NATIONAL RESEARCH UNIVERSITY HIGHER SCHOOL OF ECONOMICS

###### St. Petersburg School of Physics, Mathematics, and Computer Science

###### Department of Applied Mathematics and Computer Science

**project proposal:**

Kotlin DSL for Business Processes Automation in Telecom and Underlying Engine for their Execution

Kirill D. Kreslavski

BPM-202S

Academic Supervisor

Marina G. Ivanova, lead engineer at Nexign

Saint Petersburg

2024

**Table of Contents**

[**Abstract** 2](#_Toc161266460)

[**Introduction** 3](#_Toc161266461)

[**Background.** 3](#_Toc161266462)

[**Problem Statement.** 3](#_Toc161266463)

[**Professional Significance.** 3](#_Toc161266464)

[**Delimitations of the Study.** 4](#_Toc161266465)

[**Definitions of Key Terms.** 4](#_Toc161266466)

[**Literature Review** 4](#_Toc161266467)

[*Frameworks / libraries and products for business processes/scenarios describing and development* 4](#_Toc161266468)

[*Frameworks / libraries and products for finite state machines development* 6](#_Toc161266469)

[**Methodology** 7](#_Toc161266470)

[*Research and Analysis of Existing Analogues.* 7](#_Toc161266471)

[*Development of Kotlin DSL for Scenario Description.* 7](#_Toc161266472)

[*Implementation of Production-Ready and Testing Engines.* 7](#_Toc161266473)

[*Automated Testing for System Reliability.* 8](#_Toc161266474)

[*Validation through User Experience Feedback.* 8](#_Toc161266475)

[**Achieved Results** 8](#_Toc161266476)

[*Comprehensive Kotlin DSL.* 8](#_Toc161266477)

[*Streamlined Scenario Descriptions.* 8](#_Toc161266478)

[*Modular Separation.* 9](#_Toc161266479)

[*Automatic Scenario Description Generation* 9](#_Toc161266480)

[**Anticipated Results** 9](#_Toc161266481)

[*Enhanced Engine Functionality* 9](#_Toc161266482)

[*Automatic Scenario Description Generation* 9](#_Toc161266483)

[*Improved Multi-Threading Capabilities* 9](#_Toc161266484)

[*Expanded Documentation and Testing* 10](#_Toc161266485)

[**Conclusion** 10](#_Toc161266486)

[**References** 12](#_Toc161266487)

[**Appendices** 13](#_Toc161266488)

# **Abstract**

Nexign, a Russian IT company specializing in OSS/BSS software and business digitalization solutions, faces challenges in developing and describing high-load business scenarios efficiently. The existing approach results in lengthy and convoluted code plus some autogenerated code, based on non-human readable configurations created in GUI. The goal of this research is firstly to develop a Kotlin DSL, using cutting age technologies, world’s best practices and most modern solutions, to streamline scenario development, reducing Java boilerplate, creating one source of truth for scenarios thus minimizing human errors and secondly to create a compact engine for scenario execution resulting in improved readability and maintainability.

The paper consists of Abstract, Introduction, Literature Review, Methodology, Achieved Results, Anticipated Results, Conclusion and References.

# **Introduction**

**Background.** Nexign, a prominent IT firm based in Russia specializing in OSS/BSS software and innovative business digitalization solutions, encounters hurdles in the efficient development and description of high-load business scenarios.

**Problem Statement.** The current method yields extensive and intricate code, coupled with autogenerated code stemming from non-human-readable configurations generated within a GUI environment. This research endeavors to achieve two primary objectives: firstly, to pioneer the development of a cutting-edge Kotlin DSL by harnessing state-of-the-art technologies, best practices, and contemporary solutions. This DSL aims to streamline scenario development, curtail Java boilerplate, establish a singular source of truth for scenarios, and thereby minimize human errors. Secondly, the research aims to craft a compact engine for scenario execution, thereby enhancing readability and maintainability of the system.

**Professional Significance.** The professional significance of this project lies in its commitment to open-source principles and its potential impact on the software development community. By developing a Kotlin DSL, engine, and comprehensive documentation, all available under the permissive MIT license on GitHub, this project fosters collaboration and innovation within the industry. The open nature of the project encourages transparency, peer review, and contributions from developers worldwide. Moreover, the accessibility of the DSL and engine facilitates the adoption of best practices and modern solutions in scenario development and execution across various software projects. By democratizing access to these tools, this project empowers developers to streamline their workflow, reduce errors, and enhance the maintainability of their systems. Ultimately, the professional significance of this endeavor lies in its capacity to drive positive change and foster a culture of openness and collaboration within the software development community.

