AGILE MODEL EDITOR

AGILNÍ EDITOR MODELŮ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE
MICHAL ZAVADIL

SUPERVISOR
VEDOUCÍ PRÁCE
doc. Mgr. ADAM ROGALEWICZ, Ph.D.

BRNO 2022

Department of Intelligent Systems (DITS) Academic year 2021/2022

Bachelor's Thesis Specification

24956

Student:	**Zavadil Michal**
Programme:	Information Technology
Title:	**Agile Model Editor**
Category:	Software Engineering

Assignment:

1. Study graphical frameworks suitable for model editing.
2. Study general-purpose graphical languages used for model definition.
3. Design an editor allowing agile editing of models described in the chosen language, with the focus on complexity management.
4. Implement the editor.
5. Discuss the usability and limitations of the editor.

Recommended literature:

- Dori, Dov (2016). *Model-Based Systems Engineering with OPM and SysML*. New York: Springer-Verlag. (https://doi.org/10.1007%2F978-1-4939-3295-5)
- STPA Handbook (http://psas.scripts.mit.edu/home/get_file.php?name=STPA_handbook.pdf)
- Systems Modeling Language (SysML): https://sysml.org/
- Eclipse Sprotty, a diagramming framework for the web: https://github.com/eclipse/sprotty (https://typefox.io/sprotty-a-web-based-diagramming-framework)

Requirements for the first semester:

- First three items of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Rogalewicz Adam, doc. Mgr., Ph.D.**
Consultant:	Fiedor Jan, Ing., Ph.D., UITS FIT VUT
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	May 11, 2022
Approval date:	November 3, 2021

Abstract

The goal of this thesis is to simplify the creation and modification of complex system models and to create a reliable and performant tool to serve this purpose. Primarily, an internal data model is defined with an emphasis on effectiveness and avoiding redundancy. The editor, created to support modeling in OPM (Object-Process Methodology), is implemented with the help of web technologies and, most importantly, the web diagramming library Cytoscape.js. The editor can automatically propagate relationships to other diagrams as well as derive new relationships from existing ones. Both of which save time and make the modeling process easier.

Abstrakt

Cílem této práce je zjednodušit tvorbu a modifikaci komplexních modelů systémů a také vytvořit spolehlivý a výkonný nástroj, který je schopen splnit tento účel. Vnitřní datový model editoru je navržen s důrazem na efektivitu a vyhnutím se redundanci. Výsledný editor, který je vytvořen pro podporu modelování v OPM (Object-Process Methodology), je implementován s pomocí webových technologií, především s knihovnou pro tvorbu diagramů na webu, Cytoscape.js. Editor je schopný automaticky propagovat vazby do dalších diagramů a také odvozovat nové vazby od existujících, což šetří čas a ulehčuje proces modelování.

Keywords

model, modeling, modeling language, OPM, UML, SysML, model editor, diagram, system modeling, system architecture, master model, model-driven development, model-based system engineering, Cytoscape.js, web technologies

Klíčová slova

model, modelování, modelovací jazyk, OPM, UML, SysML, editor modelů, diagram, modelování systémů, architektura systému, master model, vývoj řízený modely, návrh systémů založený na modelech, Cytoscape.js, webové technologie

Reference

ZAVADIL, Michal. *Agile Model Editor.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Mgr. Adam Rogalewicz, Ph.D.

Rozšířený abstrakt

Návrh reálných i softwarových systémů se postupem času stává čím dál problematičtější díky jejich zvyšující se složitosti. Důkladné pochopení systému, jeho struktury, chování nebo požadavků na jeho funkcionalitu, je nezbytné pro jeho úspěšný vývoj. Pokud je toto pochopení nedostačující, může docházet k nákladným chybám, což plýtvá časem i penězi.

Jedním z osvědčených způsobů, jak bojovat se složitostí, je využívat modelů. Model je abstrakce definovaná grafickými diagramy, které popisují daný aspekt modelovaného systému. Například některé diagramy mohou popisovat strukturu systému, zatímco jiné popisují jeho chování. Dále často tyto diagramy neukazují celý systém, ale pouze jeho určité části.

Tento přístup při tvorbě systému je obecně přijímaný v tradičních inženýrských disciplínách, jako jsou letectví, stavebnictví nebo automobilový průmysl, a je především využíván v případech, kdy je kladen důraz zejména na spolehlivost a bezpečnost. Nicméně v kontextu softwarového inženýrství, které je v porovnání se zmíněnými mnohem mladším oborem, není modelování vždy zavedenou praktikou.

Model může být kreslený ručně, jako prostředek pro rychlé sdělení složité myšlenky. Obecně ale chceme od modelu to, aby byl kompletní a neobsahoval chyby. Tvorba takového modelu může být časově náročná, a proto je důležité používat spolehlivý editor modelů, který usnadňuje práci.

Tato práce má za cíl zejména vytvořit editor umožňující práci s komplexními modely. Nejdříve ale pojednává o využití modelů při vývoji softwaru, které se často nazývá *vývoj řízený modely* (angl. Model-driven development). Poté jsou představeny výhody tohoto přístupu a je zhodnocena aktuální míra použití v praxi. Aby byl model univerzálně pochopitelný a aby jeho elementy měli jednoznačný význam, musí model odpovídat pravidlům. Tato pravidla se nazývají modelovacími jazyky a určují význam elementů a říkají, jak mohou být elementy spojeny nebo kombinovány. Část práce je věnována modelovacím jazykům, jmenovitě UML (Unified Model Language), SysML (System Modeling Language) a OPM (Object-process methodology). UML a SysML – 2 nejrozšířenější modelovací jazyky – jsou popsány krátce, zatímco OPM – méně známý, ale rozšiřující se jazyk – je popsán do detailu. Výsledný editor, který je vytvořen pro modelovací jazyk OPM, umožňuje tvorbu a modifikaci komplexních modelů. Nejdůležitějším předpokladem pro tento požadavek je kvalitní a propracovaný návrh interního modelu dat, který je navržen se zaměřením na efektivitu a s vyvarováním se duplicitě dat. Dále model umožňuje jednoduché odvozování nových vazeb od již existujících.

Před implementací editoru je provedena důkladná analýza dostupných technologií, zejména knihoven pro tvorbu diagramů. Z tohoto průzkumu vzešlo rozhodnutí, že knihovna Cytoscape.js je pro účel editoru pro jazyk OPM nejvhodnější. Samotný editor je pak mimo tuto knihovnu vytvořen za pomocí webových technologií, programovacího jazyka JavaScript nebo knihovny pro tvorbu uživatelských rozhraní – React. Následuje popis implementace a ovládání editoru, přičemž některé důležité a zajímavé funkcionality jsou popsány podrobněji.

Nakonec je výsledný editor zhodnocen a porovnán s již existujícími nástroji. Kvůli časové náročnosti není editor úplný a neobsahuje některé méně podstatné funkce, které by byly důležité pro potenciální uživatele. Umožňuje ale efektivní práci s komplexními modely díky svému internímu modelu dat a narozdíl od existujících nástrojů umožňuje odvozovat nové vazby a snadnou propagaci již existujících vazeb. Obě tyto funkce zjednodušují proces modelování a šetří čas uživatele.

Agile Model Editor

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of doc. Mgr. Adam Rogalewicz, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Michal Zavadil
May 10, 2022

Acknowledgements

I would like to thank doc. Mgr. Adam Rogalewicz, Ph.D. for supervising my thesis. A big thanks deserves my consultant, Ing. Jan Fiedor, Ph.D. I am very grateful for a lot of his time, much advice and many useful ideas to improve the text as well as the editor.

Contents

Chapter 1

Introduction

When designing a system or gaining deeper knowledge about an existing one, we could encounter a great amount of complexity. As software and real systems become more and more complicated, the human mind does not adapt well. It takes time to understand complicated structures, all their parts, and how they interact. And if comprehension is poor, mistakes are being made, which wastes valuable time and money.

One way to fight the complexity is with the use of models. Models are abstractions; diagrams that demonstrate one of the system's aspects, for instance, structure or behavior. They may show a certain part of the modeled system, describing only a limited subset of related components, which facilitates understanding.

This approach to designing systems has been prevalent in traditional engineering fields, for example, automotive, aerospace, defense, and infrastructure. Especially in cases where reliability, security, and safety are of great importance. However, software engineering, as a relatively young discipline, has a long way to go in this regard.

A model should not be an arbitrarily drawn scribble. But instead, the way it is constructed should conform to a set of rules, and the elements from which the model is assembled must have a defined and unambiguous meaning. Because of this, several modeling languages have been established throughout the years, most notably: UML (Unified Modeling Language) and SysML (System Modeling Language). A new modeling language, introducing a distinct way of constructing models, has been emerging in recent years, OPM (Object-Process Methodology).

A model created in a modeling language can be drawn by hand as a way to quickly frame a thought or convey ideas to others. But many times we want a model to be complete and without flaws so that others may use it in their work. A proper diagram editor is needed for the effective creation of models. Not only must the tool be reliable, but should also save modelers' time, and be intelligent, assisting the modeler in overcoming complexity. As models are increasingly more elaborate, the performance of the editor is at the center of attention as well.

One of the outcomes of this thesis is a prototype of a diagram editor[1]. The modeling language for which the editor is implemented was chosen to be OPM. To develop a good editor, theoretical foundations must be understood first. Then, a suitable architecture can be designed for the editor to provide the required functionality.

In Chapter 2, MDD (Model-Driven Development), the model-centered software engineering practice, is examined. Its definition is presented as well as how widely it is spread

[1]Available at: https://opm-editor.netlify.app/

and the possible benefits it could bring us. This chapter also includes overviews of modeling languages that specify created models. UML and SysML are described in short, whereas OPM, the chosen language for the editor, is explained more thoroughly. In this chapter, the usefulness of models is emphasized, so right in the following Chapter 3, this principle is put into practice. Here, a data model, which is designed to satisfy the requirements of OPM, is proposed. In the next chapter, Chapter 4, a survey of available technologies is done, and the most suitable ones are chosen and justified. Additionally, this chapter presents a high-level architecture of the editor. In Chapter 5, the editor is finally created, and several interesting aspects of the implementation are discussed. Lastly, in Chapter 6, the implemented editor is assessed, compared with the competition, and its strengths and shortcomings are evaluated.

Chapter 2

Model-driven development

"Model-driven development (MDD) is simply the notion that we can construct a model of a system that we can then transform into the real thing"[11]. MDD is a software engineering practice that places the models at the focus rather than the resulting program and tries to use these models efficiently throughout the development process. MDD is also known as model-based development, model-driven engineering, or model-driven software development.

In this chapter, we will look at what MDD is, why and when it is used, and what benefits it brings us. Next, we quickly examine the most widely used modeling languages in software development: UML (Unified Modeling Language) and SysML (System Modeling Language). Following, is the OPM (Object-Process Methodology), which will be explored in greater detail as the editor is implemented for this language.

2.1 What is a model?

The word *model* could mean different things in different contexts, but generally speaking, a model is an abstraction of reality. It simplifies the reality to make it easier to understand and leaves out unnecessary details so that the final product could be easily created. A model is a collection of formal elements that represent a system and is built to serve a purpose, for example, communication of ideas, completeness checking, cost estimations, or generation of test cases [11].

In software development, we usually think of models as diagrams, which describe a modeled system, and are specified in a modeling language, e.g., UML or SysML. These diagrams show the system's various aspects, for example, its structure or behavior. An advantage of models is that they are independent of the implementation and are not bound to a specific programming language. They are easier to develop, understand, and maintain. An example of a model is shown in Figure 2.1.

2.2 Current situation

The use of models is widespread in traditional engineering fields such as the automotive and aerospace industry. In this context, the model-based approach is often called MBSE (Model-Based Systems Engineering). All parts of a product are carefully thought out before manufacturing the product itself, and models help visualize difficult problems and find their solutions. In software development, it is quite common to do no preparation at all before

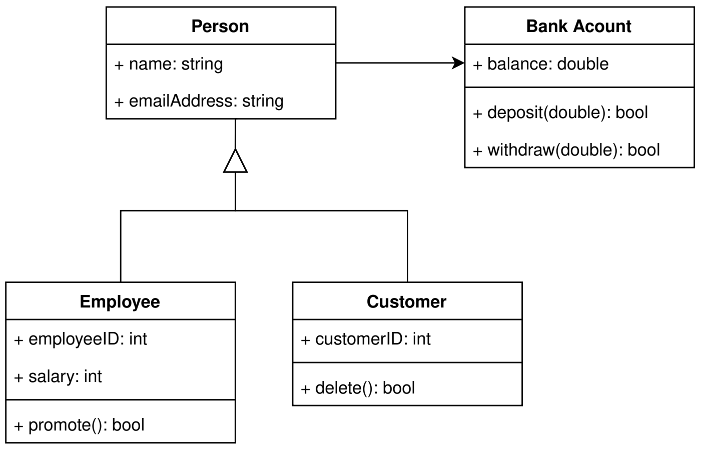


Figure 2.1: A simple example of a UML class diagram showing a part of a hypothetical banking system.

the actual implementation. One of the reasons is that the software industry is relatively new and industry-wide standards have not yet been established. This might not matter for small and short-lived projects. For bigger projects, however, making bad design decisions right at the start will certainly increase costs, waste time, or might even result in the failure of the project. Software systems are especially known for their complexity; therefore, the use of modeling techniques could be beneficial to the software development industry [15].

Table 2.1 presents the results of a survey [8] regarding the adoption of MBSE. The table shows in which aspects the respondents see the biggest advantages of the model-based approach. The conclusion is that most system engineers see the adoption as helpful, while the opinion of it being a hindrance in their work is quite rare.

2.3 Benefits of model-driven development

Using models and modeling techniques at least partly in the development process, or putting models at the center of attention and making full use of MDD's features, has its upsides. The following are the potential benefits [7]:

- **Enhanced communication:** Pictures say more than words. A diagram is easier to understand than a paragraph in a documentation. On top of that, diagrams conforming to a standardized modeling language have an expected and unambiguous meaning.

- **Reduced development risks:** A thought-out model reduces the chances of missed details and can be a subject of early validation and verification. Detecting issues before starting the implementation saves time and money later in the development process. A model also provides a more accurate foundation for determining estimates.

- **Improved quality:** In MDD, the requirements are also part of the model. The completeness of requirements and their traceability helps in achieving the best possible software quality.

	Big improv.	Some improv.	No change	Some impair.	Big impair.	Not applicable
Architecting and design	49%	25%	10%	0%	2%	15%
Requirement analysis	44%	29%	11%	0%	2%	15%
Architectural view	42%	26%	10%	2%	2%	20%
Traceability between system requirements and the realization	36%	27%	14%	2%	2%	19%
Technical reviews	23%	33%	19%	7%	0%	18%
Verification and validation	18%	38%	26%	3%	2%	13%
Technical risk assessment	13%	38%	25%	3%	2%	19%

Table 2.1: In what aspects and to what extent has the addition of MBSE helped. (Taken from [8])

- **Increased productivity** A good model promotes re-usability and saves time, especially during integration and testing phases. Automated code and documentation generation is sometimes also possible.

2.4 Modeling languages

A modeling language is a notation for expressing models. It consists of semantics, the meanings of elements, and syntax, the rules by which elements can be combined and connected. Two modeling languages have been dominating the scene and have become industry standards: UML and SysML.

