TU WIEN Informatics

Towards a Visual Design and Development Environment for the Peer Model

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Matthias Schwayer, BSc
Matrikelnummer 00925825

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dr. Dipl.-Ing. Eva Maria Kühn
Mitwirkung: Projektass. Dipl.-Ing. Stefan Craß

Wien, 1. Juni 2020

Matthias Schwayer	Eva Maria Kühn

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Towards a Visual Design and Development Environment for the Peer Model

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Matthias Schwayer, BSc
Registration Number 00925825

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dr. Dipl.-Ing. Eva Maria Kühn
Assistance: Projektass. Dipl.-Ing. Stefan Craß

Vienna, 1st June, 2020

Matthias Schwayer	Eva Maria Kühn

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Matthias Schwayer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Juni 2020

Matthias Schwayer

Acknowledgements

First of all, I would like to thank Eva Kühn for her supervision of this thesis and for providing me with interesting research opportunities in her team over the course of several years. I would also like to thank Stefan Craß for his co-supervision and his extremely helpful comments, which helped improve this thesis a lot. As soon as the modeler was usable, Maximilian Irlinger used it as a basis for his diploma thesis. I would like to thank him very much for acting as a beta tester and thus improving the result of this work.

For their continued support in all my endeavors, for teaching me early on to be curious, to think for myself, and to work precisely I would like to express my deepest gratitude to my parents, Hedwig and Karl. I also want to thank my sister, Cornelia, for accompanying me on my way, for warning me when I was about to do something stupid, and for not (always) reminding me if I did anyway.

A special thanks goes to Stephan Cejka, who has accompanied me through almost my entire studies and worked with me on countless projects and courses as well as on our joint Bachelor's thesis. Additionally, I would like to thank my fellow students Albin, Benjamin, David, Georg, Natalie, and Stefanie for their cooperation during our studies. Representing all my colleagues at work, I want to thank especially Christian and Peter, who took me in as a young colleague and became dear friends.

I would like to thank Cornelia and Stephan for proofreading this thesis on very short notice.

Finally, I would like to express my deepest thanks to my beloved girlfriend Tini, who has always motivated me over these last years to finish this thesis, even when I was less than pleasant to be around.

In one form or another, every single one of them has made me who I am today and without any of them, this thesis would not have been finished.
Thank You!

Some of the icons used in the modeler are designed by Freepik[1] or Smashicons[2] from Flaticon (`www.flaticon.com`).

[1] `https://www.flaticon.com/authors/smashicons`
[2] `https://www.flaticon.com/authors/freepik`

Kurzfassung

Immer mehr Geräte des täglichen Lebens können automatisiert, ferngesteuert und mit Diensten oder anderen Geräten verbunden werden. Der Informationsaustausch zwischen ihnen muss koordiniert und der Prozess, an dem sie beteiligt sind, muss orchestriert werden.

Das Peer Model ist ein Ansatz zur Modellierung der Koordination nebenläufiger Systeme, der auf den Prinzipien des Aktormodells, der Petri-Netze und der Tupelräume basiert und von der Forschungsgruppe für Space Based Computing der TU Wien entwickelt wurde. Um das Peer Model wurde eine Toolchain aufgebaut, die eine domänenspezifische Sprache, mehrere Implementierungen, sowohl für unternehmensweite als auch eingebettete Anwendungen, und eine grafische Notation umfasst. Bisher war die einzige Möglichkeit, grafische Modelle für das Peer Model zu entwerfen, die Verwendung generischer Zeichenwerkzeuge oder Handzeichnungen.

Die zentrale Forschungsfrage dieser Arbeit ist, wie die Software-Architektur einer visuellen Entwicklungsumgebung für das Peer Model, das sich in aktiver Entwicklung befindet und sich somit ständig weiterentwickelt, so gestaltet werden kann um mit möglichst geringem Aufwand neue Funktionen zu unterstützen. Diese Arbeit steuert daher den ersten grafischen Modellierer bei, der speziell für das Peer Model entworfen wurde, mit dem Ziel einer besseren Benutzerunterstützung und einer breiteren Akzeptanz des Modells. Um dies zu erreichen, sind die Arbeitsabläufe der Benutzeroberfläche so einfach, intuitiv und benutzerfreundlich wie möglich gestaltet. Der Modellierer ist in C++ implementiert und verwendet die Qt-Bibliothek als Grundlage für die grafische Benutzeroberfläche. Da das Peer Model aktiv entwickelt wird, basiert der Modellierer auf einer erweiterbaren Architektur und einer wartbaren Codebasis. Darüber hinaus speichert der Modellierer nicht nur die Diagramme, sondern pflegt zusätzlich ein zugrunde liegendes semantisches Modell, welches das mit Hilfe des Peer Models entwickelte Modell repräsentiert. Dieses zugrundeliegende Modell ist auch die Grundlage für Funktionen wie Plausibilitätsprüfungen und Codegenerierung, die beide über ein Plugin-System implementiert werden. Die Implementierung des Modellierers basiert auf der Evaluierung mehrerer bestehender grafischer Modellierer für andere Kommunikations- und Koordinationsmodelle. Eine erste Benutzerstudie wurde durchgeführt, um die Benutzerfreundlichkeit der Anwendung zu evaluieren.

Abstract

An ever-growing number of everyday devices come with the ability to be automated, remote-controlled, and connected to services or other devices. The information exchange between them needs to be coordinated and the process they are involved in needs to be orchestrated.

The Peer Model, developed by the Space Based Computing Research Group of TU Wien, is an approach of modeling coordination of concurrent systems based on principles of Actor Model, Petri Nets and Tuple Spaces. A toolchain has been built around it, comprising a domain-specific language, several implementations, both for enterprise and embedded applications, and a graphical notation. Until now, the only way to design graphical models for the Peer Model was to use generic drawing tools or hand drawings.

The main research question of this thesis is how the software architecture of a visual development environment for the peer model, which is in active development and thus constantly evolving, can be designed to support new features with minimal effort. This thesis therefore contributes the first graphical modeler designed specifically for the Peer Model, with the aim of better user support and wider adoption of the model. To achieve that, the user interface workflows are designed to be as simple, intuitive, and user-friendly as possible. The modeler is implemented in C++ and uses the Qt library as its foundation for the graphical user interface. Since the Peer Model is being actively developed, the modeler is based on an extensible architecture and a maintainable code base. Further, the modeler not only stores the diagrams but additionally maintains an underlying semantical model, which represents the model developed with help of the Peer Model. This underlying model is also the foundation for features like plausibility checks and code generation, both of which are implemented via a plugin system. The modeler implementation is based on an evaluation of several existing graphical modelers for other communication and coordination models. An initial user study was conducted to evaluate the application's usability.

Contents

CHAPTER 1

Introduction

There are more and more devices that come with the ability to be automated, remote-controlled, and connected to services and other devices. These have wide fields of application including home automation (e.g. lights, sunblinds, heating), household appliances (e.g. dish washers, washing machines and dryers, refrigerators), consumer electronics (e.g. televisions, audio systems, hard disk recorders), wireless sensor networks, and many more. Increasing numbers of these devices also increases the need to coordinate information exchange and data transfers, and orchestrate processes involving them.

Since a lot of these tasks cannot be run effectively on a single processor, an ever growing number of tasks are distributed and executed on many processors. This is where communication middleware and coordination models come into play. They support for example the communication between distributed processes or the coordination of multiple participants in a workflow. Communication middlewares achieve this by providing communication features, e.g. for synchronization, message flow control, or guarantees regarding message delivery. Coordination models, on the other hand, provide high-level abstractions, and thus simplification, for possibly complex coordination mechanisms. Using these supporting technologies usually results in shorter development time and less errors, since a lot of the complex and error prone workflow is done by the middleware or coordination model.

Many graphical notations are available for different situations to be used by the individuals dealing with the development of process communication or coordination. Graphical notations use abstractions to make it easier to deal with the complex situations described by the respective models. Additionally, graphical models can give an overview of a system in relatively short time that would otherwise require much more time when written in prose or code. According to [29] the benefits of software visualization tools and therefore also the reason they are used in the industry are:

- to save time and money,
- to comprehend software better,
- to improve productivity and quality,
- to cope with complexity,
- to find errors.

Petre argues in [72] that graphical notations for software do not necessarily guarantee clarity, but that the use of a so-called *secondary notation* often determines how a graphical model is perceived. The *primary notation* is, e.g., the graphical representation of individual model elements or their relations, while the secondary notation includes for example the way how whitespace is used to visually group logically related elements. In addition to the graphical notation itself, the quality of a model design also depends on the skill of the respective designer. This results in differences between the designs of novices and experts. Usually experts are better at using secondary notation, which leads to more easily comprehensible model designs, e.g. due to a more coherent layout.

The clear benefits of using coordination and concurrency models can be seen by the fact that there are several well-known and extensively researched models to orchestrate processes and coordinate workflows, e.g. Petri Nets [18, 73, 75], Reo [6], WS-BPEL [64], and BPMN [66]. Because of the aforementioned advantages of using a graphical notation for systems design and to support all the different kinds of users, like domain experts for the modeled process or application developers, graphical editors have been created for these models, e.g.:

BPEL	BPEL Designer Project [1, 30]
BPMN	BPMN2 Modeler [2]
Petri Nets	CPN Tools [75]
Reo	Extensible Coordination Tools (ECT) [3, 7]

This thesis subsequently deals with another coordination model, called Peer Model, which does not yet have a graphical modeler, but one is developed in the course of this thesis.

1.1 Peer Model

The Peer Model (PM) [52, 53, 54] is being developed by Eva Maria Kühn et al. at the Space Based Computing Research Group of TU Wien. It is an approach to model coordination of concurrent systems based on principles of Petri Nets [18, 73, 75] and Tuple Spaces [35] and is partly inspired by Abstract State Machines (ASMs) [17] and Actors [4].

The PM's main building blocks are termed peers, which are addressable resources that encapsulate behavior. They have input and output containers holding typed data items with coordination properties named entries. Their properties are used to, e.g., transport application data or to influence how the PM handles entries. The internal behavior of a peer is modeled using wirings that consume, transform and/or create entries. A wiring comprises guards as input links that establish a precondition, optional service invocations and actions as output links that do post-processing. Peers can also contain subpeers to further abstract self-contained parts of their behavior. Wirings can use links to read from and write to a subpeer's containers. The basic workflow inside a peer starts with entries being written into a container (explained in-depth in Chapter 3). Guard links of wirings check, read, or take these entries and once all guards are satisfied move them into the wiring's internal container. Then the wiring can continue by executing its services and finish by executing the action links.

1.2 Motivation

The Peer Model's environment is aligned towards the toolchain in Figure 1 (which is based on a figure from [20]). At its center stands the PM-DSL [37], which stands for Peer Model domain-specific language. All the other tools in the PM environment are originally planned around that CTL. The PM is actively developed and researched, which becomes apparent from the already available parts that are shown as grey hexagons in the figure.

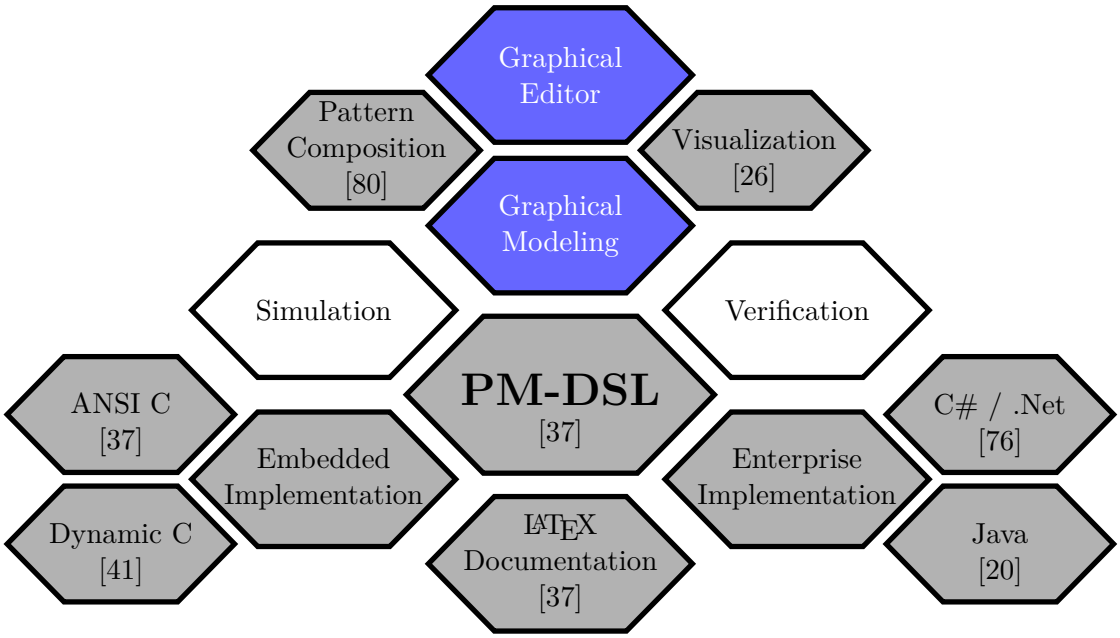


Figure 1: Extended Peer Model Toolchain [20]. Blue parts are presented in this thesis, grey parts are already available from other papers, and white parts are planned for future PM research.

Two previous theses already dealt with the field of graphical modeling for the PM. The first thesis extended the PM with pattern composition [80], which provides the possibility of designing coordination patterns in form of reusable partial PM models. These patterns can then be inserted and parameterized in other models. This work covered the semantics and relations with the existing PM and a matching graphical visualization. In hand with the second thesis a post-mortem visualizer of PM executions [26] was developed. This enabled users to analyze PM traces graphically.

However, the PM users currently still can only use drawings either by hand or in a generic drawing tool, like e.g. Microsoft Visio[1], for visual design of models. This is due to the fact that the PM lacks a graphical modeler that is designed specifically for it and thus can use information about the underlying model to better support its users. Therefore this thesis covers the PM toolchain's graphical modeling and editor parts highlighted in blue in Figure 1, to enable more users to use the PM for their system models and the design of concurrent processes and to develop an application using the graphical notation because they can rely on the tool support offered by a domain specific editor.

1.3 Contribution

This thesis aims to lay the groundwork for a visual development environment, called "Peer Model Modeling Tool" (PMMT), to work with PM models. Therefore, the first version of the PMMT was developed within the scope of this thesis. It contains a graphical editor that is specifically designed for working with the Peer Model, a plausibility checker that aids users by pointing out errors within the modeled system, and a code generator that increases development productivity by creating the code scaffolding for PM implementations. Such code generators are important to easily deploy the developed applications to targeted devices and also simplify platform independence of applications. The plausibility checker and code generator are both built upon a plugin system and corresponding interfaces to provide developers with an easy method of supporting newly developed PM implementations. For documentation purposes the models are exportable.

The visual representation of the designed model is an important part of understanding how the application will work. It should show as little information as possible, but as much as is needed to find flaws in one's design. Therefore, this was one of the priorities for this thesis and the PMMT. The ability to see the big picture with little details and drilling down into the more detailed parts of the model eases the analysis of complex models. This enables the developer to fully grasp the properties of the application. The visual representation itself is only in part a result of this thesis, since the graphical notation already specified in [52, 53, 54] was used as a starting point and adjusted to fit a graphical modeler as well as exported diagrams.

Additionally, this thesis focuses on a clean and extensible architecture for the PMMT to enable future application extensions with as little boilerplate code as possible. Another

[1] https://products.office.com/en-us/visio/flowchart-software/

important point is the usability of the graphical user interface (GUI), which is designed to be comfortable, intuitive, and easy to understand. Furthermore, constantly considering the application's performance during its development also goes hand in hand with the focus on usability.

1.4 Methodological Approach

First of all related modelers were evaluated regarding their general structure and ergonomic workflows with regard to the supported coordination model.

Based on that evaluation of existing modelers and specific features of the PM the requirements for a modeler were identified. Since the PM is an active research topic at the Space Based Computing Research Group of TU Wien, the specification has evolved since the start of this thesis. Therefore the supported PM features were specified together with the PM Technical Board.

After that, a technology for the PMMT implementation was chosen with the found requirements in mind and the software architecture with a plugin system was designed. The design phase also includes decisions for some general GUI workflows. This thesis focuses primarily on the visual modeling of PM models, which not only includes ergonomic model designing, but also how to aid the users in finding errors within their models. Therefore, some fundamental workflows, like scrolling and zooming behavior, were evaluated.

