Complex Systems Informatics and Modeling Quarterly (CSIMQ)
eISSN: 2255-9922
Published online by RTU Press, https://csimq-journals.rtu.lv
Article 143, Issue 25, December 2020/January 2021, Pages 1–18
https://doi.org/10.7250/csimq.2020-25.01

A Modeling Method for Model-Driven API Management

Andrei Chiș*

Faculty of Economics and Business Administration, Babeș-Bolyai University,
Teodor Mihali Street Nr. 58-60, Cluj-Napoca, 400591, Romania

andchis@gmail.com

Abstract. This article reports on the Design Science engineering cycle for implementing a modeling method to support model-driven, process-centric API management. The BPMN standard was hereby enriched on semantic, syntactic and tool levels in order to provide a viable solution for integrating API requests with diagrammatic business process models in order to facilitate the documentation or testing of REST API calls directly in a modeling environment. The method can be implemented by stakeholders that need to map and manage their API ecosystem, thus gaining more API management agility and improving their software engineering productivity. By assimilating API ecosystem conceptualization in the modeling environment, the proposal differs from both RPA (which typically employs non-BPMN process diagramming e.g., in UIPath) and BPM Systems (which typically isolate all API-related semantics outside the process modeling language to keep the diagrammatic representation standard-compliant).
Keywords: BPMN, API Modeling, Model-Driven Software Engineering, API Ecosystem Management, REST API Semantics.

1 Introduction

This article reports on Design Science work aiming to engineer a modeling method that establishes a functional bridge between the BPMN standard and executable REST APIs. As a functional artifact, the result takes the form of an extension for the open source Bee-Up tool – a multi-language modeling tool supporting BPMN, UML, ER, EPC and other notations [1], [2] that is also open to extensions and experimentation.

In order for a business actor to maintain a strong market position, it is of paramount importance to have maximum efficiency in its internal business processes and the capability to easily adapt to ecosystem changes. The expansion of organizations brought new challenges regarding business operations management, optimization of strategic business decisions and the improvement of business workflows. Consequently, business ecosystems are being complemented by API ecosystems for which business processes act as orchestrations [3].

* Corresponding author

Reference: A. Chiș, "A Modeling Method for Model-Driven API Management," Complex Systems Informatics and Modeling Quarterly, CSIMQ, no. 25, pp. 1–18, 2020. Available: https://doi.org/10.7250/csimq.2020-25.01

Additional information. Author's ORCID iD: A. Chiș – https://orcid.org/0000-0003-0173-7250. PII S225599222000143X. Received: 31 October 2020. Accepted: 9 December 2020. Available online: 31 December 2020.

However, there is a lack of modeling methods for managing those API ecosystems and for mapping them on business process models that may already exist in an organization's process repository.

The motivation for this project derives from larger work investigating possibilities to bridge the "design-time vs. run-time" gap in terms of API management, in a Business Process Management context. Traditional BPM systems (e.g., Camunda [4]) prefer to decouple the API ecosystem from the modeling environment – i.e., to keep the modeling method standard-compliant (e.g., BPMN-based) and have all API-related details as external "task workers". The alternative approach proposed in this article is to capture a conceptualization of the API ecosystem in the modeling environment, thus allowing its diagrammatic configurations before any actual data must be passed to run-time APIs. An agile metamodeling approach based on the Agile Modeling Method Engineering (AMME) [5] methodology is employed to tailor an open source BPMN implementation for the aforementioned purpose.

The remainder of the article is structured as follows: Section 2 discusses the problem statement and presents a solution overview. Section 3 provides a short review of relevant related works, while Section 4 gives detailed information about the research and engineering methods hereby employed. Section 5 discusses the design of the modeling tool and its implementation, while Section 6 presents the artifact evaluation and assesses the key criteria taken into account for validating the modeling method. Section 7 concludes the article.

2 Problem Statement and Solution Overview

In order to achieve efficiency and adaptability in complex business scenarios that rely on API ecosystems, traditional companies have to evolve in the direction of ambidextrous organizations [6]. Business process modeling languages and methods, alongside process automation techniques support this kind of enterprise evolution. Some of the popular solutions for automating processes are Robotic Process Automation (RPA) [7], API-based automation, or a mix of those, as RPA platforms are gradually incorporating back-end automation features.

Currently, despite the existence of a variety of languages for API descriptions, such as RAML [8] or API Blueprint [9], there is a lack of domain-specific conceptual modeling methods for designing API calls for both analysis and execution purposes, thus ensuring a bridging between the design-time and run-time aspects while enabling a semantic coupling between BPMN process models [10] and diagrammatic API models. The need for such tooling was derived from the success of the Swagger interface definition language, typically employed as API documentation [11]. The core semantics captured in a Swagger documentation (i.e., of an HTTP request configuration) are assimilated in the hereby proposal in a business process modeling environment by applying the agile customization made possible by the AMME methodology.

The benefits are the ability to integrate the proposed extension in already existing BPMN analysis mechanisms (e.g., model queries, path analysis reports), the possibility to implement API-specific mechanisms (also via model queries over the modeled API ecosystem) and at the same time to run and test API calls directly from the business process modeling environment.

The method's prototype also suggests the possibility of designing a model-driven API management system that could be integrated with more diverse business modeling languages [1]. By doing so, process modeling can evolve from diagrammatic design of business processes to automating, streamlining and executing processes based on BPMN extended with aspects of the run-time environment – something that is currently decoupled from the process modeling environment: in RPA tools like UiPath [12] the *process diagramming component ignores the existence of BPMN*, whereas in BPMN-driven platforms like Camunda [4] the *run-time API aspects are not captured on a diagrammatic level* since they are out of BPMN's scope (limited to providing "Data store" and "Data objects" symbols with no executable granularity).

2.1 Solution Overview and Use Cases

The proposed treatment to the aforementioned Design Problem is a modeling method that extends the BPMN standard with new modeling classes, semantics, syntax and functionality for describing and executing API requests. For prototyping we used the open-source BPMN implementation made available by the Bee-Up Modeling Toolkit [2], extended for this work's purpose with the help of the ADOxx metamodeling environment [13].

The proposed modeling method includes both an API modeling language specification and a functional prototype for launching API requests to real-world REST services from BPMN models. The scripts that implement execution functionalities are built upon the novel modeling language constructs in order to ensure full integration between the components of the modeling method and the external environment, with the help of ADOxx interoperability features (its proprietary ADOScript language [14] interoperating with a PHP-based HTTP client).

Figure 1 (code snippets not intended to be readable) provides an overview of the solution's building blocks and processing flow. The process model elements (BPMN tasks) can be semantically linked to an API model comprising all the API calls available in an API ecosystem.