The following sections take a closer look at these two modeling languages together with OPM, which is has been getting traction lately, and compare their differences and similarities.

Overview of UML

UML [5], introduced in 1997, was a fusion of many visual modeling languages that coexisted in the 1990s. It was a result of a need for a versatile and standardized modeling language. UML is designed for software systems and therefore supports key software concepts like OOP (Object-Oriented Programming).

Software engineers use UML in three distinct ways [5]:

- **Sketch**: Used informally, without paying attention to details, and as a quick way to communicate ideas or explain how parts of a system work. This is how UML is used most of the time.
- **Blueprint**: Used formally, with attention to detail and completeness. It is a result of system designer's efforts that can be turned directly into implementation by programmers.
- **Programming language**: Used to generate code directly from the model. It requires advanced tooling and usually generates only a general boilerplate code or a code skeleton.

Diagram types

The most recent version (UML 2.5) includes fourteen different types of diagrams. They can be divided into two main categories: structure and behavior. The following is their list along with short descriptions of the most important ones. This taxonomy is visualized in Figure 2.2.

- **Structure diagrams:**
  - **Class diagram:** Shows the relationships between classes as well as their attributes and methods. It is the most commonly used diagram type. An example is shown in figure 2.1.
  - **Object diagram:** Shows concrete examples (instances) of classes from a class diagram.
  - **Component diagram:** Shows relationships between components (higher level than classes).
  - **Deployment diagram:** Shows how the system is deployed with an emphasis on hardware resources.
  - Others: **Composite structure diagram**, **Package Diagram**, **Profile diagram**.
- **Behavior diagrams**
  - **Use Case Diagram:** Shows how the system is used by its users.
  - **State Machine Diagram:** Shows states and state changes caused by events.
  - **Activity Diagram:** Shows the order of actions that are executed as a reaction on an event.
  - **Sequence Diagram:** Shows the order and structure of messages sent between parts of a system.
  - Others: **Communication Diagram**, **Timing Diagram**, and **Interaction Overview Diagram**.

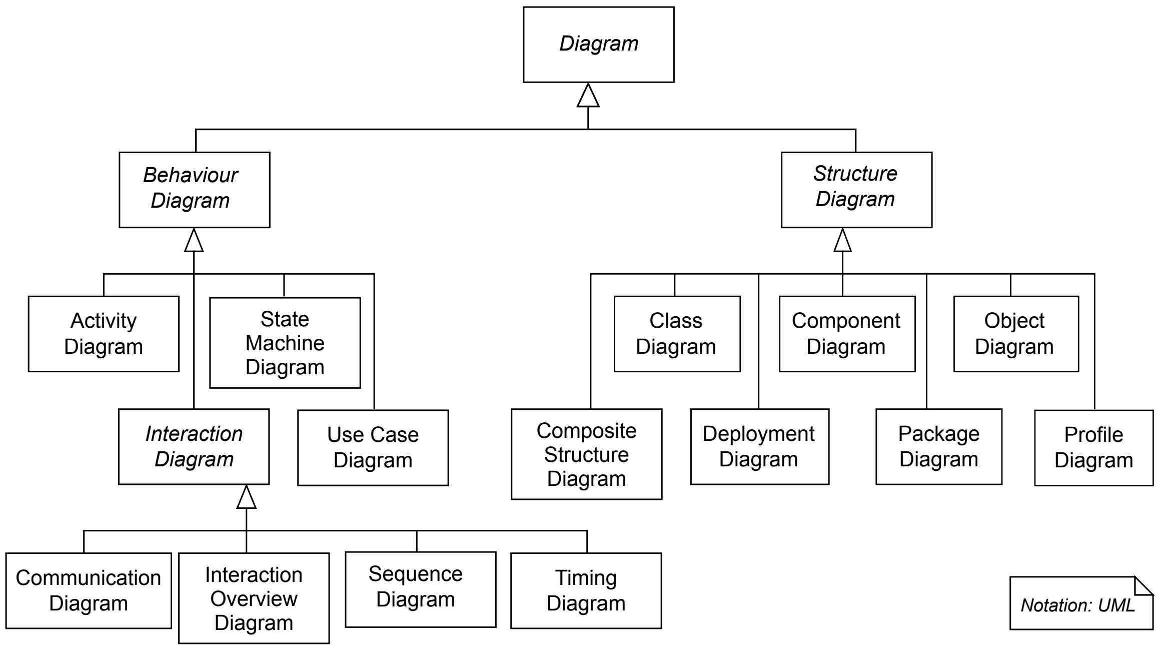


Figure 2.2: A taxonomy of UML diagram types showing the two main categories: behavior and structure. (Taken from [16])

Overview of SysML

SysML [7] is a general-purpose modeling language, which means that it is not only suitable for software systems, but also for systems of all kinds. It provides various types of diagrams, each focusing on a different aspect of the modeled system. SysML 1.0 was officially introduced in 2007 and is heavily inspired by UML.

SysML is a reaction to UML's code-centric nature. It takes seven of the original fourteen UML's diagrams and adds additional two. SysML is much more expressive, allowing to model a wide range of systems. The smaller set of diagrams results in the language being simpler and easier to understand.

Diagram types

SysML is a collection of a total of nine different types of diagrams, each describing the modeled system from a different point of view. They can be further divided into four categories called pillars (Structure, Behavior, Requirement, and Parametric). Diagrams from structure and behavior pillars are either the same as in UML or slightly modified to be less software-based, while the requirement and parametric diagrams are entirely new additions to SysML. Figure 2.3 presents the entire taxonomy of SysML diagrams along with differences with UML. The following are short descriptions of the diagrams [7].

- **Structure diagrams:**
  - **Block Definition diagram:** Shows relationships between system elements, i.e., their hierarchy, dependencies, and associations. Based on UML's Class diagram.
  - **Internal Block diagram:** Shows the internal structure of elements. Based on UML's Composite Structure diagram.

  - **Package diagram:** Shows parts of the model grouped into packages and presents a high-level view of the system.

- **Behavior diagrams**
  - **Activity diagram** Modified from UML with added support for modeling continuous systems and probabilities.
  - **Sequence diagram**, **State Machine** and **Use Case diagram** are identical to UML and are already described in 2.4.

- **Requirement diagrams:** Shows the system requirements and their relationships to the model elements.

- **Parametric diagrams:** Shows the constrains of the system. Could be in the form of a mathematical or physical equation.

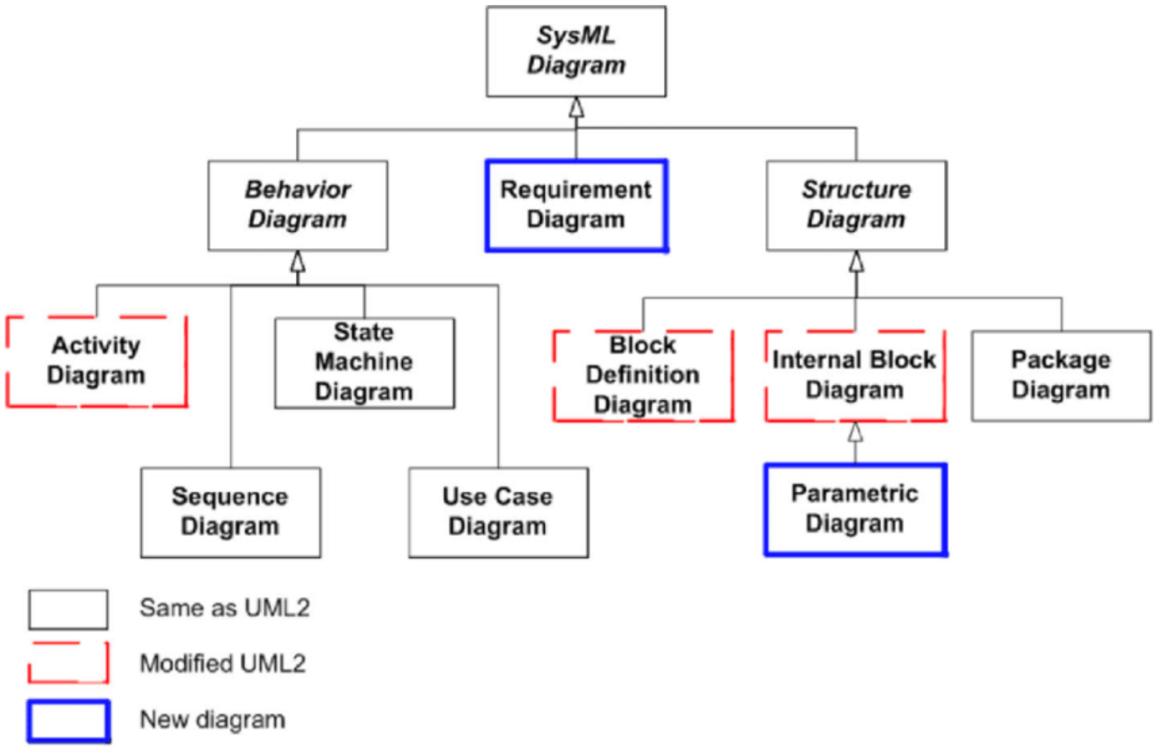


Figure 2.3: SysML diagram taxonomy. Diagram types devided into four main pillars: Behavior, Requirement, Structure and Parametric. Figure also depicts similarities and differences with UML2. (Taken from [10])

Overview of OPM

OPM [4] is a modeling language first introduced by Dov Dori in 1995 and then standardized by ISO in 2014 as ISO 19450[1] [14]. What makes OPM stand out from other modeling languages is its simplicity. It builds on the idea that the two main building blocks of

[1] https://www.iso.org/standard/62274.html

every system are objects and processes. An OPM model does not consist of multiple types of diagrams. Instead, OPM uses only one type of diagram to model all aspects of the modeled system. Another notable difference is that a proper OPM model consists of two parts: OPD (Object-Process Diagram), the graphical representation of a model, and OPL (Object-Process Language), the textual representation. The next following sections explain all the essential parts an OPM model is compiled of.

OPM Things

Elements in OPM are called things, and they can be either objects or processes. This is the smallest set of elements that a language must contain to model any system. This fact is sometimes called a minimal ontology. According to [4], a language with a smaller set of elements (that is, a smaller ontology) should be preferred to a language with a larger set of elements.

Objects, stateful or stateless, are the things that exist. They are parts of the system or things that interact with it from the outside. Processes alter objects. They create or consume objects or change objects' states.

Both objects and processes can be further characterized by two attributes: essence and affiliation. The essence value determines whether the object is physical or informatical. Physical objects in an OPD are denoted by a shadow. Affiliation can be systemic or environmental and describes whether an object is part of the system or its environment. Environmental things are denoted with a dashed border, and systemic things are denoted with a solid border. The visual representations of OPM things as well as different attribute representations of essence and affiliation are shown in Figure 2.4.

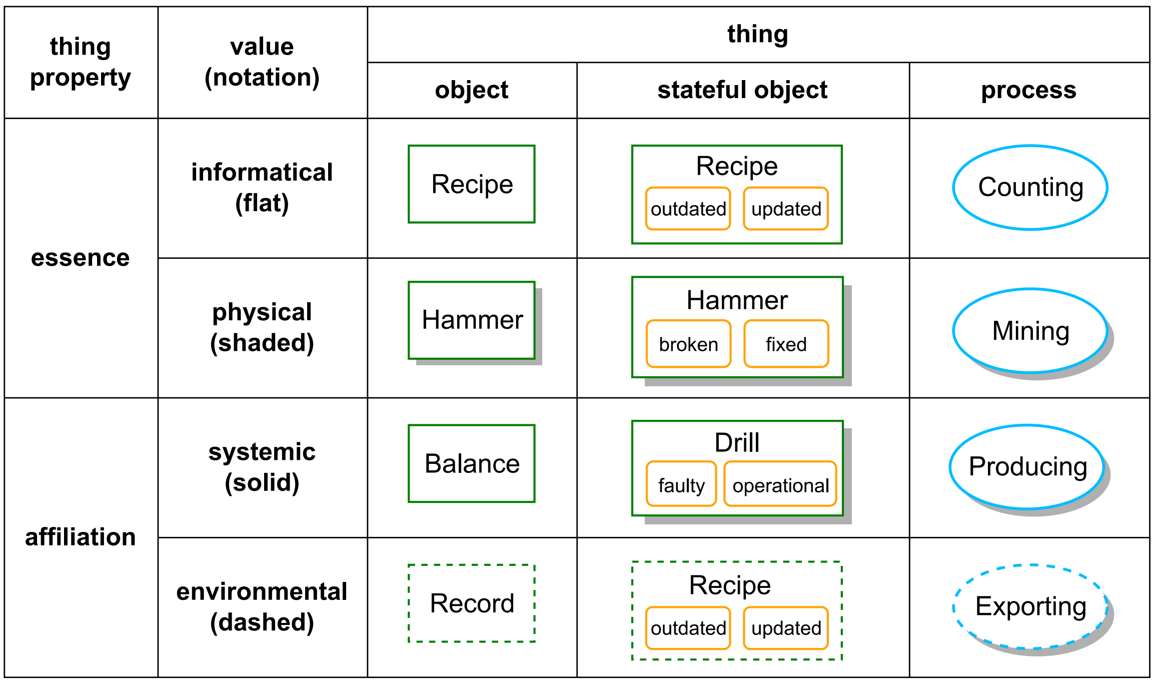


Figure 2.4: Visual representations of objects, stateful objects, and processes with different combination of attributes. (Redesigned from [4])

OPM Edges

Edges between things represent their relationship. OPM edges can be divided into two categories: structural and procedural. Structural edges express static relationships, while procedural edges express a dynamic aspect of a system. The following is a list of the most common OPM edges with brief descriptions. The visual representations of all edges are shown in Figure 2.5.

- **Fundamental structural edges:** A relationship between one superior thing and at least one inferior thing.
  - **Aggregation-participation:** Between a whole and its parts.
  - **Exhibition-characterization:** Between a thing and its features (attributes).
  - **Generalization-specialization:** Between a superclass and its subclasses.
  - **Classification-instantiation:** Between a class and its instances.
- **Tagged structural edges:** Edges with a label that explicitly states the nature of their relationship.
  - **Unidirectional tagged edge:** The relationship goes from a source to a target.
  - **Bidirectional tagged edge:** One edge that states two relationships.
  - **Reciprocal tagged edge:** The relationship goes both ways.
- **Procedural transforming edges:** Edges that express a dynamic aspect of a system.
  - **Consumption edge:** A source object is consumed by a target process.
  - **Result edge:** A target object is created by a source process.
  - **Effect edge:** A process has an effect on an object.
  - **In-out edge pair:** Two edges showing that a process changes the state of an object from the input state to the output state.
- **Procedural enabling edges:** Used between a process and an object which is required for that process to occur.
  - **Agent edge:** The required object is a human.
  - **Instrument edge:** The required object is nonhuman.
- **Control edges:** Procedural edges with added semantics that are denoted by a control modifier at its edge arrow.
  - **Event edge:** The source element is also a trigger for the target process.
  - **Condition edge:** The target process is only executed if the source condition element is present or at the desired state. Otherwise, the process is skipped.