Finally, the modeler for the PM was implemented including possibilities for plausibility checking and code generation. This implementation was evaluated by comparing its features to the related modelers and by checking the fulfillment of found requirements. Additionally, a user study was conducted with a focus on the usability of the implemented modeler.

1.5 Outline

The following chapters are structured as follows:

Chapter 2 discusses related modelers and their supported coordination models. Their structure and features are analyzed in an attempt to find ergonomic workflows that aid the users in designing their models.

Subsequently, for a deeper understanding **Chapter 3** presents the details of the PM, since it has only been laid out briefly in Section 1.1.

Based on the analysis of related modelers and specific needs of the PM, a requirement analysis for the PMMT is conducted in **Chapter 4**.

An overview of the implementation of the PMMT is given in **Chapter 5**. This chapter includes architecture and software design choices as well.

Chapter 6 explores the visual modeling of PM applications.

Chapter 7 then compares the PMMT to the related modelers, evaluates the fulfillment of the introduced requirements, and presents the results of the conducted usability study.

Thereafter, **Chapter 8** proposes research topics for future work to extend the PMMT's feature set and work towards an even more convenient development environment for the Peer Model.

Finally, **Chapter 9** concludes this thesis with a summary of the achieved results.

CHAPTER 2

Related Work

This chapter introduces related modeling tools and their respectively supported modeling concepts. The selection of tools is based in part on the works of Kühn, Craß, and Schermann in [55, 80] and on other published research for modeling tools that support the coordination models related to the Peer Model as mentioned in [52, 53, 54].

Since a visual development environment with a graphical modeler for the Peer Model is also the main part of this thesis' contribution, the main focus points for tool selection are as follows:

- Availability of a graphical modeler for the model (alternatively a graphical visualizer of the model),
- Code generation or deployment support, and
- Some way of automatically checking the model for errors.

The availability or support of the following additional features were also considered for tool selection:

- Model animation,
- Simulation and model execution,
- Model checking and verification, and
- Performance analysis or profiling.

At the first glance UPPAAL[1] seemed appropriate for comparison as it is a graphical model checker tool most commonly used for verification of protocols. However, it will not be covered in this thesis, since there is no direct support for code generation.

Based on these criteria, each of the following sections presents one tool that supports a coordination model related to the Peer Model. Additionally, the Peer Model Monitoring Tool [26] is discussed, because it is designed specifically for the visualization of Peer Model traces.

2.1 AsmEE

AsmEE stands for *ASM Eclipse Environment* [32] and it is an integrated development environment for the *Abstract State Machines* (ASMs) formal method [16, 17]. It is part of the ASMETA toolset[2] [34], which also supports:

- animating executions of ASMs (AsmetaA) [14],
- simulating ASMs (AsmetaS) [33],
- validating ASMs (AsmetaV) [19],
- visualizing ASM models (AsmetaVis) [9], or
- generating code from ASM models (Asm2C++) [13].

2.1.1 Abstract State Machines

Abstract State Machines are a formal method for designing systems on a high level [15]. They are based on finite-state machines (FSMs) that operate on arbitrarily complex states with pseudo-code describing the functions and transitions. For example, Figure 2 shows a finite state machine that parses the string “FSM” in which the control states represent the input that has already been read and what input is accepted next. The labels of the arcs between the states show the character that is read to trigger this state transition. If the label starts with an exclamation mark, the match is negated, which means that the transaction triggers if any character other than the specified one is read. This FSM can also be written as a set of transitions shown in Table 1.

ASMs are a generalization of FSMs, because their state can be arbitrarily complex, e.g. structured data. Analogously to FSM transitions, an ASM is defined by a set of instructions, called rules, of the following form:

if *condition* **then** *Updates*

[1] http://www.uppaal.org/
[2] http://asmeta.sourceforge.net/index.html

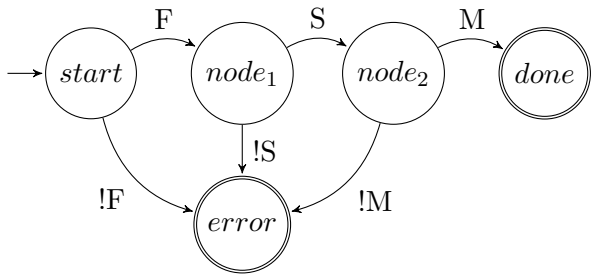


Figure 2: A finite-state machine that parses the string "FSM".

Start State	Input	End State
start	F	$node_1$
start	!F	*error*
$node_1$	S	$node_2$
$node_1$	!S	*error*
$node_2$	M	*done*
$node_2$	!M	*error*

Table 1: Transition set of the finite state machine in Figure 2.

If the *condition* is fulfilled, then the *Updates* are executed, which are a sequence of mathematical update functions that operate on the state of this ASM. This mathematical semantic model is the foundation for e.g. precise pseudo-code used by software architects and developers, or high-level process description used by application domain experts.

The ASM method covers the whole software life cycle:

Requirements Analysis supports the creation and formulation of a concise and precise ground model of the system. This model aims to meet the needs of all involved roles from users and experts in the application domain to software architects, developers, reviewers, and testers. To that end its abstraction level is at the application domain, and it is formulated with domain-specific language, so that it can be fully understood by all stakeholders. [16]

Design Refinement of the ground model is used to get from the system specification to code in multiple iterations with stepwise refinement. Every aspect of the ground model can be specified in detail with a *"mathematically precise notion of structure transforming pseudo-code"* [16, p. 102]. This means that the system architecture and components are iteratively specified in greater detail until it is specific enough for the implementation. [16]

Documentation of the model is generated partly as a side product of the refinement process. Since in every step an already fixed part of the system is described in higher detail, the resulting intermediate models also reflect the design decisions previously taken. Thus, these intermediate models explicitly describe the system and software structure. [16]

Validation and Verification of models as well as resulting software systems is supported by tools like simulation, i.e. creating a runnable model and testing it, or proving the correctness of the underlying mathematical model. [15]

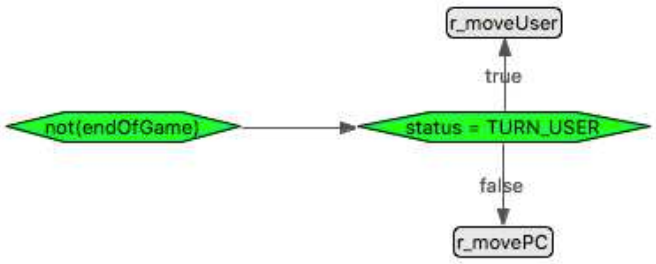
Börger states in [16] that not a single ingredient of the ASM method is original, but what makes it unique is its simplicity and the possibility to freely choose the most appropriate method for the current task at hand.

2.1.2 Integrated Development Environment

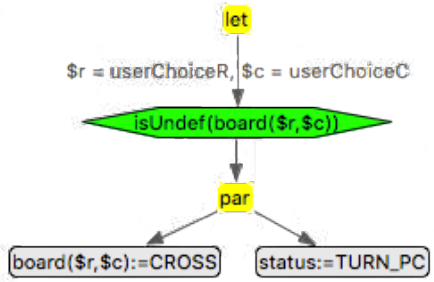
AsmEE currently features a textual editor for ASMs in the AsmetaL language [79], which itself is based on the ASM metamodel (AsmM) [77]. According to the website of the ASMETA toolset[3], a graphical editor currently is being worked on and not yet available. When creating a new ASM file in AsmEE, it contains simple example code for the children's game *tic-tac-toe* [25], which will be explained in the next section.

2.1.3 Visualization

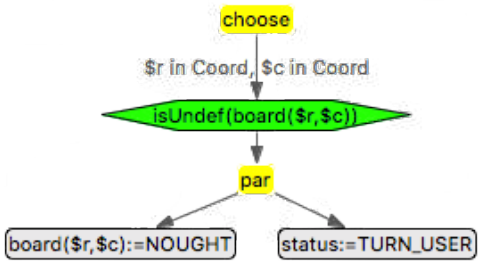
AsmetaVis [9] is the visualization tool in the ASMETA toolkit, which can visualize models written AsmEE. Figure 3 shows screenshots of the rules for the three-in-a-row children's game *tic-tac-toe* from the previous section, which are visualized by AsmetaVis. The main rule (Figure 3a) means that if the game is not over (**not(endOfGame)**), if the status is **TURN_USER** rule **r_moveUser** (Figure 3b) is executed, otherwise rule **r_movePC** (Figure 3c) is executed.



(a) Main rule **r_Main**



(b) Rule **r_moveUser**



(c) Rule **r_movePC**

Figure 3: Example ASM of tic-tac-toe game visualized by AsmetaVis [9].

2.1.4 Animation

AsmetaA [14] supports ASMs by displaying their complex state while the ASM is executed interactively. Its animation view is shown in Figure 4. The left column has two blue

[3] http://asmeta.sourceforge.net/download/index.html

buttons at the top that execute the next step(s), either one interactive or one or more random steps, where the number of random steps can be specified in the input box below. The two tables on the right contain the functions and their values at specific states. Upon starting AsmetaA all functions are in the bottom table, and they can be moved to the top table by checking the box in the first column. In this way, users can select specific functions in which they are interested, especially to maintain an overview in more complex models. The buttons on the bottom left move all functions of a specific type to the top or bottom table. The colors in the tables mark initial values (light blue) or changed values in this step (light green).

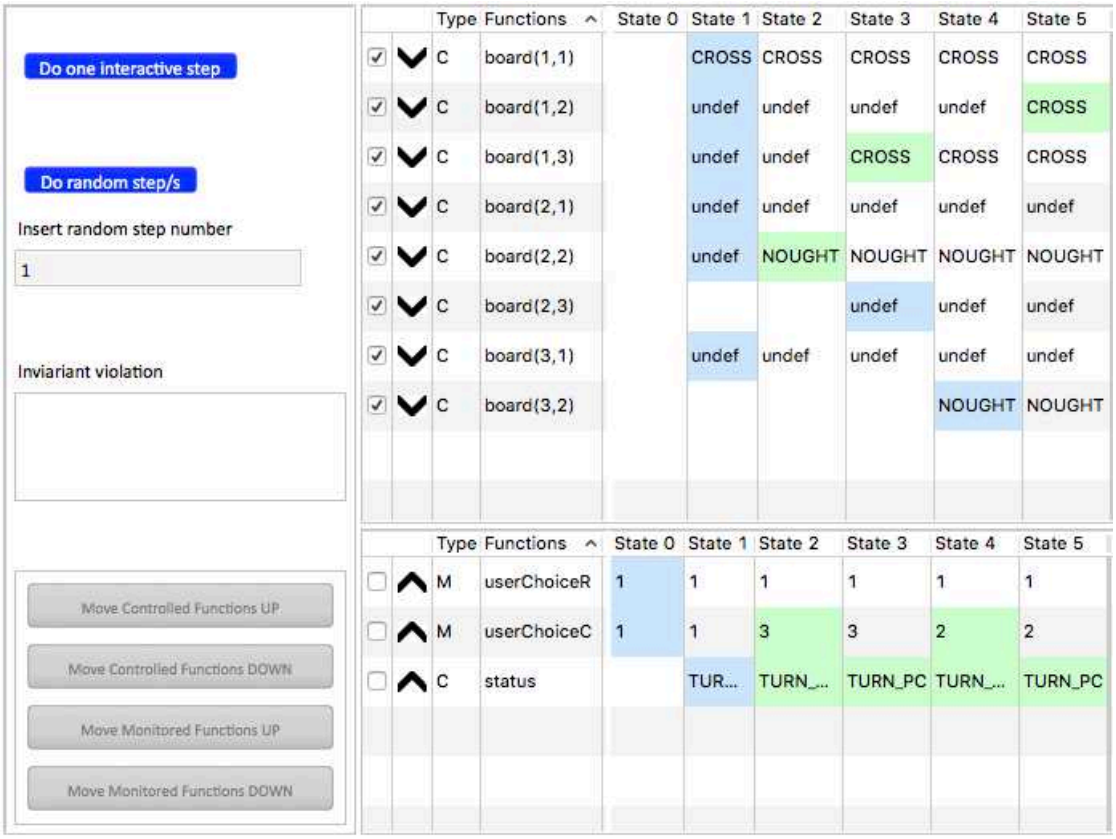


Figure 4: Animation view of AsmetaA

2.1.5 Validation and Verification

The ASMETA toolkit helps users in validating ASMs by providing *AsmetaS* [34] to simulate ASMs and *AsmetaV* [19] to validate ASMs in specific scenarios.

AsmetaSMV [8] enables users to verify ASMs by translating them to NuSMV specifications [22]. These specifications can be expressed in both Computation Tree Logic (CTL) and Linear Temporal Logic (LTL). As of version 2, NuSMV supports model checking

techniques based on propositional satisfiability (SAT) as well as on Binary-Decision Diagrams (BDD).

All three of these tools are text-based and interact with users via a console.

2.1.6 Code Generation

Bonfanti et al. presented an Eclipse plugin to generate C++ code for the Arduino platform from ASMs in [13]. However, it is not yet included in the AsmEE Eclipse Environment.

2.2 BPEL Designer Project

The BPEL Designer Project[4] is an Eclipse project that aims to support all *WS-BPEL 2.0* development tasks within Eclipse.

2.2.1 WS-BPEL

WS-BPEL stands for *Web Service Business Process Execution Language* [64] and it is an open-source standard maintained by OASIS[5]. It works with and as such is tightly coupled with other protocols of the web services protocol stack (described in [46]), like *Web Service Description Language (WSDL)*[6], *Simple Object Access Protocol (SOAP)*[7], and *Unified Description, Discovery, and Integration (UDDI)*[8]. The goal is to orchestrate web services with an executable language that enables users to specify how web services are used as single actions to compose business processes. According to the specification [64], the typical interaction models of such processes are request-response and one-way message exchange sequences between peers that are stateful and possibly long-running. WS-BPEL processes can be described in two ways: *abstract* or *executable*. Both possibilities have the same expressive power because they have the same feature set available. While executable processes are fully specified and thus can be executed, abstract processes are explicitly marked as abstract and are not intended to be executed. They are only partially specified and may hide operational details of the process that are not needed for the intended descriptive use. To that end they may use opaque tokens or omit the details entirely.

The main components used in WS-BPEL to describe business processes are:

Activities are used to describe the actions and workflow of the business process. A process has exactly one main activity, which can be composed of other activities. They are used to, e.g.:

[4] https://www.eclipse.org/bpel/
[5] https://www.oasis-open.org/
[6] https://www.w3.org/TR/wsdl20/
[7] https://www.w3.org/TR/soap/
[8] http://www.uddi.org/pubs/uddi_v3.htm

- invoke web services,
- wait for or reply to a specific message,
- update variables,
- generate faults,
- define a sequenced or parallel execution of other activities,
- conditionally or repeatedly (looped) execute activities,
- define a scoped activity with its own description of the main components,
- create a new activity type.

Partner Links describe the relationship between this process and all the partner web services that are interacted with, while the function of the web service is already described by WSDL. As the interaction is peer-to-peer, they model both the consumer and provider side of the service.

Variables are used to hold the state information that is needed for stateful interaction between web services.

Correlation Sets define the information that is used to correctly match associated messages over the course of multiple web service interactions.

Fault Handlers are used to undo the partial work of a scope in which the fault occured.

Event Handlers react to either incoming messages or alarms that are set off after a timeout of specific activities.

2.2.2 Graphical Editor

The graphical editor of the BPEL Designer (depicted in Figure 5) supports users in keeping the whole designed process in mind, including the *Partner Links*, *Variables*, *Correlation Sets*, and *Message Exchanges* in the collapsible toolbar right of the design canvas. Additionally, the designed process can be displayed as its underlying XML-based source file.

The editing workflow to place new actions and control structures in the process from the *Palette* on the right is either to drag and drop them to the desired canvas position or to first select them and then click the position on the canvas where the item should be created. The items can then be parameterized in the *Properties* view (shown in Figure 6a), where all details including partner links, ports and operations of the associated WSDL file can be selected.

More detailed information about an item is displayed in the hover info tooltip that appears when the mouse cursor is hovered above an item (see Figure 6b). The names of items can be edited in the properties view or inline directly in the editor.