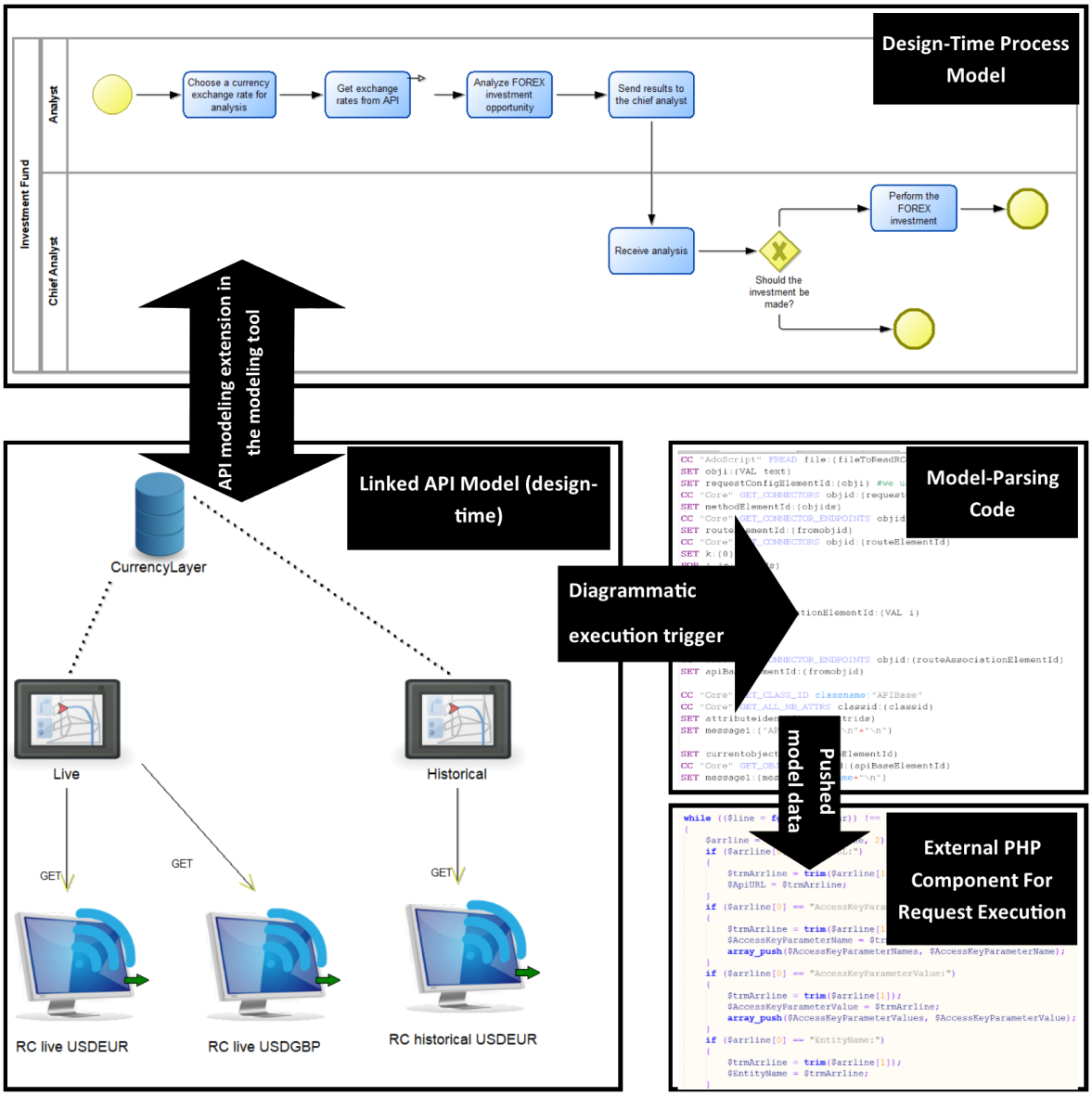

Figure 1. Solution overview

The API ecosystem can have two statuses:

- *As-Is*, if it is an existing ecosystem for which the modeling tool becomes a testing/orchestration tool;
- *To-Be*, if it is a future ecosystem for which the modeling tool becomes a design and analysis tool.

The elements from the API diagram are extensions of the BPMN, both semantically and syntactically, that can be parsed in order to compose the API calls and to execute them. This capability enhances the model-driven software engineering facet of our project, because the diagrammatic model acts as an execution driver, modifying its execution behavior in correlation with the modeling language elements in designing an API call.

The proposed extensions manifest on multiple layers of a modeling method:

1. On *graphical level*, the BPMN toolbar provides new symbols for the added concepts needed to model API calls;
2. On *conceptual level*, those symbols imply new concepts added to the BPMN metamodel, as detailed in Section 5.1, isolated in a separate type of diagram that can be connected to BPMN elements;
3. On *syntactic level*, domain, range and cardinality restrictions were introduced through the built-in settings of the ADOxx platform;
4. On *semantic level*, the new concepts were enriched with REST and HTTP-specific properties (e.g., the HTTP verb, query strings).

On *functional level*, mechanisms for triggering HTTP calls that are configured according to the diagrammatic designs are implemented.

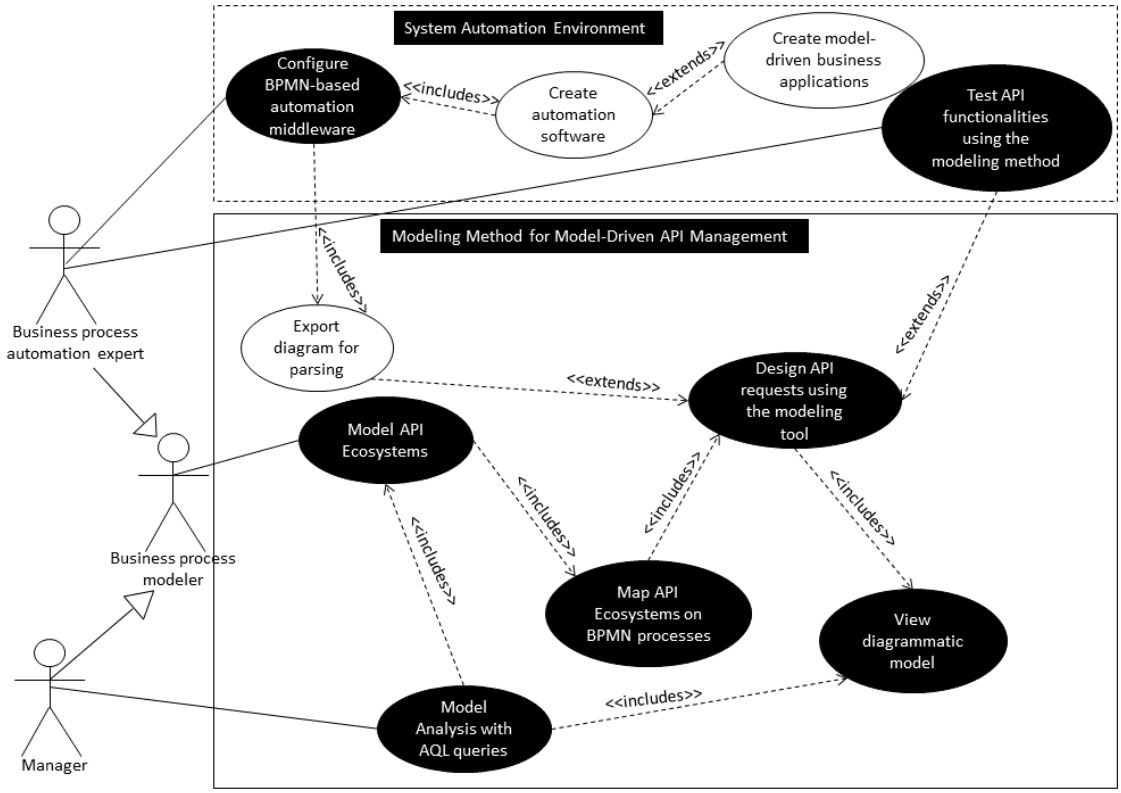

Figure 2. Stakeholders and supported use cases

The relevant use cases reflected in Figure 2 are:

- *For a business process modeler*: the ability to design an API ecosystem, mapping it to BPMN processes and detailing AP request configurations;

- *For a project management role*: the ability to perform ADOxx's model queries over the combination of BPMN-API models;
- *For a process automation expert*: the ability to configure and test run-time API calls in the BPMN modeling environment.

3 Related Works

Swagger [11] is an interface description language for APIs that inspired this research in the sense that Swagger-style API descriptions are hereby assimilated in an extended BPMN modeling language, thus acting as both a conceptual and functional bridge between business process models and API-based execution environments. In its original form, Swagger does not have a conceptual modeling method as it is strictly employed for parsing or documenting API descriptions for the clients of those APIs.

Coming from the side of business process engines, the integration between process descriptions and executable HTTP request configurations has been traditionally achieved in two ways: by completely ignoring BPMN as a process description language (in RPA tools such as UIPath [12]) or by completely decoupling API call semantics from the modeling language (e.g., in BPM systems such as Camunda [4]).

Goldstein et al. presented in [15] how MEMO OrgML can be used as a business process modeling language which could extend the capabilities of a modeling language from the design phase to the runtime phase of process execution. In our case, one of the API modeling method's purposes was to create a bridge between the design time and the run time components of an API request inside a process diagram and to enact it at runtime.

As authors of [16] state, there is an increasing need for more domain-specific targeted modeling languages in the current software engineering world – the proposed BPMN+API hybrid modeling method provides a strong base for designing and executing HTTP-based Web services (with a current focus on REST APIs, although extensions are considered for other interoperability models). In [17], authors present the implementation of the API2MoL engine, which is a rule-based language that allows mapping definitions between Java APIs' specifications and the metamodel that is used to represent them. This way, API objects can be converted into models and models can also be converted into API objects, showcasing a bidirectional model-driven engineering approach. In comparison, our work did not aim to generate new API objects from models, but rather to provide a modeling tool that can be used in order to streamline, test and orchestrate web APIs' execution in business processes. In [18] authors describe a tool for managing cloud service ecosystems using the Open Cloud Computing Interface, strengthened by model-driven engineering. In our case, we focused on API ecosystem management that includes services mostly used for retrieving or sending data, while cloud services are mostly used for acquiring superior computing power or storage space.

