**BRNO UNIVERSITY OF TECHNOLOGY**
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

**FACULTY OF INFORMATION TECHNOLOGY**
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

**DEPARTMENT OF INTELLIGENT SYSTEMS**
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

# AGILE MODEL EDITOR
AGILNÝ EDITOR MODELOV

**BACHELOR'S THESIS**
BAKALÁŘSKÁ PRÁCE

**AUTHOR** **TOMÁŠ KOREC**
AUTOR PRÁCE

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

BRNO 2021

**Brno University of Technology**
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2020/2021

# Bachelor's Thesis Specification

23038

| | |
|---|---|
| Student: | **Korec Tomáš** |
| 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 easy refactoring and regrouping of model elements.
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, 2020 |
| Submission deadline: | May 12, 2021 |
| Approval date: | November 11, 2020 |

## Abstract

The thesis aims to minimize time spent on modeling software architecture and provide a practical tool to create, order, and visualize system models. The current modeling approaches consume too much time, often creating and editing the model costs more time than implementing such a system. The work focuses primarily on representing complex models efficiently, finding the best modeling language to do this task, and developing an agile editor.

## Abstrakt

Cieľom práce je minimalizovať čas strávený modelovaním softvérovej architektúry a poskytnúť praktický nástroj na vytváranie, zoraďovanie a vizualizáciu systémových modelov. Súčasné prístupy k modelovaniu zaberajú príliš veľa času, pričom vytvorenie a úpravy modelu často stoja viac času ako ich implementácia. Práca sa zameriava predovšetkým na efektívne zobrazenie zložitých modelov, nájdenie najlepšieho modelovacieho jazyka na vykonávanie tejto úlohy a vytvorenie agilného editora.

## Keywords

modeling, OPM, system architecture, system model, modeling languages, UML, SysML, web editor, model editor

## Kľúčové slová

modelovanie, OPM, systémová architektúra, systémový model, modelovacie jazyky, UML, SysML, webový editor, editor modelov

## Reference

KOREC, Tomáš. *Agile Model Editor.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Mgr. Adam Rogalewicz, Ph.D.

# 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.

. . . . . . . . . . . . . . . . . . . . . . .
Tomáš Korec
May 11, 2021

## Acknowledgements

I would like to thank my supervisor, doc. Mgr. Adam Rogalewicz, Ph.D. for professional guidance of the bachelor's thesis. I would also like to thank my technical supervisor of the bachelor's thesis, Ing. Jan Fiedor, Ph.D., for his time, valuable advice, and factual comments that helped me to complete this work.

# Contents

**1 Introduction** **2**

**2 Editor requirements** **4**
- 2.1 Modeling practices survey . . . 4
- 2.2 Code-Centric and Model-Centric Approaches . . . 6
  - 2.2.1 Problems with the model-centric approach . . . 7
  - 2.2.2 Problems with the code-centric approach . . . 9
- 2.3 Summary . . . 10

**3 Comparison of standard modeling languages** **11**
- 3.1 Unified Modeling Language . . . 11
- 3.2 Systems Modeling Language . . . 13
- 3.3 Object Process Methodology . . . 15
- 3.4 Modeling languages comparison . . . 17

**4 Implementation** **22**
- 4.1 Web diagramming frameworks . . . 23
- 4.2 Cytoscape js . . . 24
- 4.3 Ionic Framework . . . 25

**5 Usability, limitations, and use cases** **28**
- 5.1 Application usage . . . 28
  - 5.1.1 Application frame . . . 28
  - 5.1.2 Control elements in the canvas . . . 29
- 5.2 Limitations and known issues . . . 29
- 5.3 Comparison of a model in OPM and SysML . . . 29
- 5.4 Comparison of the project and existing OPM application . . . 34

**6 Conclusion** **36**

**Bibliography** **37**

**A All tested frameworks** **38**

**B Used Cytoscape extensions** **39**

**C Data model** **40**

**D Content of the attached media** **41**

Chapter 1

# Introduction

Modeling is broadly used in both science and engineering to provide abstractions of a system at some level of detail and precision. The model is analyzed in order to obtain a better understanding of the system being developed. As reported by the Object Modeling Group (OMG): „modeling is the designing of software applications before coding“[1]. In model-based software development, modeling is used as an essential part of the software development process. Models are built and analyzed before implementing the system and are used to manage the following implementation. In this work, we are looking at using models for both software and system development.

Let us start with two questions first. Why do we create a system model? What are the benefits we can gain? These questions need to be answered before the implementation of the editor could start. If we look at what vendors of existing modeling tools are saying, we will see declarations of how models help us with software development. Let us name a few of the central claims.

- Models help us to visualize a system as it is or as we want it to be.
- Models permit us to specify the structure or behavior of a system.
- Models give us a template that guides us in constructing a system.
- Models document the decisions we have made.
- Helps to understand complex systems part by part[2].

as a result of these attributes, model-based development should also

- Improve the understandability of the system.
- Ease the integration of new team members.
- Allows the decomposition and modularization of the system.
- Facilitates system evolution and maintenance.
- Enable the reuse of parts of the system in new projects[3].

[1] https://www.uml.org/what-is-uml.htm
[2] https://www.visual-paradigm.com/guide/uml-unified-modeling-language/why-uml-modeling/
[3] https://modeling-languages.com/list-supposed-benefits-software-modeling/

Although these promises of model-centered development are very plausible, they can carry obstacles along the way. Today, model-driven development is not common practice in many companies, or it has been done only partially. The main problems of creating large system models that may first come to our minds may be that:

- It requires people trained to use a specialized, complex modeling language.
- We need to have multiple diagrams to describe different parts of the system.
- We need to have professional tooling to create and maintain complex models.
- It consumes a significant amount of time that may be better invested in developing, fixing, and testing the existing system.

The thesis first discusses the usage of modeling tools in the field and concludes which feature users need the most and what are the reasons modeling approaches are not commonly adopted in many companies. Along with that, we discuss the advantages and disadvantages of current code-based development. The rest of the work is organized as follows. In the next section, we discussed several modeling languages, their strong and weak sides. In section three, we present the different graphical web frameworks and compare them against each other. Next, we discuss the vital parts of editor implementation and the main obstacles that the development faced. Section 4 presents the final project, provides the OPM model created in the implemented editor, and compares the same model modeled in the SysML modeling language. Finally, the last section summarizes the work done and discusses some ideas for follow-up works.

# Chapter 2

# Editor requirements

First, we look for references in the field and see which are the biggest challenges that modeling approaches meet. As there are some studies done on evaluating the effectiveness of modeling systems, we can use them to assess user needs and notice this information in designing the editor.

## 2.1 Modeling practices survey

There was a study [5] conducted in 2007 about the usage of models in software development. The survey aimed to reveal attitudes and experiences of software professionals about software modeling and develop approaches that avoid modeling. The study was motivated by observations that modeling is not widely adopted since many developers continue to take a code-centric approach. It consists of 113 software practitioners with an average of 14 years of experience. With demographics, about two-thirds of the respondents were from Canada or the United States, and the last third respondents worldwide.

Most participants work on business software, followed by design and engineering software and website content management.

Type of software that participants work on (in percentage):

- 46% business software
- 25% design engineering software
- 23% website content management
- 6% other

The sample included participants from management positions and developer posts, each performing at least some design or modeling activity.

Involved in leadership roles (team leader, project manager)

- 90% at least sometimes
- 53% very often or always

Work with source code (developing new code, maintaining and bug fixing)

- 86% at least sometimes
- 49% very often or always

Performing design or modeling

- 95%+ at least sometimes
- 57% do this very often or always

The survey has 18 questions, most of which involve different sub-questions with 5-point scales. The scale ranges from strongly disagree to strongly agree, or from never to always.

The first question is discussing the most common answers for creating models and how they were received, represented in Table 2.1.

The survey found out that creating models was the most frequent way of drawing or writing on a whiteboard [5]. The second most frequent was using diagramming tools and word-processing software and finally, very few used drawing software to maintain models. Note that respondents can agree on using more than one way or not responding to some option at all, so the sum of the percentage does not need to be 100%.

Table 2.1: Responses for Question: Way of creating models [5].

| Participants opinion | agree | disagree |
|---|---|---|
| Drawing or writing on a whiteboard | 45% | 33% |
| Diagramming tools | 37% | 42% |
| Word-processing software | 27% | 46% |
| Other | 22% - 30% | - |
| Drawing software | 13% | 72% |

The survey interestingly found that sources of receiving models are drastically different from the ways people create models. All responses are included in Table 2.2.

Table 2.2: Responses for Question: Source of design information [5].

| Frequency of usage | very often | never |
|---|---|---|
| Word of mouth | 55% | 24% |
| Word processors | 48% | - |
| Whiteboards/diagram tools | 42% | - |
| Fully-integrated modeling tools | 32% | 33% |
| Handwritten material | 20% | 24% |

As we can see, the first source of design information was word of mouth. The second most important source was material created in word processors and diagramming tools (that can create structured diagrams but not integrated models). The interesting observation was that fully integrated modeling tools were one of the least important source of the information material.

The primary usage of modeling tools was designing a software system or transcribing a design into a digital format [5].

Code generation from the model was not commonly used. It may be because participants do not need it or it was not done the way expected. Brainstorming design ideas are also not a very common activity done in modeling tools, mainly because whiteboards seem sufficient to provide the desired outcome.

The main uses of modeling tools (participants responses in percentage)

- 48% to develop the design of a software system
- 39% transcribe a design into a digital format
- 23% to brainstorm about possible design ideas and alternatives
- 18% to generate part of code
- 14% to generate all code

Now when we know how models are most often created and received, we can define more precisely what model-centric and code-centric approaches are.

## 2.2 Code-Centric and Model-Centric Approaches

The following section is discussing the differences between code and model-centric approaches highlighting their problems and advantages.

The model-centric approach uses the model to see the overall design, any change to the project is first performed in the model. In this approach, developers perform modeling as a primary activity, and code is being generated, or it is written strictly according to the model [5].

The code-centric approach focuses on code as the main source of understanding the design, and every change is done directly in code.

The survey asked the participant about their perceptions of which approach works best for various activities. The answers are presented in Table 2.3. Where N stands for the number of people who responded to the question, mean is a central value calculated from the weight of each response multiplied by its percentage and divide by range, and s.d. is standard deviation, a measure of how close responses are to the average value.