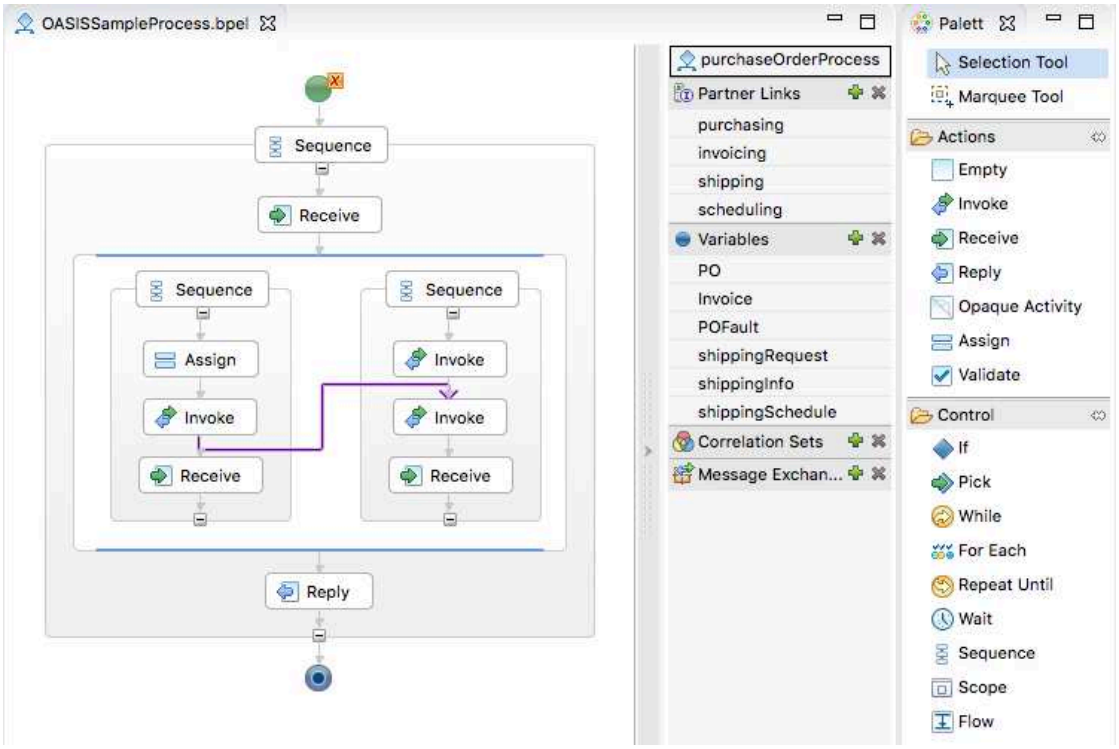


Figure 5: The BPEL Designer editor and palette views

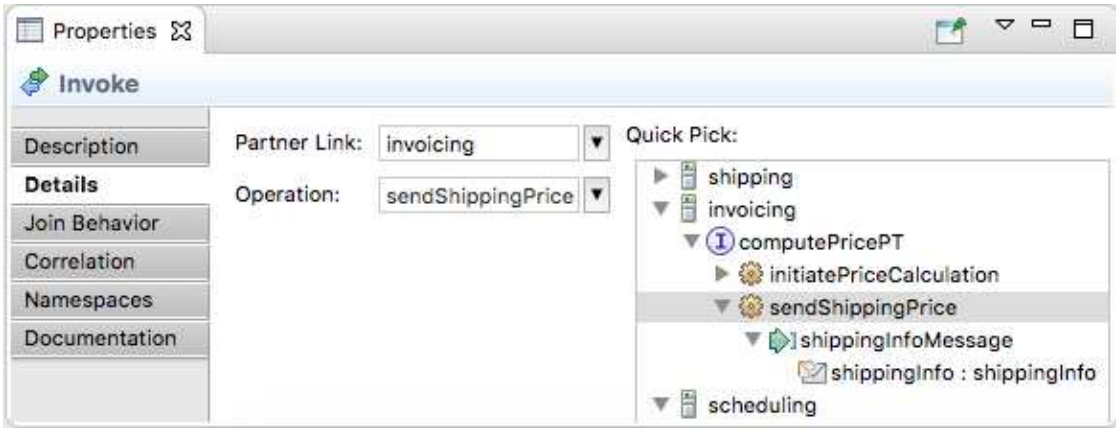
Items can be selected by clicking or by dragging a selection rectangle like a lasso around them. The properties view switches its contents to show the currently selected reference item's data. The selected items can be moved in the BPEL process by dragging them with the mouse to the desired position.

Furthermore, the used WSDL files are not only available as text files, but can be visualized as depicted in Figure 7. All the ports and operations are visualized together with their respective inputs, outputs, and faults.

2.2.3 Process Execution and Deployment

The designed processes can be executed, for example, on an Apache Orchestration Director Engine (ODE) Server[9], which is software specifically designed to execute web services written in WS-BPEL. Therefore, after adding a server to the project, the designed business process can be deployed to that server. After successful deployment, the application runs on the server and can be tested with the *Web Services Explorer* to check if everything is as expected. BPEL Designer also supports debugging of the running processes.

[9] `https://ode.apache.org/`



(a) Properties view

(b) Hover info tooltip

Figure 6: BPEL Designer's item information views

2.2.4 Model Validation

Whenever the designed business process is edited, a background process checks the model for validity. In case there is a problem, this is indicated in the graphical editor with an error icon, in the textual editor with a red squiggled underline of the problematic parts, and as entries in the table of the *Problems* view.

2.3 BPMN2 Modeler

The BPMN2 Modeler[10] is an Eclipse Project for creating business processes with the aim to provide a graphical modeling tool for BPMN2, which is easily adaptable to different BPMN2 execution engines. It is part of the Eclipse SOA Platform Project[11].

2.3.1 BPMN2

BPMN stands for *Business Process Model Notation* and is a standard for modeling business processes that is managed by the *Object Management Group (OMG)* [66]. An

[10] https://www.eclipse.org/bpmn2-modeler/
[11] https://projects.eclipse.org/projects/soa

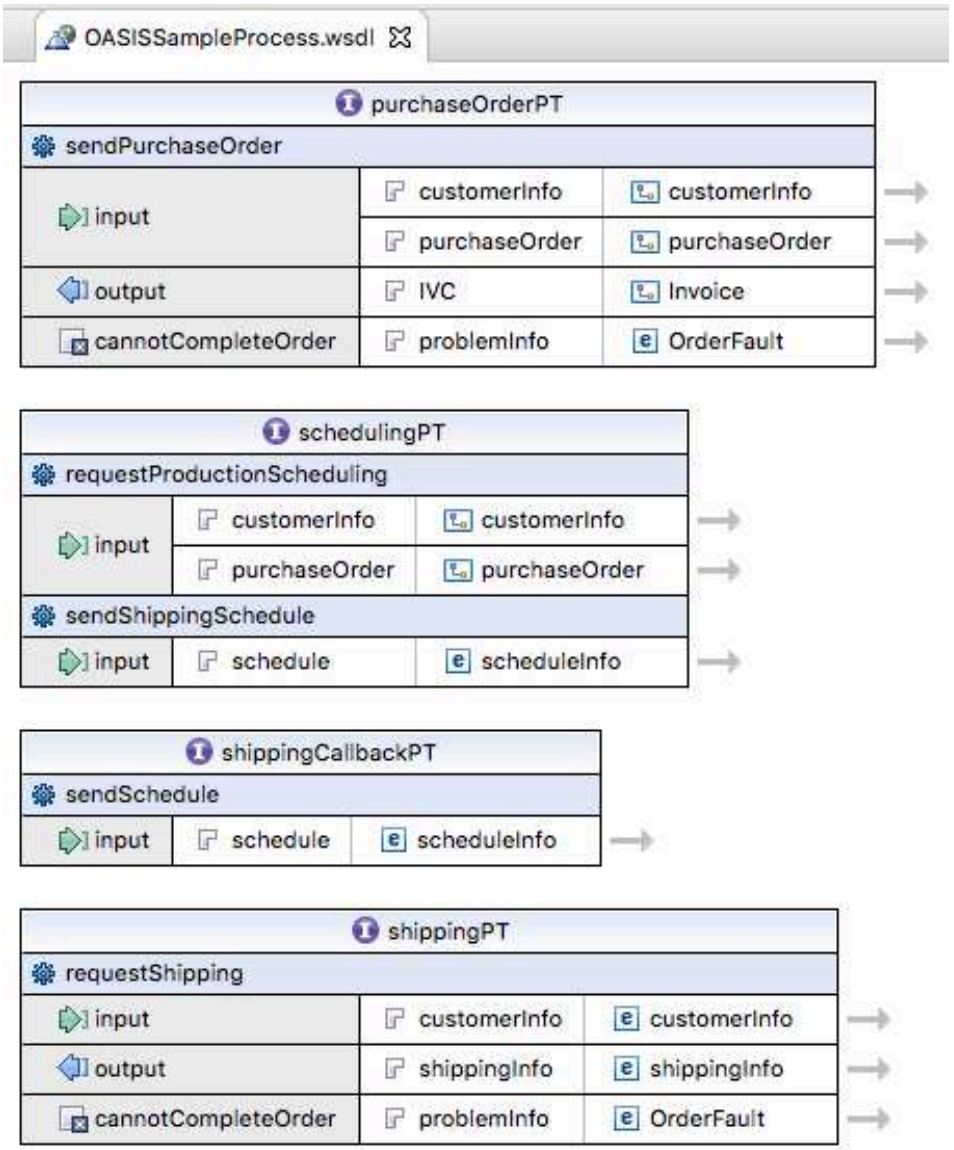


Figure 7: Visualization of the WSDL file.

easily understandable notation for all business users and thus a bridge for the gap between design and implementation of business processes is its primary goal.

The elements of BPMN are grouped into four basic categories:

Flow objects are the main elements of BPMN and comprise *Activities*, *Events*, and *Gateways*.

An Activity is an action that has to be executed (e.g. the **Develop Product**

Task in Figure 8). Activities can be atomic (*Tasks*), compound (*Sub-Processes* or *Transactions*), or call other existing activities (*Call Activity*) to reuse them.

In contrast to activities, events happen and the business process reacts to them. The *Start events* trigger the start and *End events* represent the result of the business process. *Intermediate events* represent something happening during the process and can either *Catch*, i.e. react to another event, or *Throw*, i.e. send a message to another *Pool* (will be explained in the *Swim Lanes* category).

Gateways are junction points, where the flow of the business process can split up into or join multiple paths. There are several different gateways. At the *Exclusive* gateway only one of the alternative paths will be followed, whereas at the *Inclusive* gateway all alternative paths will be executed. At an *Event Based* gateway the path taken depends on the evaluation of an event. *Parallel* gateways will create parallel paths without evaluating conditions and *Complex* gateways can be used to model complex synchronization behavior. Finally there are two combined gateways: *Exclusive Event Based* and *Parallel Event Based*. At the former an event is evaluated to determine the path that will be followed. For the latter an event triggers the execution of all parallel paths without evaluating the event itself.

Connecting objects connect flow objects and are divided into *Sequence flows* that show the sequence of the business process (depicted in Figure 8 by the arrows connecting events and activities), *Message flows* showing the transfer of messages across organization borders (e.g. between *Pools*), and *Associations* that connect *Artifacts* or text to flow objects.

Swim Lanes are used to graphically organize activities and can either be a *Pool* or a *Lane*. Analogously to a swimming pool, pools can contain one or more lanes and are used to model different organizations within the process. The BPMN specification explicitly leaves the meaning of lanes to the modeler, but they usually model the different functions, systems, or internal departments. Lanes are used to categorize and organize activities.

Artifacts are used by designers to add more information to the model or diagram and thus improve its readability. There are three different artifacts: *Data Objects*, *Groups*, and *Annotations*. A data object defines either required or produces data of an activity, groups are used to group activities together without affecting the flow, and annotations are used to explain some part of the diagram in more detail.

The BPMN specification not only contains a graphical notation but also a mapping to WS-BPEL and since version 2.0 the specification also defines execution semantics for its elements [66].

2.3.2 Graphical Editor

The graphical editor of the BPMN2 Modeler enables users to design business processes in a visual development environment. Figure 8 shows the editor view of a process diagram with the *Palette* on the right. From there, users can select new items to place in the process.

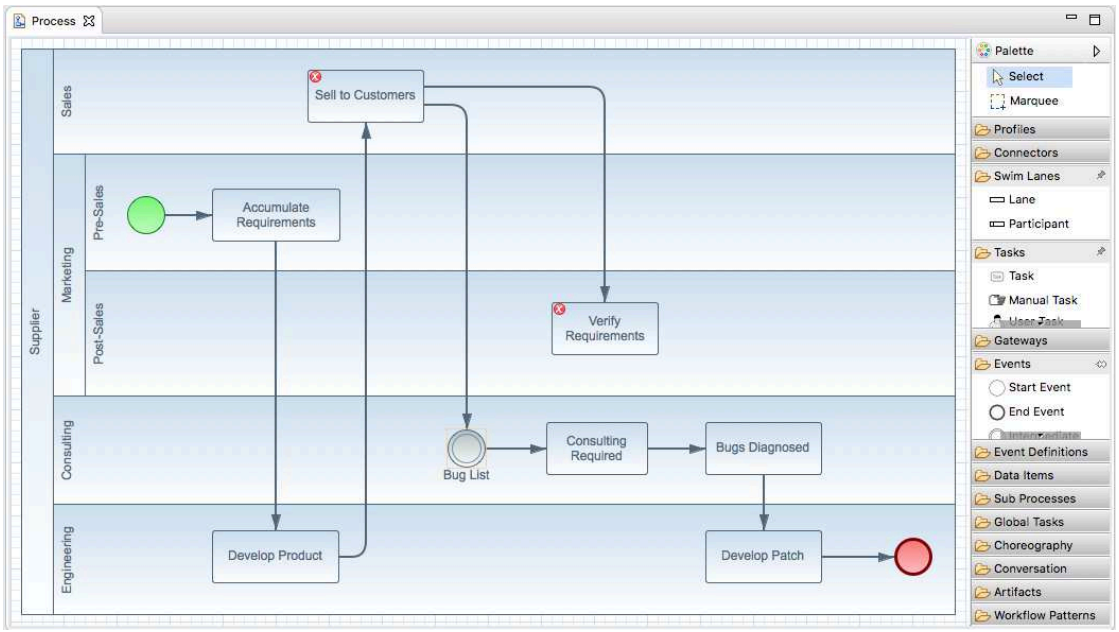
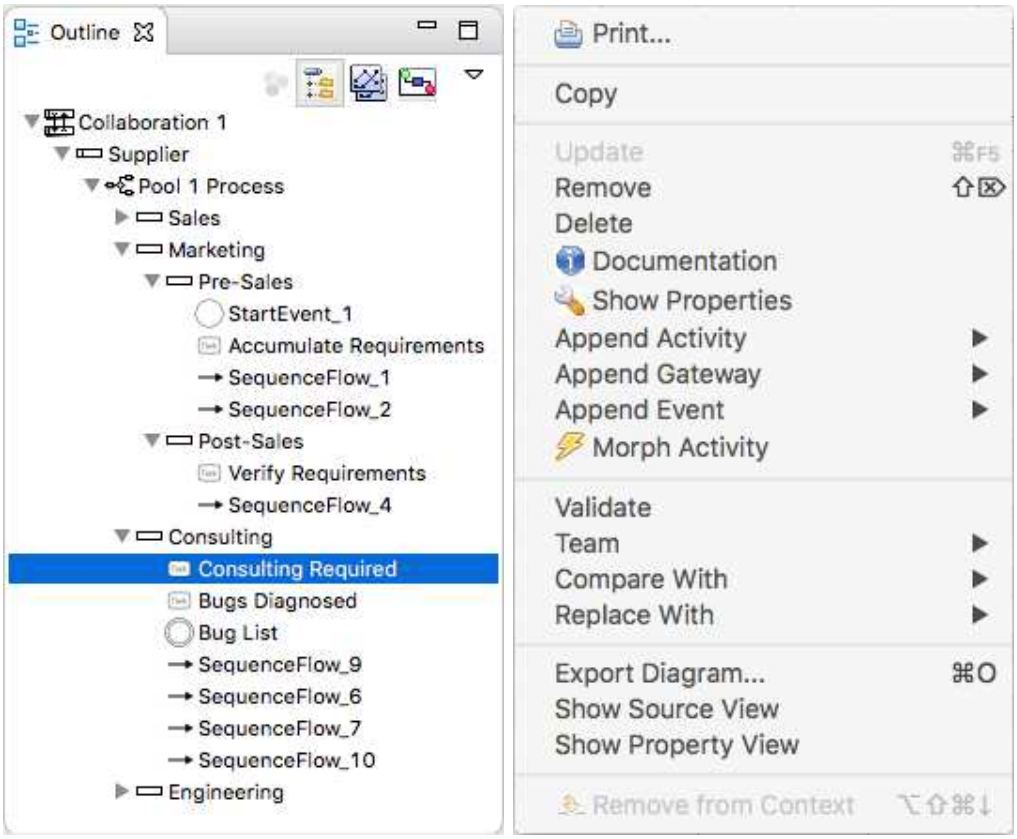


Figure 8: Editor view of a process in BPMN2 Modeler with Palette view.