**Delimitations of the Study.** This study will not bring up such topics as creating high-load distributed systems, as well as their maintaining. In terms of architecture of distributed OSS/BSS we will only discuss the smallest, yet one of the most important ones if not the most important – scenario execution engine and worker. We will also talk about documenting and autogenerating pictures, describing the scenarios for those of the global system users, who are far from the source code itself and are mostly some business process architects or managers.

## **Definitions of Key Terms.**

* OSS/BSS – Generic term that refers to the key back-office software systems required to run a cellular network. More commonly known as business support system (BSS)/operations support system (OSS)
* DSL – Domain Specific Language
* GUI – Graphical User Interface
* CI/CD – Continuous Integration / Continuous Development
* BPMN – Business Process Model and Notation

The paper consists of Abstract, Introduction, Literature Review, Methodology, Achieved Results, Anticipated Results, Conclusion and References.

# **Literature Review**

The following paragraph contains review of existing analogues of different types, describing their advantages, disadvantages and common issues.

## *Frameworks / libraries and products for business processes/scenarios describing and development*

**Cadence**. An open-source BPMN engine developed primarily in Golang, boasts extensive functionality across various programming languages, including Go, Java, Python, and Ruby, as documented on its official website (Cadence Documentation, n.d.). However, while its comprehensive feature set makes it a powerful tool, its complexity may prove overwhelming for certain use cases. The engine's expansive nature encompasses the entire breadth of BPMN functionality, which can lead to an abundance of options and preferences, as noted in the Cadence documentation. This level of flexibility, while beneficial in some scenarios, may also result in over-complexity and over-expandability, rendering it unsuitable for more straightforward tasks. In essence, Cadence can be deemed as overkill for projects requiring simpler BPMN solutions (Cadence Workflow Documentation, n.d.). Shortly said – it is an overkill for the task.

**Temporal**. An open-source BPMN framework based on Cadence and primarily written in Java, emphasizes enhanced customization options, as outlined in the official Temporal documentation (Temporal Documentation, n.d.). While offering greater flexibility for tailoring workflows to specific needs, Temporal may present challenges in readability, as noted by some users.

**iWF**. Yet another BPMN framework built upon the Cadence engine, offers a streamlined alternative to Temporal, as highlighted in the documentation provided by Indeed Engineering (Indeed Engineering, n.d.). While iWF boasts simplicity in comparison to Temporal, users have reported challenges stemming from scattered transition definitions within the framework. This suggests that while iWF may offer advantages in terms of ease of use, developers may encounter difficulties in maintaining cohesive transition definitions throughout their workflows.

Common issue – transitions definition is smeared in all across the code, no single-point for scenario description, not humanly readable.

## *Frameworks / libraries and products for finite state machines development*

**KStateMachine.** Maintained by the KStateMachine organization, offers a compact solution for managing state machines (KStateMachine, n.d.). While it simplifies state machine implementation, users have noted limitations in decomposition and visualization capabilities. However, the framework's modular design allows for easy implementation of custom execution engines (KStateMachine, n.d.).

**StateMachine** (tinder.com). StateMachine, maintained by Tinder, provides developers with a straightforward approach to building pure state machines (Tinder, n.d.). However, users have noted shortcomings in the framework, including the absence of an execution engine and difficulties in managing code organization, leading to code sprawl. This suggests that while StateMachine simplifies the process of defining state machines, developers may face challenges when it comes to implementing and executing these state machines within their applications. Additionally, the framework does not include features for generating visual representations of state machines, which may hinder developers' ability to visualize and debug their workflows effectively. Despite these limitations, StateMachine remains a useful tool for building basic state machines within software applications (Tinder, n.d.).

**KFSM**. Developed by Open Jump Co., offers a Kotlin DSL API tailored for scenario description, as documented in the repository provided by the organization (Open Jump Co., n.d.). However, despite its provision of a convenient DSL, users have noted the need for extensive code when describing scenarios, even for relatively simple state machines. This observation suggests that while KFSM streamlines the process through its DSL, developers may still face challenges in managing code complexity. On the positive side, KFSM offers automatic image generation, which enhances visualization and understanding of the state machines created within the framework.