Table 2.3: Responses for Question: Tasks that are better in a model-centric versus code-centric approach [5].

| Available activities | N | mean | s.d. | % Much easier in Models (1) | % Easier in Models (1 + 2) | % Easier in Code (4 + 5) | % Much easier in Code (5) |
|---|---|---|---|---|---|---|---|
| Explaining a system to others | 92 | 1.7 | 1.1 | 61.1 | 81.8 | 7.6 | 6.5 |
| Comprehending a system's behavior | 89 | 2.0 | 1.3 | 51.7 | 71.9 | 15.7 | 5.6 |
| Creating a new system overall | 92 | 2.2 | 1.3 | 43.5 | 68.5 | 20.7 | 7.6 |
| Creating a re-usable system | 92 | 2.2 | 1.3 | 44.6 | 63.0 | 15.2 | 9.8 |
| Creating a system that most accurately meets requirements | 91 | 2.2 | 1.3 | 42.9 | 67.0 | 19.8 | 8.8 |
| Modifying a system when requirements change | 91 | 2.5 | 1.4 | 34.1 | 54.9 | 24.2 | 13.2 |
| Creating a usable system for end users | 92 | 2.7 | 1.3 | 26.1 | 42.4 | 22.8 | 10.9 |
| Creating a prototype | 92 | 2.9 | 1.5 | 26.7 | 43.0 | 32.6 | 22.8 |
| Creating a system as quickly as possible | 92 | 3.0 | 1.5 | 23.9 | 46.7 | 42.4 | 23.9 |
| Creating efficient software | 92 | 3.1 | 1.4 | 16.3 | 35.9 | 43.5 | 21.7 |
| Fixing a bug | 90 | 3.2 | 1.5 | 21.1 | 28.9 | 43.3 | 25.6 |

Note: Values range from Much easier in a model-centric approach (1), to much easier in a code-centric approach (5).

The answers indicate that most activities are performed easier in a model-centric approach. There was no prejudice about the model-centric approach. Also, tasks such as creating efficient software were examined by nearly half of the respondents as achievable.

### 2.2.1 Problems with the model-centric approach

The biggest problem of model-centric approaches is keeping the model updated with the code [5]. Also, it can be because participants did not want to generate code from models so much. The results highlighting the problems with a model-centric approach are presented in Table 2.4.

Table 2.4: Responses for Questions: Problems with a model-centric approach [5].

| **Potencial problems** | **N** | **mean** | % Strongly Disagree (1) | % Disagree (1+2) | % Agree (4+5) | % Strongly agree (5) |
|---|---|---|---|---|---|---|
| Models become out of date and inconsistent with code | 92 | 3.8 | 7.6 | 16.3 | 68.5 | 37.0 |
| Models cannot be easily exchanged between tools | 91 | 3.3 | 15.4 | 26.4 | 51.6 | 17.6 |
| Modeling tools are 'heavyweight' (to install, learn, configure, use) | 92 | 3.1 | 10.9 | 31.5 | 39.1 | 12.0 |
| Code generated from a modeling tool not of the kind I would like | 91 | 3.0 | 18.7 | 39.6 | 38.5 | 16.5 |
| You cannot describe the kinds of details that need to be implemented | 89 | 2.8 | 23.6 | 43.8 | 36.0 | 7.9 |
| Creating and editing a model is slow | 92 | 2.7 | 17.4 | 43.5 | 22.8 | 12.0 |
| Modeling tools change, models become obsolete | 92 | 2.7 | 22.8 | 44.6 | 32.6 | 5.4 |
| Modeling tools lack features I need or want | 89 | 2.6 | 19.1 | 44.9 | 21.3 | 5.6 |
| Modeling tools hide too many details that would be visible in the source code | 92 | 2.6 | 19.6 | 44.6 | 23.9 | 1.1 |
| Modeling tools are too expensive | 90 | 2.6 | 26.7 | 46.7 | 26.7 | 6.7 |
| Modeling tools do not allow be to analyze my design in ways I would want | 90 | 2.5 | 28.9 | 51.1 | 25.6 | 6.7 |
| Organization culture does not like modeling | 92 | 2.5 | 31.5 | 48.9 | 23.9 | 4.3 |
| Semantics of models different from those of programming Languages used for implementation | 90 | 2.4 | 31.1 | 56.7 | 23.3 | 8.9 |
| Modeling languages are not expressive enough | 91 | 2.4 | 28.6 | 54.9 | 17.6 | 2.2 |
| Modeling language hard to understand | 91 | 2.2 | 28.6 | 62.6 | 9.9 | 3.3 |
| Have had bad experiences with modeling | 91 | 2.2 | 39.6 | 63.7 | 16.5 | 6.6 |
| Do not trust companies will continue to support their tools | 89 | 2.0 | 44.9 | 67.4 | 10.1 | 0.0 |
| Note. Values range from Not a problem (1), to Terrible problem (5). | | | | | | |

The study also identified that UML was the dominant modeling language with 52% usage among the survey participants. Also, the difficulty of modeling languages is not the limiting factor to adopting model-centric since they were considered easy to understand.

The authors also pointed out that text and source code has a longer lifespan and that it will be a good idea to explore better ways to render models in a textual manner that is human editable. This option is based on the fact that 34% of participants felt that a model's underlying storage format would become obsolete.

### 2.2.2 Problems with the code-centric approach

Participants were asked about problems involved with code-centric approaches to software development. All the results are included in table 2.5.

Table 2.5: Responses for Questions: Problems with a code-centric approach [5].

| **Potencial problems** | **N** | **mean** | **% Strongly Dis-agree (1)** | **% Dis-agree (1 + 2)** | **% Agree (4 + 5)** | **% Strongly Agree (5)** |
|---|---|---|---|---|---|---|
| Hard to see overall design | 94 | 3.8 | 4.3 | 13.8 | 66.0 | 35.1 |
| Hard to understand behavior of system | 94 | 3.6 | 4.3 | 19.1 | 60.6 | 21.3 |
| Code becomes of poorer quality over time | 92 | 3.4 | 9.8 | 28.3 | 55.4 | 25.0 |
| Too difficult to restructure system when needed | 93 | 3.4 | 8.6 | 22.6 | 51.6 | 17.2 |
| Difficult to change code without adding bugs | 93 | 3.4 | 9.7 | 22.6 | 50.5 | 18.3 |
| Changing code takes too much time | 94 | 2.8 | 20.2 | 39.4 | 27.7 | 8.5 |
| Our programming language leads to complex code | 94 | 2.5 | 26.6 | 51.1 | 20.2 | 8.5 |
| More skill is required than is available to develop high quality code | 91 | 2.5 | 29.7 | 53.8 | 22.0 | 6.6 |
| Programming languages are not expressive enough | 91 | 2.1 | 46.2 | 64.8 | 14.3 | 5.5 |
| Organization culture does not like the code-centric approach | 92 | 1.9 | 58.7 | 72.8 | 14.1 | 4.3 |
| Our programming language is likely to become obsolete | 93 | 1.9 | 51.6 | 75.3 | 9.7 | 3.2 |
| Note. Values range from Not a problem (1), to Terrible problem (5). | | | | | | |

Participants experience that code-centric approaches fail to deliver a high-level view of the system, and over time expect the situation only gets worse. Programming languages, in contrast to modeling tools, are not likely to become obsolete. The participants were divided as to whether or not changing the code takes too much time.

We see that both approaches have their own problems and advantages. While there is clear evidence that we can benefit from both, the synchronization and maintenance of both approaches together is a pressing issue. The survey conclusion and more detailed analysis from the survey authors are presented in the next section.

## 2.3 Summary

The study concluded that most participants have an expansive view of what modeling is. They consider informal material such as hand-drawn diagrams to be a model. However, over a third of participants never use formal modeling tools. The dominant modeling language is UML but rarely used differently than as an information carrier. The central use of modeling tools is to create documentation of the system or transform the design into a digital format and are rarely used to generate code [5].

According to the authors of the study, model-centric tools may benefit from features that help to better:

- „Synchronize code and models to reduce inconsistencies.
- Provide better traceability between models and code to help identify relationships among code and model artifacts to help indicate aspects of models that may require maintenance should the code change (and vice-versa).
- Provide better modeling capabilities and expressions within the programming code to reduce the need for external and disjoint modeling artifacts. “ (Forward, Andrew and Lethbridge, Timothy, 2008, page 7 [5])

Participants believe that the model-centric approach has multiple advantages, and they also have a desire to incorporate more modeling into their workflow, but it is hardly achievable since most of the participants work in a code-centric environment.

Now that we have expertise from the field, we can continue and find formal modeling language that will suit these requirements the best.

#### Chapter 3

# Comparison of standard modeling languages

This chapter discusses different modeling languages (ML) that are used today. The goal is to determine the most suitable ML for system description. The most well-known formalisms for system modeling are UML, SysML, and OPM, and they are the main focus of our research. We look at several basic parameters. We discuss basic units, the type of diagrams, and complexity management for each language.

Diagrams essentially present how compact the language is, and it can make a possible estimation of the learning curve to handle the whole language.

Complexity management is an essential part of the modeling language. If we try to incorporate all the details into one diagram, the amount of drawn symbols gets very large, and their interconnections quickly become a complicated web. Furthermore, the whole model structure will be hard to follow. So a system modeling languages include their integral mechanisms for controlling and managing its complexity, such as presenting and viewing the system at various levels of detail. Now we can start with an analysis of the most common language - UML.

## 3.1 Unified Modeling Language

Unified Modeling Language (UML) is a general-purpose modeling language in software engineering that is intended to provide a standard way to visualize the design of a system. UML has 14 diagram types. The current version is 2.5. It is mainly used in modeling business processes and in software to analyze and design software implementation [1].

### Diagram types

UML diagrams can be divided into two main groups Structure and Behavior diagrams. Furthermore, the Behavior diagram group also includes Interaction diagrams.

As the name indicates, structure diagrams describe static application structure. These include six types: Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.

Following group behavior diagrams describe general behavior of a system and are three of them: Use Case Diagram, Activity Diagram, and State Machine Diagram and also included four that represent the aspect of interactions: Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram. For a better understanding of UML diagrams structure, they are all presented in Figure 3.2.