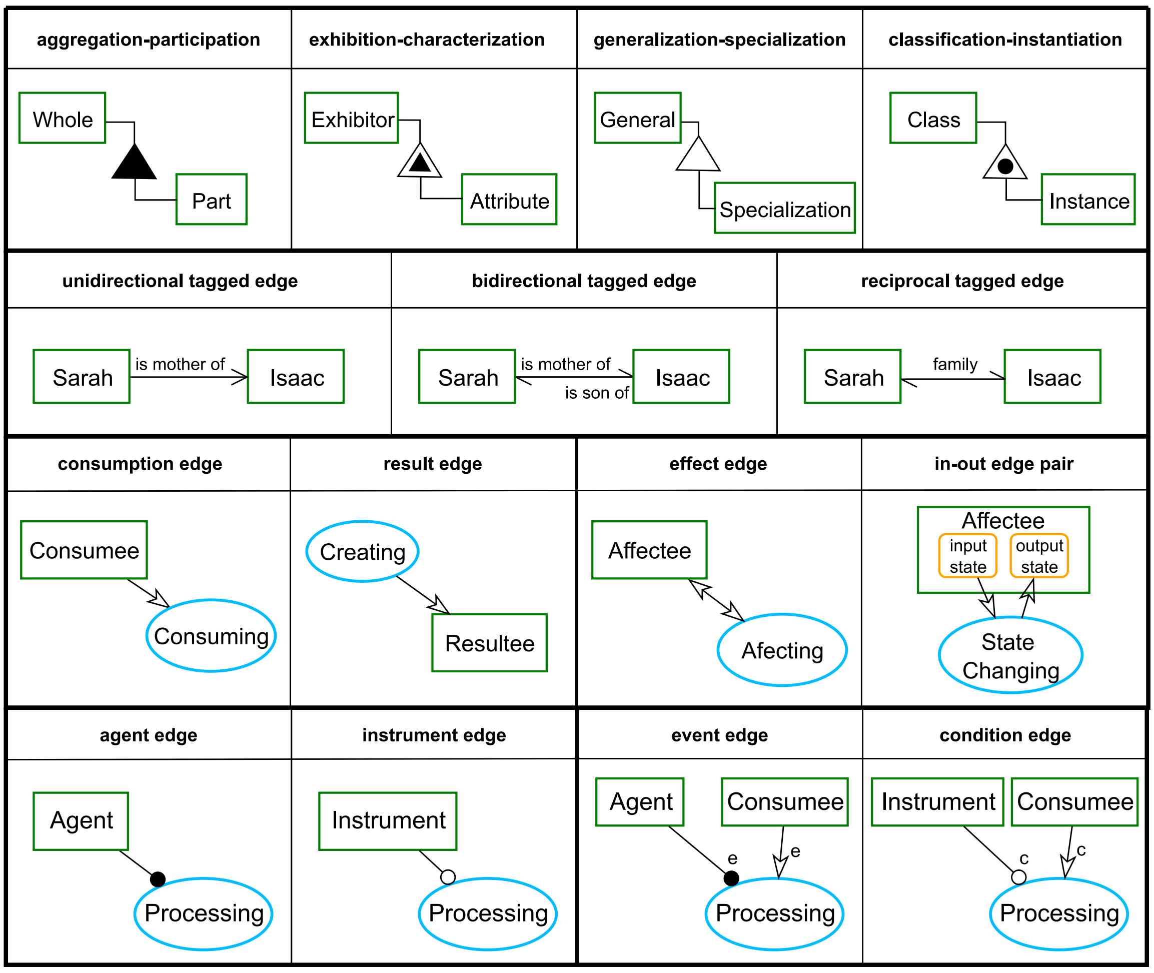


Figure 2.5: Visual representation of the most common OPM edges. (Redesigned from [4])

Object-Process Language

In addition to the visual diagram (OPD), another form of representation is provided by the OPM diagramming tools, OPL (Object-Process Language). OPL is a textual representation of a model and is expressed as a series of simple English sentences. The visual and textual representations are semantically equal, so that one can be deduced from the other. This duality makes understanding the model easier for people who are not so familiar with the methodology. It also brings another form of verification, as it is possible to compare the generated OPL sentences with the diagram and verify that they match our intent.

An example is shown in Figure 2.6. In this figure, there is a simple diagram that contains an object, a process, and a consumption edge between them. At the bottom, there are three OPL sentences describing the aforementioned diagram. The first two sentences describe the existence of two things, and the third sentence describes the edge between them. Additionally, the things mentioned in the sentences are color-coded to distinguish objects and processes.

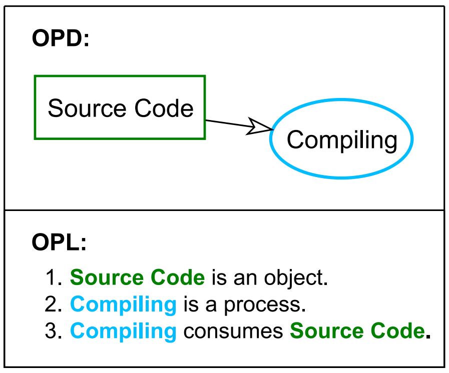

OPD:
Source Code
Compiling
OPL:
1. Source Code is an object.
2. Compiling is a process.
3. Compiling consumes Source Code.

Figure 2.6: The relationship between an OPD (Object-Process Diagram) and an OPL (Object-Process Language)

Complexity management of OPM

Rather than utilizing many types of diagrams, OPM only uses one type to model all aspects of a system. To deal with complexity, models are hierarchical, consisting of many diagrams arranged in a tree-like structure (Diagram Tree). Each diagram describes a different part of a system at a different level of detail. Diagrams at the top of the hierarchy are the most abstract ones, presenting a high-level view of a system, while diagrams at the bottom are the most detailed and concrete ones. A modeler should not clutter top-level views with unnecessary details. But instead, refine the parts of the system in a new descendant diagram. To facilitate this, OPM provides multiple ways of decomposition and refinement.

The most common method is called *in-zooming*, which is used when we need to model parts of an element. An in-zoomed element is enlarged, and additional elements can be added into the in-zoomed element's body. This action is much more common with processes than with objects and is an idiomatic way to model processes' details. There are two kinds of of in-zooming: *in diagram in-zooming* and *new diagram in-zooming*. In diagram in-zooming is to be used when we want to show process's refinement in the current diagram. However, this is not as widely used as it adds extra elements to a possibly already overcrowded diagram. Instead, new diagram in-zooming is much more common, as it creates a new descendant diagram and creates a copy of the in-zoomed proess. After that action, the process exists in two diagrams at once. In the parent diagram, the process is shown as an abstraction and without its parts, while in the descendant diagram, the process could be described in great detail, showing its parts as well as outside elements the process interacts with.

An example of new diagram in-zooming is presented in Figure 2.7. In-zooming of **Process 1** results in the creation of a new diagram **SD1**, where **Process 1** is duplicated and placeholder subprocesses are added. The diagram **SD1** is added as a descendant of **SD**, which is depicted in the **Diagram tree**. **Object 1** and the connected edge are also added to the new diagram because they are linked to the in-zoomed process. This behavior is present in the existing OPM diagramming tools.

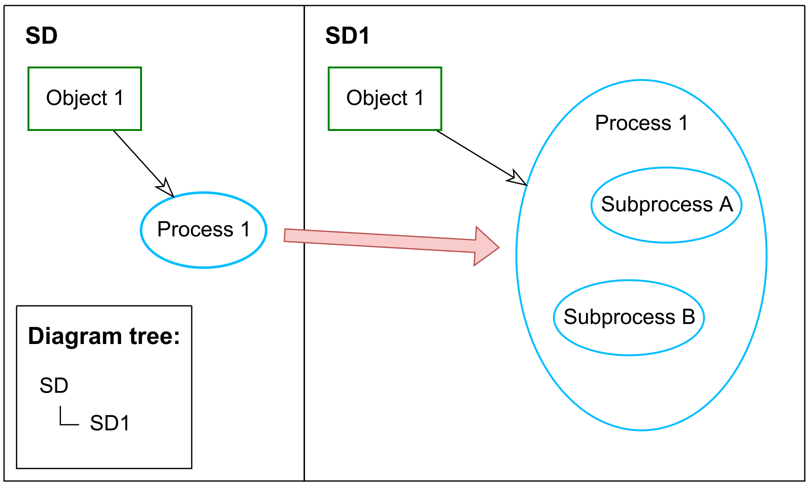


Figure 2.7: An example of in-zooming.

Chapter 3

Proposed data model

As systems get more complex, outdated model editors cease to be effective, and modeling with them becomes slow and inconvenient. Therefore, new tools that are more suitable for large models need to be created. The editor's performance will be highly influenced by its data model, so in this chapter, the internal data model is designed. And just like in the fashion of Model-Driven Development (MDD), described in Chapter 2, this approach could save us time later on in the development process.

The requirement for the data model is that it should allow for automatic displaying of elements and relationships where appropriate. For example, if we choose to display multiple elements from different diagrams, the relationships between them could be easily read from the data model and shown. To do these actions automatically means that a modeler would not have to do so explicitly, which saves time. Another important feature that the data model should enable is the derivation of new relationships from existing ones. For instance, a relationship between two unrelated child elements can be applicable to their parents and can be derived.

To facilitate this, there should be one data structure that contains all elements and relationships between them. In doing so, even if the system model is composed of multiple diagrams and elements may appear in multiple of them, all elements are stored at one place, and no redundant copies are created. This structure is often called *master model* and its usage also prevents inconsistencies from being made. A change of an element's attribute (for instance, name) would only have to be made in the master model, and then this change would spread to all diagrams through references.

The master model can be divided into two parts: *element model* and *edge model*. There is one additional auxiliary data structure that is needed for the implementation of the editor, *diagram tree model*. The hierarchy of these data structures is shown in Figure 3.1.

The diagram tree model contains individual diagrams, which are the views of the system showing only a limited subset of the master model. Diagrams are dependent on the master model, as they reference its elements, whereas the master model is an entirely independent entity. This approach to structuring data is common in the design of current diagramming editors.

The following are the aforementioned data structures described in more detail.

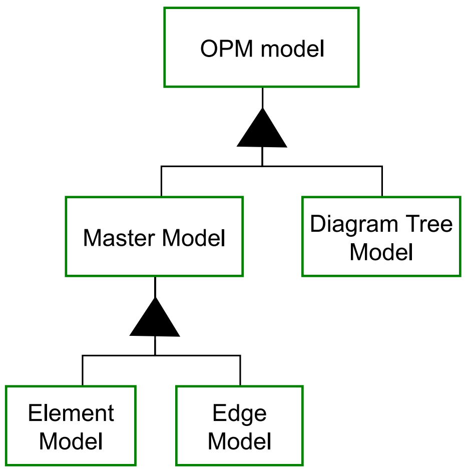


Figure 3.1: Hierarchy of data structures that represent an OPM model.

3.1 Element model

The element model is a tree[1] that contains all elements (OPM things) present in the model. The root node is the only exception; it does not represent any model element and serves purpose only as a reference to the element model. An empty model contains only the root node, and adding elements to a root diagram (SD) adds them as children of the root. Transformation from a leaf node to a parent node is facilitated by zooming in, as described in Section 2.4. Adding elements to a diagram of an in-zoomed element adds them as children to that particular in-zoomed element.

An example explaining the connection between the actual diagrams and the element model is shown in Figure 3.2. The top two boxes present two OPDs (Object Process Diagrams), the left one being the root diagrams **SD** and the right one being its descendant **SD1**. This relationship is depicted in the bottom-left box that contains the diagram tree. The hierarchy of elements, defined in the top diagrams, is reflected in the element model on the bottom right. The current state could be achieved from an empty model in these steps:

1. Adding **Object 1** and **Process 1** into the root diagram (**SD**). The same elements are added as children of root in element model
2. In-zooming of **Process 1**. This action creates a new child diagram **SD1**, with **Process 1** as its main element. The diagram tree is updated accordingly as well.
3. Adding **Subprocess A** and **Subprocess B** and **Object 2**. Note that existing OPM editors add a couple of subprocesses as placeholders by default.

3.2 Edge model

A proper OPM model contains not only elements (OPM things) but also relationships (edges) between them. Just like elements, edges in OPM commonly appear in multiple diagrams, and therefore storing them in one place facilitates consistency. For this purpose,

[1] https://en.wikipedia.org/wiki/Tree_(data_structure)

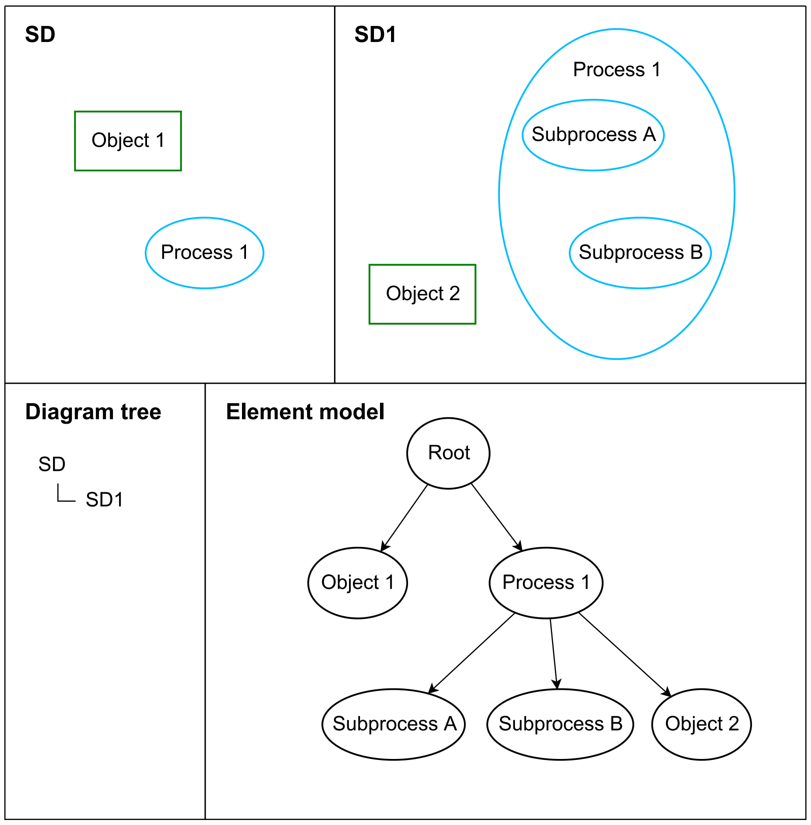


Figure 3.2: A relationship between two OPM diagrams and the element model.

a simple collection (array, list) could be used and its fields would hold information about every individual edge, for example, name, source element, target element, and edge type (see 2.4).

This collection is not sorted by default, but additional index structures could be easily added to enhance the performance of frequent queries. For example, an index with edges arranged by their source and target elements could speed up the resolution of finding all edges that are connected to a given element.

Figure 3.3 shows two OPDs at the top, similar to those in Figure 3.2, and the state of the edge model collections at the bottom. In **SD**, there are two edges, while in **SD1**, there is only one edge. **Edge x** is present in both OPDs, and these two occurrences of the same edge hold the same reference to the edge model. Therefore, the original edge collection contains two edges in total, and the collection of derived edges, which will be discussed subsequently, is empty.