The work presented in [19] focuses on modeling RESTful conversations, abstracting the structure of HTTP interactions. This is accomplished by extending the notation that is used in BPMN choreography diagrams. The main purpose in [19] is to represent the interaction sequences that appear in a RESTful API conversation, while our work focuses on linking RESTful APIs to the business processes in which they can be employed. After the linking part, our modeling tool offers the possibility to launch the requests from the modeling environment.

The project presented in [20] showcases the definition of a microservice composition approach that enables the creation of a composition in a BPMN model which improves engineering decisions' analysis quality. Moreover, the referenced work presents how the created BPMN model for microservice implementation can be split into fragments that are executed through an event-based choreography form.

The authors of [21] introduced RESTalk which is an extension to the BPMN choreography diagrams, that allows API developers to render more accurately the client-server interactions of a RESTful API in a diagrammatic form. This enrichment was proposed in order to increase

developers' efficiency and facilitate better understanding of the API structure that needs to be implemented.

The work done in [22] supports the introduction of a platform for defining cloud-specific workflows in BPMN business process engines. The BPMN specification was extended in order to support the orchestration of cloud-specific workflow activities and a new metamodel was proposed in order to map orchestration workflow elements onto extended BPMN elements.

Unlike the referenced works, this article showcases the viability of the AMME methodology for enriching BPMN with aspects pertaining to REST API management and run-time.

The work reported in [23] showcases MetaEdit+ which allows fast method prototyping to enhance domain-specific modeling agility, a goal that is also pursued by this article, but with the help of the Agile Modeling Method Engineering framework, promoted in OMiLAB's ecosystem [24] for conceptualization and operationalization of conceptual modeling methods.

4 Research and Engineering Method

The Design Science research methodology [25] was applied as a research frame for developing the modeling method starting from requirements captured in the use cases presented in Figure 2. AMME [26] was the engineering methodology.

4.1 Research Method

A typical Design Science development cycle is structured in 6 main stages: Problem identification and motivation, Objectives statement, Design and development, Demonstration, Evaluation, and Communication [27], conducting to the development of a usable artifact.

- In the Problem identification and motivation phase we surveyed the state of the art literature, leading to the conclusion that an API domain-specific conceptual modeling method is missing in the API management and process automation environments, with a total decoupling between process modeling standards and API specifications.
- In the Objectives statement step we set the requirements as suggested in Figure 2.
- During the Design and development phase, we employed the AMME methodology, to be detailed in the next subsection, arriving at low technological readiness prototype.
- In the Demonstration step, we tested the API modeling method on several web APIs varying in terms of used parameters, types of keys or headers, in order to show that the treatment has a broad level of application.
- During the Evaluation phase, the treatment was compared to a previous model-driven semantic orchestrator that only read API settings from BPMN annotations, presented in an earlier publication [28]. It was a typical case of comparing a model-driven software engineering artifact (the current one) to a model-aware software engineering artifact, with "model-awareness" referring to when run-time components only query model contents, rather than being generated from model contents [29]; the version reported in this article produces full HTTP request scripts in PHP from API-specific diagrammatic contents.
- The Communication phase consists in disseminating the result and the article at hand is one step in this direction.

4.2 Engineering Method

In order to develop the API modeling method in a way that is coupled with the BPMN 2.0 modeling language, we used the AMME development model. As [5] states, a modeling method comprises a modeling language (syntax, semantics and notation), a modeling procedure and functional mechanisms. AMME [26] is a modeling tool development framework which applies Agile principles [30] to the practice of modeling method engineering. Since it is iterative and aims to produce an artifact tailored for a limited context, it can act as the engineering cycle

implied by the Design Science methodology. The design decision and implementation details derived from applying AMME for the purpose of this research will be detailed in the subsequent sections.

5 Design Decisions and Implementation Details

5.1 Conceptual Design

Bork et al. reported in [31] several approaches to specifying a metamodel – in this article we use a simplified UML class diagram. Figure 3 depicts the BPMN metamodel enriched with the language constructs from the API modeling tool. It contains 3 core classes – *APIBase, Route, RequestConfiguration* and 2 relation classes – *RouteAssociation and Method*. The *APIBase, Route and RequestConfiguration* classes are subclasses of the *APINode* class, which has only a grouping scope in the metamodel.

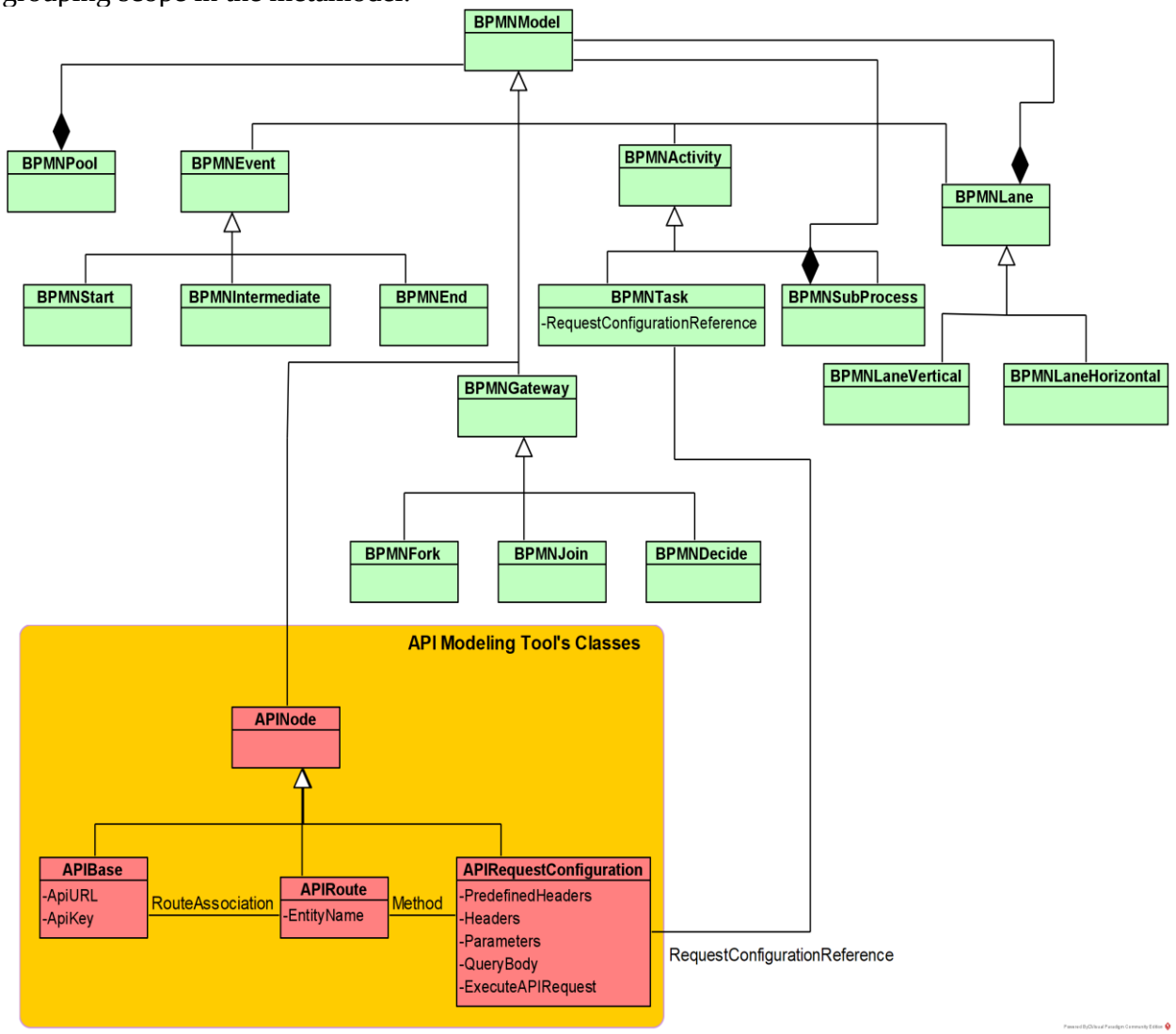

Figure 3. The BPMN metamodel, based on [32] and enriched with API ecosystem concepts

The *APIBase* class stores information about the API's basic elements like the URL or the access key. The *Route* class is used to indicate the route chosen in the API request, while the *RequestConfiguration* class stores request's parameters in order to perform the desired action. This class is also referenced by a specific task from the business process model, which must be executed using a certain API. Alongside this, the *RequestConfiguration* class also offers the

possibility to launch into execution an API request using its embedded functionalities that were programmatically developed during the course of this research project.