# **Methodology**

*Research and Analysis of Existing Analogues.* In the initial phase of our project, we conducted a thorough examination of existing BPMN frameworks and state machine libraries to identify their strengths and weaknesses. This research allowed us to gain insights into the landscape of available tools and understand the challenges faced by developers in scenario development. Through this analysis, we identified Cadence, Temporal, iWF, KStateMachine, and StateMachine as notable analogues, each offering unique features and functionalities. While Cadence provided extensive functionality, it suffered from complexity and over-expandability. Temporal offered easier customization but lacked readability, while iWF presented a simpler alternative but suffered from scattered transition definitions. KStateMachine and StateMachine focused on pure state machine building but lacked certain essential features such as execution engines and image generation capabilities.

*Development of Kotlin DSL for Scenario Description.* The centerpiece of our project is the development of a Kotlin DSL for scenario description, aimed at enhancing readability and conciseness. Leveraging the concept of type-safe builders in Kotlin, as outlined in the official Kotlin documentation on type-safe builders (Kotlinlang.org, n.d.), we aimed to create a DSL that would enable developers to define scenarios in a clear and intuitive manner. Drawing inspiration from examples such as VillageDSL (zsmb13/VillageDSL, n.d.), we designed our DSL to represent scenarios as graphs built of operations, such as an example, provided in Appendices as Picture 1. Provided Scenario as a code snippet can be found in Appendices as Picture 2. Notably, operations can be instantiated via functional interfaces, providing a simple and elegant way to define scenario behavior (Kotlinlang.org, n.d.). A code example can be seen in Appendices as Picture 3.

*Implementation of Production-Ready and Testing Engines.* To ensure the robustness and reliability of our solution, we implemented production-ready and testing engines for scenario execution. Our approach involves a mix of Java ThreadPools and Kotlin Coroutines technologies to enable efficient multithreading capabilities. This hybrid approach allows for optimal utilization of system resources while ensuring scalability and responsiveness.

*Automated Testing for System Reliability.* In addition to developing production-ready engines, we prioritized the implementation of automated tests using the JUnit framework. This approach allows us to systematically validate the functionality of our system and identify any potential issues or bugs. By incorporating automated testing into our development workflow, we aim to ensure the reliability and stability of our solution under various scenarios and use cases.

*Validation through User Experience Feedback.* Finally, to validate the proof of concept and assess the usability of our solution, we plan to seek feedback from specialists in scenario development and scenario architecture. These experts will review the entire solution and provide insights into its practicality and potential for adoption in production environments. By soliciting feedback from domain experts, we aim to ensure that our solution meets the needs and expectations of real-world users, ultimately enhancing its effectiveness and usability.

# **Achieved Results**

*Comprehensive Kotlin DSL.* The development process yielded a robust Kotlin DSL, meticulously designed with a formal language structure architecture. This DSL serves as the foundation for scenario description, providing developers with intuitive syntax and powerful abstractions to define complex business scenarios efficiently.

*Streamlined Scenario Descriptions.* Through careful design and implementation, scenario descriptions were streamlined to ensure conciseness and clarity. Transitions between states were optimized for readability, while high-level error handling mechanisms were integrated to enhance system reliability and resilience.

*Modular Separation.* One of the key achievements was the modular separation of DSL, engine entities, and scenario packages. This modular architecture promotes code reusability, maintainability, and scalability, allowing for easy integration of new features and enhancements in future iterations.

*Automatic Scenario Description Generation*. An essential feature of the developed system is the automatic generation of human-readable scenario descriptions and execution logs. This capability enhances developer productivity by eliminating the need for manual documentation and ensuring consistency across scenarios.

# **Anticipated Results**

*Enhanced Engine Functionality*. The next phase of development aims to enhance the engine with support for external dependencies and REST API services. By integrating these features, the engine will become more versatile and interoperable, enabling seamless integration with external systems and services.

*Automatic Scenario Description Generation*. Another anticipated result is the implementation of automatic generation of scenario descriptions as XML specifications and schematic graph images. This feature will provide developers with visual representations of scenarios, aiding in understanding and debugging complex workflows.