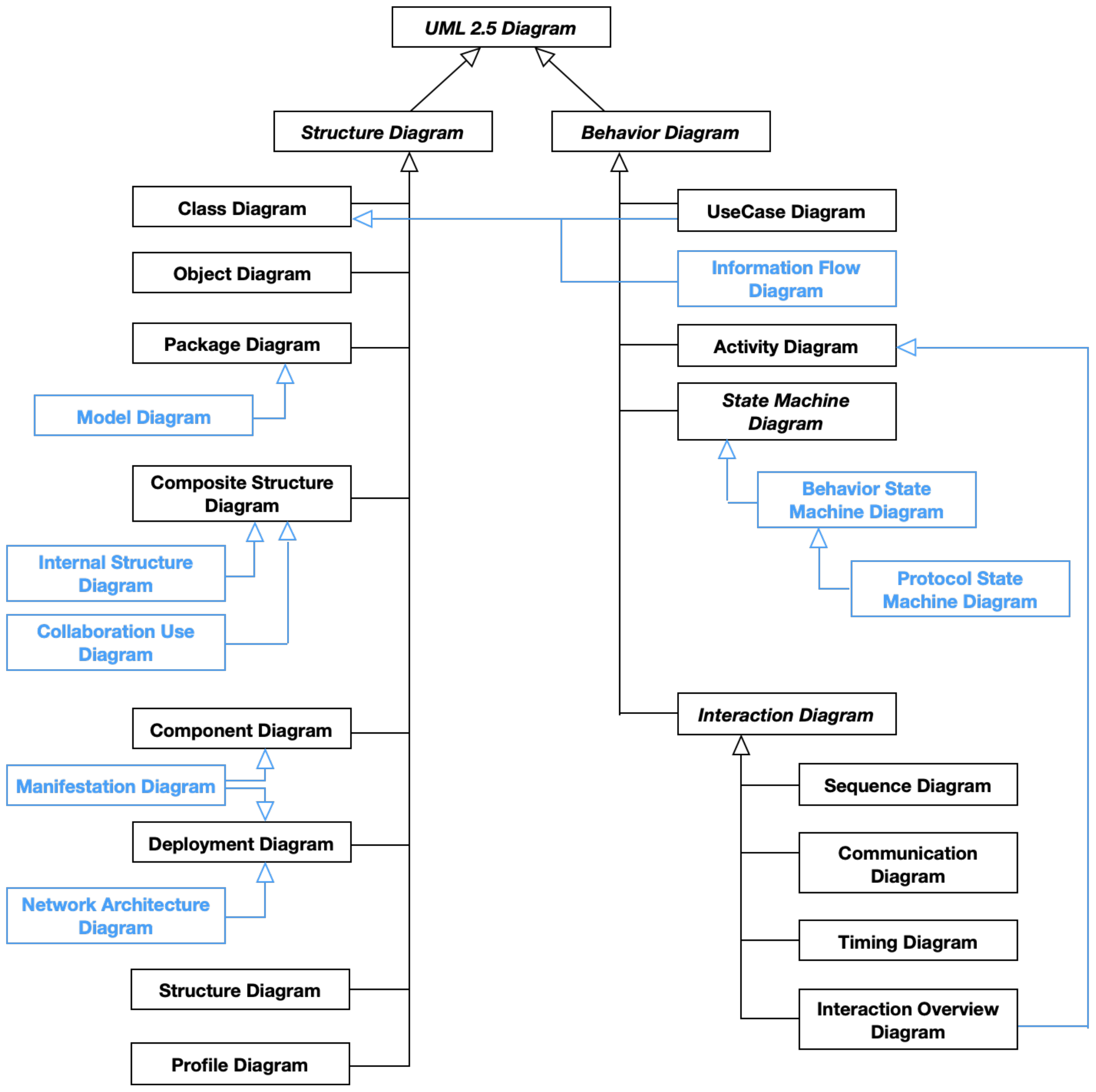


Figure 3.1: UML 2.5 diagrams[1].

Note that the items in Figure 3.1 shown in blue are not part of the official UML 2.5 taxonomy of diagrams.

UML has various diagram types suited for specific modeling needs, thus missing one underlying diagram that connects them. The most significant advantage of UML is that it is very well known and has a large ecosystem of modeling tools.

[1]diagram taken from: https://www.uml-diagrams.org/uml-25-diagrams.html

#### Complexity management

Complexity management evolved mainly from UML version 2.0, where the new version can nest models inside the component that manages them. In other words, nearly every building block is a classifier, including classes, objects, components, and behaviors such as state machines and others, and those classifiers can be nested on to each other. This feature allows building complex structures and behaviors with interactions that are described sideways in Interaction Overview Diagram.

That level of complexity gives a choice to present an abstraction of the system in various ways. For example, UML 2 allows modeling a state machine inside the class that implements it. However, with variability comes the cost of an increased effort to keep all components synchronized and the need for at least one extra diagram to describe integrations between all layers and diagram types, making it vulnerable to become inconsistent over time.

Some other of UML's criticism is that the language is too „software-centric“ so it can be harder to describe other than a software type of system[2]. Since we had described all critical parts of UML, let us move into examining its descendant SysML.

## 3.2 Systems Modeling Language

Systems Modeling Language (SysML) is a general-purpose graphical modeling language and represents an extension of a subset of UML 2. The portion of which SysML has in common with his ancestor is graphically represented in Figure 3.2. The current version is 1.6[3], and version 2.0 is coming in the fall of this year.

The language is used for specifying, analyzing, designing, and verifying complex systems that include hardware, software, information, personnel, procedures, and facilities.

SysML leverages the OMG XML Metadata Interchange (XMI®) to exchange modeling data between tools intended to be compatible with the evolving ISO 10303-233 systems engineering data interchange standard.

[2]mentioned in MITOPENCOURCEWARE Session 3: Systems Modeling Languages course, transcript availeble: https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-842-fundamentals-of-systems-engineering-fall-2015/class-videos/session-3-systems-modeling-languages/CTVFDb44ses.pdf search „software centric“

[3]https://sysml.org/sysml-faq/what-is-current-version-of-sysml.html

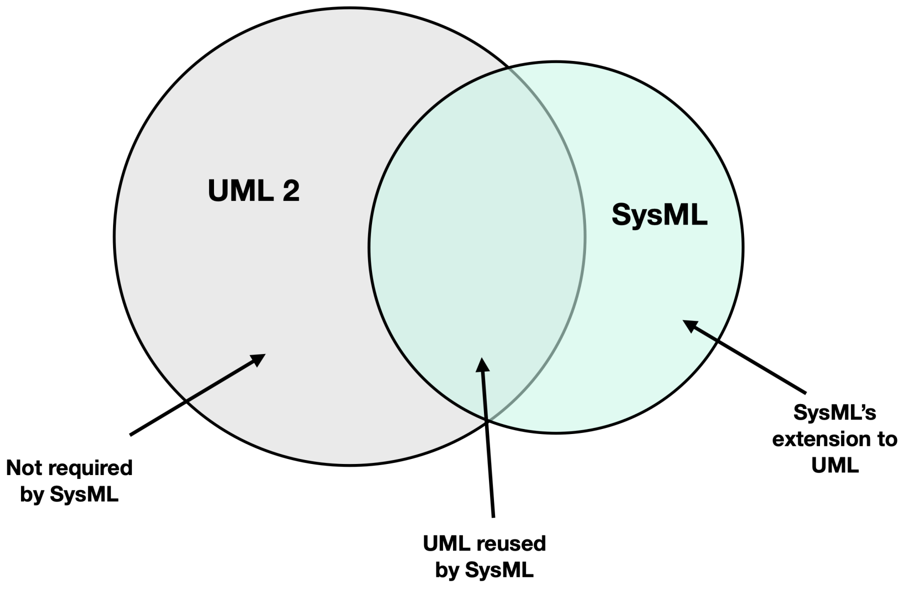


Figure 3.2: UML SysML relation [3].

### Basic unit

The block is the basic unit of structure in SysML and can represent hardware, software, facilities, personnel, or any other system element [3].

### Diagram types

SysML has 9 diagram types with four Pillars. Two pillars are the same as in UML Behavior and Structure and two more Requirements and Parametrics. From UML, SysML reuses four diagrams, namely: Sequence, State machine, and Use case diagrams. Three diagrams are taken and modified from UML: activity, block definition, and internal block diagrams. And two completely new types: Parametric and Requirement diagrams. SysML diagram structure is illustrated in Figure 3.3.

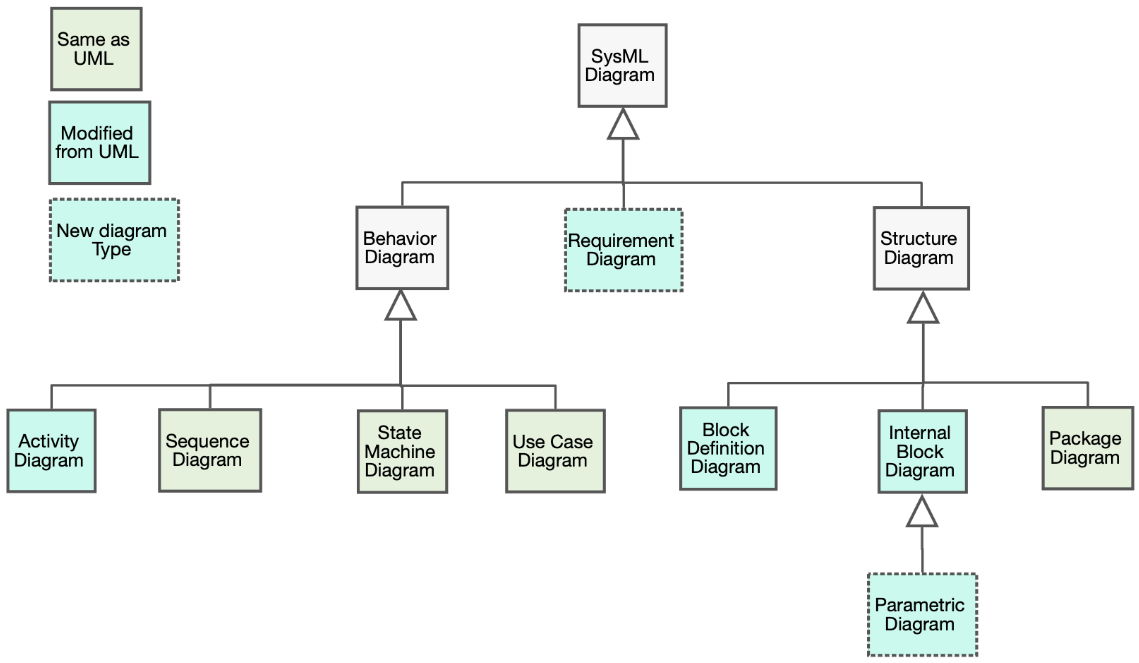


Figure 3.3: SysML Diagram types [3].

Plus, SysML also has an allocation with associates or maps model elements of different types or in different hierarchies. They enable traceability of components which is a crucial part of model-based development in general. Allocate Dependency patterns are generally helpful in improving model architecture integrity and consistency. These allocation relationships can dynamically create allocation tables to summarize connections in the model [2].

#### Complexity management

SysML's idea is to provide us with language capable of describing a system at multiple levels and describing systems further from the software domain. SysML provides relationships for Cross Connecting Model Elements, for example, mentioned allocation relations. Also, in contrast to UML, SysML model management constructs support models, views, and viewpoints.