When an edge is created and one or both endpoints are parts of another element (e.g., they are subprocesses of another process), this new relationship is also transitively applicable to the parent elements. In a diagram with a higher level of abstraction, for instance, only these parent elements might be present, and a modeler should be able to see all relationships, even if they only relate to their descendant elements. Because of this transitivity of relationships, it is convenient to store the original edges and their derived edges in separate collections. The collection of derived edges works as a permanent cache, and edges should be added or removed upon the change in state of their original, i.e., edge creation, reconnection, and deletion. Note that due to the hierarchical structure of the element model

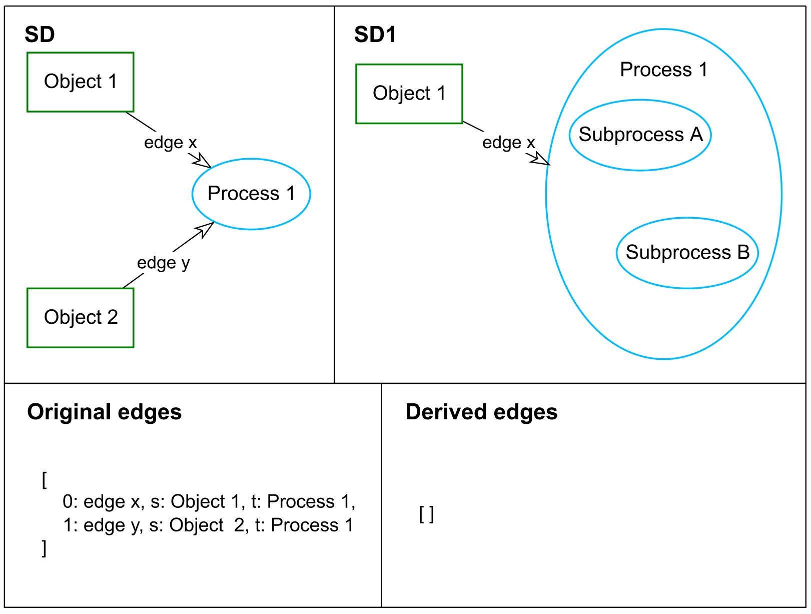


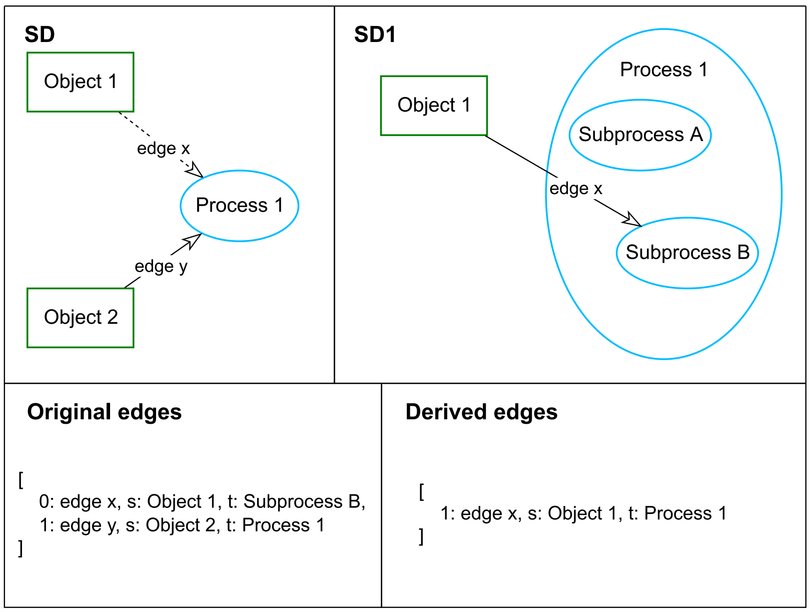
Figure 3.3: A relationship between two OPM diagrams and the edge model.

discussed above, an element could have multiple transitive parents. Therefore, deriving edges could potentially be executed for all nodes leading all the way up to the root element. The extent to which the edges are derived is up to the implementation of the diagramming tool and should ideally reflect the average user's needs. Optionally, the levels to which the edges are derived could be selected by the user himself.

It is important to be able to couple these derived edges with their originals, as derived edges should not have data of their own and should reference their original counterparts. In this way, the change of attributes to either the original or the derivative would only change the original data in one place and then would be propagated through references across the entire model.

The principle of derived edges is explained in Figure 3.4, which is slightly modified from Figure 3.3. The important difference is that **edge x** moved its target from **Process 1** to its descendant, **Subprocess B**. Because the target element has a parent element, a derived edge must be created, which points to the parent element. Therefore, the new derived edge is added to the collection of derived edges and visualized in the **SD** diagram.. The derivative is dashed.

A situation can arise in which multiple edges need to be abstracted and merged into one edge. This one edge is effectively a derivative of all the merged originals. An example of this phenomenon is presented in Figure 3.5. When the original edges are of the same type and have no tags, the situation is quite simple, the edges are abstracted into an edge of their type. If they are tagged, a good solution might be to aggregate the tags at the abstraction. If the edge types differ, one of them should be selected based on their predetermined priority, or they can be combined. This is only possible with a consumption and a result edge that combine into an effect edge; see Figure 3.5.



```
[
  0: edge x, s: Object 1, t: Subprocess B,
  1: edge y, s: Object 2, t: Process 1
]
```

```
[
  1: edge x, s: Object 1, t: Process 1
]
```

Figure 3.4: Relationship between two OPM diagrams and edge model with an emphasis on derived edges.

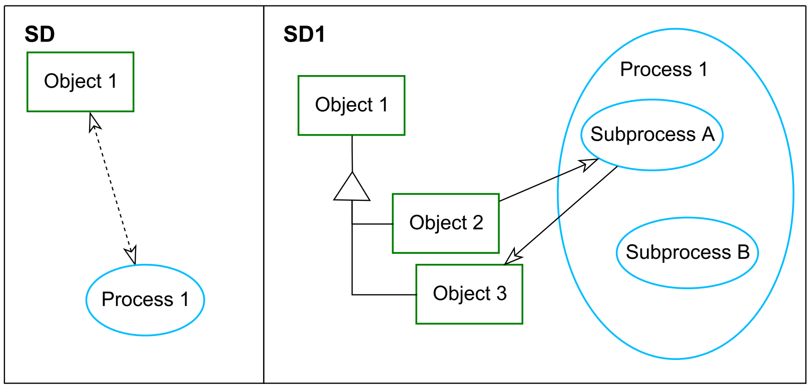


Figure 3.5: An example of abstracted merged edges.

3.3 Diagram tree model

One last auxiliary data structure is needed to finish the design of the editor's model, the diagram tree. It is a tree structure very similar to the element model explained previously, and its main function is to store data specific to each diagram. Each node holds data, such as a list of elements present, their position, or their size. This information could not be part of the element model or edge model, as multiple occurrences of elements or edges will have different positions, for example. Note that this aspect is closely related to a functionality of a diagramming library, discussed further in the following chapter.

Figure 3.6 shows two representations of the same diagram tree model. The right one is more of a traditional way of depicting trees, while the left one is being used in OPM diagramming tools, as it is more compact and better suited for graphical user interface.

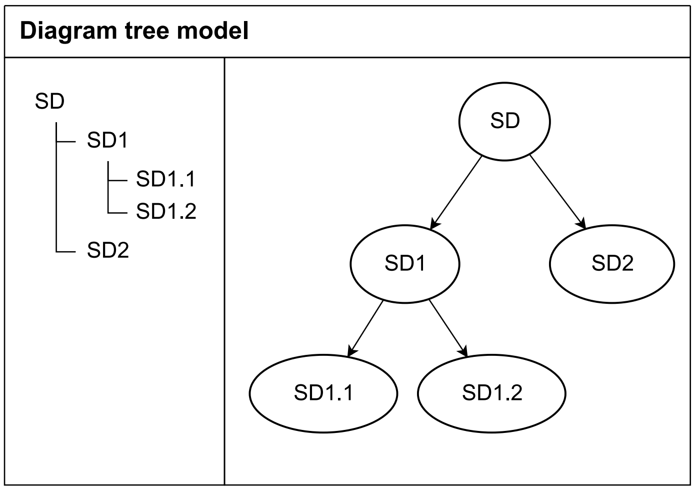


Figure 3.6: An example diagram tree model showed in two different representations.

Chapter 4

Editor design

Now that the editor's data model has been presented, technologies, frameworks, and libraries can be chosen to facilitate implementation. In this chapter, important architectural design decisions are made and justified.

The first important decision is about the platform on which the editor should be implemented. Creating the editor as a web-based application seems to be the best approach nowadays and has the following benefits over traditional software applications [12]:

- It is compatible across all platforms, as the only tool required is the web browser.
- It does not need to be installed and, therefore, does not take up any space on the hard drive.
- The application is always up-to-date and there is no need to worry about different versions.
- It is accessible from anywhere.
- It demands lower technical requirements from users.
- The need for maintenance and support from the developer is reduced.

Moreover, a prototype of the editor, which focuses mostly on the diagramming capabilities, can be client-only. Advanced functionalities (e.g., user authentication and cloud diagram storage) would require a back-end server, but that is not the main focus of this thesis.

For front-end web development, the most common choice of a programming language is JavaScript, which is widely used and offers a wide variety of open source libraries. Additionally, its syntactical superset, TypeScript, is more suited for larger projects and provides many solutions to the shortcomings of JavaScript. It introduces static typing, which yields the main following benefits:

- Catching of common and runtime errors during the compilation stage
- Better code readability
- IDE support and Intellisense

There are many JavaScript UI (User Interface) frameworks available; among the most popular are: React, Angular, and Vue.js. The purpose of the UI framework is to create a layout of the application and define elements for diagram modification, such as buttons, menus, and modals. As all the frameworks satisfy the requirements, the choice is arbitrary and, therefore, React was chosen because of my personal preference and prior experience with it.

In React, the user interface is divided into components. A programmer can make his own components, or a third-party library can be utilized to import already defined components that just need to be customized. This approach also ensures a unified design, as global themes can be easily specified. The Ant Design[1] library was chosen, due to a wide selection of components, especially the tree component that is well equipped to implement the visuals of the diagram tree (see Figure 3.6).

4.1 Web diagramming libraries

By far the most important design decision is the selection of a web diagramming library. This choice greatly influences the final result, as a great library could make key features possible and save a lot of time during implementation. Therefore, before the actual implementation, a review of existing diagramming libraries was done and the most fitting one was chosen. Libraries were evaluated on the basis of several factors, for example, the presence of key functionalities, quality of documentation, provided demo examples, or community size. Furthermore, simple prototypes were made to verify the presence of key functionalities for some of these libraries.

The following is the list of rejected candidates, as well as their quick assessment and reasons why they were not selected. The chosen library, Cytoscape.js, is then described in more detail in the following section.

JointJS[2] is a free open-source version of a commercial library Rappid, which is specifically made to build diagramming tools. Moreover, the newest and feature-rich OPM tool, OPCloud, is written with the help of this commercial library. However, the free version is very limited in its functionality. During prototyping, it proved to be especially difficult, for example, to add new diagram elements on user action. This feature is possible in the paid variant and, on top of that, it offers a broad variety of customizable toolbars. The documentation is quite extensive; with many demo examples. Nevertheless, due to the lack of key features, the JointJS library is not usable for our purposes.

GoJS[3] is also a commercial library, with the possibility of a free trial. The reason why it is included here is that it is very extensive and is suitable for the purpose of creating a diagramming tool. On top of that, the documentation is detailed and comprehensive with a huge number of examples. However, because the library is not open-source, it is not possible to use it.

jsPlumb[4], in contrast to the previous libraries, is quite simple. It facilitates draggable HTML elements and edges between them. The functionality of the library is rather lim-

[1] `https://ant-design.gitee.io/`
[2] `https://www.jointjs.com/`
[3] `https://gojs.net/latest/index.html`
[4] `https://jsplumbtoolkit.com/community`

	JointJS	**GoJS**	**jsPlumb**	**mxGraph**	**Cytoscape.js**
made for diagrams	yes	yes	usable	yes	usable
functionality	limited	extensive	limited	extensive	extensive
documentation	great	great	sufficient	sufficient	great
active	yes	yes	yes	no	yes
open-source	yes	no	yes	yes	yes
github stars	3.6k	6.1k	6.9k	6.3k	8.3k

Table 4.1: Comparison of all the tried-out diagramming libraries.

ited, and the absence of advanced features would lead to slow development and unnecessary obstacles.

mxGraph[5] is an extensive and time-tested library. It is used in many projects, most notably the draw.io[6] online diagram editor. The only downside, and an important one, is that the library is no longer maintained. This fact could cause serious issues with compatibility in the future, and it is advised not to start new projects with this library.

A comparison of the aforementioned libraries as well as the chosen library, Cytoscape.js, is shown in Table 4.1. An exhaustive list of graphical JavaScript libraries is provided here [13]. In addition, several other libraries are thoroughly compared in this thesis [9]. The thesis also focuses on the implementation of a diagramming editor and comes to the same conclusion that Cytoscape.js is the best option for our purpose.

4.2 Cytoscape.js

Cytoscape.js[7] [6] is an open-source library released under the MIT license. At this time, it is still actively maintained by its substantial community and has more than 8000 stars on its github repository[8]. Its documentation is quite thorough and detailed and includes interactive code examples.

Cytoscape.js uses the HTML canvas[9] element and is a highly optimized graph library for large and complicated networks. The library is mostly used as a graph visualization tool that draws elements, and a user is able to interact with these elements. In addition, it contains many useful functions for graph theory and graph analysis, as well as functions for filtering elements, traversing the model, or creating animations. The library can even be run headlessly, that is, without the visualization and only in a terminal. It is not specifically made for building diagram editors, but its vast functionality makes it fitting even for this use case. The library is built with extensibility in mind, as it provides an API (Application Programming Interface) for attaching extensions. Therefore, it is possible to create an extension or use one of many developed by the community.

In the following sections, some concrete features of the library are described and explained how to utilize them in the implementation of the final editor.

[5]https://jgraph.github.io/mxgraph/
[6]https://app.diagrams.net/
[7]https://js.cytoscape.org/
[8]https://github.com/cytoscape/cytoscape.js
[9]https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

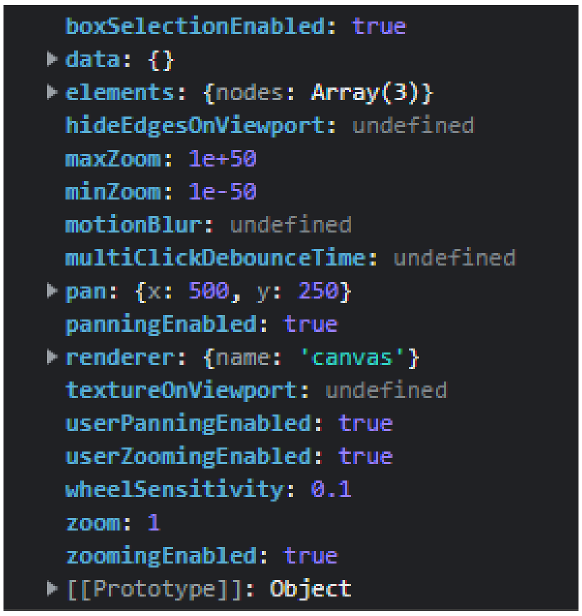
Cytoscape.js data model

As discussed in Section 2.4, an OPM model consists of multiple diagrams. During modeling, switching between diagrams is an essential part of the process, and an OPM diagram editor needs to provide ways to accommodate that. Cytoscape has a simple API for exporting a currently displayed diagram and importing a saved one into the canvas container. This feature makes diagram switching and their serialization easy to implement. The following JavaScript code snippet demonstrates diagram switching.

```
let oldDiagram = cy.json();
cy.json(newDiagram);
```

In the code snippet, `cy` is the cytoscape instance. It is registered to a particular DOM element in which the diagrams should be rendered. Calling its method `json`, with an empty argument list, returns data of the currently displayed diagram. Note that these data are not a string in the JSON format as the name of the method suggests, but a regular JavaScript object. This object can then be serialized and saved afterwards. On the second line, calling the same function with a diagram data object re-renders an old diagram with the newly provided one.