The workflow of graphically editing processes starts by creating a new or opening an existing process, which opens a window with the editor view showing a canvas. Like in BPEL Designer new items can be placed on the canvas either via drag and drop or select and click.

Figure 9a shows the *Outline View*, which gives a hierarchical overview of the model in the currently active editor view. It is a tree view where child nodes can be collapsed and expanded.

Existing items can be edited in a number of different ways. When the mouse cursor hovers near or over the item, a contextual toolbar pops up (see Figure 9c) and shows buttons to interact with the item. The waste bin icon on top deletes the item from the model and the icons on the left open a new window with either a text editor for the item's documentation (top icon) or all available item properties to edit (bottom icon). Using the bottom icons users can do the following actions (starting from the left): a) create a connection to another activity by dragging the mouse cursor, b) morph the activity's type to another one, c) append a new event, d) append a new gateway, or e) append a new activity.



(a) Outline view of the process (b) Context menu for activity item

(c) Contextual toolbar

Figure 9: Selected editor features of the BPMN2 Modeler.

Items can be selected by clicking the item or by clicking the canvas and dragging a lasso around the items to select. Selected items can be resized using the displayed grabbing handles in the corners. The text of an activity can be edited inline in an overlay text box that appears after clicking the items text.

The item context menu (as shown in Figure 9b) provides all the actions from the context toolbar as well as some general actions like *Print*, *Copy*, or *Validate* for the selected items,

some version management actions (*Team*, *Compare With*, *Replace With*), *Export Diagram* to create a picture file of the selection, *Show Source View* to open the underlying XML code describing the process, and *Show Properties View* to open the properties of the selected item as a docked window.

Users can also use the keyboard to interact with the selected items in the editor view. The cursor keys with a pressed modifier key can be used to move the currently selected items and pressing the delete key will delete them.

2.3.3 Deployment

BPMN2 Modeler is a useful tool not only to design business processes, but also to support software architects in directly deploying these business processes to a business process engine, which in turn can execute and run the business process. For that the BPMN2 Modeler can be extended by *Target Runtimes* that support the deployment of designed business process to so called *Workflow Engines*. The *jBPM Target Runtime*[12] was developed within the BPMN2 Modeler Project to demonstrate the extension possibilities and in turn also influenced the design decisions of the extension points.

2.3.4 Model Validation

Similar to BPEL Designer (cf. Section 2.2.4), the model is validated in the background while editing it. The problems are also displayed in the editor by error icons (e.g. the *Verify Requirements* Task in Figure 8) and when hovering the mouse cursor above the erroneous task, a popup tooltip will show more details.

2.4 CPN Tools

CPN Tools[13] [75] combine a graphical editor, simulator, and analyzer for timed and untimed, hierarchical Colored Petri Nets (CPNs). The tools support among others

- designing Petri Nets using the graphical editor,
- syntax checking and error highlighting in the editor,
- simulating and verifying designed Petri Nets, and
- analyzing the performance of Petri Nets.

2.4.1 Petri Nets

First presented in the thesis of Carl Adam Petri [73], Petri Nets or place/transition nets (PT nets) are mathematical models for describing and reasoning about concurrent and

[12] http://www.jbpm.org/
[13] http://cpntools.org/

distributed systems. Since then, they have been widely adopted, e.g. for the following application areas:

- software design,
- modeling, simulation, and reasoning of concurrent systems,
- verification or performance validation of protocols, or
- workflow management systems.

The main components of Petri Nets are *places*, *transitions*, and *tokens*. The latter represent units of resources that reside in places and are graphically represented by dots. Places are resources that can contain zero or more tokens and are graphically represented by circles. Transitions are the possible events in the modeled Petri net and are represented by bars. Places can be connected to transitions with directed edges called *arcs*. A transition can have input and output places depending on the direction of the connecting arc. Once all of a transition's input places have tokens, it is said to *fire*, which means that the transition consumes the tokens from its input places and generates tokens at its output places. This means that input places are pre- and output places are post-conditions. Arcs can also have a so-called *multiplicity* that describes how many tokens will be consumed or generated upon firing of the transition. The multiplicity also means that a transition will only fire if all input places have enough tokens to satisfy the arc's multiplicity.

Over the years some extensions for Petri Nets have been introduced.

Colored Petri Nets [47] enable the distinction between different types of tokens by adding colors to them. The colors can also be used to model arbitrarily complex data that is carried by the tokens. Transitions are aware of token colors, which enables them to only fire when tokens of specific colors are residing in the input places.

Places also have a *type* that determines the colors of the tokens that can be held. Additionally, another abstraction layer is introduced with the concept of *variables* and *bindings* [57].

Furthermore, Colored Petri Nets also introduced support for hierarchies and reuse of Petri Nets to cope with the fact that large Petri Nets easily became confusing. Therefore subnets, called *pages*, can be referenced in other nets via so-called *substitution transitions*. In the subpages there are special tags for *port* and *socket places*. They build the interface through which the subpage communicates with its calling Petri net's surroundings.

Time in Petri Nets has been introduced independently by Merlin [62] and Ramchandani [74] in 1974. There are basically three different ways to represent time in Petri Nets: a) *firing*, b) *holding*, and c) *enabling durations*.

Firing durations assign each transition a duration, which changes the semantic of the execution. The transition will still consume the tokens from its input places (according to the arcs) when it becomes enabled, but it will only create the tokens at its output places after the assigned firing duration has elapsed.

Holding durations are also assigned to transitions, but the execution semantic stays unchanged, because instead of keeping the tokens in the transition they are classified as available or unavailable (graphically represented by non-filled dots). Such unavailable tokens become available once the holding duration elapsed and only then can enable a transition. Previously the durations have also been assigned to places instead, which has the same modeling power [10, 81].

Enabling durations can be assigned to a transition, which then must be enabled for the duration before it fires. If, on the other hand, the durations are assigned to a transition's input arcs, the transition must either be enabled for the longest duration [88] or every arc starts its countdown once the input place satisfies the arc [28]. The latter changes the execution dynamic greatly, but is equally expressive as the former as shown in [28].

According to [18] the names for Petri Nets incorporating time depend very much on the particular author, but in general enabling durations are associated with *stochastic Petri Nets* and *time Petri Nets*, while firing and holding durations are used in *timed Petri Nets*.

Object Petri Nets extend tokens to model nets themselves, and thus fit the description of "nets within nets". These approaches enable modeling dynamic aspects of systems and intend to overcome the static nature of Petri Nets. On one hand there have been proposals that extend CPNs [60, 61], and on the other hand Valk proposes the so-called *Elementary Object System (EOS)* in [86, 87]. An overview of the different approaches and their analytical methods is presented in [63].

2.4.2 Graphical Editor

Upon starting CPN Tools an empty window is presented with only the so-called *Index*, which is the light blue column on the left depicted in Figure 10. From there, one can create new or load existing nets via *marking menus* [58] (shown in Figure 11a). These marking menus are essentially circular context-sensitive menus, which are activated by holding the right mouse button down. When it is released while the mouse cursor is above a menu entry, this command is executed. The study conducted by Kurtenbach and Buxton in [58] shows that marking menus are very efficient for a small number of commands that require a screen position as input.

The nets and palettes are displayed in tabbed windows (called *binders*), where users have a lot of freedom in rearranging the binders, dragging tabs to other binders or creating a new one. The user interface of CPN Tools tries to leave as much freedom to users as possible and support different individual workflows. The nets in Figure 10 show the included example **Hierarchical Protocol**, with the top level net in the top left binder.

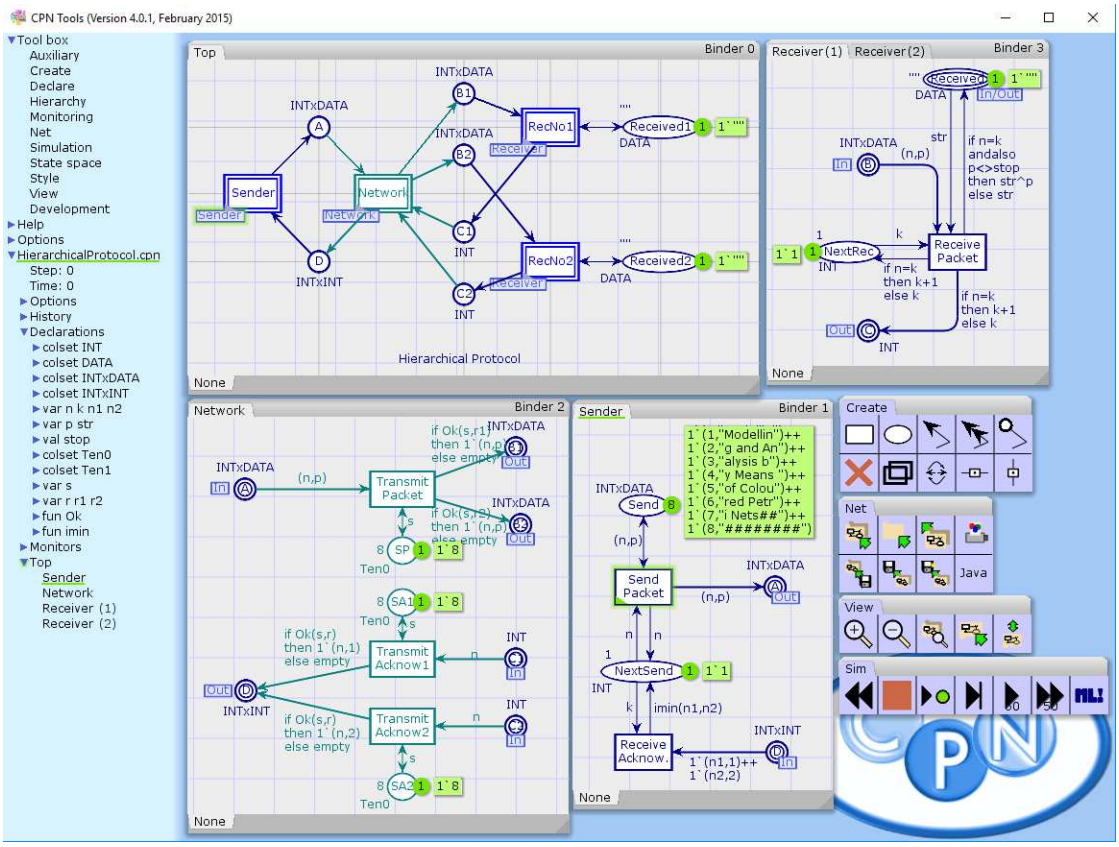
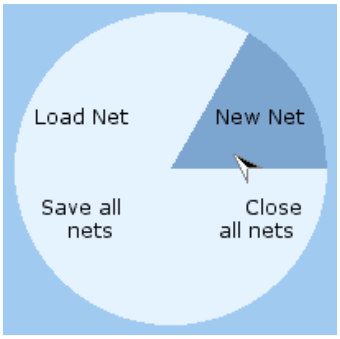


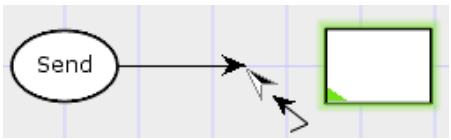
Figure 10: CPN Tools with Index on the left, some binders with nets, and some toolbox palettes on the bottom right.

That net contains substitution transitions (recognized by the double borders) for the **Sender** (bottom right binder), **Network** (bottom left binder), and two **Receiver** nets (top right binder with two tabs). Each of these subpages have input and output places with the same names as in the top level net (again recognizable by the double borders).

As previously mentioned, the user interface supports multiple different workflows. For example to create new model items like places, transitions, and arcs the user needs to click the desired item in the create toolbox and then click inside the net in the position, at which the item shall be created. For arcs, the start and end points are selected separately by two sequential clicks (depicted in Figure 11b). Another workflow is to open the marking menu in the position of the net, where the item shall be created and execute the according action. Again for creating arcs the end point has to be selected now. If the arc should not go in a straight line from start to end point, clicking on free canvas space will create intermediary points through which the arc will go. Alternatively, the arc can be bent after creation by dragging an arbitrary point on its line to another position. An intermediary point will be created at the position the mouse button is released.



(a) Marking menu



(b) Select end point of new arc

Figure 11: Overview of CPN Tools' editor features

The two example workflows essentially differ in two things. They are started either via marking menu or via toolbox. Additionally, when starting from the marking menu, the cursor is reset to normal selection mode after workflow completion. If on the other hand starting from the toolbox, the cursor stays in creation mode until it is explicitly deactivated or changed to another item in the toolbox.

In CPN Tools' editor there are no extra windows to display or enter information about the designed model. All names, expressions, conditions, and variables are shown and edited in the designer directly beside or inside the model items themselves. This eliminates the distance between a specific property and its editing interface and keeps the user's focus on the model. All the enumerated properties can be changed directly in the editor.

2.4.3 Checks and Code Generation

Syntax checks are run automatically as background tasks, while the nets are edited. The checks are optimized and run only for the parts of the net that are affected by changes. The results are visible via orange (unchecked or in progress) and red (for errors) underlines of the net's names in binder tabs, or outline glow of arcs, places, and transitions. Further, speech bubbles show error messages for the failed syntax checks next to the corresponding items. Figure 12 shows the red outline glow for the **Sender** substitution transition in the top level net, the red underlines for the **Sender** net binder, and the error speech bubble for the **Send Packet** transition within the net.

Code generation is coupled with the syntax checks. When nets are syntactically correct, the simulation code is generated automatically in a background task. Furthermore, code generation is run incrementally, such that even when some parts of a net contain syntax errors, others that are correct can still be simulated.

Additionally, Simonsen presented a tool called *PetriCode*[14] in [83], which takes models of CPN Tools and generates code via templates. PetriCode requires the CPN model to be

[14] `http://kentis.github.io/petriCode/`

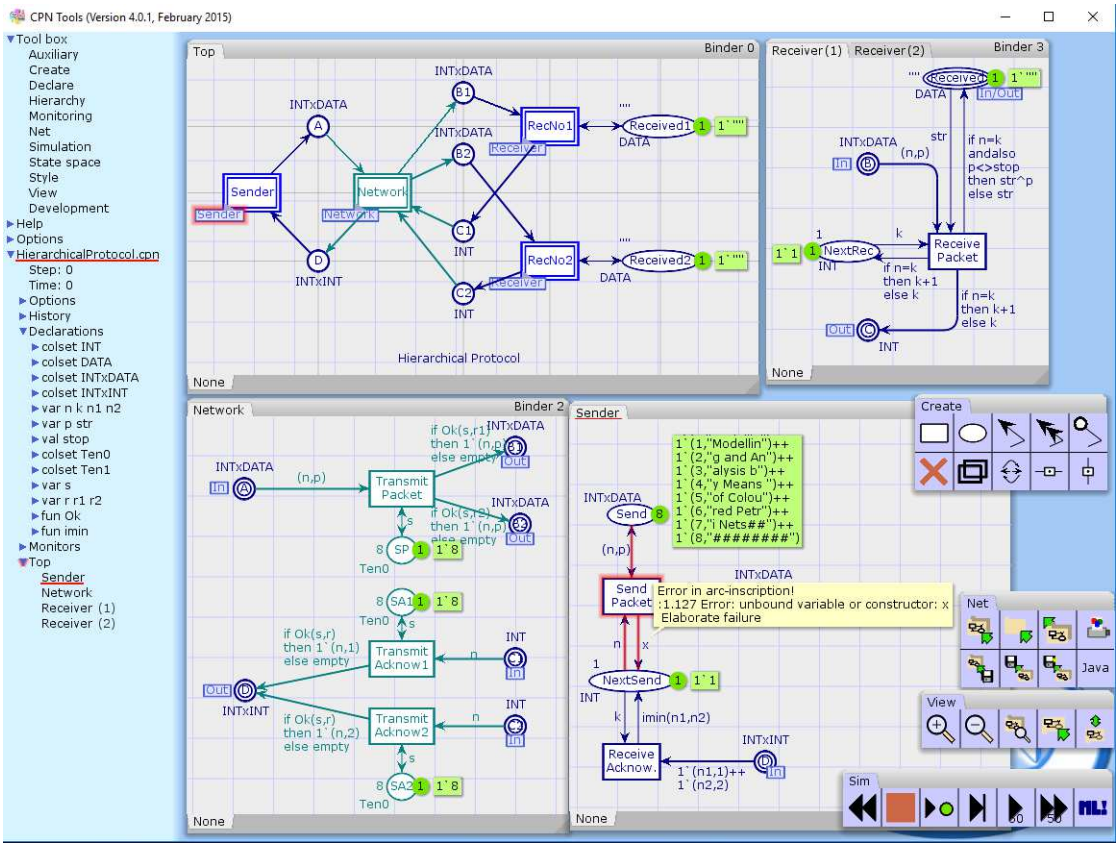


Figure 12: The syntax check feedback in CPN Tools.

annotated with *pragmatics* [82], which are annotations that guide the code generation. Currently, there are templates available for the JVM-based programming language *Groovy*[15].