SysML shows an effort to connect pieces of models together in order to see the picture of the whole system in fewer diagrams. There is also an idea to start supporting multiple levels of the system at different layers, which increases the flexibility of presenting the complex and extensive system in a more accessible manner.

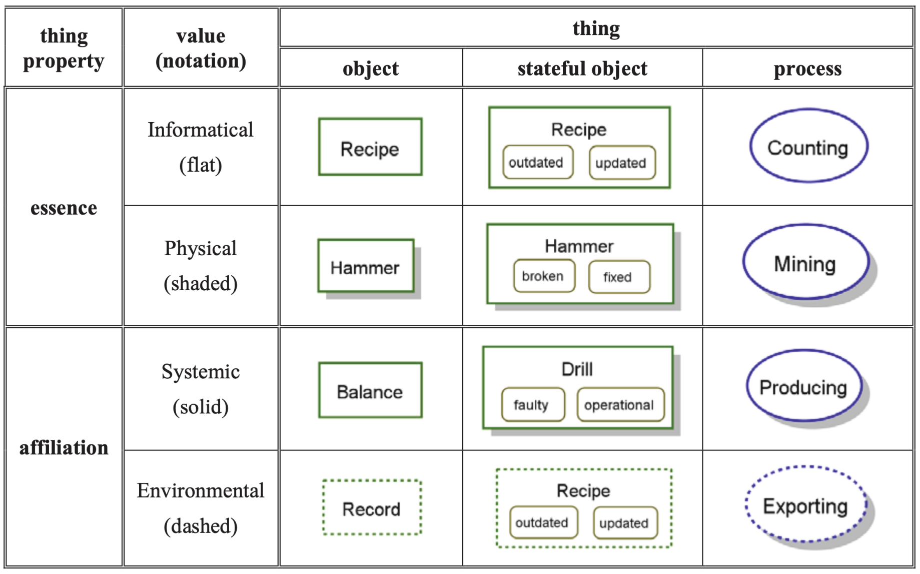
Finally, let us present the last discussed language - OPM.

## 3.3 Object Process Methodology

Object Process Methodology (OPM) is a conceptual modeling language and methodology for capturing knowledge and designing systems, specified as ISO/PAS 19450. OPM is designed to be able to model any system no matter the domain. In contrast to UML-like languages, OPM does not separate behavior and structure in different diagrams. Another cornerstone of OPM is its bi-modal graphical-textual representation called Object Process Language (OPL). It is generated by each connection in Object Process Diagram (OPD) and is translated to a subset of natural English.

## Basic unit

OPM builds on a minimal set of concepts. The fundamental element in the diagram is not a block but a „thing“ representing either process or object. Attributes to a stateful object are called „states.“ Things are also distinguished by their physical or informatical existence. All possible types of „things“ are presented in figure 3.4



| thing property | value (notation) | thing | | |
|---|---|---|---|---|
| | | object | stateful object | process |
| essence | Informatical (flat) | Recipe | Recipe: outdated, updated | Counting |
| | Physical (shaded) | Hammer | Hammer: broken, fixed | Mining |
| affiliation | Systemic (solid) | Balance | Drill: faulty, operational | Producing |
| | Environmental (dashed) | Record | Recipe: outdated, updated | Exporting |

Figure 3.4: OPM Things (Dori, 2008, page xxi [4])

## Diagram types

OPM has one kind of diagram called Object Process Diagram (OPD), and it is the only kind of diagram of OPM. The primary promise of OPM is that it can show everything in one diagram type, so the functions, the functional attributes, the objects, different types of objects, operands, system components, consumables, the attributes of those objects, and then the links.

The author of OPM made a point of the simplicity of language that is worth mentioning; he states: „We cannot do much about the inherent complexity of the system, but by using a simple modeling framework, we can significantly reduce the system’s complicatedness.“(Dori, 2008, page 295 [4])

## Complexity management

OPM has mechanisms to handle the complexity of the model, and it is based on managing multiple layers represented by a set of hierarchically organized OPDs.

To move between the model layer, in/out-zooming and folding methods are used. With these, the OPM can conceptually model systems at any level of complexity with no limited number of nested levels.

Whenever an OPD becomes hard to comprehend due to an excessive amount of details, a new descendant OPD needs to be created. The determination of when an OPD becomes too complex is left to the responsibility of the modeler because it is hard to define by static references such as the number of model elements or other characteristics.

Now as we presented the languages let us move to their comparison.

## 3.4 Modeling languages comparison

In this section, we compare the three modeling languages discussed in the previous sections. Table 3.1. lists the parameters of each of the ML's. As this work focuses on the modeling aspect of these languages, the most important properties to look at are the theoretical foundations, support for decomposition of complex models, learning complexity, and existing tooling support. These properties are discussed in more detail in the rest of this section.

Table 3.1: UML vs SysML vs OPM[4]

| Feature | UML | SysML | OPM |
|---|---|---|---|
| Theoretical foundation | UML; Object-Oriented paradigm | UML; Object-Oriented paradigm | Minimal universal ontology; Object-Process Theorem |
| Standard documentation number of pages | 1400 =700 (UML Infrastructure) + 700 (UML Superstructure) | ~1670 =UML + 270 (OMG SysML) | ~180 =100 (ISO 19450 main standard) + 80 (append) |
| Standardization body | OMG and ISO | OMG | ISO |
| Number of diagram | 14 | 9 | 1 |
| Top-level concept | Block (UML object class) | Block (UML object class) | Thing (object or process) |
| Complexity management guiding | Aspect-based decomposition | Aspect-based decomposition | Detail-level-based decomposition |
| Number of symbols | Many | ~120 | ~20 |
| Graphic modality | Yes | Yes | Yes |
| Textual modality | No | No | Yes |
| Built-in physical-informatical distinction | Yes | Yes | Yes |
| Systemic-environmental distinction | Partial (using boundaries) | Partial (using boundaries) | Yes |
| Logical relations (OR, XOR, AND) | No | No | Yes |
| Probability modeling | No | No | Yes |
| Execution, animated simulation, validation and verification capability | Partial (in some tools for some diagram kinds) | Partial (in some tools for some diagram kinds) | Yes |
| Tool availability | many | many | 1 limited in functionality |
| Registered in | 1997 In 2005 by ISO | September 2007 | December 2015 by ISO |

### The object-oriented paradigm vs the minimal universal ontology

The first significant difference is the difference between the Object-oriented paradigm (OOP) and minimal universal ontology (MUO). Let us describe what these two approaches are and how they differ.

Ontology is a set of concepts and their relations in some domain of debate [4]. The Minimal Ontology principle states that if a system can be specified at the same level of accuracy by two languages of different ontology sizes, then the language with the smaller size is preferred over the other. Minimal universal ontology is formed by stateful objects,

[4]modified table from (Dori, 2008, p. 297 [4]), added UML column

processes, and relations among them and can conceptually model any system in any domain. Therefore minimal universal ontology means OPM can be used to describe itself and not only itself, but it is sufficient to model the universe and systems in it.

Object-oriented paradigm (OOP), on the other hand, is closely related to Object-oriented programming and software development. It puts an object as a center of attention and represents a real-world element in an object-oriented environment that may have a physical or a conceptual existence. In the OOP view, every system is described by its objects. We can say that in the ability to model different systems, SyML is able to describe only a subdomain of what can be described with OPM.

Now, when we defined the terms, we can highlight differences. Objects and processes are the two types of OPM's universal building blocks, and processes are modeled as equals that are not inferior to objects. This object-process orientation is a primary difference from the object-oriented (OO) software paradigm, which places objects as the only dominant players.

The table is also pointing to top-level concept differences. In OOP, processes are referred to as methods or services and can not exist without the object itself. In OPM, system-level processes are as important as the objects in the system; thus, they are independent of objects and can be modeled separately.

### Aspect and detail based decompositions

Another significant difference is in the decomposition of complexity. UML and SysML address the problem of managing systems complexity primarily by aspect decomposition - dividing the system model into 14 (UML) and 9 (SysML) different diagram types for modeling various aspects of the system – structure, dynamics, state transitions, timing, and others [4].

The OPM approach is orthogonal, detail-based decomposition: Rather than applying a separate model for each system aspect, OPM handles the internal system complexity by decomposing the system into a hierarchy of diagrams of the same kind. The entire system is completely specified through its Object Process Diagram (OPD) set, and each of the diagrams provides a partial view of the system, which together provide a complete picture of the system.

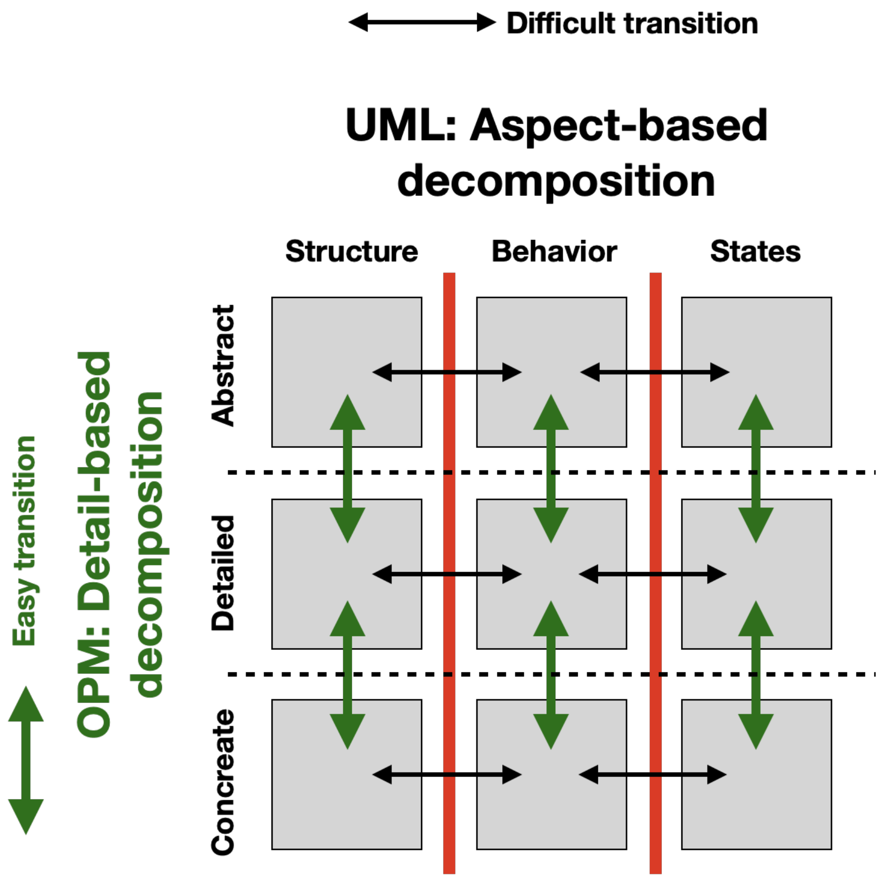


Figure 3.5: Decomposition (Dori, 2008, p. 297 [4])

Figure 3.5 shows the two orthogonal complexity management strategies. In the aspect-based decomposition, two solid vertical lines separate the structure, behavior, and state transition aspects. The thin bidirectional horizontal arrows across these lines symbolize difficult transition among the various models. The detail-based decomposition is represented by the two dashed horizontal lines separating the various levels of detail from abstract to detailed and concrete. The green bidirectional vertical arrows symbolize easy transition between the detail levels. The diagram is schematic, and the number of transitions, levels, or diagram types does not matter.