Figure 4.1 shows an example of what the diagram data object could look like. The most important field is `elements`, which contains elements (in Cytoscape.js these are both nodes and edges) present in the diagram. Each element could hold generic data (e.g., position and size) or custom data specified by the user, e.g., label. Other fields could determine the position and behavior of the viewport, for example, `pan` and `zoom`.



```
  boxSelectionEnabled: true
▸ data: {}
▸ elements: {nodes: Array(3)}
  hideEdgesOnViewport: undefined
  maxZoom: 1e+50
  minZoom: 1e-50
  motionBlur: undefined
  multiClickDebounceTime: undefined
▸ pan: {x: 500, y: 250}
  panningEnabled: true
▸ renderer: {name: 'canvas'}
  textureOnViewport: undefined
  userPanningEnabled: true
  userZoomingEnabled: true
  wheelSensitivity: 0.1
  zoom: 1
  zoomingEnabled: true
▸ [[Prototype]]: Object
```

Figure 4.1: Example of diagram data in JSON format exported from cytoscape.js

Compound nodes

Compound nodes are one of the key features for which Cytoscape.js was selected. It allows the creation of elements that are nested and hierarchical and is a fitting way of implementing in-zoomed nodes (see 2.4). Most importantly, the size of a parent node is adjusted according to the position of its children to always include them. Figure 4.2 shows two examples of compound nodes.

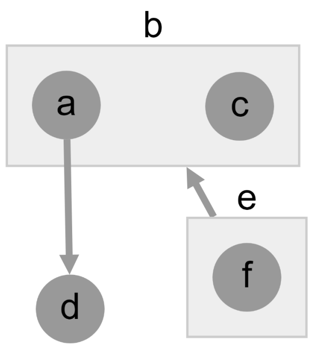


Figure 4.2: Examples of Cytoscape.js compound nodes. (Taken from the Cytoscape.js documentation [1])

4.3 Architecture

Now that the key components of the application are known and their functionality has been explained, it is possible to propose a high-level architecture. The architecture is based on the principles of MVC (Model-view-controller).

Figure 4.3 describes the design of the top-level architecture. The **Controlling** process handles the actions made in **Cytoscape Canvas** and functions of **UI Toolbar**. It takes care of modifying the state of the model and displaying the diagram in **Cytoscape Canvas**. Conceptually, there are two models: **Cytoscape Model** (a model of a currently displayed diagram, which is handled by Cytoscape.js) and **Master Model** (described in Chapter 3). Both the actions of **Cytoscape Canvas** and **UI Toolbars** cause changes in either or both models. The logic of the controller decides which model should be affected. For example, changing the position of an element only affects **Cytoscape Model**, while adding a new element concerns **Cytoscape Model** as well as **Master Model**. Additionally, the figure shows that the **Cytoscape Model**, the currently displayed diagram, is one of many diagrams present in the **Diagram Tree Model**.

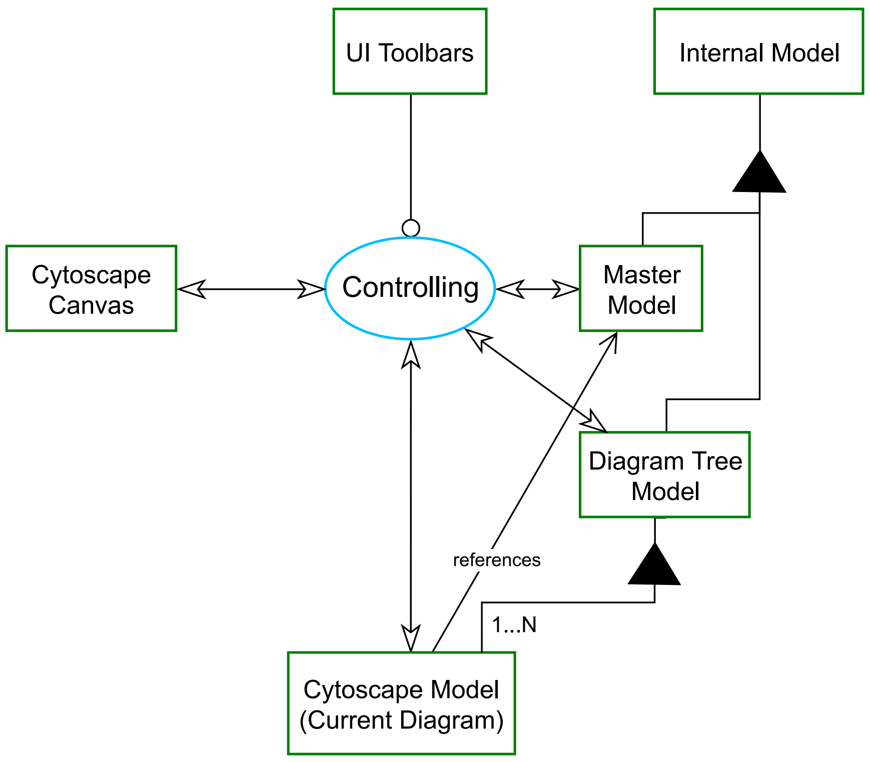


Figure 4.3: The top-level architecture of the editor specified in the OPM notation.

Chapter 5

Implementation of the agile editor

In the previous chapter, the data model was designed, and the fitting technologies were chosen. This chapter describes main features of the editor and specifies details of their implementation. The application is available online on a free hosted website[1] and the source code is available on a public GitHub repository[2].

Figure 5.1 shows the GUI (Graphical User Interface) of the application. In the center is the diagram canvas, which displays the currently selected diagram and allows for interaction with it. This canvas is handled by the Cytoscape.js library, discussed in Section 4.2, and its extensions.

In the left sidebar, there is the diagram tree, described in Section 3.6. The top toolbar contains the rest of the application's functionality. The top toolbar sections are as follows, listed from the left: propagation selection, demo example selection, export to PNG button, export to JSON button, and import from JSON button. All of these functions are presented in the following sections. In this chapter, the term *diagram entity* is used to describe any part of an OPM diagram, which means an object, a process, a state, or an edge.

5.1 Context menu

Actions regarding modifications of a diagram (that is, element addition, element deletion, change of attributes, etc.) are handled using a context menu, which can be invoked by right-clicking on either a diagram entity or the diagram canvas. The context menu is provided by an extension of Cytoscape.js, cytoscape-context-menus[3] (its creation is associated with this paper [2]). With the help of the extension, it is possible to specify menu options and define actions that are executed upon clicking them. In addition, selectors can be added to cause the options to show only on the specified targets. This is shown in Figure 5.2. On the left, there is a context menu that is displayed upon right-clicking on the diagram canvas, while on the right, there is a context menu for an object.

The following actions are available through the context menu: adding objects and processes, adding states to objects, in-zooming processes, changing essence and affiliation of objects and processes, hiding and removing of all diagram entities, option to show all hidden diagram entities, options regarding the edge editing (adding bend and control points and

[1] `https://opm-editor.netlify.app/`

[2] `https://github.com/micholenko/OPM_Editor`

[3] `https://github.com/iVis-at-Bilkent/cytoscape.js-context-menus`

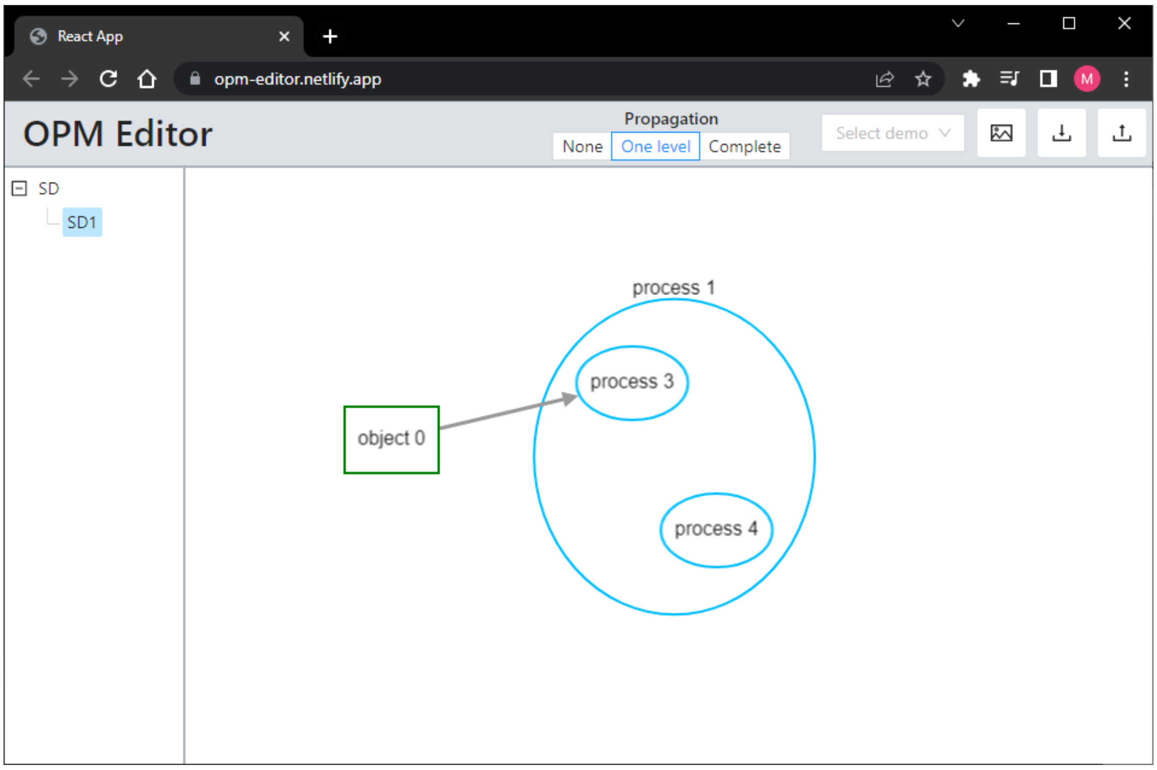


Figure 5.1: Graphical user interface of the implemented OPM editor

their removal), bringing all states of a given object and bringing connected edges of a given element.

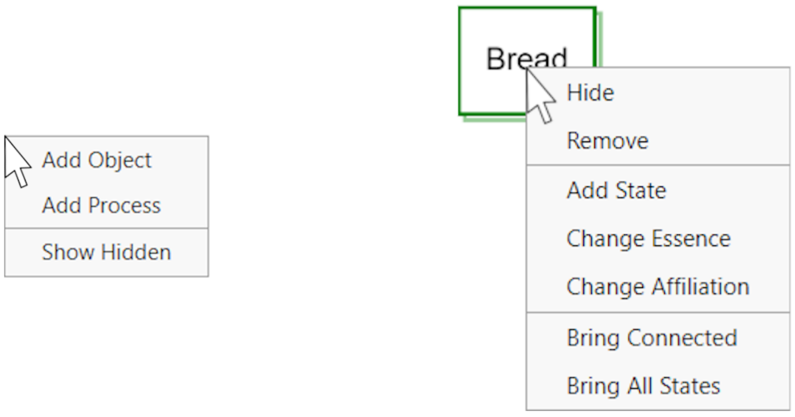


Figure 5.2: Context menus difference based on the right-click target.

Bringing states and edges works in a similar manner; the master model is queried for diagram entities, which is an inexpensive and a straightforward action. The `Bring All States` option works as a way to propagate states associated with a certain object from different diagrams. The `Bring Connected` option is more selective. It presents a list of possible relationships that were modeled in different diagrams and can be propagated to the current one. These options are listed in a pop-up modal shown in Figure 5.3 (implemented with the help of the Ant Design modal[4] component). To render each option, an instance of the Cytoscape.js library is conveniently used. A user is then able to select an edge and this edge along with the connected element is added. Both of the aforementioned options also take into account hidden entities; they are revealed or presented as an option to be revealed.

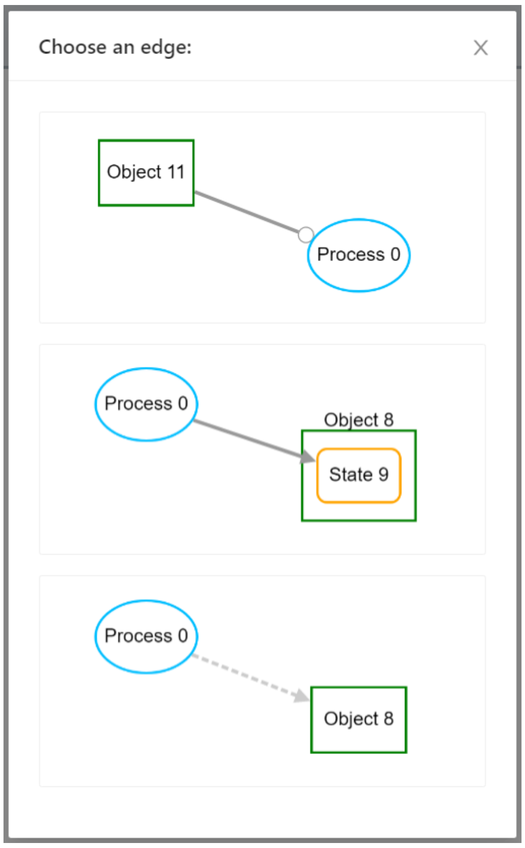


Figure 5.3: List of options of edges that can be propagated to the **Process 0**. This modal is displayed after selecting the context menu option, `Bring Connected`.

It is also worth mentioning the difference between hiding and removing diagram entities. Hiding deletes the entity from the current diagram, and it is possible to restore these hidden entities with the `Show Hidden` option, while removing deletes the element from the entire model irreversibly and, therefore, in all diagrams.

Most of the other options are fairly trivial and are not worth being described in detail. They simply modify the master model, using methods of the master model, and the current diagram, using the Cytoscape.js API (Application Programming Interface).

5.2 Propagation

The propagation is a relatively unique feature that is not implemented in present-day diagramming tools. Propagation aims to save time by automatically duplicating added diagram entities to other affected diagrams. The modeler would then not have to repeat the changes manually for every individual diagram. Note that propagation only affects the

[4]https://ant.design/components/modal/

creation of diagram entities, whereas when it comes to deletion, the options are either to hide in the current diagram or to remove it from the entire model.

A user can choose between three levels of propagation that change the behavior of the editor: `None`, `One level`, and `Complete`. The propagation selection is located in the top toolbar (see Figure 5.1).

`One level` is the default propagation option. It causes the created diagram entities to be replicated to diagrams one level above and below in the diagram tree, but only when it is possible. The `Complete` mode attempts to replicate the created diagram entities across all the diagrams. And the `None` mode does no propagation at all, and the diagram entities are unique to the current diagram.