2.4.4 Simulation and Analysis

The generated code is used for simulating designed nets. Users can control simulation with the *Sim* palette shown in Figure 13. The controls are similar to those of a music or video player's and support from left to right:

Rewind to reset the net to its initial marking.

Stop a running simulation.

Single step to execute the selected enabled transition or any enabled transition of a page if none is selected.

[15] http://www.groovy-lang.org/

Play executes a specific number of steps that can be defined by the user, and the user interface will be updated after each step.

Fast forward also executes a specific user-defined number of steps, but the user interface will only be updated after all steps are completed.

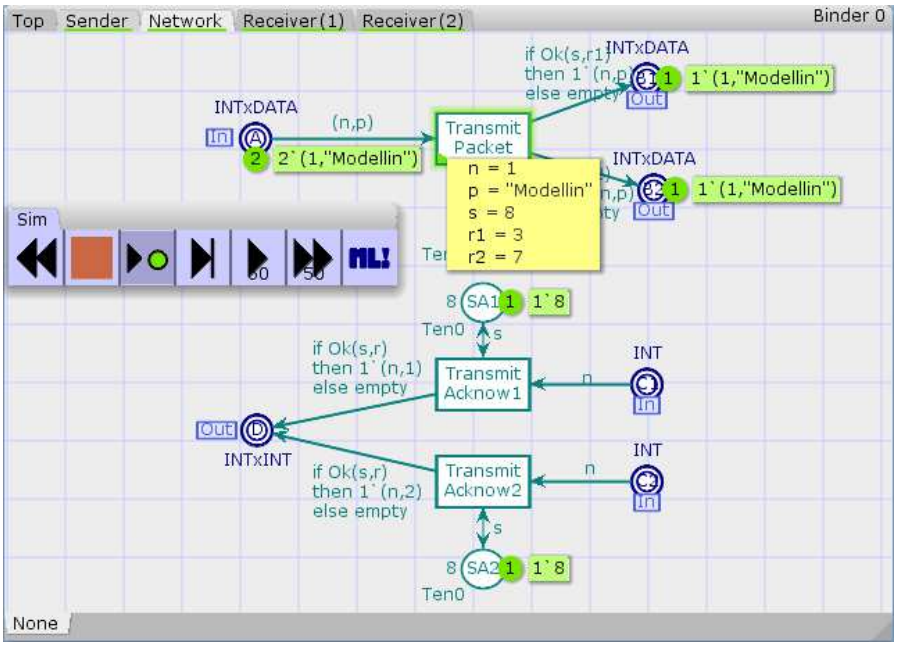


Figure 13: Simulation controls and feedback in CPN Tools.

During simulation, enabled transitions are visualized with a green outside glow, and pages with enabled transitions have a green underline in their binder tabs and in the index. The token count at places is shown in green circles, and their contents are displayed next to them in light green boxes. Bound variables are shown in yellow boxes with their values. They all are updated during the simulation as described before.

Additionally to simulation, state space analysis is supported by CPN Tools, to aid users in analyzing the designed nets. There are constraints as to which nets can be analyzed. For example all places and transitions must have unique names, and all arcs must have inscriptions, which are responsible for selecting the tokens from the input places. While these constraints need not be met for simulation, state space analysis depends on them. Each of the calculated markings has a unique number and users can switch from such a marking to the simulator and inspect the marking and enabled transitions and thus scrutinize the condition under which this marking came to be. Also the inverse direction to switch from the simulation to the state space is possible. Among other things this mechanism is useful for inspecting states that result from errors in the designed net.

Furthermore, [89] introduced simulation-based performance analysis of CPN models. *Monitors* can be used to control and inspect a model without modifying the model's

structure. There are several types of monitors like *simulation breakpoint*, *place contents*, *transition enabled*, or *data collection* monitors. All of them can be used to observe the model, but especially the latter is useful for collecting statistical data about the model over the course of a simulation. It is also possible to run multiple independent simulation repetitions and collect data over the course of all of them to increase the accuracy of calculated confidence intervals.

2.5 Extensible Coordination Tools

The Extensible Coordination Tools (ECT)[16] [7] are a set of plugins for the Eclipse platform that build an environment to work with Reo [6] connectors. These plugins include development tools for design and verification as well as runtime engines that support execution of Reo connectors. Following next, a brief overview of Reo will be given and the development tools comprising the ECT will be explained subsequently.

2.5.1 Reo

Reo [6] is a coordination language that models coordination using connectors. Every connector models a specific coordination pattern between connected components that are running their I/O operations through the connector. The individual components do not need to know about any of that. Reo emphasizes that only the coordination logic is modeled in a connector, not any application logic. This is called *exogenous coordination* [5], which means "coordination from outside" of the coordinated entities.

Channels are the basic building blocks used in Reo. They represent a point-to-point communication imposing a constraint on the data flow between their two ends. The ends can be either source or sink ends. Source ends accept data into and sink ends dispense data from the channel. The type of the channel is defined by the user and it specifies the channel ends. There are some standard channel types (e.g. synchronous, asynchronous, FIFO, filter, drain), which can be used to build sophisticated connectors.

Channel ends can be logically joined at so called *nodes* to create complex connectors. Reo distinguishes *source nodes* (only source ends), *sink nodes* (only sink ends), and *mixed nodes* (both source and sink ends). At nodes the data from sink ends is duplicated to each connected source end. When there are multiple sink ends, one is selected non-deterministically.

2.5.2 Graphical Editor

The graphical editor depicted in Figure 14 is arguably the most important tool, because here the designers model their coordination system as Reo connectors. The depicted connector **Order** has two sources (nodes **a** and **b**) and one sink (node **c**). Nodes **a** and

[16] Since the start of this thesis the ECT have been deprecated (see `http://reo.project.cwi.nl/v2/tools/`), because the project wants to lose dependencies on Eclipse and Adobe Flash.

b are connected by a *SyncDrain* channel, **a** and **c** by a *Sync* channel, and **b** and **c** by a *FIFO* channel. The sources of **Order** are each linked with a writer and the sink is linked with a reader. The connector orders all incoming data by synchronously taking a datum from both writers. Since nodes **a** and **b** are connected to each other by a *SyncDrain* channel, both of them must have a datum at the same time. The ordering of their data is implemented by having a *Sync* channel between **a** and **c**, and a *FIFO* channel between **b** and **c**. This causes the data from **a** to be delivered to **c** at the same time as the data from **b** is put into the *FIFO* channel. Once **c** has delivered the data from **a** to the reader, the data from the *FIFO* channel can be accepted and delivered to the reader.

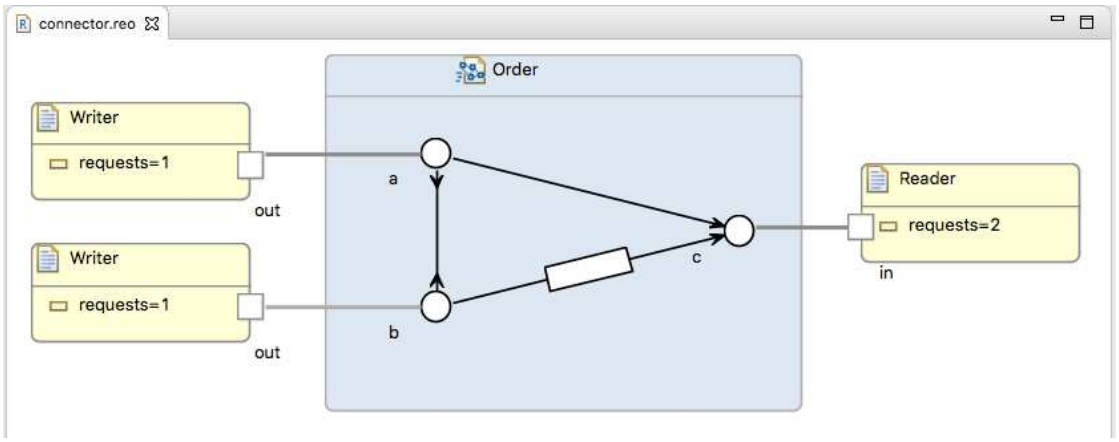


Figure 14: The graphical editor in ECT showing an ordering connector.

Different components and channels are presented to the user in a *Palette* window (depicted in Figure 15). From there, the users can see available parts and select which ones to use in their model via the select and click mechanism already described in previous sections.

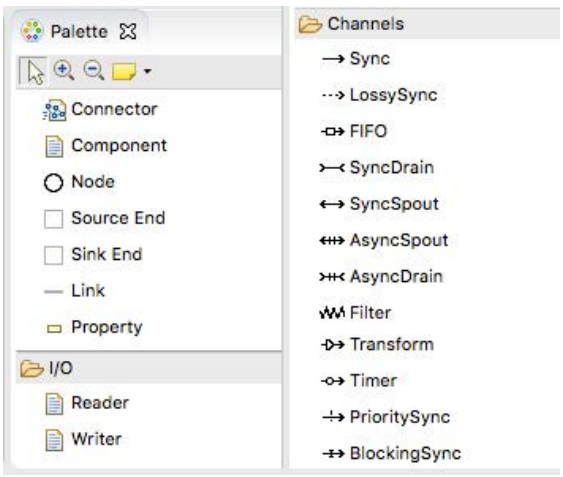
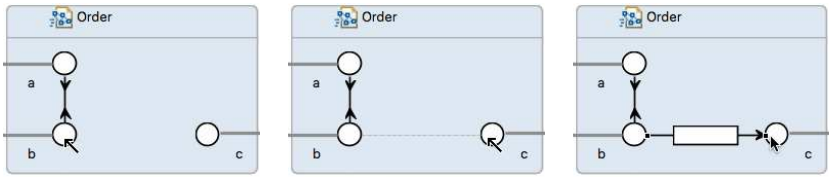


Figure 15: The palette for the graphical editor in ECT.

Figure 16 shows the individual steps of how a new channel is created. After choosing the channel type in the palette the channel is created by clicking the source node **b** (Figure 16a), dragging the mouse to the sink node **c** (Figure 16b) and then releasing the mouse button to commit the creation (Figure 16c). Items can be intuitively selected, rearranged, and resized in the editor via grabbing handles and descriptions are edited also directly in the editor.

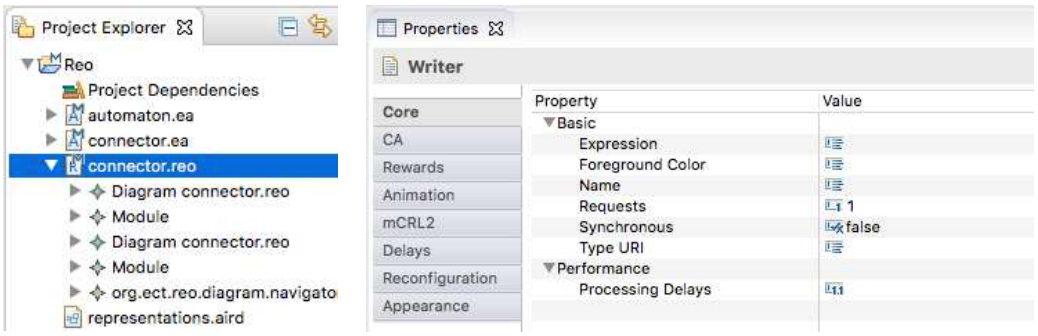


(a) (b) (c)

Figure 16: Creating a new channel

The *Project Explorer* view (see Figure 17a) gives a hierarchical overview of all open projects in the current Eclipse workspace. The items that have been placed in the graphical editor are presented in a tree structure, so that users can drill down from top to bottom to inspect the items they are looking for.

Information about the current selection is displayed in the *Properties* view (cf. Figure 17b). The selection can be one or more items in the editor or an item in the *Project Explorer* view. Furthermore, all details of the selected items can be edited in the *Properties* view.



(a) Project Explorer (b) Properties view

Figure 17: Overview of ECT's features

2.5.3 Animation

ECT includes an animation view (depicted in Figure 18) to visualize the individual steps of the coordination within a connector. For this an Adobe Flash animation of the selected connector is generated and displayed in the window. On the left side there is a list of animations that can be started by clicking on them. For more complex connectors

there can be multiple different animations in the list, which depend on the coordination modeled by the connector and the connected coordinated components (displayed as writers and readers). The data tokens are shown as blue and green pentagons and the synchronous data flow is highlighted violet-blue along the channels. The red triangle marks the sink of node **c** that is non-deterministically not executed in the current step of the coordination sequence in Figure 18.

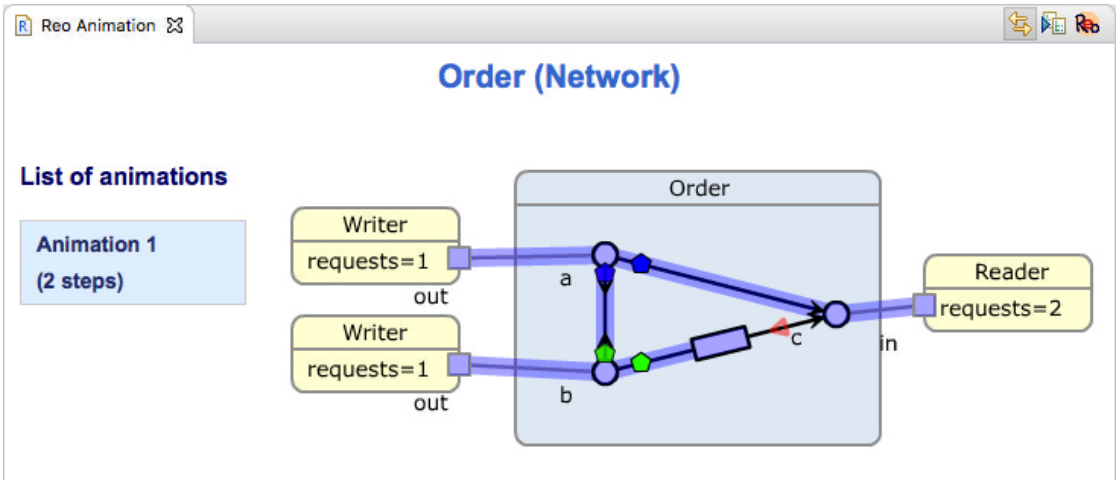


Figure 18: The animation view of the order connector in ECT.

2.5.4 Model Checker

For the support of model checking the designed connectors can be converted to so-called *Constraint Automata* [12]. These Constraint Automata are semantic models that describe the synchronization sequence executed by the connectors. Furthermore these automata support composition as one can compose the automata of the used channels to derive the automaton of a connector. The actual model checker uses logic similar to CTL and was implemented as a model check view for ECT by Klüppelholz and Baier [45]. It also has support for the verification of connector properties. For example, it can verify that a connector can never get stuck in a deadlock, or it can check whether two connectors have equivalent behavior.

2.5.5 Performance Analyzer

To analyze the performance of connectors, ECT supports another semantic model, namely *Quantitative Intentional Automata (QIA)*, which extend Constrained Automata with quantitative properties like average delays of data flow between and arrival rates at ports. Subsequently, the QIA are converted to continuous-time Markov chains [40], which in turn are used as input to the PRISM model checker [59]. This approach supports reasoning about average response times, whether deadlines are met, or other best and worst case scenarios. Furthermore, QIA support composition, like Constraint Automata.

2.5.6 Code Generator

The ECT support generation of executable Java code. This is achieved by first converting the connectors to Constraint Automata, like with the model checker, which in turn is used to create executable Java code. Jongmans et al. extend the features of the ECT with the possibility of generating code for the orchestration of WSDL web services [44].

2.6 Papyrus

Papyrus[17] is an Eclipse Modeling Framework-based tool for graphical UML2 modeling [36]. It also supports SysML diagrams (cf. Figure 19) via the *Papyrus SysML 1.4*[18] extension.