**Learning difficulty**

As we mentioned, the time and resources required to create a valid system model are crucial for system modeling. Suppose we do a simulation of resources needed to integrate each ML to a production flow properly.

When we look at the second and third rows in the table, we can deduct the time needed to master each language. The learning curve of each language depends on how long the specification is and how complex the language is. To have full knowledge of a UML, we would need multiple qualification courses with a few years of expertise in the field. About SysML, let us assume a few months of training can be enough. OMP, with its short and straightforward specification, we will be okay with one book and possibly a few weeks to learn all specifics of the language. OPM also provide automaticaly generated description of model. This allows people not knowing the notation (language) to understand the diagrams.

### Tooling

If we look at tool availability, UML and SysML are usually covered by one tool for complete UML tooling. However, these tools are huge, complex, and expensive pieces of software. The good news is that we can choose a relatively large base of software tools. On the other hand, only one editor is available for OPM, which can be enough if it will provide the satisfactory features required. Unfortunately, from practical experience, the current tool for OPM is not optimal, which is also the motivation and goal of this work to provide a concept for a better solution.

# Chapter 4

# Implementation

The technical decision was to make the editor web-based. The advantage to this approach is that the application does not require a specific operating system to run, and the only required software is a web browser. There is no need to install extensive desktop tools, and so the application is more easily accessible to users.

The languages chosen to write the application were javascript and typescript mainly because of the well-extended graphical and UI libraries.

Frameworks, especially graphical ones, play an important role and are core to providing the editor's main feature. Choosing the suitable framework was done by examining predefined parameters mentioned in the next section, testing available demonstration examples, carefully studying each framework documentation, and looking at GitHub parameters regarding the community.

The project structure is divided into two main graphical parts, canvas and frame of the application. Canvas has a primary role in manipulating the diagram and frame components in managing the entire project.

## 4.1 Web diagramming frameworks

For choosing the proper graphical framework, there was the need to define critical properties that such a framework must-have. The first base functionality was interactive nodes along with the ability to connect elements together. The following crucial features were text and position saving and also the performance of all mentioned operations. Other features were regarded as welcoming benefits of the framework, such as grouping, nesting, undo-redo features, and context menus. All features together and a comparison of different libraries are listed in Table 4.1.

Other not graphical related parameters play a role as well. Parameters such as if the project is well documented, still supported, provide enough helpful demonstration examples, and community involvement. Also, an essential requirement is that the selected framework must be open-source.

Table 4.1: Framework Comparison

| | Cytoscape | D3 | jgraph (Draw.io) | Draw2D | Sprotty |
|---|---|---|---|---|---|
| Interactivity | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |
| TextWrite | 🟨 | 🟨 | 🟩 | 🟩 | 🟨 |
| Connecting | 🟩 | 🟥 | 🟩 | 🟩 | 🟥 |
| Saving position | 🟩 | 🟨 | 🟨 | 🟥 | 🟨 |
| Saving text | 🟨 | 🟨 | 🟨 | 🟥 | 🟨 |
| Resizing | 🟥 | 🟥 | 🟩 | 🟩 | 🟥 |
| MultipleSelection | 🟩 | 🟥 | 🟩 | 🟩 | 🟥 |
| Grouping/Nesting | 🟩 | 🟩 | 🟩 | 🟩 | 🟥 |
| Expansion/Folding | 🟩 | 🟩 | 🟥 | 🟥 | 🟥 |
| Visibility | 🟩 | 🟩 | 🟥 | 🟥 | 🟥 |
| Undo/Redo | 🟩 | 🟥 | 🟩 | 🟩 | 🟥 |
| Contex menu | 🟩 | 🟩 | 🟩 | 🟩 | 🟥 |
| Active project | yes | yes | no | yes | yes |
| Documentation | great | great | medium | medium | poor |
| Commits: | 5200+ | 4000+ | - | 202 | 237 |
| Used by: | 3500+ | 204k | - | 62 | 100 |
| Contributors: | 80+ | 128 | - | 6 | 14 |
| Stars: (Community) | 7k | 96k | - | 509 | 314 |

The green background color in the table indicates that the mentioned framework provides such a feature, yellow means it can be done with some workaround, and red means it is not included in the framework, and it is possibly hard to include them. Note that other tested frameworks failed to provide at least four working features, so they are not included in the table. A complete list of all tested frameworks can be found in appendix A.

As the table indicates, the best score has Cytoscape js framework. It was also chosen for its modular nature and large community.

## 4.2 Cytoscape js

Cytoscape.js is an open-source JavaScript-based graph library [6]. It is most often used as a visualization software component and to render interactive graphs in a web browser. Nevertheless, it also contains a graph theory model, and it is used for graph analysis and visualization of relational data, like biological data or social networks. As analysis is an integral part of model-based development, this focus of the library fits our needs as well.

In our case, the Cytoscape canvas is responsible for diagram manipulation. Most of the essential operations are done by a special circle context menu adapted to be used in touchscreens.

For functionalities like connecting diagrams, grouping, and collapsing are used different Cytoscape extensions listed in appendix B. The most challenging part of the implementation was to find a proper way to integrate them together and implement the missing functions.

### Missing functionalities

Text editing was the first bottleneck of the framework, although it was finally solved by adding an Html element (via a library named popper js[1]) near the canvas node and transferring text into the internal data structure.

With regards to styling, element Cytoscape provides nearly all CSS-like properties to style elements inside the canvas. However, when it comes to providing an elliptic style of so-called 'parent compound nodes,' it fails to include all child elements inside the boundaries of the ellipse. It is because the bounding box is always rectangular. The solution to this problem was to recalculate the ellipse diameter and rewrite it in code directly.

To calculate the proper size of the ellipse, we need to know three points, center, and two on the circumference. We also need to ellipse to retrace the rectangle. We take two edge points of the rectangle, set them as circumference points, and set the center 0,0.
Taking an equation of ellipse:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{4.1}$$

with folding points taken from rectangle where a1, b1 are coordinates of a right-up border and -a1, b1 are coordinates of a left-up border:

$$S[0,0], A[a_1, b_1], D[-a_1, b_1] \tag{4.2}$$

we calculate the a and b using the following equations where a and b represent each point co-ordinates of the ellipse border:

$$a = \sqrt{\frac{-{a_1}^2 * 2b_1^2}{b_1^2 - 2b_1^2}} \tag{4.3}$$

$$b = \sqrt{2}b_1 \tag{4.4}$$

Another missing functionality was resizing elements. The only way of modifying the size allowed in the framework is at the parent compound node, which is resized according to its children. The solution to this problem was to append two control nodes that serve as a size modifier of the parent element and are shown only with cursor selection. Sizing non-compound elements was not necessary because its size is calculated according to its actual label content.

[1] https://popper.js.org/

## 4.3 Ionic Framework

All other components are written with typescript and with the use of react and ionic components. The reason for choosing ionic was that it provides tooling for creating a hybrid-web application. Ionic framework is an open-source mobile UI toolkit for building cross-platform interfaces[2]. Hybrid-web applications blend native and web solutions where the core of the application is written using web technologies[3].

There is starting to be a standard for modern applications to be available across all platforms, including phones, tablets, desktop applications, and web applications. There are basically ways to provide applications available at all platforms, either develop a native app on each platform or use hybrid solutions. Both ways have their advantages. The main difference is the time and resources needed to develop software per platform. In our case, going to develop a web application, we decided to have the ability to extend it across platforms only with web technologies.

In short, the possible ways to develop software are summarized in table 4.2 and also underlines that the solution with minimal effort to achieve cross-platform compatibility is via hybrid web-based technologies.

[2] https://ionicframework.com/
[3] https://ionic.io/resources/articles/what-is-hybrid-app-development

Table 4.2: cross-platform framework options [9]

| | Native | Hybrid-Native | Hybrid-Web |
|---|---|---|---|
| Examples | iOS and Android SDKs | React Native, Xamarin, NativeScript, Flutter | Ionic |
| Languages | Obj-C, Swift, Java | JS + Custom UI Language / Interpreter | HTML + CSS + JS |
| Code Reuse | Totally Separate Code Bases per Platform | Shared Business Logic with Different UI Codebases | One codebase, UI codebase stays the same |
| Target Platforms | iOS & Android Native Mobile Apps | iOS & Android Native Mobile Apps | iOS, Android, Electron, Mobil and Desktop Browsers as a Progressive Web App, and anywhere else the web runs |
| Investment | Largest investment in staff and time | Medium investment in staff and time | Lowest investment in staff and time |
| UI Elements | Native UI independent to each platform | A selection of Native UI elements for iOS and Android UI elements are specific to the target platform and not shared Custom UI elements begin to require split UI code bases | Web UI elements that are shared across any platform, conforming to the native look & feel of wherever they are deployed |
| API Access / Native Features | Separate Native API & Codebases for each App | Abstracted Single-Codebase Native Access through Plugins (with ability to write custom Plugins) | Abstracted Single-Codebase Native Access through Plugins (with ability to write custom Plugins) |
| Offline Access | Available | Available | Available |
| Performance | Native Performance with well written code. | Indistinguishable difference on modern devices with well written code. | Indistinguishable difference on modern devices with well written code. |

Currently, the application is now available only on a web platform. Making the app available on other platforms is planned as future work.

Chapter 5

# Usability, limitations, and use cases

## 5.1 Application usage

In these sections, we will discuss editor controls and the features they provide. The final look of the application is presented in 5.1. figure. The design of the application is focusing on simplicity and a clean white look.