The relation classes: *RouteAssociation and Method* are used to link the main classes and are expressed here as associations, considering the fact that the BPMN relation classes are also not represented to keep the metamodel simple. The *RouteAssociation* class has simply a diagrammatic role, informing the modeler which routes are available for a specific API (the basic elements of the API are represented in the *APIBase* class). The Method class is used to link a *Route* to a *RequestConfiguration* and it is also used to define the HTTP method of the request (GET, POST, PUT, PATCH, DELETE).

Some key attributes have been highlighted: the *RequestConfigurationReference* attribute, which was added to the Task class from the BPMN language in order to make links between tasks and their API request implementations through the *RequestConfiguration* class of the API modeling tool's language class structure. Also, the attributes from the *RequestConfiguration* class are the ones that provide the request's functional specificity by allowing the user to choose the type of headers, add request parameters, query body, if needed, and link the *RequestConfiguration* through the suite of request execution scripts using the *ExecuteAPIRequest Programcall* type attribute.

5.2 Syntactic Design

According to [33] any graphical representation of a modeling language "must effectively communicate with business stakeholders" and "support design and problem solving by software engineers". In other words, at least some of the symbols used in a new modeling method prototype should be strongly correlated with the domain-specificity of the application area. Figure 4 shows the domain-specific graphical symbols of the modeling language classes. The APIBase has a database-related symbol because its purpose is to store some of the elements that form the base target for any API request, such as the URL or the API key. The HTTP Route class is represented by a GPS because it must communicate to the modeler the possibility to add one or more endpoints of an API in the diagram. The RequestConfiguration class is the one that effectively launches the request so it must symbolize the connection to a web service. The RouteAssociation class has a graphical role on the diagram helping the modeler to differentiate between different implementations of an API, using the same base and different routes. The HTTP Method class has the shape of an arrow because it can symbolize the transfer of a request through an REST method.

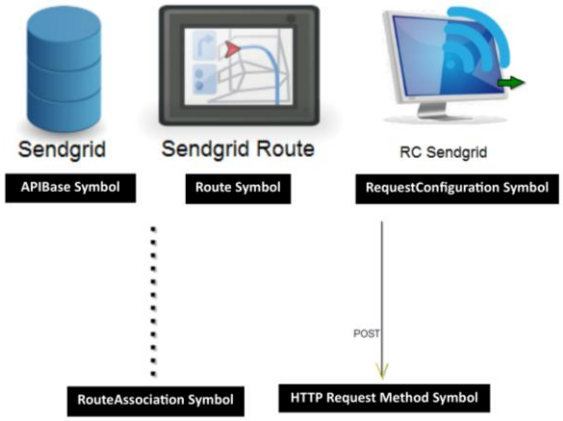

Figure 4. The graphical notations of the modeling tool

Just as important as the visual domain-specificity is the interactive nature of graphical symbols, with hotspots that trigger inter-model semantic links (from BPMN elements to API elements) or execution of external components (HTTP requests to the modeled APIs). These are

the key UI-level bridge between the design-time and run-time of proposed models, suggested in Figure 5. Execution results are also displayed directly in the modeling environment.

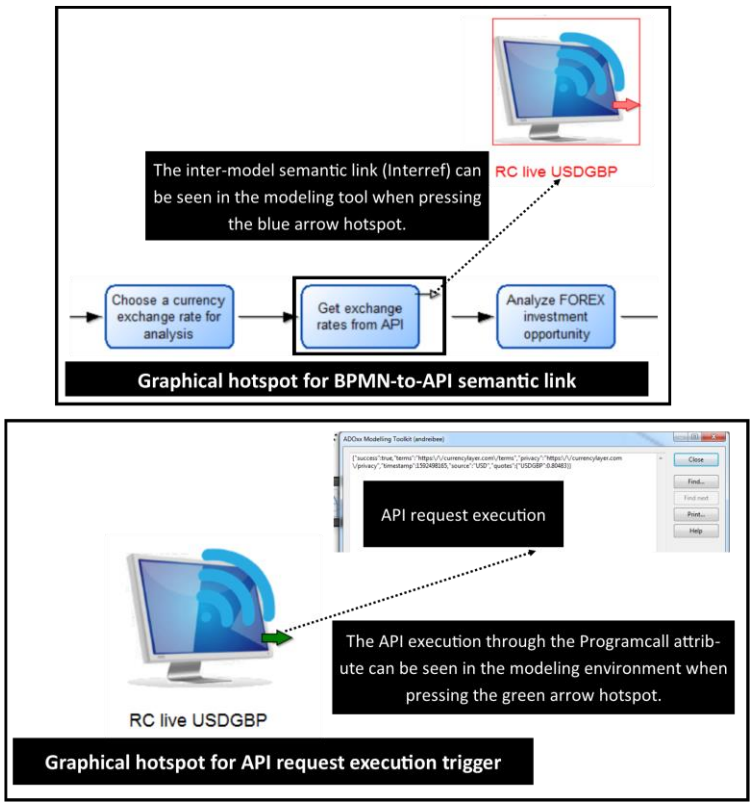

Figure 5. Visual cues acting as interactive triggers

5.3 Functional Components

The executable bridge between the modeling environment and any existing APIs relies on scripting mechanisms created with a mix of ADOxx's internal programming language (ADOScript) and external HTTP management (PHP-based, as it is easier to handle JSON payloads than handling ADOxx's built-in data structures).

The main algorithms that employ the run-time functionalities of the API modeling method are the diagram path parsing algorithm (in ADOScript) and the API request execution algorithm (in PHP). They interoperate in order to achieve the goal of supporting model-driven process automation based on APIs.

Consequently, API models work as controllers for API execution. They can be considered code generators because the PHP-based code handling HTTP requests is dynamically built from diagrammatic design. The RequestConfiguration concept works as an orchestrator. It sends to the model parsing algorithm the necessary data for parsing the path chosen by the modeler when clicking on the RequestConfiguration's visual hotspot. After that, the request execution algorithm launches the API call based on the data pushed by the diagram path parsing algorithm. After the execution phase it sends the results back to the ADOxx environment.

The diagram path parsing algorithm is divided into two main processes, while the API request launching algorithm follows a single process, as shown in Figure 6.

On the left side of Figure 6, a flowchart shows actions that are executed by the subroutine that reconstructs the API diagram path necessary for executing a certain API request. After the modeler chooses a certain request implementation by clicking on its RequestConfiguration object, the algorithm collects all the objects that are needed to successfully launch the API call.

The center of Figure 6 shows the subroutine gathering from the API diagram all the parameters such as API key, URL, request headers or query body that are used in order to launch

the API call. This algorithm parses the objects discovered in the diagram path reconstruction step and then gathers the needed attributes' data from them.

The path reconstructing sub-process and the data gathering sub-process can be regarded as the pillars of the model-driven facet employed in this project.