*Improved Multi-Threading Capabilities*. To enhance performance and scalability, efforts will be made to improve multi-threading capabilities within the engine. By leveraging the power of Kotlin coroutines and Java ThreadPools, the system will be optimized to handle high-load scenarios more efficiently.

*Expanded Documentation and Testing*. Finally, the project aims to expand documentation and incorporate testing into CI/CD pipelines. Comprehensive documentation will serve as a valuable resource for developers, while automated testing will ensure the reliability and stability of the system in real-world scenarios.

# **Conclusion**

In conclusion, the development of the Kotlin DSL and compact engine presented in this research aligns seamlessly with the identified aim and objectives, addressing the challenges faced by Nexign in scenario development and execution. Through meticulous research and analysis of existing analogues, we have identified the shortcomings of current methods and outlined a clear path towards improvement. By pioneering the development of a cutting-edge Kotlin DSL, we have successfully addressed the need for a more efficient and readable approach to scenario description. Through the implementation of state-of-the-art technologies and best practices, we have created a tool that not only reduces Java boilerplate but also establishes a singular source of truth for scenarios, thereby minimizing human errors.

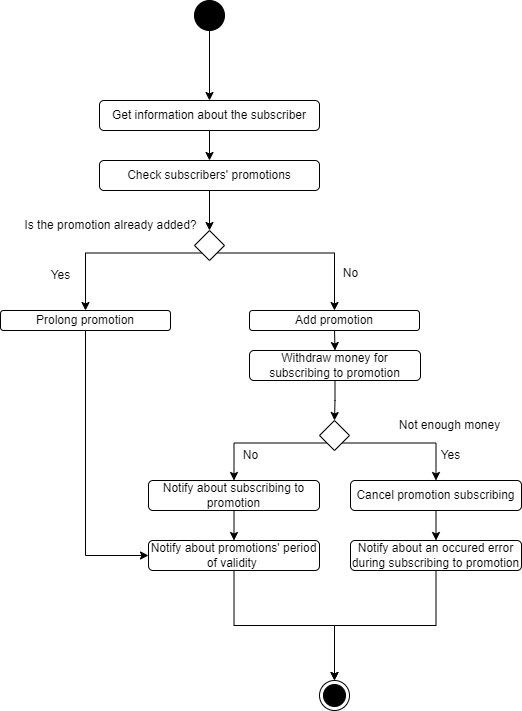
Furthermore, the crafting of a compact engine for scenario execution represents a significant step towards enhancing the readability and maintainability of the system. By leveraging the capabilities of Kotlin and integrating multithreading technologies, we have created a robust engine that ensures optimal performance and scalability. The professional significance of this project cannot be overstated, as it embodies Nexign's commitment to open-source principles and collaboration within the software development community. By making the Kotlin DSL, engine, and comprehensive documentation available under the permissive MIT license on GitHub, we have democratized access to these tools, empowering developers worldwide to streamline their workflow, reduce errors, and enhance the maintainability of their systems.

In summary, the proposed Kotlin DSL and compact engine represent a promising solution to the challenges faced by Nexign and the broader software development community. Through innovation, collaboration, and a commitment to openness, this project has the potential to drive positive change and foster a culture of excellence within the industry.