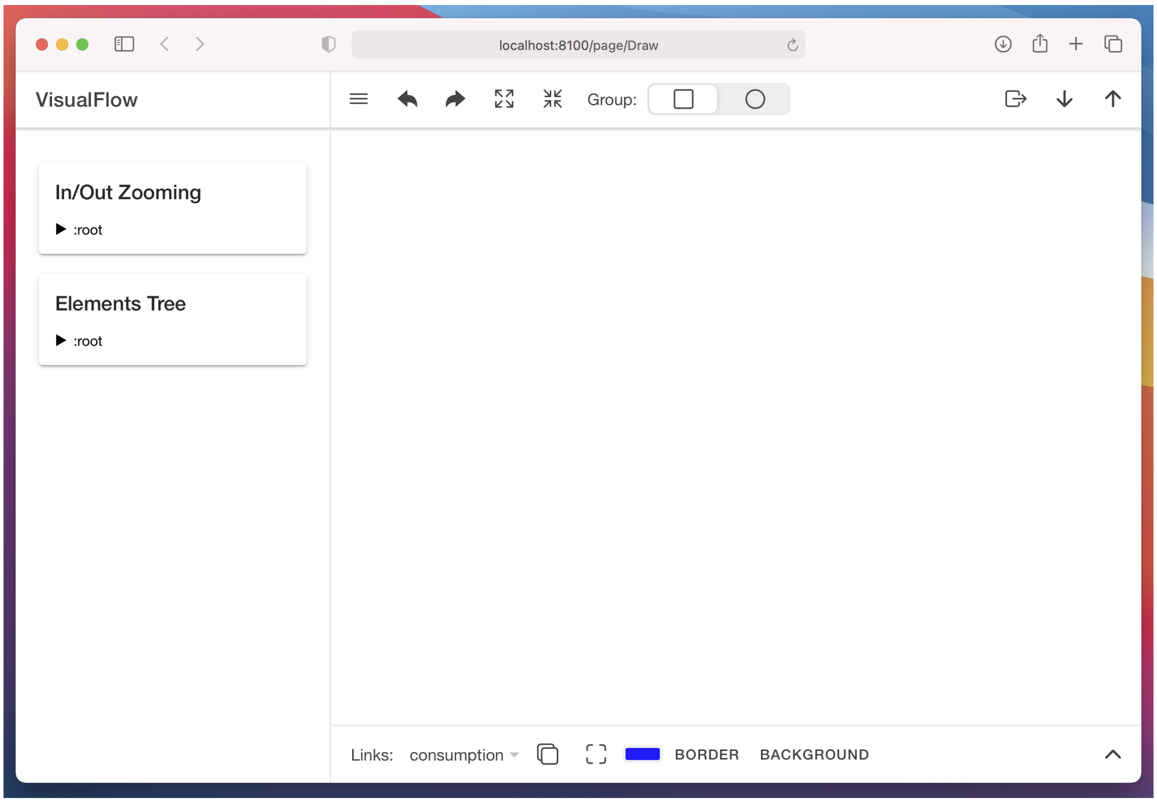


Figure 5.1: VisualFlow application design

### 5.1.1 Application frame

The application frame presents all components outside of the canvas - which is in the application center. This section describes only features present in the frame.

The editor supports folding and unfolding of nodes done by selecting the parent and clicking at two buttons in the left corner of the upper bar. The editor also supports importing and exporting the project in JSON format, exporting SVG pictures, and handling the generation of OPL. Generated text appears dynamically at the bottom text field and is triggered by a creating link connection. The OPL section also highlights elements when we hover over the text. At the bottom bar, there are controls used to style elements properties like color, shadow, and dotted borders.

The editor also has in/out zooming inside the left menu. The menu contains a representation of nested structures as a simple folding list. After a new level of the hierarchy is created, a new element in the list is dynamically created and removed when the element is deleted.

### 5.1.2 Control elements in the canvas

As we mention, the canvas is directly in the application center, and its primary is controlled by the left click of a mouse or tapping with two fingers on the touchpad to show the context menu. There are three main control elements in the application at the canvas.

The first control element is the context menu, with the ability to create and delete nodes and update text. There is support for nesting from upside-down via inserting an element into an object or process and composing from the bottom up via creating groups. There is also copy-paste functionality inside a context menu that copies and pasted all previously selected nodes.

The second control is the edge handle in the form of a little dot created on hover when the cursor passes over the element that can connect nodes with a selected link.

Third, there are two small squares shown when the parent element is selected. Their function is to control parent element size.

## 5.2 Limitations and known issues

The first jet unsolved issue is that the arrow selection closes the sidebar and disables expansion of the list. It is due to the bug in the ionic menu component. The workaround to this is to repeat the selection of link type, and the list should function normally.

The next known issue is that export is not working when nodes are collapsed. It is due to the now solved bug in one of the Cytoscape extensions, and the bug will be fixed when the solution is propagated to a stable branch of extension. A temporary solution is to expand all nodes by clicking on the „:root“ element on the second list, which will expand all nodes before exporting.

## 5.3 Comparison of a model in OPM and SysML

This section compares two modeling languages (SysML and OPM) by modeling a real-life system - a distiller.

#### Distiller problem

A commonly used example of the modeled system is to model the water distiller process [8]. A distiller is a system for purifying dirty water. It consists of three fundamental components Counter Flow Heat Exchanger, Boiler, and Drain.

The whole process starts with heating the dirty water in the Heat Exchanger, and then the dirty water boils in the Boiler, and the steam is condensed. Lastly, the residue is drained by the Drain.

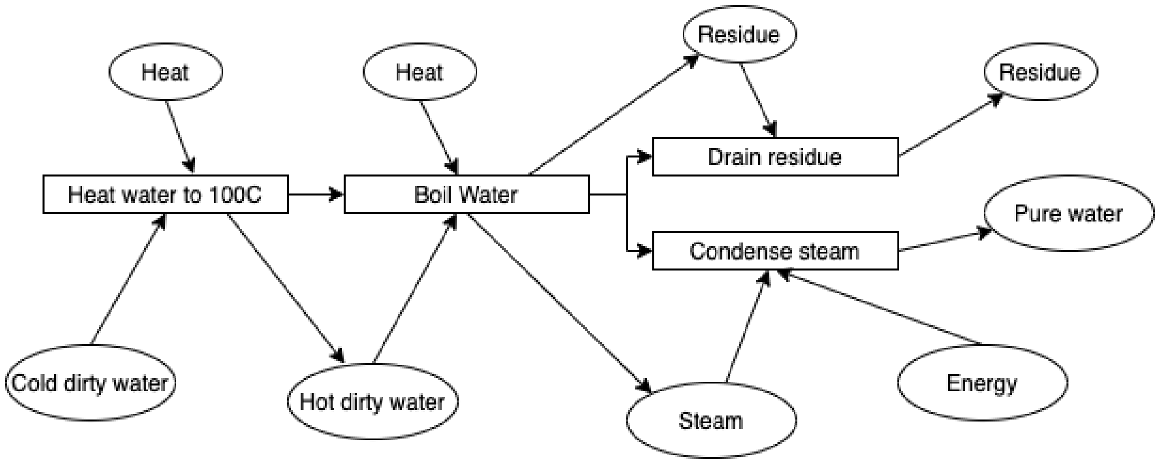


Figure 5.2: Distiller behavior diagram

Figure 5.2 displays a simplified behavior diagram of this distiller. The rectangles present system processes, while the ellipses present process inputs and outputs.

### Creating the distiller model with SysML

SysML modeling is a three-step process. The first step is to identify and organize the required libraries, resulting in a list of requirements and assumptions. The second step is to define the behavior and structure of the model and the list of constraints. Last, in the third step, the model is verified against the requirements and constraints.

The second and third stages are repeated until the model fits all the requirements and constraints.

So to account for the first step Package Diagram and a Requirement Diagram are built. The Package Diagram includes six packages: Distiller requirements, Distiller use case, Distiller structure, Distiller behavior, Value types, and Item types [7].

For simplicity, we include only two diagrams, requirements and behavior.

The Requirement Diagram carries two parts: one for the distiller specifications and one for assumptions. The requirements diagram is exposed in Figure 5.3.

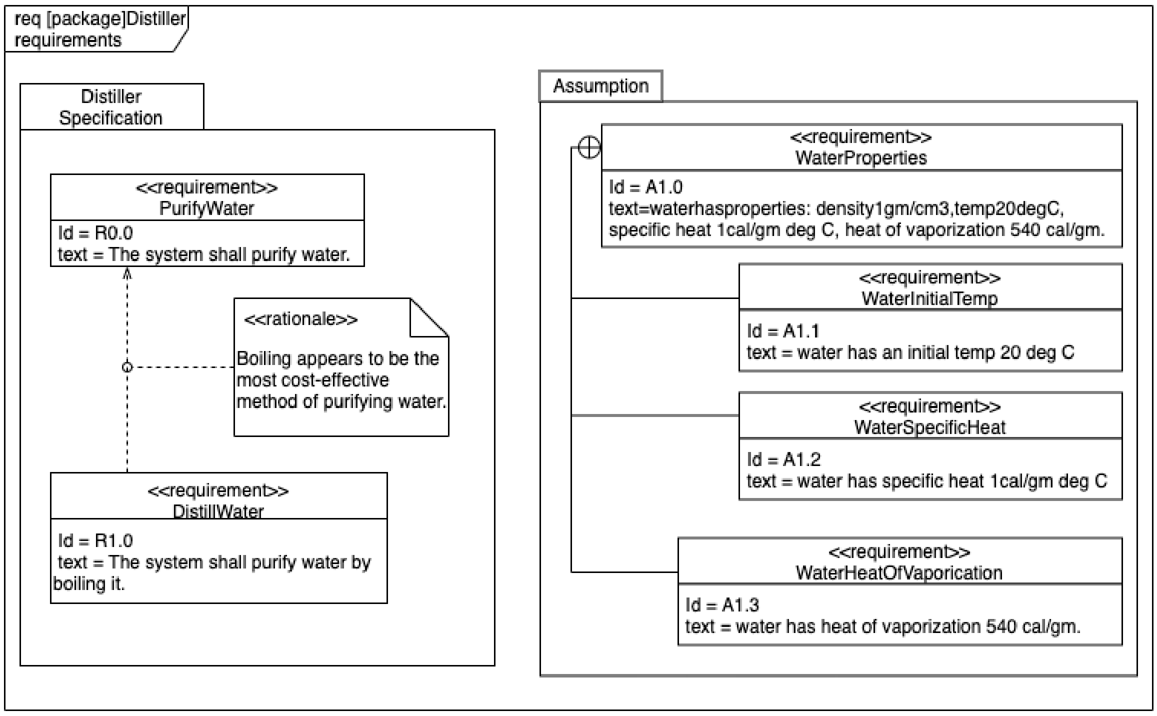

req [package]Distiller requirements
Distiller Specification
<<requirement>> PurifyWater
Id = R0.0
text = The system shall purify water.
<<rationale>>
Boiling appears to be the most cost-effective method of purifying water.
<<requirement>> DistillWater
Id = R1.0
text = The system shall purify water by boiling it.
Assumption
<<requirement>> WaterProperties
Id = A1.0
text=waterhasproperties: density1gm/cm3,temp20degC, specific heat 1cal/gm deg C, heat of vaporization 540 cal/gm.
<<requirement>> WaterInitialTemp
Id = A1.1
text = water has an initial temp 20 deg C
<<requirement>> WaterSpecificHeat
Id = A1.2
text = water has specific heat 1cal/gm deg C
<<requirement>> WaterHeatOfVaporication
Id = A1.3
text = water has heat of vaporization 540 cal/gm.