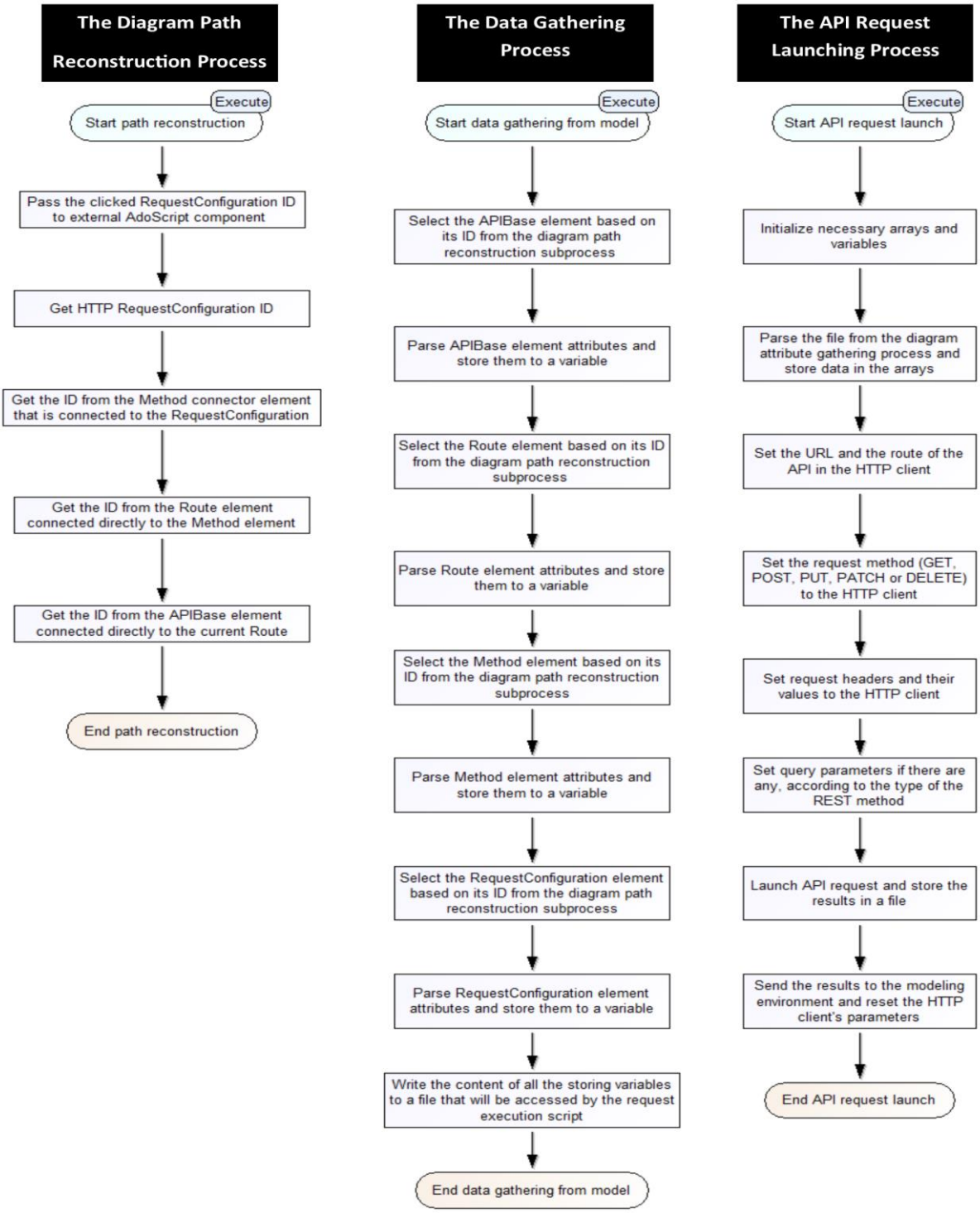

Figure 6. Flowcharts describing the diagram path reconstruction (left), the model parsing and data gathering process (center) and the API request execution process (right)

The following step is the launch of the API request by using the HTTP client. This process is performed after the API parameters are gathered and stored in a file that is accessible to the HTTP request launching script.

The right side of Figure 6 shows the code logic employed in the request execution step. By using a PHP-based HTTP client, the request parameters form the parameter gathering step are set and the request is executed. After that, the response is decoded and sent to the modeler directly in the modeling environment.

5.4 Method Implementation

In order to develop the API modeling tool and its related ecosystem of classes and functionalities we chose a multi-layer architecture that separates the core language constructs from the code of an API request launch.

Figure 7 shows an API description diagram that was built using the modeling method (for a single API of an ecosystem; the simple grouping of multiple such elements in the same canvas acts as a containment relationship, separating ecosystems in different diagrams). The right side shows attributes of the RequestConfiguration class as seen by the process modeler. The PredefinedHeaders attribute is a tabular attribute that consists of a drop-down list from where the user can chose header types – e.g., Content-Type, in the PredefinedHeaderKey field, while the PredefinedHeaderValue field takes its data from another drop-down list which stores some header types such as application/json or application/graphql. We chose to offer these predefined headers and values because they are the most used while interacting with web services through APIs and this way we wanted to limit any possible spelling mistakes. The Headers attribute is a tabular attribute where the user can explicitly set other headers besides Content-Type and Accept, that might be used in specific API requests – e.g., the Authorization header that is needed for some web services that do not store the authentication key inside a request parameter but they get it from a request header. The Parameters attribute is another recordset attribute, used for specifying request parameters that usually act as GET parameters that get appended in a human readable format to the URL of the request. It has two fields: Parameter – where the user sets the parameter's name and Value where the user sets each parameter's value. The QueryBody attribute is of type string and it is used mostly for POST-type requests in which the user can explicitly set the whole body of an API query – e.g., querying a GraphQL [34] or SPARQL endpoint [35].

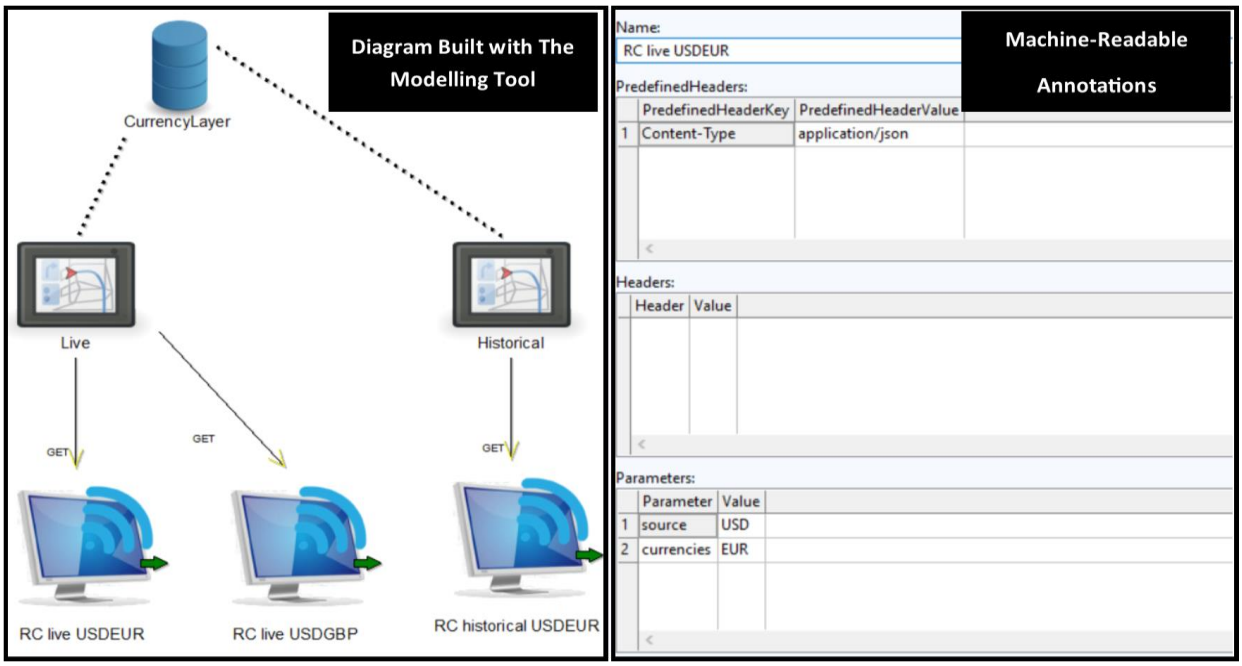

Figure 7. API Diagram built with the presented modeling tool (left) and machine-readable annotations of an HTTP Request Configuration (right)

The Execute API Request attribute is an execution trigger reflected visually as an interactive hotspot, launching the PHP component that will actually execute the API request and return the results in the modeling environment.

Attributes can be visualized and set in the other classes similarly to the way they are here, through the built-in linking UI available in the original Bee-Up implementation.

Figure 8 presents the user interface for creating a semantic link for a BPMN Task.

The model-driven functionality turns diagrams into execution drivers, therefore we have a model-driven software engineering approach that is typically found in process automation, however without capturing the API ecosystem conceptualization in the modeling environment itself – e.g., UIPath stays away from BPMN entirely, whereas BPM platforms like Camunda keep BPMN standard-compliant while decoupling the API semantics completely from the modeling environment.