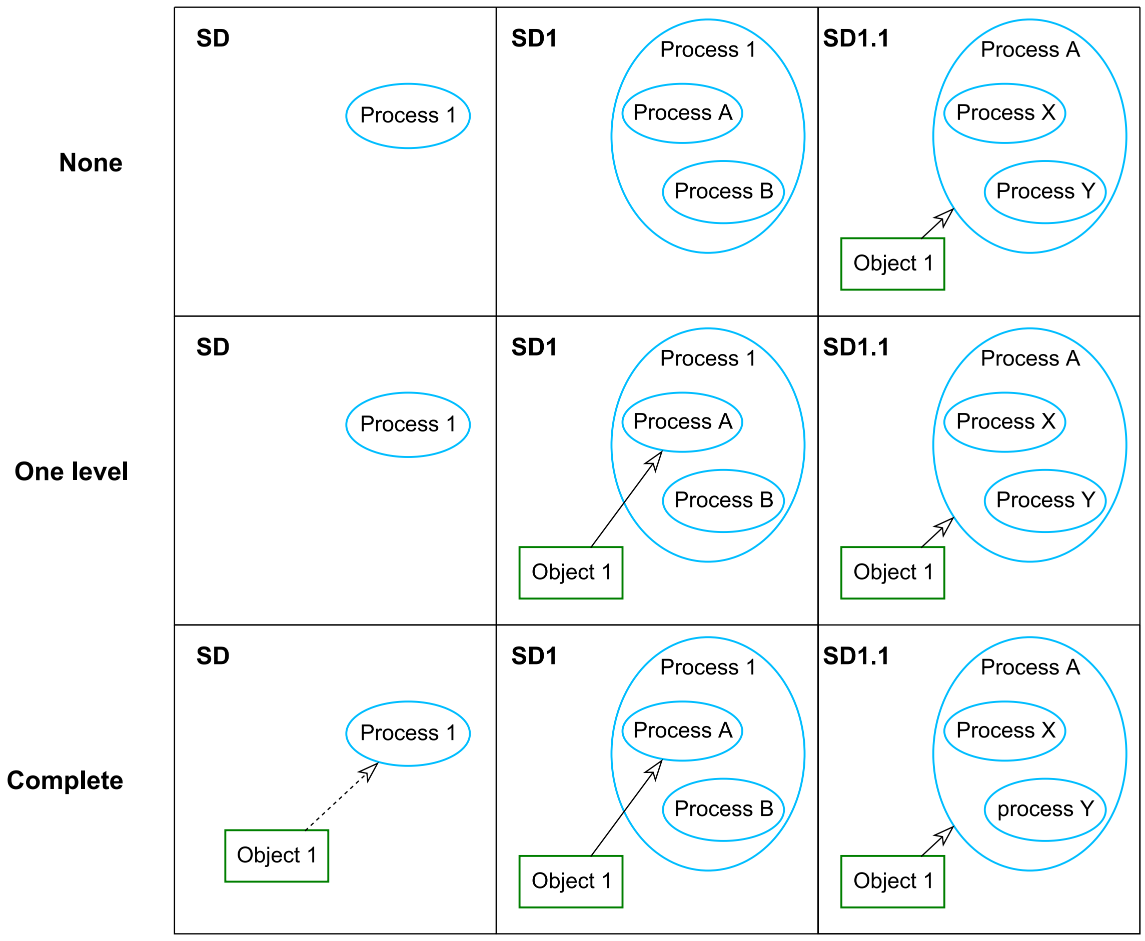


Figure 5.4: Comparison of effects on diagrams of the three modes of propagation: `None`, `One level` and `Complete`. The state of the each row is different after connecting the **Object 1** in the third diagram.

To explain this more clearly, an example is shown in Figure 5.4. Each row shows the same three diagrams after connecting **Object 1** to **Process A** in the diagram **SD1.1**. Depending on the propagation mode, **Object 1** along with its connected edge is replicated in the other diagrams. The situation in the first row is straightforward, as no diagram entities were propagated. In the second row, propagation occurs, as the diagram entities added from **SD1.1** are copied over to **SD1**. Notice that the two edges share the source and target elements in both diagrams and are, in fact, two projections of the same edge.

However, in the third row, the edge added in the **SD** diagram is not the same as the other two edges. The target of the edge is **Process 1**, which is the parent of **Process A**, and therefore it is a derived edge. For more information on the derived edges, see 3.4. Note that in this figure and many other figures shown in this thesis, the derived edge has a dashed line style for visualizing purposes. In the actual editor, derived edges are solid, but with a slightly lighter gray color.

5.3 Edge creation, reconnection and editing

Edges are created by right-clicking on the source element, holding the right mouse button, and releasing it on the desired target element. The cytoscape-edgehandles[5] extension facilitates this functionality. Upon the release of the right mouse button, a pop-up modal appears with a selection of edge types, and the edge is created after the user selects the type. Figure 5.5 shows the pop-up modal. Potential derived edges, presented in Section 3.2, are also created at this point.

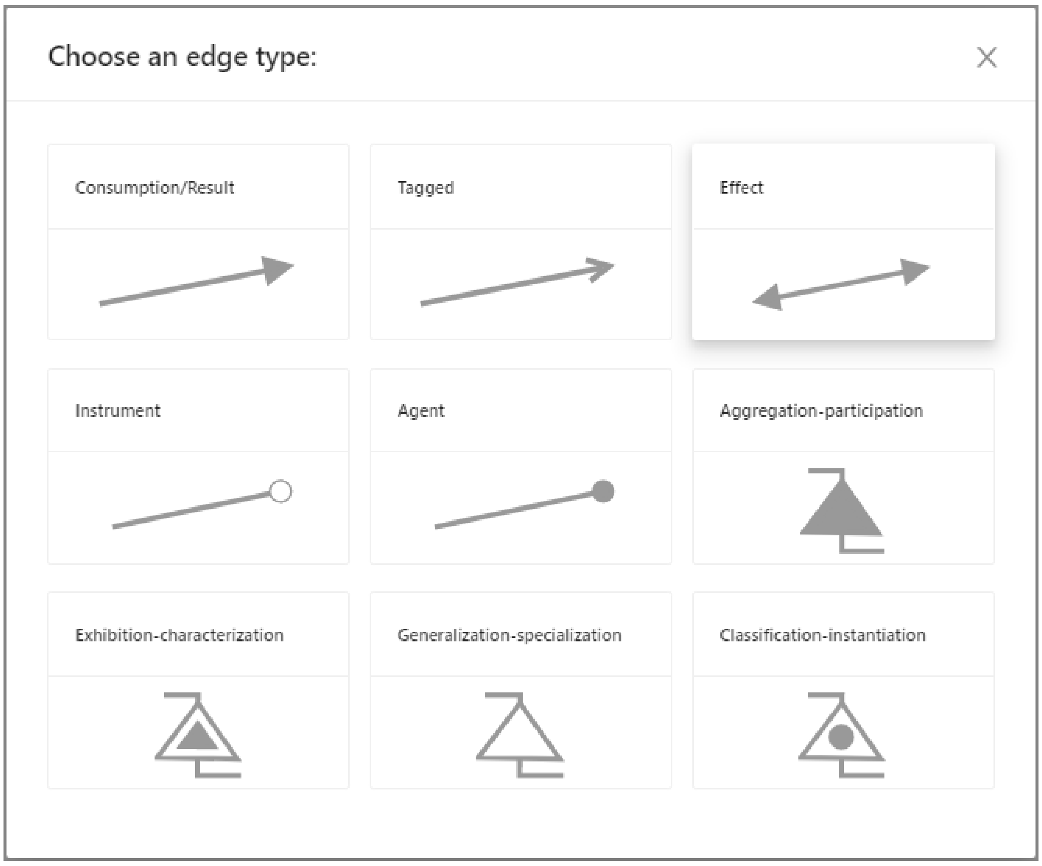

Choose an edge type:
Consumption/Result
Tagged
Effect
Instrument
Agent
Aggregation-participation
Exhibition-characterization
Generalization-specialization
Classification-instantiation

Figure 5.5: Edge type selection modal.

Reconnecting an edge to a different source or target (referred to only as reconnection) is provided by another Cytoscape.js extension, cytoscape-edge-editing[6] (its creation associated with this paper [2]). It adds two anchor points at both ends of an edge. By clicking and

[5]https://github.com/cytoscape/cytoscape.js-edgehandles
[6]https://github.com/iVis-at-Bilkent/cytoscape.js-edge-editing

dragging the anchor to a different element, it is possible to change the source or target of an edge. Reconnection of an original edge could cause its former derivatives to no longer be relevant and, therefore, they need to be removed in most cases. Reconnecting a derived edge promotes the edge to be an original edge and removes its former associated edges: the original and other derivatives. This solution may not be ideal for some use cases and may seem confusing to some users. When it comes to the reconnection and many other features, concrete behavior is the subject of debate.

The same extension also facilitates the option of editing edges by adding bend points or control points. To add them to an edge, the edge context menu is extended with associated options. The bend points, control points, and anchor points are shown in Figure 5.6. Note that adding control points to any edge makes it a Bézier curve.

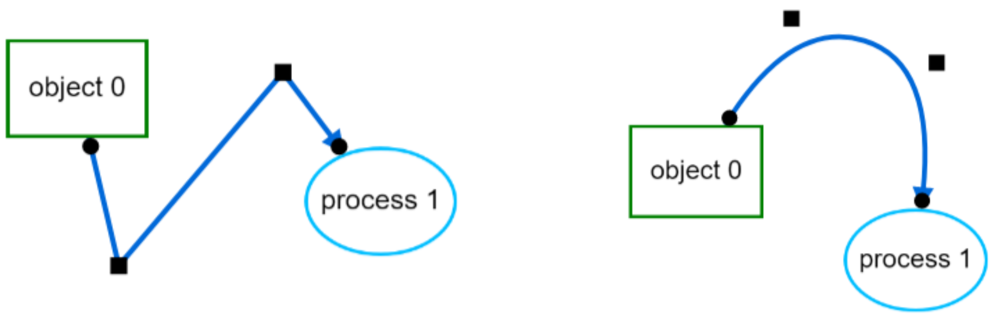


Figure 5.6: Bend points and control points allowing for interactive edge editing. The anchor points on the ends of edges used for reconnection are shown as well.

5.4 Name editing

Changing names of diagram entities is an essential functionality that any diagram editor should provide. This feature is implemented with the help of the cytoscape-popper[7] extension, which dynamically adds HTML elements and positions them next to the associated entities and over the diagram area (HTML canvas element). Double-clicking on an entity invokes a prompt dialog with an input field to change the name. An example of such a dialog is presented in Figure 5.7.

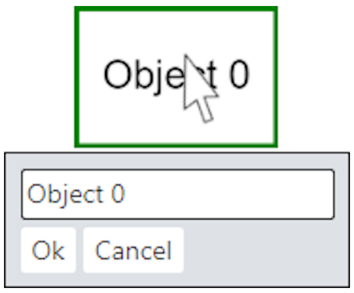


Figure 5.7: Name editing of an object.

[7]https://github.com/cytoscape/cytoscape.js-popper

5.5 Diagram switching

Diagram switching occurs after two user actions: process in-zooming and selection of a diagram in the diagram tree shown on the left in Figure 5.1.

In-zooming is simpler, as it creates a new diagram with the in-zoomed process as its main element and optionally copying connected elements. Note that whether the connected elements are copied over is based on the chosen propagation mode. In summary, `One level` and `Complete` modes would propagate the connected elements, while `None` would not. An example of in-zooming is shown in Figure 2.7.

Switching to an already existing diagram is a bit more complicated. While diagrams are not displayed, they are not updated, as it would be time-consuming and resource-intensive to identify all affected diagrams on every model change. In addition, a user might take his changes back, which would make the modifications pointless. Instead, a diagram is updated on demand when a user wants to see it.

First, the potentially outdated diagram is imported into the Cytoscape.js library. This process is explained in more detail in Section 4.2. Names and attributes are automatically updated as they are referenced from the master model. All diagram entities need to be checked for existence because if they were removed, they would have to be removed from the current diagram explicitly. When it comes to the removed elements and states, their connected edges need to be removed as well. Then, for every individual element or state, the master model is searched for new connected edges, and, if found, they are added. This is how propagation is implemented. Reconnection of an edge could result in the removal of its derived edges or its original edge, but this is determined when the reconnection event occurs.

5.6 Changes in cytoscape source code

Some functionalities were not attainable with the features provided by Cytoscape.js or its extensions. There is also a possibility to create our own extensions to add new features. In our case, however, the issues do not stem from the lack of functionality but rather from the unsuitability of the library's inherent behavior. Instead, it was necessary to directly modify the source code of the library. This fact means that moving to newer versions of the library will be more difficult, as newer releases would have to be rebased on our modified version, and merge conflicts may occur. Arguably, these issues might happen only rarely and could be resolved quickly, as large code refactorings are not to be expected given the maturity of the library.

Cytoscape.js does not support compound nodes that are ellipses. The reason for this might be the fact that checking the boundaries of a circular shape is computationally more expensive, and therefore developers chose not to include this option. In order to enable this functionality, first, an ellipse had to be added to the list of supported shapes. Second, because the ellipse bounding box would always be rectangular, the calculation of the compound node size had to be changed.

The library offers only a limited number of arrow types and does not provide any way to add new ones. In order to attain the exhibition-characterization and classification-instantiation edge arrow types, it was necessary to implement them directly in the library code.

Another notable modification was to the behavior of the taxi edges (right-angled edges used, for example, for the aggregation-participation edge). In this case, the problem was that multiple edges would not reliably converge into one when approaching the superior element. Due to this fact, the calculations of the taxi edge bend points had to be tweaked.

There are several other cases where it is necessary to change the behavior of the library. If the editor were used by real users, these issues would need to be addressed. Modifying the source code directly is a bit more time-consuming, but certainly possible, as shown above. This fact reaffirms that the open-source approach and the chosen library are viable.

5.7 Export and import of diagrams

The application allows for export to PNG and JSON. Exporting to PNG works only for the currently displayed diagram, although extending this feature to export all diagrams would not be a difficult task. This conversion is handled by Cytoscape.js, which also supports the JPG format. The SVG format is often problematic, as various implementations are not always compatible. Converting a diagram to SVG is currently not supported by the library or the implemented application, but the cytoscape-svg[8] extension could potentially be utilized for this purpose.

Exporting to JSON functions is a way to save the modeler's work and is a bit more complicated, as it is not concerned with just a single diagram but should encompass the state of the whole model. This means that the resulting JSON file should include the diagram tree model, which contains data of all diagrams, the element model, and the edge model. All of this is discussed in Chapter 3.

In JavaScript, the standard way to serialize data to JSON format is through the built-in call `JSON.stringify()`. However, this way does not work well with complex data structures because of two reasons: the call fails when circular references are encountered, and it resolves objects to their attributes every time their reference is encountered, creating redundancy. Both of these problems are present in our case. Although the circular references could theoretically be removed, multiple references on the same objects are not avoidable. Another important drawback of this approach is that using the built-in parsing call `JSON.parse()` will not restore the previous model structure with all its interconnected objects and references.

A way to solve this would require manually serializing all model elements, defining a data format in which the model would be stored, and then implementing a functionality that would recreate the model structure based on the stored data. This seems like a non-trivial and time-consuming undertaking; fortunately, there are libraries that perform this function well.

An open-source library called TeleJSON[9] solves all the aforementioned problems. The library serializes each object only once and references objects with an absolute path. Through functions provided by this library, it is possible to serialize complex object structures and then deserialize them to their previous form. The only downside is that, through this process, prototypes of the objects are lost, which means that the objects lose their methods, and calls to these methods would result in an error. Because of this, the structures need to be iterated over to reassign the prototypes. However, this is a fairly simple task.

