] Gabriel Juhás, Milan Mladoniczky, and Lubos Petrovic. “Practical Experience with Petriflow: Enriched Process Models Serving as Implementation”. In: *Modellierung 2024 - Workshop Proceedings, Potsdam, Germany, March 12-15, 2024*. Ed. by Holger Giese and Kristina Rosenthal. Gesellschaft für Informatik e.V., 2024, p. 24. DOI: 10.18420/MODELLIERUNG2024-WS-024.

- [KP09] Ekkart Kindler and Laure Petrucci. “Towards a Standard for Modular Petri Nets: A Formalisation”. In: *Applications and Theory of Petri Nets*. Ed. by Giuliana Franceschinis and Karsten Wolf. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 43–62. ISBN: 978-3-642-02424-5. DOI: 10.1007/978-3-642-02424-5_5.
- [KWM04] Olaf Kummer, Frank Wienberg, and Daniel Moldt. “Renew: The Reference Net Workshop”. In: *Tool Presentations, 25th International Conference on Applications and Theory of Petri Nets*. 2004. URL: <https://www2.informatik.uni-hamburg.de/tgi/publikationen/public/data/2003/Kummer+03/Kummer+03.pdf>.
- [Val78] Rüdiger Valk. “Self-Modifying Nets, a Natural Extension of Petri Nets”. In: *ICALP*. Vol. 62. Lecture Notes in Computer Science. Springer, 1978, pp. 464–476. DOI: 10.1007/3-540-08860-1_35.
- [Vog02] Walter Vogler. “Partial order semantics and read arcs”. In: *Theor. Comput. Sci.* 286.1 (2002), pp. 33–63. DOI: 10.1016/S0304-3975(01)00234-1.
- [Wes13] Michael Westergaard. “The CPN Tools 4: Multi-formalism and Scalability”. In: *Application and Theory of Petri Nets and Concurrency*. Vol. 7927. LNCS. 2013, pp. 400–409. URL: https://ceur-ws.org/Vol-1021/paper_3.pdf.