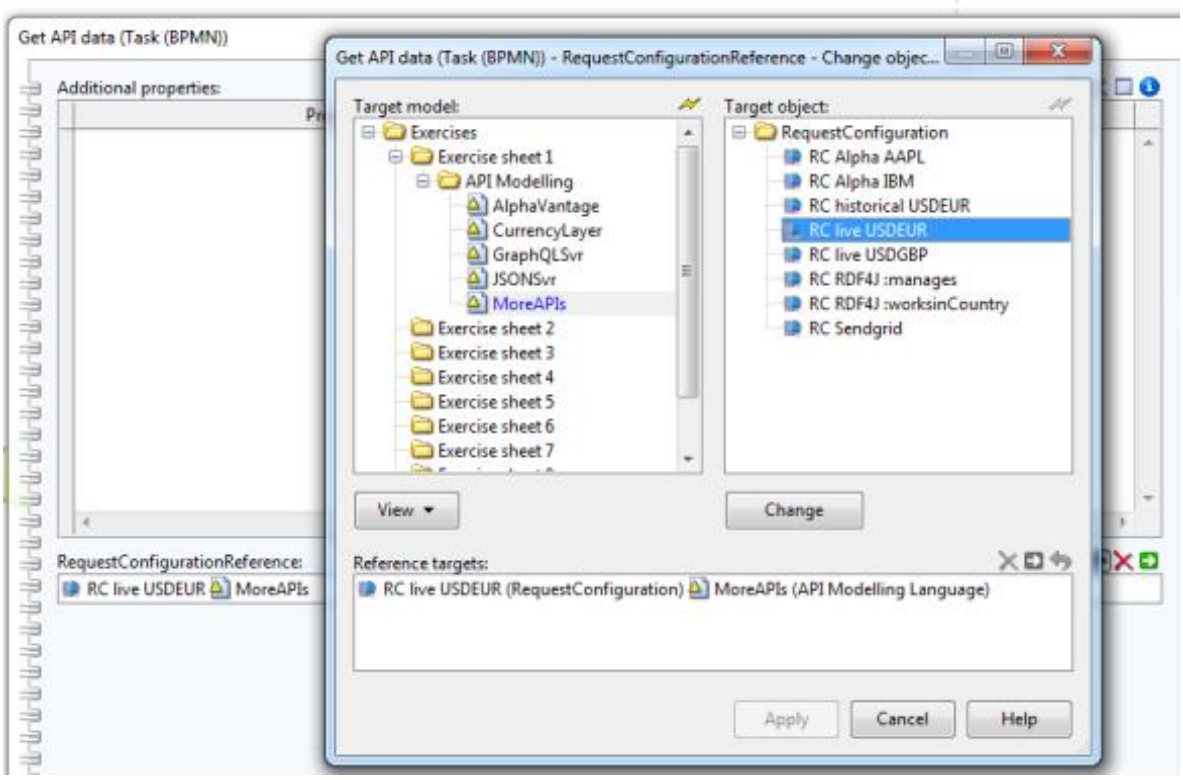

Figure 8. Semantic linking between a BPMN Task and a RequestConfiguration

Figure 9 shows the cycle that is executed every time an API request is launched.

In the beginning, the modeler must design a model that contains the elements necessary for the API request(s) launch. When pressing the RequestConfiguration's execution hotspot, a model-parsing ADO script reconstructs the chosen diagram path and extracts the request-related information from the design-time model. This operation leads to the creation of model-generated code necessary for launching an API call. This code is then pushed to the PHP request launching script that uses a convenient HTTP client. After the response from the API is received, the same PHP script extracts the necessary data and pushes it back into the modeling environment, where the modeler can confirm it, not unlike the way tools like Postman [36] are used for API test purposes – however, with the advantage that here it is done in a modeling environment that (a) acts as a diagrammatic repository of API descriptions; (b) maps those API descriptions on BPMN process tasks, potentially allowing a diagrammatic orchestration of managed APIs.

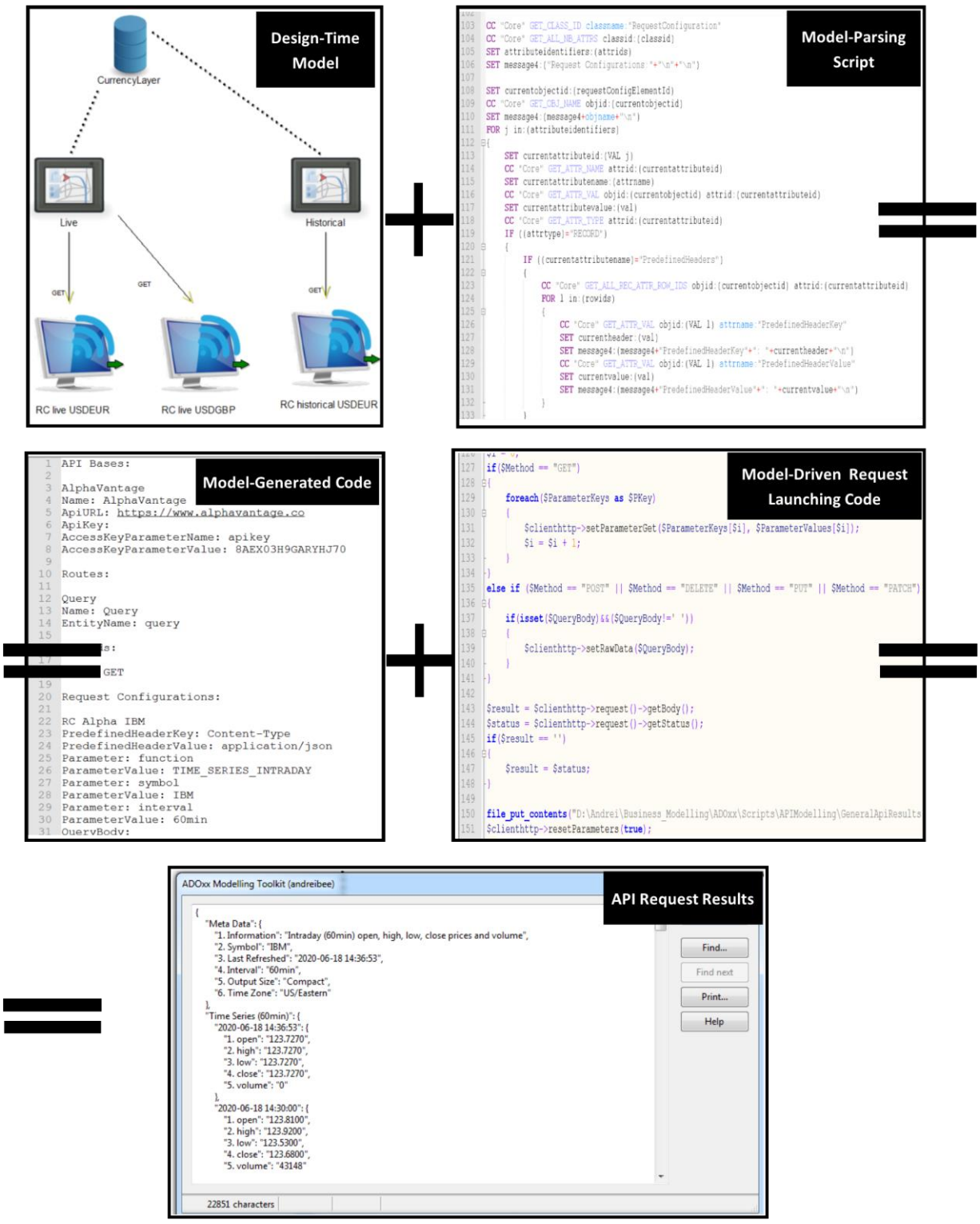

Figure 9. The model-driven API request execution cycle

Figure 10 depicts a code sample from the AdoScript diagram parser that is used for gathering data from the model, employing the model-driven functionality. It displays a part of the attribute parsing and data gathering for the APIBase class. Later, the values of the attributes will be stored in a file that will be consumed by the PHP script in order to launch the API requests, as the presented algorithms show.

```
CC "Core" GET_CLASS_ID classname:"APIBase"
CC "Core" GET_ALL_NB_ATTRS classid:(classid)
SET attributeidentifiers:(attrids)
SET message1:("API Bases:"+"\n"+"\n")

SET currentobjectid:(apiBaseElementId)
CC "Core" GET_OBJ_NAME objid:(apiBaseElementId)
SET message1:(message1+objname+"\n")
FOR j in:(attributeidentifiers)
{
    SET currentattributeid:(VAL j)
    CC "Core" GET_ATTR_NAME attrid:(currentattributeid)
    SET currentattributename:(attrname)
    CC "Core" GET_ATTR_VAL objid:(currentobjectid) attrid:(currentattributeid)
    SET currentattributevalue:(val)
    CC "Core" GET_ATTR_TYPE attrid:(currentattributeid)
    SET message1:(message1+currentattributename+": "+(currentattributevalue)+"\n")
    IF ((attrtype)="RECORD")
    {
        IF ((currentattributename)="ApiKey")
        {
            CC "Core" GET_ALL_REC_ATTR_ROW_IDS objid:(currentobjectid) attrid:(currentattributeid)
            FOR l in:(rowids)
            {
                CC "Core" GET_ATTR_VAL objid:(VAL l) attrname:"AccessKeyParameterName"
                SET accesskeyparamname:(val)
                SET message1:(message1+"AccessKeyParameterName"+": "+accesskeyparamname+"\n")
                CC "Core" GET_ATTR_VAL objid:(VAL l) attrname:"AccessKeyParameterValue"
                SET accesskeyparamvalue:(val)
                SET message1:(message1+"AccessKeyParameterValue"+": "+accesskeyparamvalue+"\n")
            }
        }

    }
}
SET message1:(message1+"\n")
```

Figure 10. Gathering the attributes of the APIBase class in AdoScript

5.5 Deployment Architecture

Figure 11 shows the deployment diagram of all involved components: The BPMN modeling tool communicates with the API modeling tool in order to integrate the business process modeling phase and the API request modeling phase. The AdoScript script parses the diagram obtained by using the API modeling method and then launches into execution the PHP script that communicates with the web API through the HTTP client.