Additionally, the resulting JSON files are created deterministically, which means that if the model has not been changed, the JSON files will be indistinguishable. A change in a model would be reflected only in the relevant JSON parts, so two files can be checked for

[8]https://github.com/kinimesi/cytoscape-svg
[9]https://www.npmjs.com/package/telejson

differences. Models are often stored in textual form and versioned by tools like Git, and for these purposes, it is important that the differences are minimal. This is not always the case as (for example, element IDs) could be generated randomly, resulting in many changes every time models are exported. Minimal differences are also important for porting a model between different editors and for a possible collaborative extension of the editor.

Chapter 6

Evaluation

The editor, implemented in the previous chapter, is evaluated in this chapter. First, a non-trivial example model is created using the editor to prove that it is capable of serving its purpose. Next, the implemented editor is compared to the already existing OPM diagramming tools. Finally, the shortcomings of the editor are discussed.

6.1 An example use case

In this section, a model is created with the help of the implemented editor to show its usability. The models used in this section were created in the editor and then exported.

As an example, a bread baking model is used, which was inspired by [3]. The creation of the model will not be as straightforward as is the case in the cited source. Instead, the model is created gradually and with the aim of highlighting the full potential of editor's functionalities, namely propagation and derived edges. This is arguably much closer to the reality of modeling, as it is usually hard to get the model right on the first try.

Figure 6.1 shows the first step in the modeling process. In **SD**, there is a high-level view showing the process of **Baking** consuming **Ingredients** and producing **Bread**. **Equipment** is necessary for **Baking** to occur. On the right, in **SD1**, there is the **Baking** process zoomed in. Here, **Baking** is refined into multiple subprocesses (**Mixing**, **Forming**, **Heating**, **Cooking**) and transient objects (**Dough** and **Loaf**). At this level, it also makes sense to add states to the **Bread** object and model the state transition by parts of the **Baking** process. Notice the derived edges that are denoted with the dashed edge style. This is only for the purpose of visualization, and in the actual editor, the derived edges only have a slightly lighter gray color.

Next, we want to add **Baker** to both levels. To spare the modeler from repeating the same action for multiple diagrams, the propagation feature could be used. Setting the propagation mode to `One Level`, and subsequently creating the **Baker** object along with an agent edge linked to **Baking**, will duplicate the action for both diagrams. This addition could be done in either diagram and would be propagated into the other. Then **Energy** is added as an instrument for **Heating**. Let us assume that it is only integral to the **SD1** diagram and not **SD**. To do this, the propagation mode is set to `None`. In this way, only the current diagram would be affected. The results of these actions are shown in Figure 6.2.

Finally, in **SD1**, **Ingredients** and **Equipment** are specified and modeled in more detail. To do this, the aggregation-participation and generalization-specialization edges are used, respectively. The objects **Flour**, **Water**, **Yeast**, and **Salt** are modeled as parts of

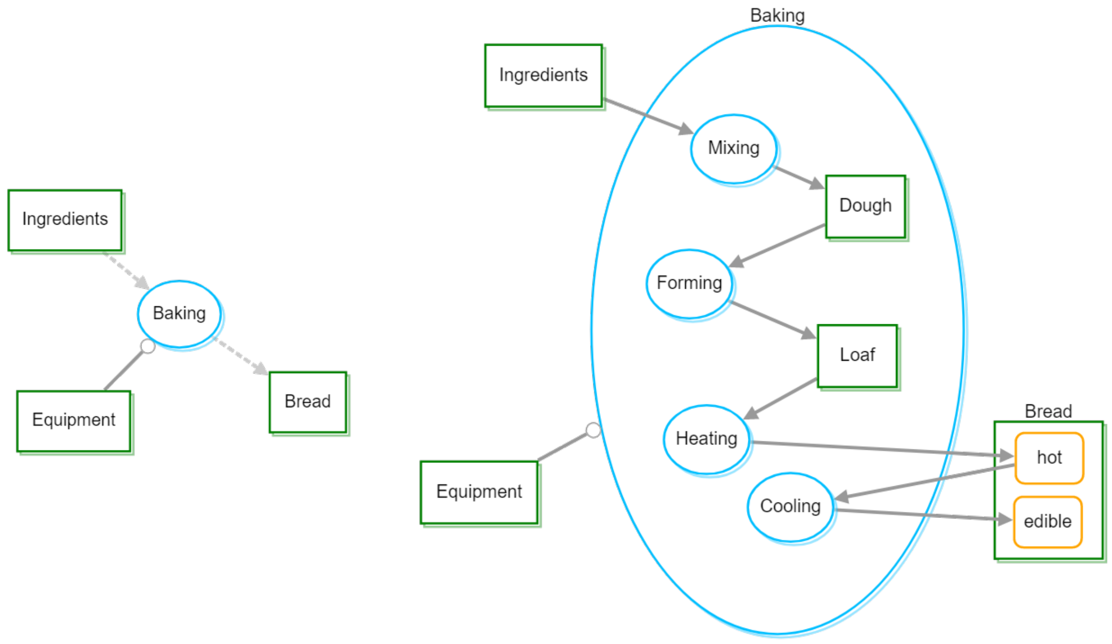


Figure 6.1: Step 1 in modeling bread baking.

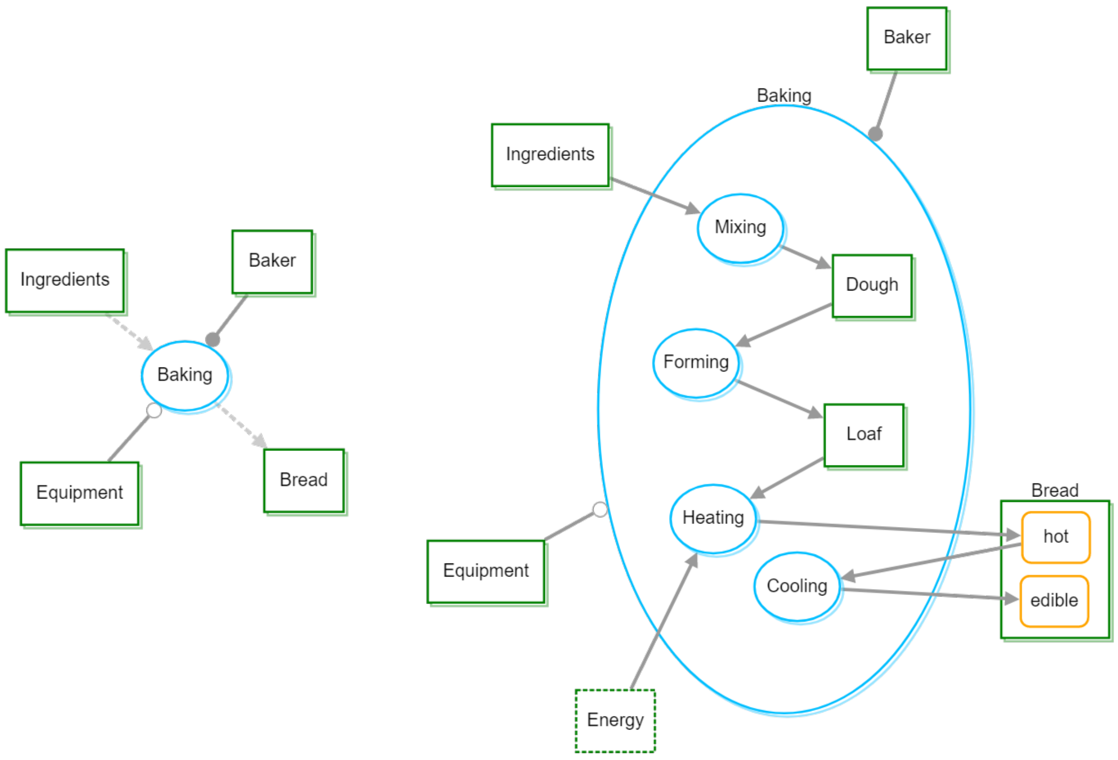


Figure 6.2: Step 2 in modeling bread baking.

Ingredients, and **Mixer** and **Oven** are represented as specializations of **Equipment**. The None propagation level ensures that these structural relationships are unique to **SD1**. Furthermore, the newly added procedural edges of **SD1** are abstracted into one edge in **SD**. This phenomenon is described in detail in Section 3.2.

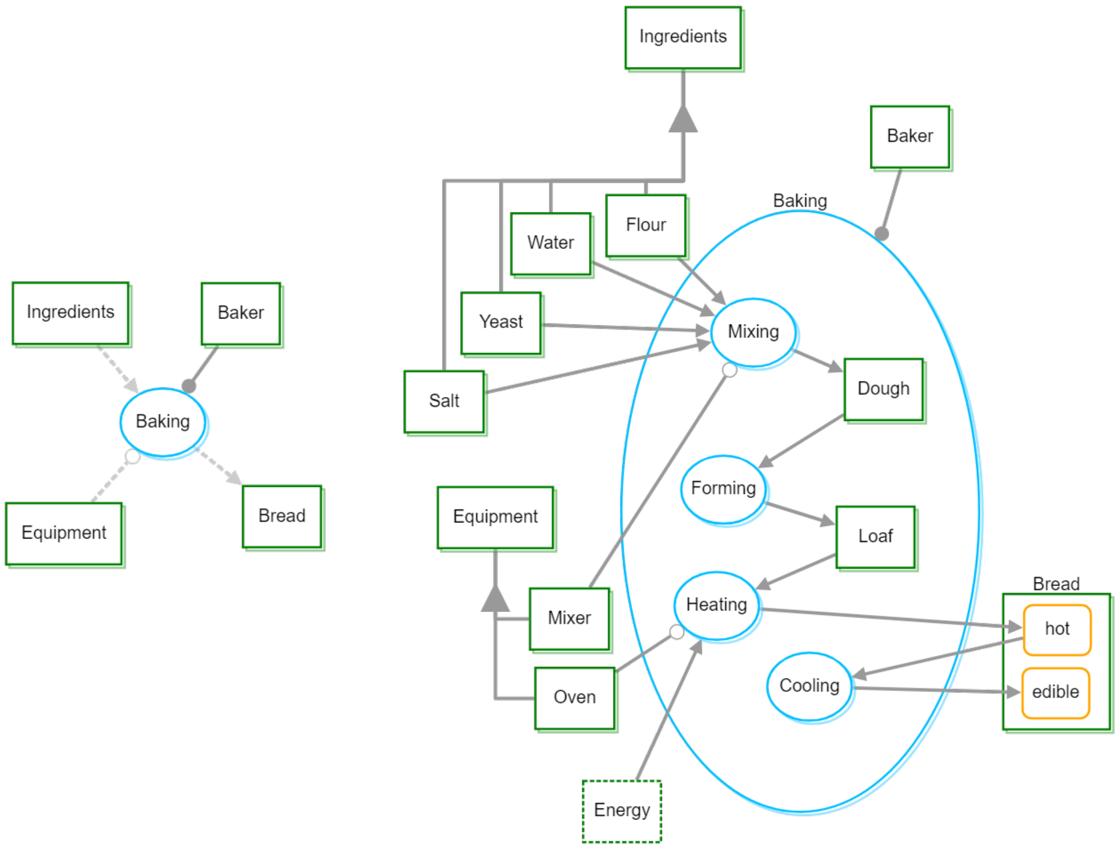


Figure 6.3: Step 3 in modeling bread baking.

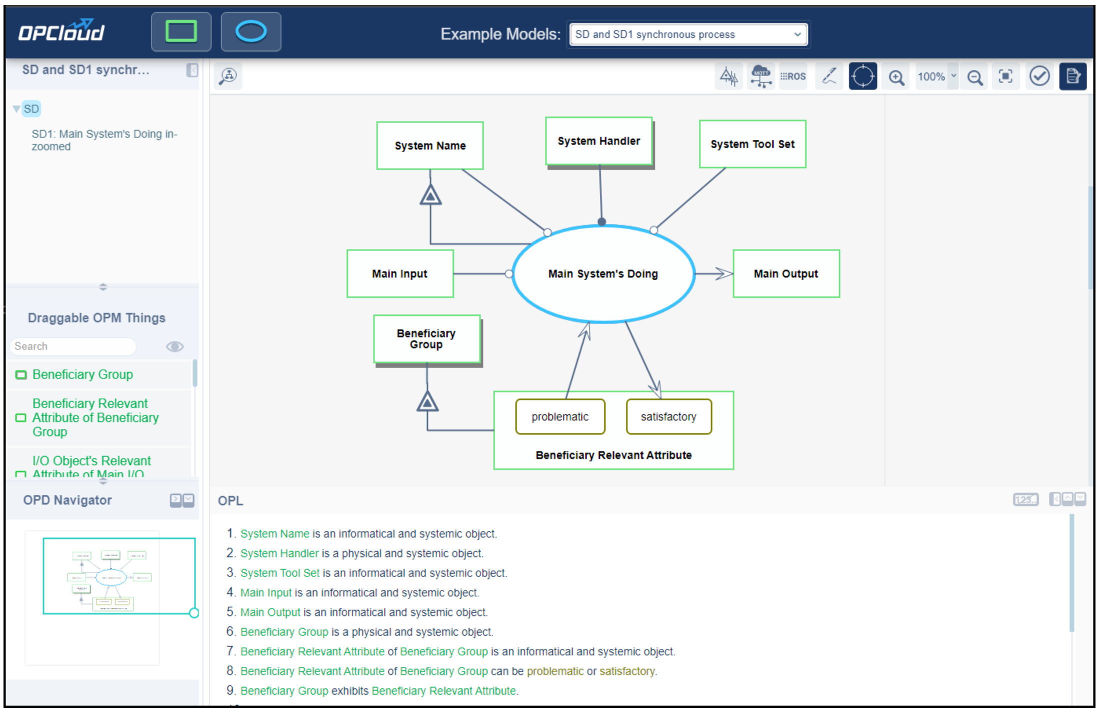
6.2 Comparison with existing OPM editors

As of now, there are two currently available official OPM diagramming tools: OPCloud[1] and Opcat[2]. Their GUIs (Graphical User Interfaces) are shown in Figures 6.4 and 6.5, respectively.

Opcat is a free desktop application written in Java. However, it is an older one of the two. In fact, it is no longer maintained, and the newer tool, OPCloud, is its successor.