2.6.1 SysML

SysML stands for *Systems Modeling Language* and it is managed by the *OMG* [67]. SysML started out as an open source project, was adopted by the OMG as specification *OMG SysML* in 2006, and was published as ISO standard ISO/IEC 19514 [43] in 2017. SysML extends UML2 [69] as a profile, which means it can strictly extend UML2 and must not contradict any of its semantics.

The goal of SysML is to provide a modeling language not only for software systems, but systems in a more general and broader sense. Thus, it strives to supply systems engineers with the tools to analyze, design, specify, and verify complex systems [39]. Further, like with the ASM method (cf. Section 2.1.1), SysML aims to improve data exchange between different tools, and to bridge the gap between different engineering disciplines [67].

To reach that goal, SysML incorporates four types of diagrams, which are also known as the four pillars of OMG SysML [39], describing:

- system requirements,
- behavior,
- structure, and
- parametric relationships.

As can be seen in the SysML diagram taxonomy in Figure 19, SysML uses some of the existing UML2 diagram types unchanged, some diagram types are modified, and two new diagram types are introduced, which are derived from existing UML2 diagram types.

[17] https://www.eclipse.org/papyrus
[18] https://www.eclipse.org/papyrus/components/sysml/0.10.0/

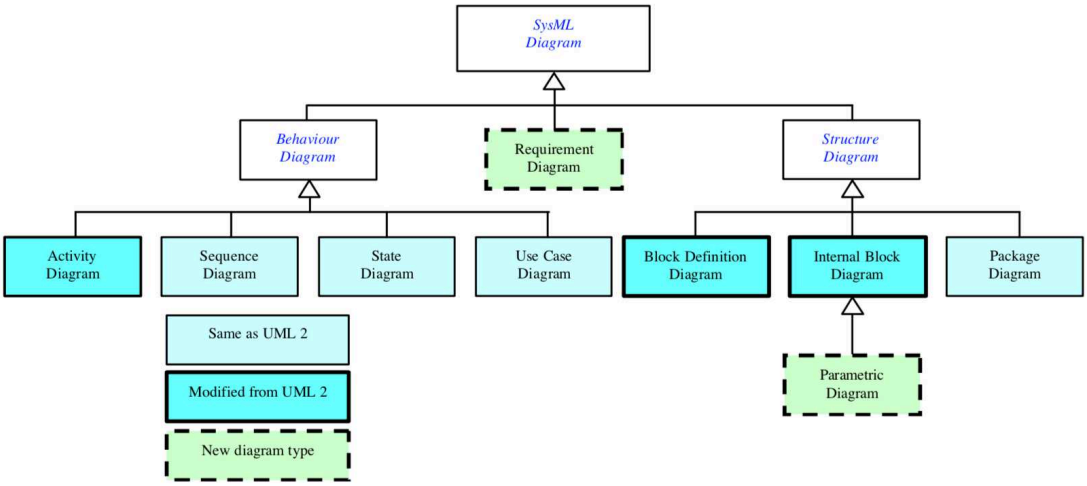


Figure 19: The diagram taxonomy in OMG SysML. (Figure taken from [39])

Requirements Diagram

The *Requirements Diagram* is introduced to capture system requirements, map relationships between them, and most importantly connect the system model to text-based requirements management tools. Users had to rely on use case diagrams for the description of high-level requirements when using only pure UML2.

Each requirement has an identification and a textual description. Requirements can be decomposed into more fine-grained requirements using the UML containment relationship. Requirement derivation can be modeled with the *deriveReqt* dependency, to explicitly show that a requirement is derived from another one. *Rationale* concepts can be added to justify design decisions and provide contextual information. [39, 67]

Further, the Requirements Diagram supports cross connections to other diagrams in the model. The *refine* dependency is used to show that other SysML model elements (e.g. Activity or Use Case Diagrams) are refinements of a requirement. Model elements that are designed to satisfy a requirement are linked to the specific requirement via the *satisfy* dependency, and test cases are linked to requirements with the *verify* dependency. [39, 67]

Parametric Diagram

In UML2 it was not possible for users to precisely describe constraints on system parameters in a straightforward way. The *Parametric Diagram* was introduced to cope with that. It is a specialization of the *Internal Block Diagram*, and constraints on system parameters are described with *constraints blocks*. They contain a mathematical formula to model the actual constraint, and the needed parameters of the formula. [39, 67]

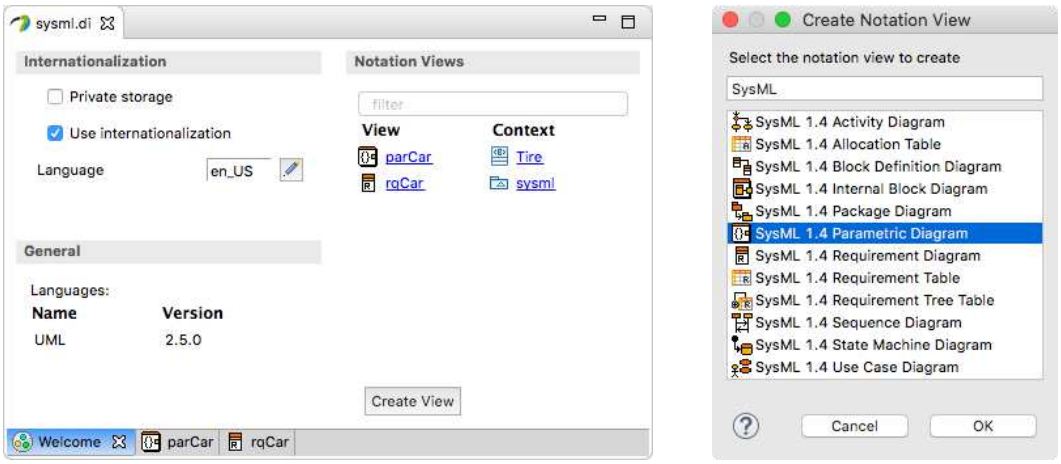
The constraint blocks are used to: [67]

- describe the physical properties of the modeled system,
- express performance and quantitative constraints on system parameters,
- express relationships of functions within the model, and
- specify relationships between different variables of the system model.

Further, the constraint blocks are used to build a coherent network of constraints on system parameters to enable a connection of the SysML model to engineering analysis tools and analysis models, like performance and reliability models. These constraint networks enable analysis tools to find critical performance parameters and their relationships to other parameters, and they also make it possible to track these critical parameters over the entire system life cycle. [67]

2.6.2 Graphical Modeler

The project welcome screen (shown in Figure 20a) shows some general information about the SysML model and its views and provides a button to create new views (see Figure 20b). The tab bar on the bottom of the overview window shows a tab for the welcome screen and other already open views. If a view is not shown as a tab it can be opened by clicking it in the *Notation Views* section in the right column of the welcome screen.

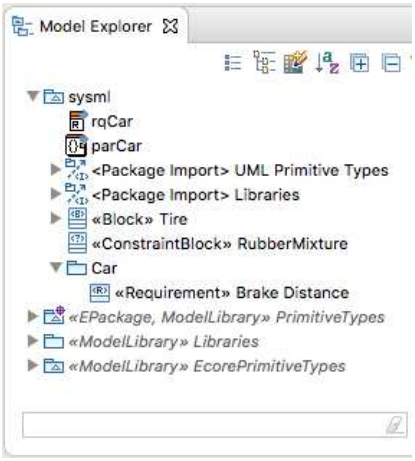


(a) Papyrus project overview (b) Create new view

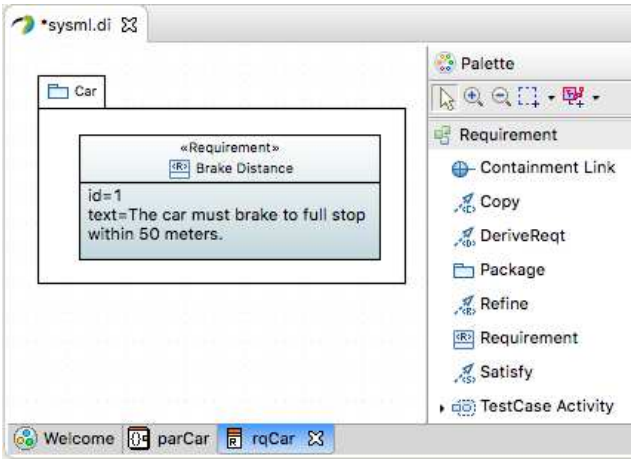
Figure 20: Papyrus SysML project information and views

The Model Explorer provides an overview of the current model as a tree (cf. Figure 21a). From there, users can inspect the model hierarchy and open specific elements.

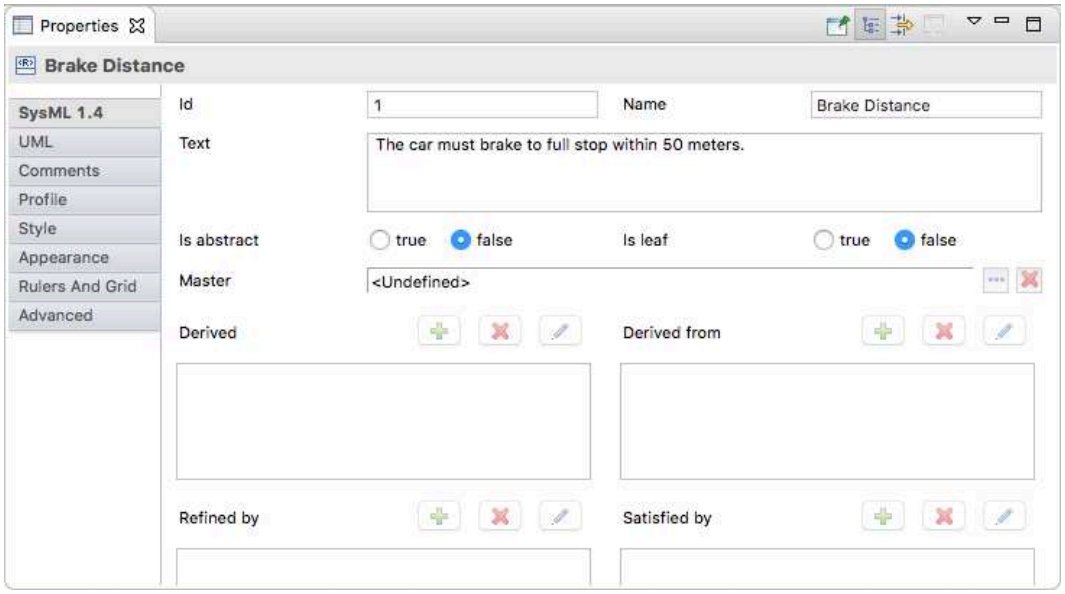
When creating a new view, it opens automatically in a new tab of the Papyrus SysML model and shows the editor with a palette on the right (see Figure 21b). The palette is context-sensitive and shows only the items relevant for the currently open diagram type. From there, new model elements can be placed in the diagram via the drag-and-drop or select-and-click workflows already described in previous sections. Right after the placement of the new item, its name is selected for editing and can be entered right in the editor. Users can edit specific information of the currently selected model element via the context-sensitive properties view (shown in Figure 21c). Some of the properties can also be edited directly in the editor.



(a) Model Explorer view

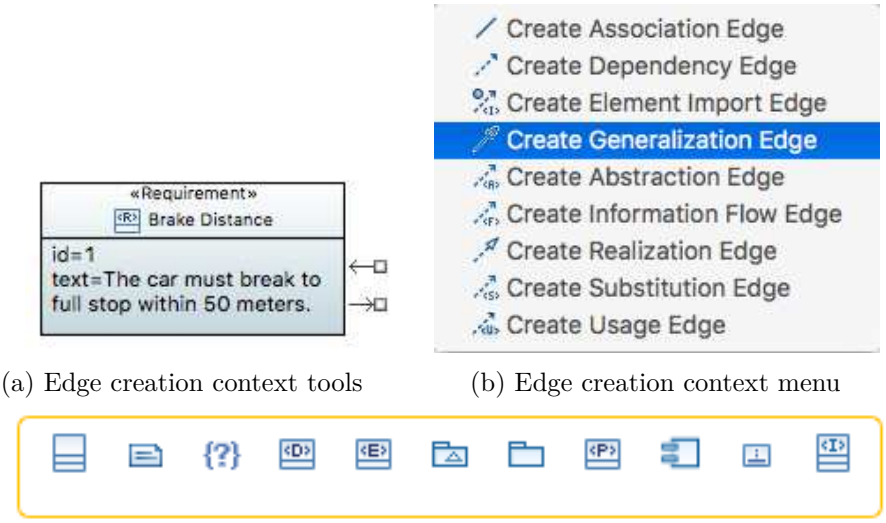


(b) Graphical editor with palette



(c) Context sensitive Properties view

Figure 21: Papyrus views of the model



(a) Edge creation context tools (b) Edge creation context menu

(c) Shape creation context tools

Figure 22: Overview of Papyrus editor features

Figure 22a shows the context-sensitive edge tools that appear when hovering above an item. An edge in incoming or outgoing direction to another item can be created by dragging the respective icon to the item that shall be connected. Then the context menu in Figure 22b will be presented, where the actual type of the edge that will be created can be chosen. After that, the name of the newly created edge is selected and can be edited right in the editor. When hovering with the mouse cursor for a few seconds, the context-sensitive shape creation tools shown in Figure 22c are displayed, from where different shapes can be created right in the editor without the palette.

2.6.3 Model Execution

A requirement for the ability to execute models is a formal specification of the semantics for the UML/SysML parts that are to be executed. The OMG standardized the execution semantics of a subset of UML called Foundational UML (fUML) [70]. SysML inherits the execution semantics, since it is a profile of UML. Another important part to enable model execution was standardized as Action Language for fUML [68]. It formally specifies the notation to represent structural model elements and their actions.

Based on these standards Tatibouët et al. presented a methodology for the formalization of UML profile semantics in [85]. As they rely entirely upon the standards, their approach ensures that the models using the formalized profiles are directly executable.

Papyrus provides model execution through an extension called Moka[19]. Moka includes execution engines based on either fUML [70] or Precise Semantics of UML Composite Structures (PSCS) [71]. The latter extends fUML with the specification of formal

[19] https://marketplace.eclipse.org/content/papyrus-moka

semantics for composite structures in UML, e.g. an fUML class with internal structure. Using one of these execution engines, the model can then be run interactively, including the ability to debug the model, set breakpoints, or inspect variables during the run. The model is also animated while it is being executed.

2.6.4 Code Generation

Papyrus SysML supports code generation via extensions. For example, the Papyrus Software Designer[20] is another extension to the Papyrus UML editor that provides the ability to generate source code for the C++ and Java programming languages. Additionally, it is also possible to reverse-engineer the model from Java source code. Another possibility is the template-based Acceleo[21] extension developed by the OMG inside the Eclipse project. It is an implementation of the *MOFM2T* specification [65] by the OMG. In [84] the authors use it to generate Simulink models from SysML models.

2.7 Peer Model Monitoring Tool

The Peer Model Monitoring Tool was developed by Csuk as the contribution of his diploma thesis [26]. It is a post-mortem analysis tool of PM execution traces (see Figure 23). It automatically creates a visualization of the traced Peer Model in the *Main View* and a *Timeline* of the execution trace. For that two input files are analyzed. The first file contains the static structure of the model, and the second contains the trace with the runtime information logged during Peer Model execution. Users can then investigate the state of the model at every single point in the timeline with help of the provided animation and the controls in the *Sidebar*.

Since the input file with the static model structure does not contain positional data on the individual Peer Model items, the PM Monitoring Tool automatically calculates a layout for the items and routes the links between containers and wirings. This algorithm minimizes link crossings to provide an easy-to-understand view of the model. The static view of the Peer Model shows the complete model in one pannable window to enable users to easily trace the transported data. Peers can be collapsed to hide their contents when they are not needed. The dynamic view, which becomes available after loading an input file with the execution trace, is presented as an overlay on top of the static model. Users can step through the events individually and thus trigger animations of the entries moving along the links that transport them through the model. Entries are visualized as color-filled circles that vary in size and opaqueness depending on some system properties (see 24a). Additionally, the entries feature a detail view showing property keys and values that can be activated for each individual entry by the user (cf. 24b).