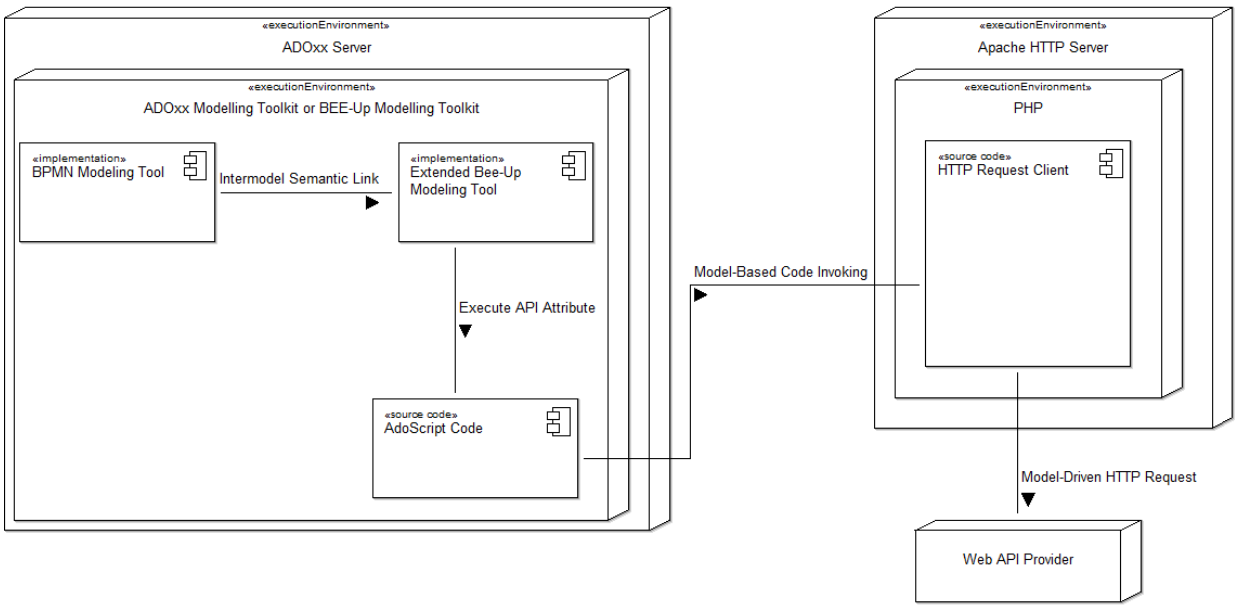

Figure 11. Deployment diagram

6 Artifact Evaluation

6.1 Quantitative Evaluation

We conducted a series of performance evaluations, testing prototype capabilities with multiple Web APIs, where the request payload takes various forms (JSON, SPARQL queries). We also wanted to see how an artifact that follows the model-driven software development pattern compares to an orchestrator that was built following the model-aware development model described in [28].

Table 1 displays the execution time for some of the API requests that were integrated with the API modeling method. Considering the selected requests, the average execution time was around 6.409 seconds.

Table 1. The execution times of different web API requests

API Request	Request Description	Execution Time (s)
Currencylayer USDGBP	Sending a JSON request to a currency exchange API, retrieving the USD-GBP rate	6.333
Currencylayer USDEUR	Sending a JSON request to a currency exchange API, retrieving the USD-EUR rate	6.396
Currencylayer historical USDEUR	Sending a JSON request to a currency exchange API, retrieving the USD-EUR rate from 01/01/2020	6.708
AlphaVantage AAPL	Sending a JSON request to a stock exchange API, getting the AAPL daily quotes	6.459
AlphaVantage IBM	Sending a JSON request to a stock exchange API, getting the IBM hourly quotes	6.723
RDF4J server request 1	Sending a SPARQL request to an RDF server, getting the structure of a company	6.209
RDF4J server request 2	Sending a SPARQL request to an RDF server, getting data about the employees of a company	6.037

Table 2 shows the comparison between the model-driven and the model-aware approach in terms of execution time. With the API modeling method, process modelers can choose which API request is sent to the execution engine at run-time. This functionality enables the model-driven approach to have a higher grade of usability as the modeling environment becomes a request control panel, even if it might seem slightly slower than the model-aware approach.

Table 2. Comparison between averages in the model-driven approach and the model-aware approach

Method	Execution Time
API modeling method (model-driven software development)	6.409
Semantic orchestrator (model-aware software development)	6.396

The system that was used for developing and testing both the API modeling tool and the semantic orchestrator is represented by a personal laptop that runs Windows 7 Professional 64-bit and has an available RAM memory of 8GB. The processor type of the system is Intel Core i7-47410HQ CPU 2.50Ghz.

6.2 Semantic Coverage

AQL Queries [37] already available in Bee-Up can now extend their scope over the metamodel extension, allowing to perform query-based analysis of the hybrid BPMN-API models. Below are two use case examples in this respect.

Example 1: Gather the linked RequestConfiguration of a certain BPMN task (the Gather exchange rates from API task from one of our BPMN models)

AQL code: `({"Get exchange rates from API"}-->"RequestConfigurationReference").`

Example 2: Get the name of the APIBase linked to a certain RequestConfiguration element (the RC live USDEUR RequestConfiguration element)

AQL code: `(({"RC live USDEUR"}<-"Method")<-"RouteAssociation").`

Figure 12 displays the results of the AQL queries that can support the analysis of BPMN-API hybrid model by retrieving some common information assuming a complex model.

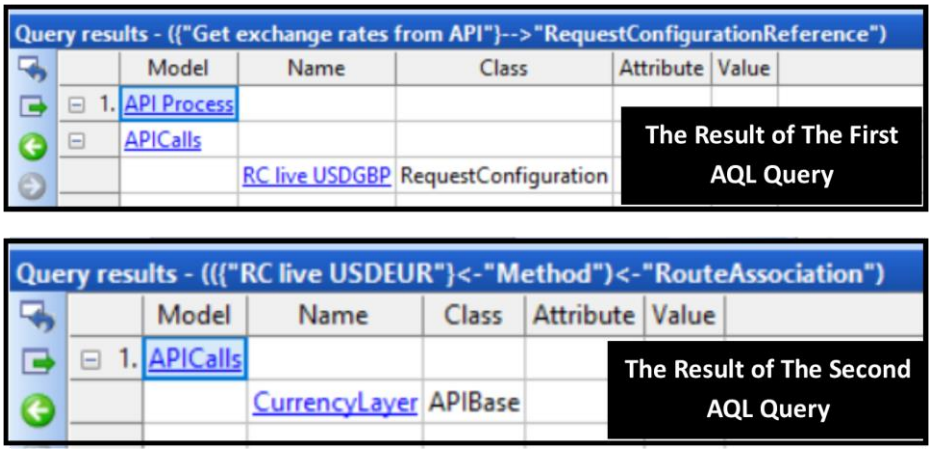

Figure 12. AQL query examples

6.3 Usability Evaluation

A usability evaluation hereby reports on the number of clicks that a modeler has to perform in order to link the tasks from process model to their API implementations and to launch the needed API call, as seen in Table 3.

Table 3. Modeling effort for linking APIs to processes and launching API requests

Operation	**Number of Clicks Necessary**
Linking task to API implementation	10
Launching the API request linked to a certain task (starting from the process model)	3

7 Conclusions

This article reports on a new modeling method for model-driven API management which enriches the BPMN standard by adding new semantic elements, as well as functionality on prototype level. The proposed artifact couples design-time process modeling to run-time execution of requests based on models. In today's software engineering world, model-driven approaches tend to get more traction as applications need to work in a frequently reconfigured ecosystem of APIs, and a model-driven treatment of this challenge is the idea advocated by this work.

The modeling method created this way can be used for integrating API requests in traditional business processes but it also provides a good solution for API testing, automation or API ecosystem management. This way, BPM agility can be improved by a more transparent relation between design-time and run-time.