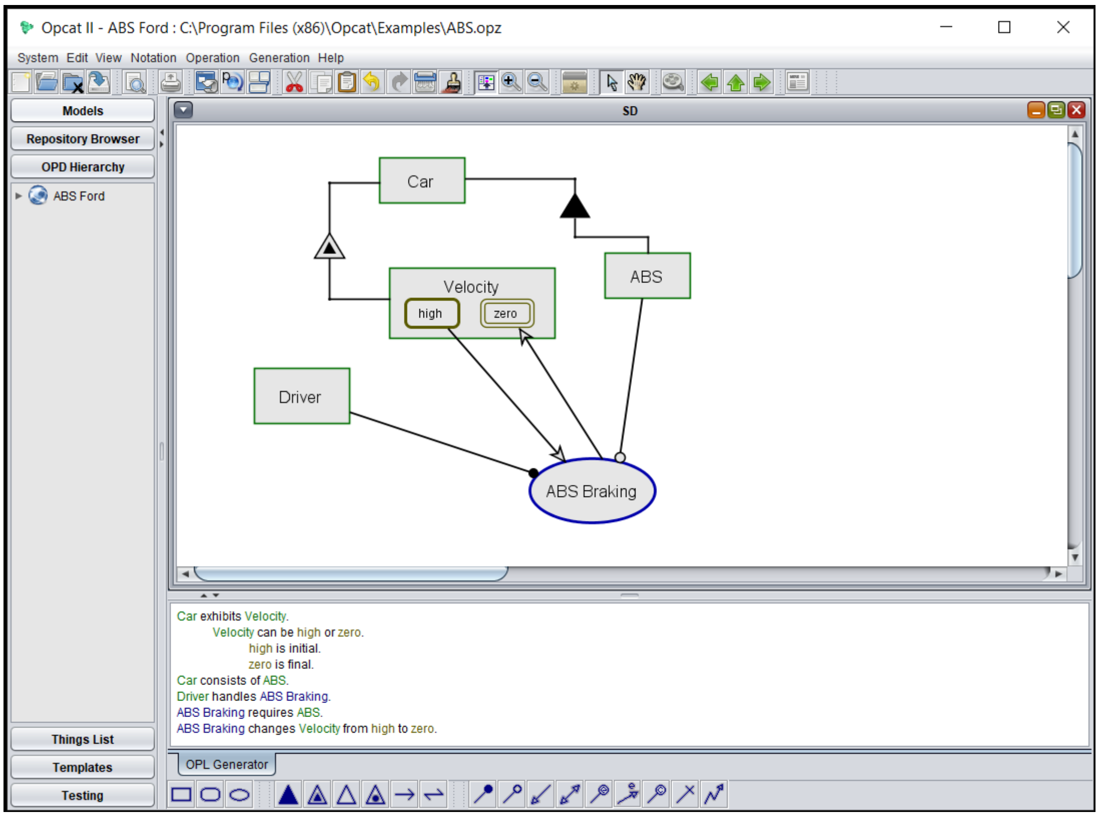
OPCloud is a web-based diagramming tool that has been developed quite recently using Angular and a web diagramming library, Rappid (discussed in 4.1). It is a full-featured diagramming editor with plenty of advanced functionality, such as model simulation and execution, or support for collaborative work of multiple users at the same time. Unfortunately, OPCloud is not free software, although a free account is provided for academic purposes. The demo version, available on the public OPCloud website, offers only a limited

[1]https://sandbox.opm.technion.ac.il/
[2]https://esml.technion.ac.il/opm/opcat-installation/



1. System Name is an informatical and systemic object.
2. System Handler is a physical and systemic object.
3. System Tool Set is an informatical and systemic object.
4. Main Input is an informatical and systemic object.
5. Main Output is an informatical and systemic object.
6. Beneficiary Group is a physical and systemic object.
7. Beneficiary Relevant Attribute of Beneficiary Group is an informatical and systemic object.
8. Beneficiary Relevant Attribute of Beneficiary Group can be problematic or satisfactory.
9. Beneficiary Group exhibits Beneficiary Relevant Attribute.

Figure 6.4: OPCloud, the web-based OPM diagramming tool (demo version).



Car exhibits Velocity.
Velocity can be high or zero.
high is initial.
zero is final.
Car consists of ABS.
Driver handles ABS Braking.
ABS Braking requires ABS.
ABS Braking changes Velocity from high to zero.

Figure 6.5: Opcat, the desktop OPM diagramming tool.

functionality and, for example, lacks even essential importing and exporting. The conclusion is that for now, there is no free, full-featured, and currently maintained OPM editor. This makes the development of a new alternative a worthwhile investment, as the result could be used by the open-source community.

There are a few features that make the implemented editor stand out from OPCloud. The automatic propagation (`None`, `One level`, `Complete`) as well as the selective propagation (`Bring Connected` and `Bring States` context menu options) and derived edges might be difficult to comprehend at first, but once learned, they have the potential to save time and make the modeling process easier. The automatic propagation is discussed in Section 5.4, the selective propagation is described in Section 5.2, and the derived edges are presented in Section 3.4.

To summarize, Table 6.1 presents a comparison of all existing editors.

	Opcat	**OPCloud (demo)**	**OPCloud (full version)**	**OPM Editor (this thesis)**
Open-source	No	No	No	Yes
Basic diagramming capabilities	Yes	Yes	Yes	Yes
Advanced OPM features (simulation, code generation, ...)	Yes	No	Yes	No
Collaborative editing	No	No	Yes	No
Export/Import	Yes	No	Yes	Yes
Automatic relationship derivation	No	No	No	**Yes**
Automatic propagation	No	No	No	**Yes**
Selective propagation	No	No	No	**Yes**
Master model (no duplication of data)	No	No	No	**Yes**

Table 6.1: Comparison of Opcat, OPCloud demo version, OPCloud full version and the implemented editor.

6.3 Missing functionality

The aim of this thesis was to implement an agile OPM diagram editor that focuses on complexity management. This goal was achieved. However, creation of a full-featured application would require more time, and therefore, the editor could be thought of as a prototype. Some features are absent because it was decided that they were too difficult or time-intensive to implement and not important enough for the usability of the editor. The following is a list of improvements to the application.

- **Allow the diagrams to move in the diagram tree.** The tree component of the Ant Design library is easily extensible with this functionality.
- **Changing names of diagrams in the diagram tree.** Again, the tree component was selected with this functionality in mind.

- **Resizing elements.** An extension[3] is available, but due to compatibility issues, it is not usable. To implement this feature would likely require modifications in the said extension.
- **Changing edge types on already created edges.** A workaround is to remove the edge and create a new one, but that is not an ideal solution.
- **Sidebar for advanced element modification.** Currently, all the modification of elements is done by right-clicking and the context menus. It would be convenient to have a sidebar for advanced element modification (for example, width or height) that would get invoked on element selection.
- **Undo/Redo.** To support this feature for a single diagram is quite trivial as the cytoscape.js-undo-redo[4] extension does exactly that. However, this approach would only take care of the visual modifications handled by Cytoscape.js, but user actions often affect the whole model. Because of this, a possible solution might not be trivial.
- **OPL generation.** The OPL (Object-Process Language, see 2.6) sentences could be advantageously generated from the used master model. Potentially, this could be handled externally, by a different application, and only displayed in the editor.
- **Restrict the options of edge types in edge creation based on source and target.** The absence of this feature allows users to create relationships that are invalid according to the OPM standard.
- **Export to SVG.** Possibly solvable by the existing cytoscape-svg[5] extension.
- **etc.**

[3] `https://github.com/iVis-at-Bilkent/cytoscape.js-node-editing`
[4] `https://github.com/iVis-at-Bilkent/cytoscape.js-undo-redo`
[5] `https://github.com/kinimesi/cytoscape-svg`

Chapter 7

Conclusion

The objective of this thesis was to create a prototypeof an OPM diagram editor[1] that is able to handle complex models efficiently. The most important aspect in achieving this goal is a well-designed internal data model without data duplication. This allowed efficient implementation of derived relationships and easy realization of data propagation. Other notable outcome is the survey of suitable technologies for diagram editors, most importantly the web diagramming libraries.

First, MDD (Model-Driven Development) was introduced with all its potential benefits. Then, overviews of three modeling languages, UML (Unified Modeling Language), SysML (System Modeling Language) and OPM (Object-Process Methodology), were presented, and their differences were highlighted. All important aspects of OPM were described, since the editor was implemented to support modeling with this language. The data model, whose structure was designed to facilitate the modeling nature of OPM, was then proposed. Next, suitable technologies were chosen for the implementation, and a thorough evaluation of the diagramming libraries was carried out. Out of this assessment, a library called Cytoscape.js proved to be the most appropriate for our purpose. In the following part, the editor's implementation was described, its interesting aspects were pointed out as well as obstacles encountered during the development process. And lastly, the editor was put to the test by presenting a non-trivial use case while showcasing its functions. In addition, this part included a comparison with its potential competitors.

The implemented prototype is usable but still offers many possibilities for improvement and extensibility. Apart from smaller features, these are some non-trivial ways in which the editor could be extended.

- Collaborative editing of multiple users at once
- Integration of the editor into an existing development support tools, such as Jira
- Code generation from models
- Animated simulation

A closely related thesis by Jana Treláková from Masaryk University – Faculty of Informatics is being created in parallel and in cooperation with this one. Her thesis focuses mainly on the collaborative aspect but uses the same data model, which is an important precondition for the integration of the results of these two theses.

[1] Available at: https://opm-editor.netlify.app/

Bibliography

[1] Cytoscape.js creators. *Cytoscape.js documentation.* 2022. [Online; accessed 13-April-2022]. Available at: https://js.cytoscape.org/.

[2] Dogrusoz, U., Karacelik, A., Safarli, I., Balci, H., Dervishi, L. et al. Efficient methods and readily customizable libraries for managing complexity of large networks. *PLOS ONE.* may 2018, vol. 13, p. e0197238. DOI: 10.1371/journal.pone.0197238.

[3] Dori, D. *Object-Process Methodology for Structure-Behavior Co-Design.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. 209–258 p. ISBN 978-3-642-15865-0. Available at: https://doi.org/10.1007/978-3-642-15865-0_7.

[4] Dori, D. *Model-Based Systems Engineering with OPM and SysML.* 1stth ed. Springer Publishing Company, Incorporated, 2016. ISBN 1493932942.

[5] Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* 3rd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321193687.

[6] Franz, M., Lopes, C. T., Huck, G., Dong, Y., Sumer, O. et al. Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics.* september 2015, vol. 32, no. 2, p. 309–311. DOI: 10.1093/bioinformatics/btv557. ISSN 1367-4803. Available at: https://doi.org/10.1093/bioinformatics/btv557.

[7] Friedenthal, S., Steiner, R. and Moore, A. *A Practical Guide to SysML: the Systems Modeling Language.* January 2008.

[8] Huldt, T. and Stenius, I. State-of-practice survey of model-based systems engineering. *Systems Engineering.* 2019, vol. 22, no. 2, p. 134–145. DOI: https://doi.org/10.1002/sys.21466. Available at: https://onlinelibrary.wiley.com/doi/abs/10.1002/sys.21466.

[9] Korec, T. *Agile Model Editor.* Brno, CZ, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/23038/.

[10] Ma, Z. Business ecosystem modeling- the hybrid of system modeling and ecological modeling: an application of the smart grid. *Energy Informatics.* november 2019, vol. 2. DOI: 10.1186/s42162-019-0100-4.

[11] Mellor, S., Clark, A. and Futagami, T. Model-driven development - Guest editor's introduction. *IEEE Software.* 2003, vol. 20, no. 5, p. 14–18. DOI: 10.1109/MS.2003.1231145.

[12] Indeed Editorial Team. *What Is a Web Application? How It Works, Benefits and Examples*. 2021. [Online; accessed 1-April-2022]. Available at: https://www.indeed.com/career-advice/career-development/what-is-web-application.

[13] Ed-douibi, H. *20+ JavaScript libraries to draw your own diagrams (2022 edition)*. February 2022. [Online; accessed 12-April-2022]. Available at: https://modeling-languages.com/javascript-drawing-libraries-diagrams/.

[14] Wikipedia contributors. *Object Process Methodology — Wikipedia, The Free Encyclopedia*. 2022. [Online; accessed 31-March-2022]. Available at: https://en.wikipedia.org/w/index.php?title=Object_Process_Methodology&oldid=1072017082.

[15] Selic, B. The pragmatics of model-driven development. *IEEE Software*. 2003, vol. 20, no. 5, p. 19–25. DOI: 10.1109/MS.2003.1231146.

[16] Wikipedia contributors. *Unified Modeling Language — Wikipedia, The Free Encyclopedia*. 2022. [Online; accessed 3-April-2022]. Available at: https://en.wikipedia.org/w/index.php?title=Unified_Modeling_Language&oldid=1084962438.

Appendix A

Manual

The following is a quick reference to the editor's functions:

Basic functions:

Adding objects – Right-click on the diagram canvas and choose *Add Object* in the context menu.

Adding processes – Right-click on the diagram canvas and choose *Add Process* in the context menu.

Creating edges – Right-click on the desired source element, hold, and release on the target element. In the pop-up modal, select an edge type.

In-zooming processes – Right-click on the a process and choose *In-zoom* in the context menu.

Edge editing:

Reconnecting edges – Select an edge by left-clicking on it. Two anchors should appear at both ends of the edge. By left-clicking, drag either of the anchors to a new endpoint.

Adding bend points edges – Select an edge by left-clicking on it. Right-click on the selected edge and choose *Add Bend Point* in the context menu. Position the bend point to edit the edge.

Adding control points edges – Select an edge by left-clicking on it. Right-click on the selected edge and choose *Add Control Point* in the context menu. Position the bend point to edit the edge.

Removing bend points – Select an edge by left-clicking on it. Right-click on a bend point and choose *Remove Bend Point* in the context menu.

Removing control points – Select an edge by left-clicking on it. Right-click on a control point and choose *Remove Control Point* in the context menu.

Element editing:

Add states to objects – Right-click an object and choose *Add State* in the context menu.

Change affiliation – Right-click on an object or process and choose *Change Affiliation* in the context menu.

Change essence – Right-click on an object or process and choose *Change Essence* in the context menu.

Hide/Remove:

Hide elements or edges – Right-click an element or edge and choose *Hide* in the context menu. This action hides the target in the current diagram.

Remove elements or edges – Right-click an element or edge and choose *Remove* in the context menu. This action removes the target from all diagrams.

Show all hidden – Right-click on the diagram canvas and choose *Show hidden* in the context menu.

Propagation:

Bring connected elements – Right-click an element and choose *Bring Connected* in the context menu. A modal appears with options of elements connected to the selected element. Click on an option to add it into the current diagram.

Bring states – Right-click an object and choose *Bring States* in the context menu. This action automatically adds states of the selected object that are not present in the current diagram.

Propagation selection – Choose a mode of propagation in the selection located in the top toolbar. The options are *None*, *One Level* and *Complete.* Elements will be automatically propagated (or not) based on the selected mode.

Select demo – In the top toolbar, select a demo to import an already pre-made model.

Export current diagram to PNG – Click on the associated button located in the top toolbar.

Export/import the whole model to/from JSON – Click on the associated button located in the top toolbar.

Appendix B

Attached CD contents

```
/
├── thesis.pdf ........................ Text of the thesis
├── thesis_source/ .................... Latex source files and images
└── application/ ...................... Files related to the application
    ├── build/ ........................ Packaged application ready to be run in a browser
    ├── public/ ....................... Static files
    ├── src/ .......................... Application source code
    │   ├── components/ ............... React components definition
    │   ├── controller/ ............... Controller functions
    │   ├── css/ ...................... Stylesheets
    │   ├── model/ .................... Definition of model classes
    │   ├── options/ .................. Initilizing options for Cytoscape.js and its extensions
    │   ├── data/ ..................... Application data (images)
    │   └── index.tsx ................. React entry point
    ├── dependencies/ ................. Dependencies not handled by NPM (modified Cytoscape.js)
    ├── demos/ ........................ Pre-made model examples available in the application
    ├── package.json .................. NPM dependencies declaration
    ├── package-lock.json ............. NPM dependencies declaration
    ├── tsconfig.json ................. TypeScript configuration
    ├── LICENSE ....................... MIT license file
    └── README.md ..................... Readme file with manual and installation info
```