[20] https://wiki.eclipse.org/Papyrus_Designer
[21] https://www.eclipse.org/acceleo/

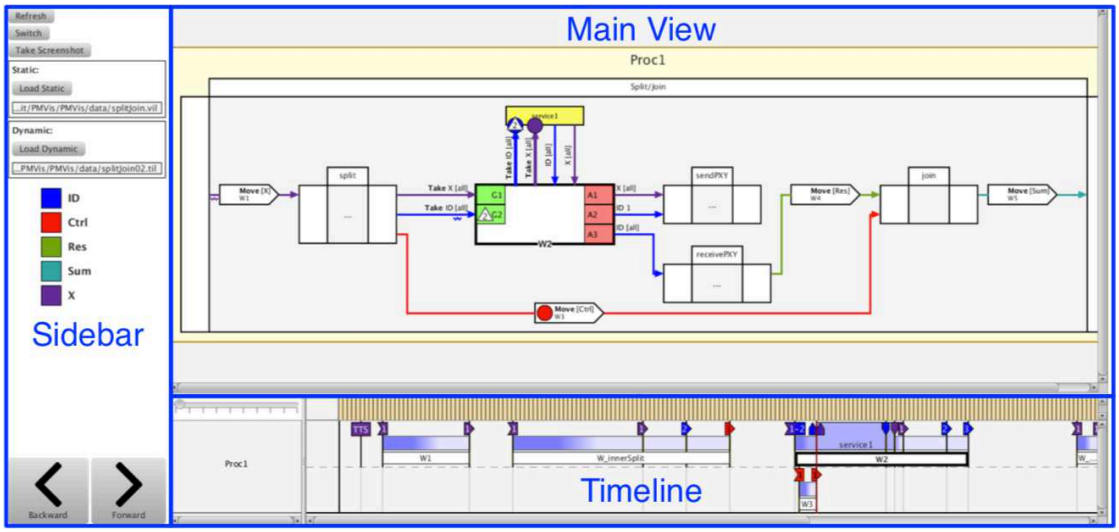


Figure 23: The window of the Peer Model Monitoring Tool, with the main sections highlighted (taken from [26]).

2.8 Discussion

In this section we compare the related tools regarding their respective features for graphical modeling based on the selection criteria from the beginning of this chapter.

As most of the compared modelers are based on Eclipse (AsmEE, BPEL Designer Project, BPMN2 Modeler, Extensible Coordination Tools, and Papyrus), their user interfaces have large overlaps. That primarily becomes apparent as they use the infrastructure provided by the Eclipse project, like:

- the frame application called *workbench*,
- *workspaces* to organize projects (on disk or just logically),
- *perspectives* to provide plugin-specific views,
- the *model explorer* to display the model in a tree view,
- the *properties* view to edit details of different elements,
- the *palette* view to provide elements to place in the models, and
- the *problems* view to inform about errors in the model.

CPN Tools is not based on a framework but developed from the ground up with some unusual controls and user interfaces. On one hand this is owed to the encouragement of using two or more pointing devices, like a mouse and a trackball. On the other hand the used marking menus [58] are a little unconventional in current software, as most operating systems, applications, or tools present context menus as regular linear menus.

Table 2 attempts to give an overview of the related tools comparison at a glance. Columns correspond to the analyzed systems, while the rows represent the compared features. Each element shows the related tool's fulfillment rating of the feature.

Feature	AsmEE ASMETA	BPEL Designer	BPMN2 Modeler	CPN Tools	ECT	Papyrus	Peer Model Monitoring Tool
Graphical Editor	○○	●●	●●	●●	●●	●●	○○
Automatic Layouting		○○	○○	○○	○○	○○	●●
Context Menus		●○	●●	●○	●○	●●	●○
Drag and Drop							
Creating		●○	●○	●●	○○	●○	
Moving		●○	●○	●●	●●	●○	●●
Resizing		●○	●○	●●	●●	●●	
Lasso Selection		●●	●●	○○	●●	●●	
File Export		●○	●○	○○	●●	●○	
Information Overlay		●●	●○	●●	●○	●○	●●
Inline Text Editing		●○	●○	●●	●○	●○	
Item Palette		●●	●●	●●	●●	●●	
Hierarchical Overview		●●	●●	●●	●●	●●	
Keyboard Shortcuts		●●	●●	●●	●●	●●	
Code Generation	●○	○○	○○	●○	●●	●●	○○
Deployment Support	○○	●●	●●	○○	○○	○○	○○
Error Checking	●○	●○	●○	●●	●○	●○	●○
Simulation	●●	○○	○○	●●	●●	○○	●○
Model Checking and Verification	●●	○○	○○	●●	●●	○○	○○
Performance Analysis	○○	○○	○○	●●	●●	○○	○○

Legend: ●● completely fulfilled ●○ partially fulfilled ○○ not fulfilled

Table 2: Feature comparison of related tools

As the graphical editor feature of the compared tools is the most significant for this thesis, it is split into subcategories to be compared in higher detail. The graphical editor's subfeatures are each compared on their own row, but their feature names are indented. Since AsmEE currently does not feature a graphical editor, it does not have ratings in the subfeature rows. The Peer Model Monitoring Tool also does not support creating models. However, it has ratings in some of the graphical editor subfeatures, because it supports customizing the display of the monitored models. The following sections discuss the similarities and differences of all examined tools in detail.

2.8.1 Graphical Editor

The graphical editors are the main focus point of the related work, because they also are the primary point of user interaction and, as such, probably the most important expected outcome of this thesis.

AsmEE is the only selected editor that currently does not have a released graphical editor, but the team behind the ASMETA toolset are working on one, which is still experimental and not yet available for the public. The Peer Model Monitoring Tool is, as the name already reveals, a monitoring tool that can be used to analyze logs of PM implementations post mortem, and as such it also does not have a graphical editor.

All the other tools feature graphical editors with similar user interfaces designed for interaction with a pointing device.

Automatic Layouting

Only the Peer Model Monitoring Tool automatically calculates a layout for the model and positions the items in an optimal way such that the crossings of links are minimized. The other tools rely on users to position the model items.

Context Menus

They all provide context menus, usually activated via a right mouse click, to access actions related to the current context. The reason only BPMN2 Modeler and Papyrus are marked as completely fulfilled is that they also feature contextual toolbars, which are displayed when hovering the mouse pointer above or in the vicinity of an item.

Drag and Drop

Drag and drop gestures for the pointing device are in some kind supported by all editors, but to get a more detailed comparison, this feature has also been subcategorized into creating, moving, resizing, and selecting items.

Only CPN Tools supports creating all items by drag and drop. BPEL, BPMN2, and Papyrus support that for some but not all possible items, and ECT do not support drag-and-drop item creation at all. The latter only features a select-and-click workflow, which also means that there is no preview before the item is actually created.

For moving and rearranging existing items in the designed model, all tools support a drag-and-drop workflow. However, just CPN Tools, ECT, and the PM Monitoring Tool have complete fulfillment ratings, because they display the moved items at all times just as they will be displayed after dropping, while BPEL, BPMN2, and Papyrus have a somewhat limited preview and thus only partial fulfillment ratings.

The same arguments apply for the fulfillment ratings of resizing existing items. CPN Tools, ECT, and Papyrus have full preview during the resize, while BPEL and BPMN2 have only limited preview with the new outlines.

Selecting multiple elements by dragging a "selection rectangle" around them is supported by BPEL, BPMN2, ECT, and Papyrus. CPN Tools on the other hand does not support the selection of multiple elements, neither by selection rectangle nor by clicking with a modifier key.

File Export

Of the compared tools with a graphical editor, only CPN Tools does not support exporting the model as graphics files. All others support at least the export as pixel graphics files. But as pixel graphics have serious limitations when being scaled, only ECT has a complete fulfillment rating, because it also can export the models as vector graphics files, which support scaling the model to all possibly needed sizes.

Information Overlay

Complete fulfillment of this feature is achieved by BPEL, because all the additional information about individual orchestration items is displayed when hovering above them with the mouse pointer. Even though CPN Tools does not show temporary information overlays in the same way as BPEL, it gets a complete fulfillment rating, because all necessary information is displayed next to the respective items directly in the editor view. The PM Monitoring Tool provides additional detail information directly in the main view via, e.g., a entry detail view and thus also receives complete fulfillment ratings. BPMN2, ECT, and Papyrus fulfill this feature partially, because they have tooltips with limited extra information about the items.

Inline Text Editing

CPN Tools is the only one that allows to edit all texts, descriptions, expressions, and properties directly in the editor and therefore is the only tool with a complete fulfillment rating. All others partially fulfill this feature with the ability to edit at least some information and names directly within the editor, while most of the expressions have to be entered in a separate properties view.

Item Palette

All compared tools feature some possibility to choose new elements to be created in the model. The Eclipse-based editors (BPEL, BPMN2, ECT, Papyrus) have a palette view and CPN Tools has a create toolbox, which shows available item types to be created in the current context.

Hierarchical Overview

All editors have a view that shows a hierarchical overview of the designed model. The Eclipse-based editors have a model explorer view as stated at the beginning of this section

and CPN Tools have an Index, which shows a top level entry for each currently open model. Below that top level entry the model is listed in a tree-like structure.

Keyboard Shortcuts

All the compared tools support some keyboard shortcuts, e.g. the delete key for deleting selected items or key combinations to copy or cut selected items to and paste them afterwards from the clipboard.

2.8.2 Code Generation

The evaluated tools need the ability to generate source code files, which represent the designed model, to fulfill the code generation feature. It is further required that these files can be compiled, interpreted or otherwise executed to obtain a runnable instance of the designed model.

As such, the ASMETA toolset, CPN Tools, ECT, and Papyrus provide code generation either via plugins (e.g. in ECT and Papyrus) or external tools that use designed models as input files (e.g. ASM2C++ of the ASMETA toolset and PetriCode for CPN Models).

BPEL Designer Project and BPMN2 Modeler at the time of writing this thesis do not provide the ability to generate code. Although, in theory it is possible to provide such features analogously to Papyrus, because both tools are based on the Eclipse Modeling Framework. As such it is possible to develop Eclipse extensions that use one of several provided model transformation frameworks[22].

2.8.3 Deployment Support

Deployment means the ability to create a runnable version of the designed model and deploy it to a running server that in turn will execute the model. The BPEL Designer Project supports deploying a designed model to an Apache Orchestration Director Engine server. BPMN2 Modeler can deploy designed models to workflow engines, which can be specified as a target runtime. These target runtimes are implemented as an extension point of the modeler to easily support new runtimes. One such target runtime called jBPM has been developed within the BPMN2 Modeler project.

2.8.4 Error Checking

In software development it is commonly accepted that the earlier in the software development life cycle an error is found, the cheaper the correction of that error is [38]. Therefore, it is paramount in software development to find errors early to keep the cost low. Additionally to highlighting errors in the graphical editor, this section evaluates how identified errors are presented to users and whether there is support for guiding users to the erroneous items.

[22] https://www.eclipse.org/modeling/transformation.php

All modelers, except ECT, perform error checks either when the model is saved, continuously as a background task, or when implicitly requested. Found problems are then visualized in the editor by highlighting the erroneous elements. CPN Tools also highlight the hierarchical entries in the index to give an overview of the errors in the model, and to guide the designers towards the problems. The Eclipse-based tools display the found errors in a problems view to provide designers with an overview. The PM Monitoring Tool checks the validity of input files and presents an error message to its user in case of a problem.

ECT only allows actions in the editor that keep a valid model, which prevents model inconsistencies by design. However, this covers only the actions inside the graphical editor. Input fields for model properties or for simulation options are checked when they are used. That may be fine if incompatible values are entered for simulation and users are notified when trying to start simulation on the same input form. It is, on the other hand, quite confusing if incompatible values are entered as properties of the model elements and such an error is presented to the user out of context without a precise location. In this case, the other modelers provide better guidance to find the actual problem.

2.8.5 Simulation

The support for simulating the execution of the designed model is used by the users as another tool to find design flaws in their models. Simulation means the ability to execute the model and inspect the properties of model elements during the execution.

The ASMETA framework supports the animation of the designed model using AsmetaA, as described in Section 2.1.4. In the AsmetaA window users can monitor all the values of model elements in a tabular form and execute the model one step at a time or multiple steps at once. Further, user input can be generated randomly or entered by the users. Additionally, users can validate the designed models against use case scenarios with AsmetaV (see Section 2.1.5) to check if the model behaves as expected in the different scenarios.

CPN Tools support simulating (cf. Section 2.4.4) the designed models directly in the editor using the simulation controls. Properties of the places and transitions during the simulation are also displayed directly in the editor.

ECT animates the possible flows of connectors in an animation view (see Section 2.5.3) by highlighting the respectively active channels and displaying the flow of tokens between the channels and nodes.

The Peer Model Monitoring Tool (described in Section 2.7) can replay a traced execution of a Peer Model by analyzing two files: the static model information and the execution trace. It provides a graphical view of the model, and users can use the provided timeline control to inspect the data at all moments of the trace. However, since it does not support interactive simulation this feature is only partially fulfilled.

2.8.6 Model Checking and Verification

AsmetaSMV (cf. Section 2.1.5) can be used as a stand-alone tool or integrated in AsmEE to translate the created model into a specification for NuSMV. This approach enables the analysis of CTL and LTL using techniques based on BDD as well as SAT.

CPN Tools features a state space analysis of CPNs (see Section 2.4.4) to help users understand their designed nets. Once the state space is calculated, CPN Tools support switching back and forth between a state space marking and the model.

ECT enables users to convert connectors to so-called Constraint Automata (cf. Section 2.5.4), which represent a semantic model of the connector and serve as input for the model checker.

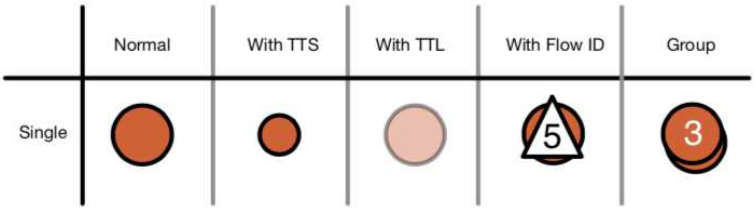
2.8.7 Performance Analysis

CPN Tools supports performance analyses by monitoring model data over the course of one or multiple simulation runs (described in Section 2.4.4). The collected data also contains confidence intervals for multiple runs.

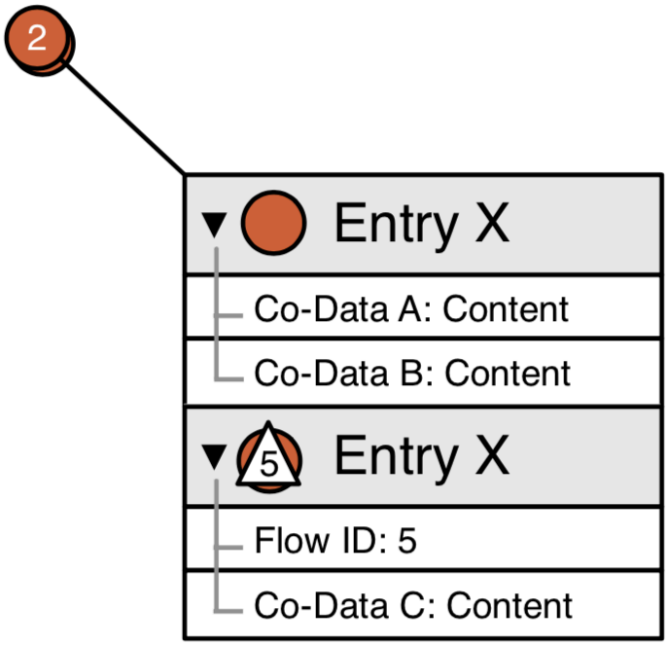
Additionally to Constraint Automata, ECT provides the possibility to convert connectors to Quantitative Intentional Automata, which are an intermediate step to get to continuous-time Markov chains, which are used as input for the PRISM model checker (see Section 2.5.5). The latter can then be used to reason about, for example, average response times, and best or worst case scenarios.

2.8.8 Conclusion

The analysis of related modelers shows a lot of features that are essential for a graphical modeler, like well designed user interaction with simple workflows and the need to offer actions directly in the vicinity of the treated items. It is also important to show information directly in the editor, because this greatly helps keeping the focus on the designed model. Additionally, ideally users get feedback from the editor whether the designed model is, at least syntactically, valid. Thus, they can use such error messages to easily and quickly find the problematic model items and fix them. Finally, generating code directly from the model helps to shorten the time from design to a running application.