In terms of limitations, the API diagrams developed with the presented modeling tool are limited to being linked to process diagrams made using the BPMN 2.0 standard in the ADOxx or

Bee-Up environments. An important step forward would be developing an API modeling tool that can be used also in other modeling environments such as Camunda or Bizagi [38].

Moreover, the work presented in this article would greatly benefit from being integrated with a more varied pool of modeling standards, not being strictly limited to BPMN 2.0.

Future work will also focus on expanding the proposed BPMN extension to cover multiple Web interoperability protocols – e.g., CoAP, MQTT, WebSocket, which, due to the status of standards of such protocols can enrich BPMN in ways that are highly reusable for software engineers.

References

[1] D. Karagiannis, R. A. Buchmann, P. Burzynski, U. Reimer, and M. Walch, "Fundamental conceptual modeling languages in OMiLAB," *Domain-Specific Conceptual Modeling*, Springer, pp. 3–31, 2016. Available: https://doi.org/10.1007/978-3-319-39417-6_1

[2] The Bee-Up Modeling Tool Download, http://austria.omilab.org/psm/content/bee-up/download?view=download. Accessed on October 22, 2020.

[3] R. C. Basole, "Accelerating Digital Transformation: Visual Insights from the API Ecosystem," *IT Professional*, vol. 18, no. 6, pp. 20–25, 2016. Available: https://doi.org/10.1109/MITP.2016.105

[4] The Camunda Modeling Tool. Available: https://camunda.com/. Accessed on October 28, 2020

[5] D. Karagiannis and H. Kühn, "Metamodelling Platforms," *Proceedings of the Third International Conference EC-Web 2002 – Dexa 2002*, Springer, pp. 182–182, 2002. Available: https://doi.org/10.1007/3-540-45705-4_19

[6] C. A. O'Reilly and M. L. Tushman, "The ambidextrous organization," *Harvard Business Review*, vol. 82, no. 4, pp. 74–81, 2004. Available: https://hbr.org/2004/04/the-ambidextrous-organization. Accessed on October 12, 2020.

[7] W. M. P. van der Aalst, M. Bichler, and A. Heinzl, "Robotic Process Automation," *Business & Information Systems Engineering*, vol. 60, pp. 269–272, 2018. Available: https://doi.org/10.1007/s12599-018-0542-4

[8] The RESTful API Modeling Language (RAML). Available: https://www.raml.org/. Accessed on October 22, 2020.

[9] The API Blueprint Description Language. Available: https://apiblueprint.org/. Accessed on October 21, 2020.

[10] The BPMN Standard. Available: https://www.omg.org/bpmn/. Accessed on October 23, 2020.

[11] The SwaggerHub API Development Tool. Available: https://swagger.io/tools/swaggerhub/, Accessed on October 28, 2020.

[12] The UiPath RPA Solution Provider. Available: https://www.uipath.com/. Accessed on October 28, 2020.

[13] The ADOxx Toolkit. Available: https://www.adoxx.org/live/home. Accessed on October 15, 2020.

[14] The AdoScript Programming Language. Available: https://www.adoxx.org/live/adoscript-language-constructs. Accessed on October 13, 2020.

[15] A. Goldstein, T. Johanndeiter, and U. Frank, "Business process runtime models: towards bridging the gap between design, enactment, and evaluation of business processes," *Information Systems and e-Business Management*, vol. 17, pp. 27–64, 2019. Available: https://doi.org/10.1007/s10257-018-0374-2

[16] B. Henderson-Sellers, C. Gonzalez-Perez, O. Eriksson, P. J. Ågerfalk, and G. Walkerden, "Software Modeling Languages: A Wish List," *2015 IEEE/ACM 7th International Workshop on Modeling in Software Engineering*, pp. 72–77, 2015. Available: https://doi.org/10.1109/MiSE.2015.20

[17] J. L. Cánovas Izquierdo, F. Jouault, J. Cabot, and J. G. Molina, "API2MoL: Automating the building of bridges between APIs and Model-Driven Engineering," *Information and Software Technology*, vol. 54, no. 3, pp. 257–273, 2012. Available: https://doi.org/10.1016/j.infsof.2011.09.006

[18] F. Zalila, S. Challita, and P. Merle, "Model-driven cloud resource management with OCCIware," *Future Generation Computer Systems*, vol. 99, pp 260–277, 2019. Available: https://doi.org/10.1016/j.future.2019.04.015

[19] C. Pautasso, A. Ivanchikj, and S. Schreier "Modeling RESTful Conversations with Extended BPMN Choreography Diagrams," *Software Architecture, ECSA 2015, Lecture Notes in Computer Science*, Springer, vol. 9278, pp. 87–94, 2015. Available: https://doi.org/10.1007/978-3-319-23727-5_7

[20] P. Valderas, V. Torres, and V. Pelechano, "A microservice composition approach based on the choreography of BPMN fragments," *Information and Software Technology*, vol. 127, 2020. Available: https://doi.org/10.1016/j.infsof.2020.106370

[21] A. Ivanchikj, C. Pautasso, and S. Schreier, “Visual modeling of RESTful conversations with RESTalk,” *Software and System Modeling*, vol. 17, pp. 1031–1051, 2018. Available: https://doi.org/10.1007/s10270-016-0532-2

[22] R. Dukaric and M. B. Juric, “BPMN extensions for automating cloud environments using a two-layer orchestration approach,” *Journal of Visual Languages & Computing*, vol. 47, pp. 31–43, 2018. Available: https://doi.org/10.1016/j.jvlc.2018.06.002

[23] S. Kelly, K. Lyytinen, M. Rossi, and J. P. Tolvanen, “MetaEdit+ at the Age of 20,” *Seminal Contributions to Information Systems Engineering*, Springer, pp. 131–137, 2013. Available: https://doi.org/10.1007/978-3-642-36926-1_10

[24] D. Bork, R. A. Buchmann, D. Karagiannis, M. Lee, and E. Miron, “An Open Platform for Modeling Method Conceptualization: The OMiLAB Digital Ecosystem,” *Communications of the Association for Information Systems*, vol. 44, pp. 673–697, 2019. Available: https://doi.org/10.17705/1CAIS.04432

[25] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*, Springer, 2014. Available: https://doi.org/10.1007/978-3-662-43839-8

[26] R. A. Buchmann and D. Karagiannis, “Agile Modelling Method Engineering: Lessons Learned in the ComVantage Research Project,” *Proceedings of PoEM 2015*, pp. 356–373, 2015. Available: https://doi.org/10.1007/978-3-319-25897-3_23

[27] K. Peffers, T. Tuunanen, M. Routenberger, and S. Chatterjee “A Design Science Research Methodology for Information Systems Research,” *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2008. Available: https://doi.org/10.2753/MIS0742-1222240302

[28] A. Chiș “Proof of Concept for a BPMN-Driven Semantic Orchestration of Web APIs,” *Proceedings of the 18th International Conference on Informatics in Economy (IE 2019)*, pp. 193–198, 2019. Available: https://doi.org/10.12948/ie2019.04.08

[29] R. A. Buchmann, M. Cinpoeru, A. Harkai, and D. Karagiannis, “Model-Aware Software Engineering – A Knowledge-based Approach to Model-Driven Software Engineering,” *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering*, vol. 1, pp. 233–240, 2018. Available: https://doi.org/10.5220/0006694102330240

[30] The Agile Manifesto, https://agilemanifesto.org/. Accessed on October 22, 2020.

[31] D. Bork, D. Karagiannis, and B. Pittl, “A Survey of Modeling Language Specification Techniques,” *Information Systems*, vol. 87, 2020. Available: https://doi.org/10.1016/j.is.2019.101425

[32] D. Cetinkaya, A. Verbraeck, and M. D. Seck, “MDD4MS: a model driven development framework for modeling and simulation,” *Proceedings of the 2011 Summer Computer Simulation Conference (SCSC’11)*, pp. 113–121, 2011.

[33] D. Moody, “The “Physics” of notations: towards a scientific basis for constructing visual notations in software engineering,” *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 756–779, 2009. Available: https://doi.org/10.1109/TSE.2009.67

[34] The GraphQL Query Language. Available: https://www.graphql.org/. Accessed on October 15, 2020.

[35] The SPARQL Query Language. Available: https://www.w3.org/TR/rdf-sparql-query/. Accessed on October 17, 2020.

[36] The Postman API Development Platform. Available: https://www.postman.com/. Accessed on October 30, 2020.

[37] The AQL Query Language. Available: https://www.adoxx.org/live/adoxx-query-language-aql. Accessed on October 1, 2020.

[38] The Bizagi Modelling Suite. Available: https://www.bizagi.com/?lang=en. Accessed on October 15, 2020.