Figure 5.3: Distiller Requirement diagram in SysML

In the second step, the structure and behavior of the system are modeled using two diagrams: Activity Diagram (behavior) and Block Definition Diagram (structure).

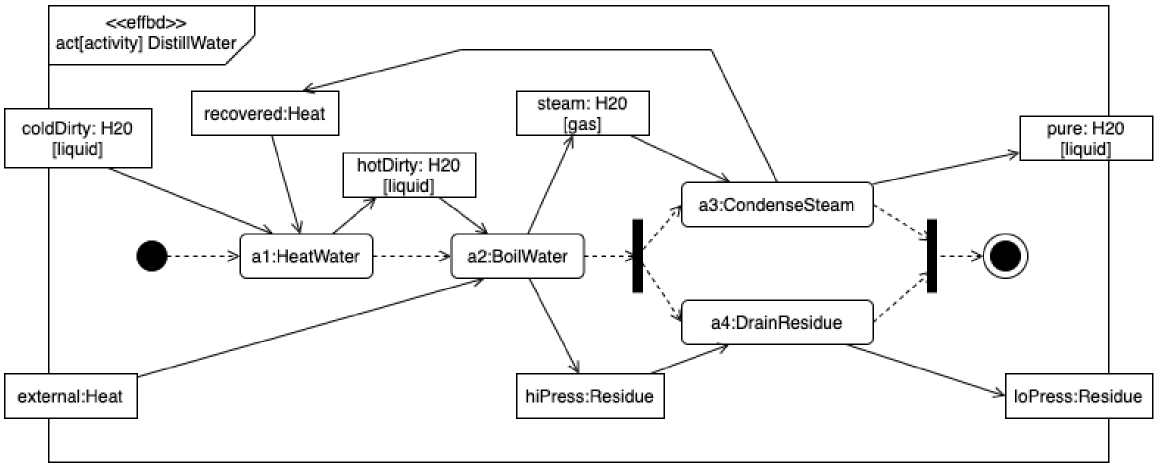


Figure 5.6: Distiller Activity diagram in SysML [8]

The distiller block diagram contains three main blocks: Heat Exchanger, Boiler, and Valve. The connection between the block diagram and the activity diagram is made by allocations. Allocations are used to allocate behavior onto the structure and flow onto I/O.

Additional diagrams such as sequence diagram and state-machine diagram could be created in order to complete the model. The second step ends when a complete model is obtained.

In the third step, the model is examined against the requirements and assumptions of the system. Stages two and three are performed repeatedly until the system meets all the requirements and constraints.

### Creating the distiller model with OPM

Similar to the SysML model approach, the process begins with the definition of the distiller requirements. The OPM does not define the precise structure of requirements, and this can be done as a form of another Object Process Diagram or by defining requirements in the table. The author of OPM proposes inserting requirements directly into the modeling tool model, making requirements definition a little bit tool dependent.

The next step included defining the system boundaries is defining its primary process and the system's principal function and, in this case, purifying being modeled as the primary process. Around this process, the associated objects are represented: affected object (Water in this case), inputs and outputs, main agents (actors) if there are some, and possible environmental (external) objects and processes involved. Figure 5.4 describes the System Diagram (the root of the OPD hierarchy tree) of the distiller system. It is the top-level diagram that highlights Purifying as the system's function as its central goal.

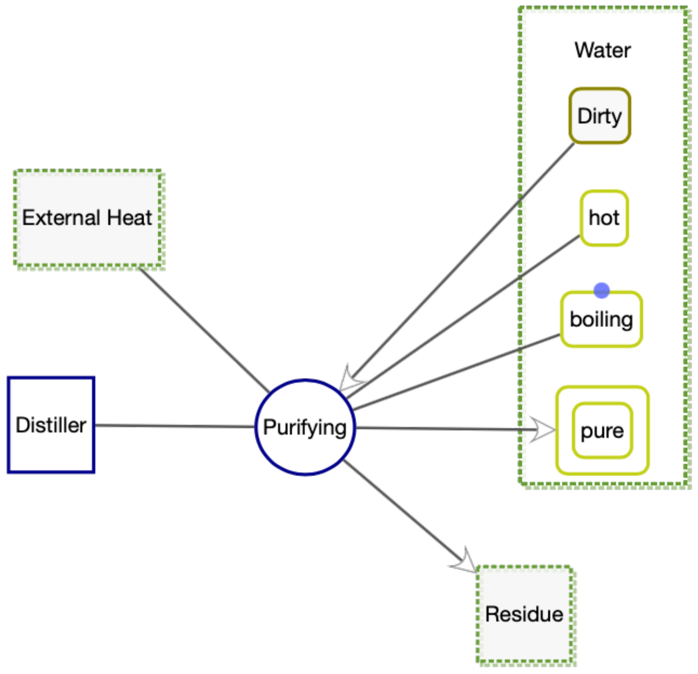


Figure 5.4: The System Diagram of the Distiller System in OPM

Next, the hierarchical refinement of the system processes and objects occurs, using the refinement mechanisms of in-zooming for processes and unfolding for objects. Figure 5.5 shows the in-zoomed Purifying process. A detailed sequence of subprocesses involved in the purifying process, shown inside the Purifying ellipse. They also change the states of other entities and require specific parameters to do that. In our case, they require subparts of the distiller to change the state of the water from dirty to boiling and pure.

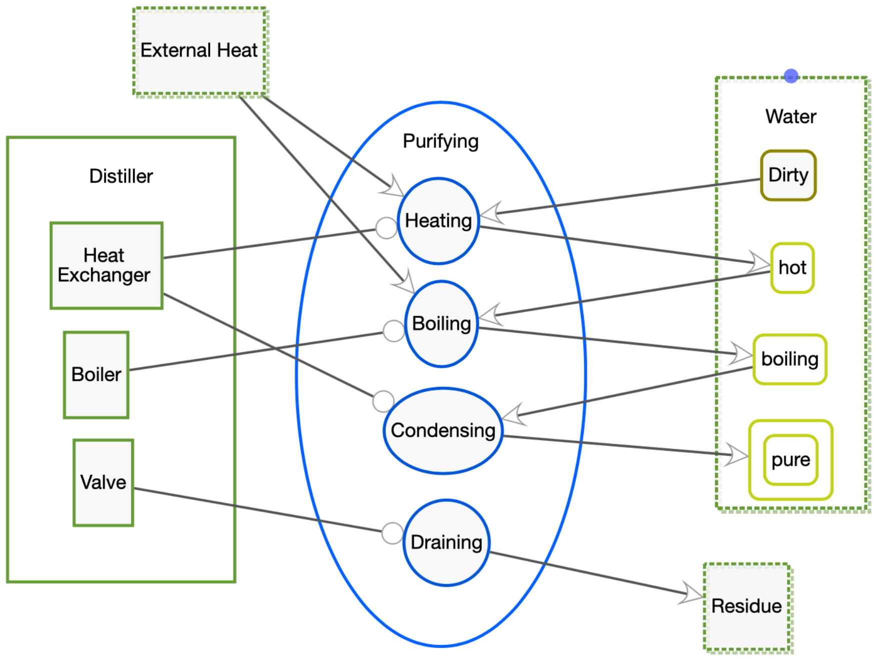


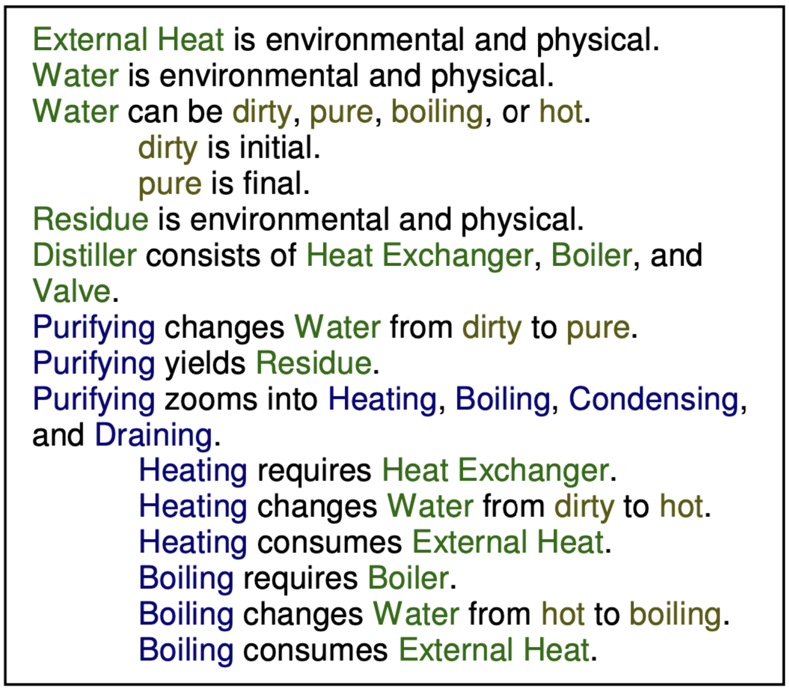
Figure 5.5: In-zoomed Purifying process in OPM

The Y-axis inside an in-zoomed process serves as a timeline. It specifies the sequencing of the operations, the first operation situated at the top. In the diagram, Heating is followed by Boiling and Condensing, and lastly, Draining.

The structure of the distiller is specified in The diagram by showing its parts, including Heat Exchanger, Boiler, and Valve.

OPM also distinguishes physical elements by the shading effect and environmental elements that are not part of the system by dashed contour.

The textual representation is then additionally generated from the graphical description. The Object Process Language text for the diagram in Figure 5.5 is listed in Figure 5.6.