# **References**

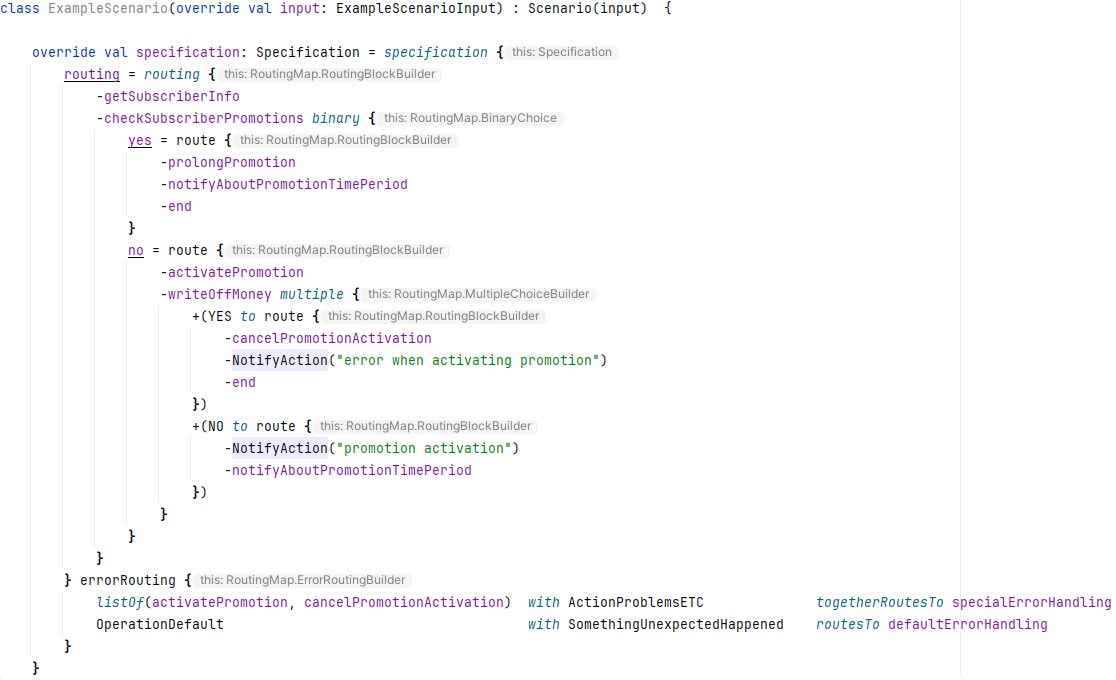
* Baeldung. (n.d.). Kotlin DSL. Retrieved from <https://www.baeldung.com/kotlin/dsl>
* Cadence Workflow Documentation. (n.d.). Retrieved from <https://cadenceworkflow.io/docs/use-cases/dsl/>
* Gartner (2011) IT Terms Definitions <https://webcitation.org/65AkPdIv3?url=http://www.gartner.com/technology/it-glossary/>
* Kotlinlang.org. (n.d.). Type-Safe Builders. Retrieved from <https://kotlinlang.org/docs/type-safe-builders.html>
* Kotlinlang.org. (n.d.). Functional Interfaces. Retrieved from <https://kotlinlang.org/docs/fun-interfaces.html>
* KStateMachine. (n.d.). kstatemachine. GitHub. <https://github.com/KStateMachine/kstatemachine>
* Open Jump Co. (n.d.). KFSM. GitHub. <https://github.com/open-jumpco/kfsm>
* Temporal Documentation (n.d.) Retrieved from <https://docs.temporal.io/>
* Tinder. (n.d.). StateMachine. GitHub. <https://github.com/Tinder/StateMachine>
* Indeed Engineering. (n.d.). iWF. GitHub. <https://github.com/indeedeng/iwf>
* zsmb13/VillageDSL. (n.d.). GitHub. <https://github.com/zsmb13/VillageDSL>

# **Appendices**

* Picture 1

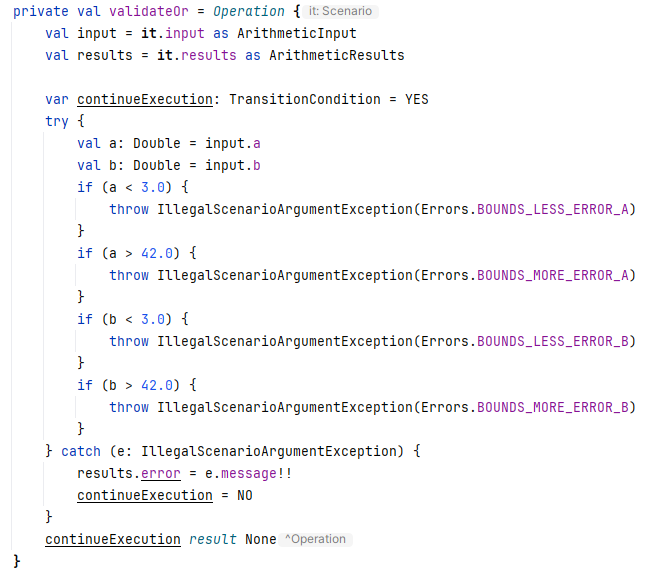
An example of business process / scenario in Telecom.

* Picture 2



An example of how the Scenario can be described in code via DSL.

* Picture 3



An example of Operation created using SAM interface.