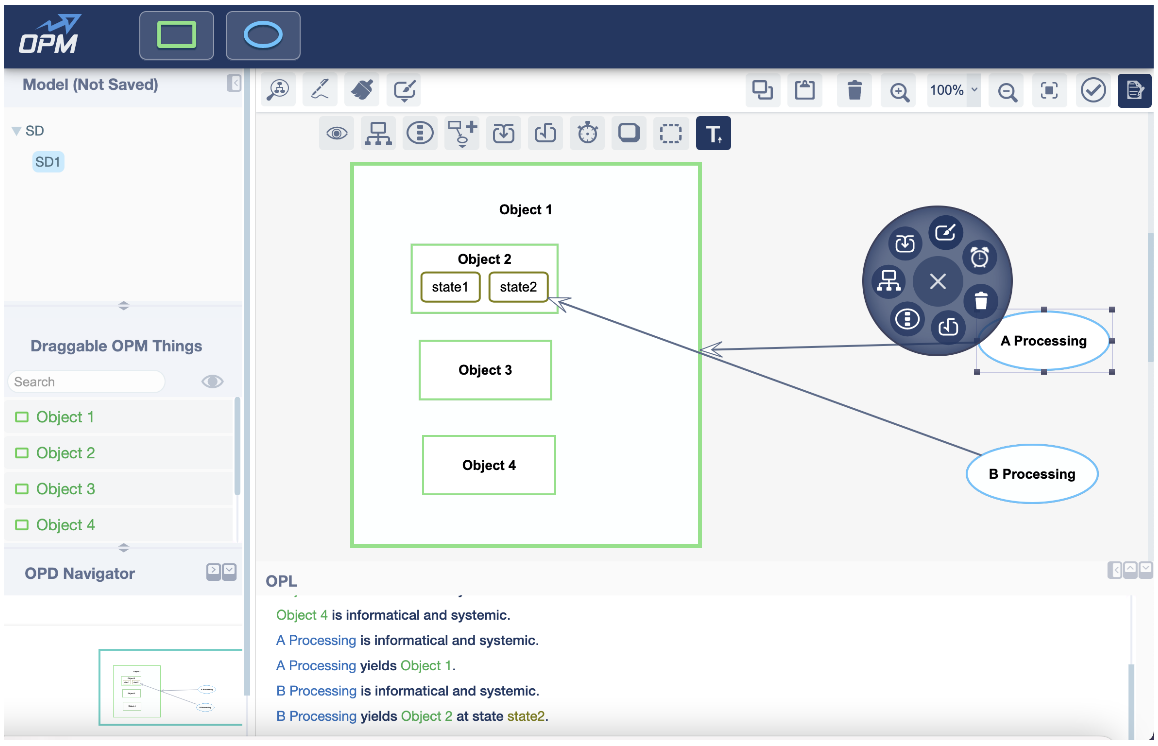


External Heat is environmental and physical.
Water is environmental and physical.
Water can be dirty, pure, boiling, or hot.
    dirty is initial.
    pure is final.
Residue is environmental and physical.
Distiller consists of Heat Exchanger, Boiler, and Valve.
Purifying changes Water from dirty to pure.
Purifying yields Residue.
Purifying zooms into Heating, Boiling, Condensing, and Draining.
    Heating requires Heat Exchanger.
    Heating changes Water from dirty to hot.
    Heating consumes External Heat.
    Boiling requires Boiler.
    Boiling changes Water from hot to boiling.
    Boiling consumes External Heat.

Figure 5.6: Distiller OPL [8]

## 5.4 Comparison of the project and existing OPM application

In recent months there has been an effort from the creator of OPM language and older desktop tool OPCAT to create an OPM web editor. Now, we can compare these two and highlight differences.



Object 4 is informatical and systemic.
A Processing is informatical and systemic.
A Processing yields Object 1.
B Processing is informatical and systemic.
B Processing yields Object 2 at state state2.

Figure 5.7: OMP cloud[1]

The main features of the OPM foundation solution are sophisticated OPL generation and the correction of possible ways to connect elements. They also provide full styling of elements, including text size and style. However, the editor's missing option is the possibility to save and upload created projects, making the app unusable for now.

On the other hand, our solution has a unique element tree to allow the user to fully control which elements he wants to see in the project and thus provide the functionality to see a coss connection between a variety of model components. So it means we have the ability to see and work with one model as a source of all information.

They use a different approach when it comes to showing in zoom's view. Each view in their case is a separate diagram, meaning the objects and processes present in multiple diagrams are copies, which requires synchronization. They do not have the „master" model that is shared by all views, and thus no sync is needed. We „hide" other elements from canvas, and they load the part of the project, which makes the transitions a bit slow.

They also choose different approaches when it comes to connecting and editing elements. Every element can change its size in its editor, but elements may not be included in the parent element. They are sometimes laid out on top of the parent element, which means the parent element does not always resize according to the parent.

In summary, the OPMCloud solution provides validation of the OPM model but lacks some of the primal functionality of managing the project. Our solution provides freedom when it comes to modeling decisions and full functionality to store the project.

[1]available at https://sandbox.opm.technion.ac.il/

# Chapter 6

# Conclusion

The aim of the work was to design and evaluate an agile editor of models of a selected modeling language. First, we examined parameters from conducted research in the field and found that the most common issue in the model-based approach was the inconsistency of model and code over time. In the second part, the OPM formalism was chosen as graphical language because of its simplicity, lack of available tools, and universality in the modeling domain. In the third part, Cytoscape was chosen for implementation because it provides most of the features required for manipulation with diagrams and its large community and well-maintained project. In the fourth part, the implemented editor was evaluated on an example of the distiller model and compared with existing editors. Along with that, the editor showed its uniqueness in allowing users to work with the master model and show all elements of the model or pick up single elements from multiple levels of hierarchies.

The central portion of the practical part was connecting different segments of the Cytoscape framework, creating an application user interface, and providing the functionality needed to work with OPM models. There were also attempts to provide an optimized backend, written in Rust language, to application for user login and saving projects to the cloud. However, due to the time demand of such a feature, the editor supports only local storage of projects. In the future, there is a probability that the backend will be in the form of a service (BaaS), and multiple providers are discussed.

The aim of the thesis was not to create an editor that could immediately compete with the existing ones, but mainly to verify the suitability and readiness of new formalisms for practical application, try new approaches to system modeling and provide a basic implementation, which can be easily expanded. This work opens up space for several future works. Among the possible extensions, we can mention integration to open a source code editors, dynamic simulation of the model, simultaneous editing, and diagrams automatic generation from code as a form of synchronization with written code. It would also be appropriate to improve the generation of OPL, enhance responsiveness of the left menu, and enable users to create their own views[1].

[1] Application as a service can be found at: https://visual-flow-9631a.web.app

# Bibliography

[1] *INTRODUCTION TO OMG'S UNIFIED MODELING LANGUAGE™ (UML®)* [online]. Object Management Group [cit. 2021-04-01]. Available at: https://www.uml.org/what-is-uml.htm.

[2] *SysML Open Source Project - What is SysML? Who created SysML?* [online]. Object Management Group [cit. 2021-04-01]. Available at: https://sysml.org.

[3] *WHAT IS SYSML?* [online]. Object Management Group, 2021 [cit. 2021-04-01]. Available at: https://www.omgsysml.org/what-is-sysml.htm.

[4] Dori, D. and Crawley, E. *Model-based systems engineering with OPM and SysML*. New York: Springer, 2016. ISBN 978-1-4939-3294-8.

[5] Forward, A. and Lethbridge, T. Problems and opportunities for model-centric versus code-centric software development: A survey of software professionals. *Proceedings of the 2008 International Workshop on Models in Software Engineering*. january 2008, p. 27–32. DOI: 10.1145/1370731.1370738.

[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., Moore, A. and Steiner, R. *OMG Systems Modeling Language (OMG SysMLTM) Tutorial* [online]. INCOSE, september 2009 [cit. 2021-04-01]. Available at: https://www.omgsysml.org/INCOSE-OMGSysML-Tutorial-Final-090901.pdf.

[8] Grobshtein, Y., Perelman, V., Safra, E. and Dori, D. Systems modeling languages: OPM versus SysML. In:. April 2007, p. 102 – 109. DOI: 10.1109/ICSEM.2007.373339. ISBN 1-4244-0771-0.

[9] Kremer, M. *Comparing Cross-Platform Frameworks* [online]. Ionic, 2021 [cit. 2021-04-01]. Available at: https://ionic.io/resources/articles/ionic-vs-react-native-a-comparison-guide.

# Appendix A

# All tested frameworks

There were 16 graphical frameworks reviewed for desirable functionality discussed in chapter 3.

## List

Frameworks that have more than four required functionalities:

- Cytoscape
- D3
- jgraph / Drawio
- Sprotty

with less than four:

- jsplumb community edition
- PixiJS
- React-diagrams

Frameworks that have at least interactive elements:

- p5.js
- Raphael
- sigmajs
- visjs

The graphical frameworks were found to have none of the required criteria for diagram manipulation:

- three.js
- statejs
- treant-js
- graphviz
- paperjs

# Appendix B

# Used Cytoscape extensions

- „cytoscape“: „3.17.0“,
- „cytoscape-automove“: „1.10.2“,
- „cytoscape-clipboard“: „2.2.1“,
- „cytoscape-cxtmenu“: „3.3.1“,
- „cytoscape-edgehandles“: „3.6.0“,
- „cytoscape-expand-collapse“: „4.0.0“,
- „cytoscape-popper“: „1.0.7“,
- „cytoscape-undo-redo“: „1.3.3“,
- „cytoscape-view-utilities“: „5.0.0“,

# Appendix C

# Data model

```json
{
    "elements": {
        "nodes": [
            {
                "data": {
                    "id": "1",
                    "shape": "rectangle",
                    "padding": 5,
                    "name": "Water",
                    "width": "name",
                    "dotted": 1,
                    "backgroundColor": "#ffffff",
                    "borderColor": "#669c35",
                    "shadow": 1,
                    "invisibleParent": 0,
                    "visibility": 1
                },
                "position": {
                    "x": 483.98238403498334,
                    "y": 591.3682723961318
                },
                "group": "nodes"
            }
        ]
    }
}
```

# Appendix D

# Content of the attached media

```
/
└── code ........................................ Code directory
    ├── public .................................. Icons directory
    │   └── assets .............................. Icons
    ├── src ..................................... Application source code
    │   ├── pages ............................... Pages directory
    │   │   └── Page.tsx ........ Drawing page includes canvas and frame
    │   ├── components .......................... Components directory
    │   │   ├── Cytoscape.jsx ................... Canvas and diagram logic
    │   │   └── Menu.tsx ........ Frame including menu, bars, buttons
    │   ├── theme ............................... Styles for app
    │   └── App.tsx ............................. App root file
    └── README.md ........... Instructions to build and run application
└── thesis ...................................... Thesis source files
```