(a) Visualization of entries depending on some properties



(b) Detail view of entries

Figure 24: Entries in the Peer Model Monitoring Tool (taken from [26])

CHAPTER 3

The Peer Model

The Peer Model is a design and programming model for concurrent and coordinated processes, which range from low-level implementations of network protocols to high-level models of business processes. It was first introduced in [52] and is under active development by Eva Maria Kühn et al. at the Space Based Computing Research Group of TU Wien [49, 50, 53, 54, 55]. It is based on principles of Petri Nets [18, 73, 75] and Tuple Spaces [35] and partly inspired by Abstract State Machines [17] and Actors [4].

The mentioned literature also provides visualizations for all the used concepts in the Peer Model. These visualizations evolved over time together with the Peer Model with the focus on getting a recognizable and understandable, but also an easily drawable visual language.

The following sections will first present the components that make up the Peer Model and then explain the behavior and dynamic interactions between these components.

3.1 Entry

The fundamental concept for data in the Peer Model is the so-called entry. Such an entry has a type and several properties (see Section 3.2). While system properties are provided by the Peer Model, entries can also have user-defined coordination properties and application properties. The entry type has a unique name by which it is referred to throughout the designed model and which thereby serves as a reference to the entry's properties for the user. System properties are needed by the Peer Model internals and they can be modified by the application to affect how the Peer Model handles the entries. Coordination properties allow to model complex application-specific coordination logic of the designed application going beyond the available system properties. Application properties can be used to store and transfer application data that can only be used by

application logic, but not for coordination, e.g. the binary data output of a service (cf. Section 3.7) needed as input by another service.

3.2 Property

A property in the Peer Model is a value with an associated name (e.g. name "ttl" and value 100). The name can be hierarchical with the dot as level separator (e.g. "ttl.exception") to group several properties together. As can be seen by the two examples, intermediate levels can be properties themselves and in that case have values. However, they may also be used just for structuring several properties and thus do not necessarily need values. Properties can affect entries, links, and wirings and they have a defined data type, which can be one of the following:

Bool	boolean value
Integer	integer number
Float	floating point number
Text	arbitrary text
Address	can point to another peer or a specific container
Duration	an amount of time

As already mentioned, system properties are provided by the Peer Model internals and needed to control how the Peer Model components handle entries. They have default values, but can also be defined by the designer. Some examples of system properties are:

TTL (time-to-live) How long is an entry valid?

The entry is invalid after this time expires, at which it is converted to an exception entry.

TTS (time-to-start) When does an entry first become valid?

The entry is inactive and thus is not considered by the runtime until this time expires.

DEST (destination) Where should this entry be delivered to?

This property holds an address if the entry should be delivered to another peer's container.

FLOW (flow identification) Which workflow does this entry belong to?

This unique identification marks data as belonging to the same workflow. This property is used to group entries together and, for example, explicitly show that they are part of the same request.

3.3 Container

Entries usually reside in containers when they are not currently handled by another component. This shared containers' functionality is based on eXtensible Virtual Shared Memory (XVSM) spaces [23], because their coordination mechanisms are configurable and they can be accessed via an extensible API.

The XVSM spaces provide several coordination mechanisms that determine which entries are selected when accessed. For example, there are predefined coordinators for several strategies:

any	select items without any particular order
random	select items at random
fifo	first in, first out
lifo	last in, first out
key	select items with an unique key
label	select items that have a specific label
linda	select items using Linda template matching [35]
query	select items based on a query, similar to SQL

Furthermore it is possible to provide custom coordinators, which makes it easy to extend for a specific use case.

3.4 Peer

Peers are the eponymous main components of the Peer Model. They have an internal structure and encapsulate behavior, which is not accessible from the outside. Peers are uniquely addressable resources, which allows data to be sent to a specific peer. To receive and send data, peers have a *PIC* (peer-in-container, similar to an inbox) and a *POC* (peer-out-container, like an outbox), respectively.

Peers can also be nested, which means that a peer can contain other peers, which are then called subpeers. This is useful for abstraction and encapsulation as a subpeer has self-contained logic and the parent peer can only interact with a subpeer via its PIC and POC.

The visual representation of a peer is depicted in Figure 25. It consists of a top rectangle with the peer's identifier, which is also used as a part of its address [48]. The rectangles on the sides represent the containers PIC (left) and POC (right), while the content of a peer is drawn inside the center rectangle. The content will be explained in detail in the following sections.

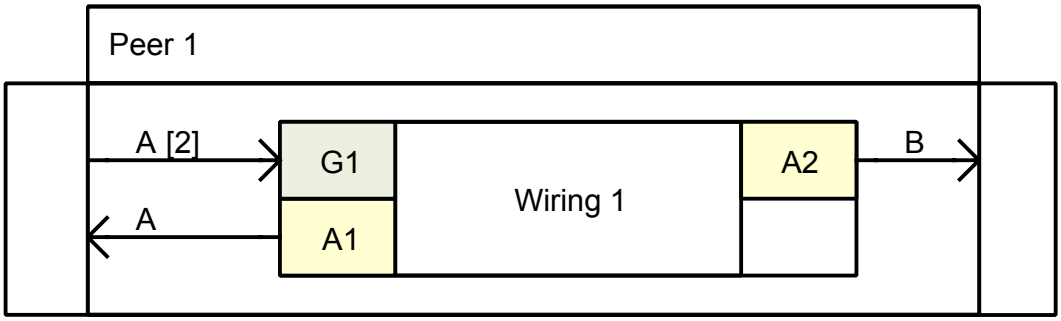


Figure 25: The visual representation of a peer.

3.5 Wiring

Comprised of links and services, wirings are responsible for the active operation of the Peer Model. The visualization of wirings has already been depicted in Figure 25, where the wiring is shown as the content of **Peer 1**. A wiring has a name, which is unique within the parent peer. In the example, the name is **Wiring 1** and it is shown in the center of the wiring.

The individual steps of a wiring's logic are represented by links. They transport entries from their source to their destination. There are two types of links: guards and actions. Each link connects the wiring with a container, except for the guard of an init wiring explained later in this section and special cases explained in Section 3.6. Guards are inputs to wirings, which means they have the container as source and the wiring as destination, and actions are outputs from wirings, meaning their source is the wiring and their destination is the connected container. Links have an identifier, which consists of a letter representing the type, G for guards and A for actions, followed by a unique number within the wiring for the type (cf. **G1**, **A1**, **A2** in Figure 25). At least one guard link is mandatory for every wiring. Additionally, a wiring can optionally have any number of services (explained in Section 3.7) and action links. The graphical representation of a link consists of a so-called base, which is located within the wiring and contains the link identifier, and a connector to the corresponding container. The wiring shown in the figure takes two entries of type **A** from the peer's PIC with guard **G1**. It then writes one entry of type **A** back into the PIC and writes one entry of type **B** into the peer's POC.

Before a wiring starts to execute its logic, the prerequisites specified by the guard links have to be fulfilled. This means that the source containers hold enough entries that satisfy each link's query (explained in Section 3.6). Once those prerequisites are fulfilled, the wiring is said to fire, which means it now executes the guard links in the mentioned order, followed by the services, and concludes with executing its action links again in order of their numbering, all in one transactional step. That means a wiring is either successfully executed as a whole, or it rolls back all the changes it made. The wiring's

links can only access the containers of its parent peer or direct subpeers.

The wiring itself has a wiring entry collection, which is used as a temporary container for the entries handled by the wiring. This means that entries delivered by the execution of guards are stored in the entry collection. Each executed service gets its input entries from there and delivers its output entries to the entry collection. The action links also take entries from the entry collection. At the end of the wiring execution all remaining entries are destroyed.

In addition to regular wirings with guards, services, and actions, there are so-called *init wirings*, depicted in Figure 26. Every peer can have only one init wiring, which is executed as soon as the peer is initialized. They are distinguished from regular wirings by only having one special guard with an asterisk as its name. Init wirings can still have services and actions just like regular wirings.

Figure 26: Visualization of an init wiring.

3.6 Link

The previous section already explained that links are used by their wirings to ensure certain prerequisites. How this is accomplished is explained in the remainder of this section. In addition to the already mentioned entry type of a link there are also other parts that comprise its definition (a complete example is shown in Figure 27).

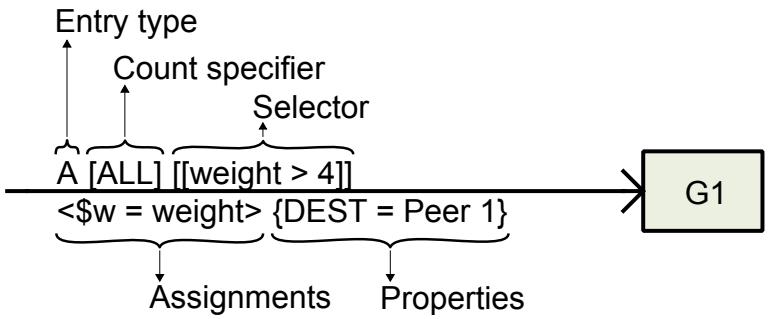


Figure 27: Visualization of a link with all the parts that make up its definition.

Above the connector are the entry type, the count specifier and the selector, which make up the query that defines what entries are handled by the link. Below the connector are assignments and link property definitions, which can manipulate data of the handled entries. The statements used in these fields may not only use fixed values, but also values of previously set wiring context variables (cf. assignments). In such a case the variable names are important, because they also specify the context of the associated value. If the name starts with $$, it denotes a variable automatically defined and provided by the

Peer Model runtime (e.g. $$PID for the identifier of the current peer). If it starts with $, then it refers to a variable that is set explicitly by the user. If the name does not start with $, it refers to a property of the transported entry.

Entry type requires the handled entries to have this type.

Count specifier specifies how many entries this link will transport and can be one of:

1. the special specifier *ALL* to select all available matching entries, which can also be 0 entries,
2. the special specifier *NONE* to require that there is no matching entry in the source container,
3. a bounded range (e.g. $\leqslant 4$ for 0 to 4 entries, or $[3, 5]$ for at least 3 and up to 5 entries),
4. an unbounded range (e.g. $\geqslant 3$ for at least 3 and up to all available entries),
5. omitted, then it defaults to exactly 1 entry.

Selector specifies which entries should be handled by the link. It has to correspond to the coordinators that are used in the connected container (as explained in Section 3.3). For example, the query coordinator provides the following possibilities:

1. An empty selector does not further restrict the considered entries.
2. Property values can be compared using the operators $<$, $\leqslant$, $\geqslant$, $>$, $=$, $\neq$.
3. The result of a comparison can be inverted with the unary boolean negation operator !.
4. The binary boolean operators *AND* and *OR* can be used to combine multiple results.

The selector in Figure 27 specifies that only entries having a weight property value greater than 4 should be considered by this link.

Assignments are a sequence of statements that can modify properties of transported entries and set variables in the context of the wiring to, e.g., store the count of actually selected entries when using ranges in the count specification. These variables can be used in the definition fields of other links that are executed afterwards or in later assignment statements of the current link, which is the reason that the order of statements is important. Additionally, functions provided by the Peer Model runtime can be used to, e.g., create a new flow identification and set it in the transported entries.

Link Property Definitions are an unordered set of statements that set link properties to fixed values or to the value of a previously set wiring context variable, e.g. when the *DEST* property is set, the link will wrap all transported entries in a single new container entry, where the count of the wrapped entries can be set as an entry property.

How a link interacts with the connected container and its wiring is specified by a source operation and a destination operation. In principle, all links must be connected to a container, unless it is specified otherwise in the following descriptions.

The source operations are:

take consumes the entries, which means they are removed from the container after reading. This is the default operation and is used in the previous examples.

read copies the entries and leaves the originals in the container.

create creates the entries specified by the link's definition (entry type, count, assignments). Guard links with this source action are not connected to a container.

The destination operations are:

write puts the entries into the container. This is the default operation and is used in the previous examples.

shift puts the entries into the container and drops entries that do not fit the container due to size restrictions, which depend on the container's XVSM Coordinator.

delete does not put the transported entries into the container. Action links with this destination action are not connected to a container. For example, this operation type is used to explicitly model entries that satisfy guards of wirings, which do not need the entries for its subsequent logic.

3.7 Service

Services are used to interface with and execute application logic from within the PM. Once the wiring has executed all guards, the specified services are executed sequentially. Each service gets the entries of the wiring entry collection (see Section 3.5) as input and also puts its output entries there.

Besides execution of application logic, services can be used to interface with legacy software. For that the service acts as a wrapper around existing software by converting the input entries to the needed input format and converting the output to entries.

Figure 28 shows a wiring **CallWrapA** that takes one entry of type **A** via guard **G1** and then calls the service **WrapA**. The service gets one entry of type **A** as input and is expected to put one entry of type **B** into the wiring entry collection as output, which then is moved by action **A1**.

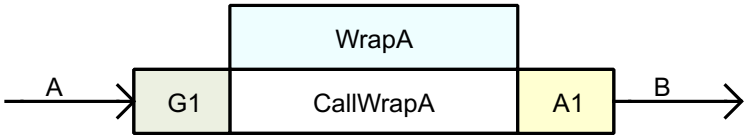


Figure 28: Visualization of a wiring with a service.

3.8 Coordination Example

A peer's behavior is modelled using wirings, links, and services. We will look into that behavior and inner workings of the peer model based on the baker peer of the bakery example from [50]. The coordination challenge in this example consists of individual bakers that get their orders from the bakery and are responsible for producing the dough for the bakery's products.

The wirings of the baker peer (shown in Figure 29) always act on a charge entry in the PIC or an exception entry in case the time-to-live of the charge entry expired. This charge entry is used to create 5 doughs sequentially, where the production of a dough entry is split up into ordering the ingredients (cf. wiring ProduceDough1) and the actual production of the dough using the delivered ingredient entries with the StirDoughService in wiring ProduceDough2. Every produced dough entry is sent to the Bakery peer as soon as it is finished, and when the baker either completes the charge of 5 doughs (cf. wiring ChargeComplete) or the charge times out (cf. wiring ChargeIncomplete), the Bakery peer is notified to deliver the produced doughs of this charge using its flow identifier. The remainder of this section explains the baker peer's coordination logic in detail.

Upon startup of the Peer Model runtime, when the **Baker** peer is instantiated, the **Init** wiring gets executed and action link **A1** creates[1] an entry of type **charge** in the PIC with properties for a new flow identifier *fid*, a dough count (k) of 0 kg, *phase* set to 1, and a time-to-live of 500.

The next wiring, **ProduceDough1**, takes a charge entry from the PIC with a dough count less than 5 and phase equal to 1. Then action **A1** puts the charge entry back into the PIC and sets the phase property to 2. As last action of this wiring **A2** creates an entry of type **sendIngredients** and the properties *fid* with the current wirings flow ID ($$FID), *baker* set to the current peer's identifier ($$PID), *neggs* marking the needed eggs unit count to 4, *nflour* marking the needed flour unit count to 10, and a time-to-live of 10. Additionally, the destination (*dest*) of the created entry is set to Bakery, so that it is sent to the bakery, which can then send the incredients to this peer's PIC.

Wiring **ProduceDough2** starts by taking the charge entry from the PIC with **G1** if it has a dough count less than 5 and phase equal to 2, then it takes 4 entries of type **egg** with guard **G2** and 10 of type **flour** with guard **G3**, as they were ordered in wiring

[1] The action's source operation create is symbolized by the * notation and the not connected line.

ProduceDough1. After all guards are satisfied, the taken entries are put into the wiring's entry collection and the **StirDoughService** is executed, which uses the egg and flour entries to produce the dough. Once the service is finished, action **A1** increases the dough count k on the charge entry and sets it back to phase 1. At last, action **A2** puts the produced dough entry with the current wiring's flow ID and destination Bakery into the POC.

The **ChargeIncomplete** wiring deletes[2] an exception entry with guard **G1**, which is created when the time-to-live of the charge entry expires. The exception entry's *etype* property is set to charge in order to show what the original entry's type is. Additionally, the original entry itself is embedded within the exception entry, but not further needed in this coordination logic. Then action **A1** creates[1] a new charge entry in the PIC with the same properties as in the Init wiring, and action **A2** creates a **deliver** entry in the POC with the current wiring's flow ID and destination set to Bakery, such that the bakery delivers the incomplete dough charge to the oven.

The last wiring **ChargeComplete** deletes[2] a charge entry with dough count equal to 5 from the PIC with guard **G1**. Then actions **A1** and **A2** are the same as in the previous wiring to first create[1] a new charge entry in the PIC and secondly create a deliver entry in the POC to be sent to the bakery, to initiate the delivery of the complete charge of doughs to the oven.

[2] The guard's destination operation delete is symbolized by the not connected line.

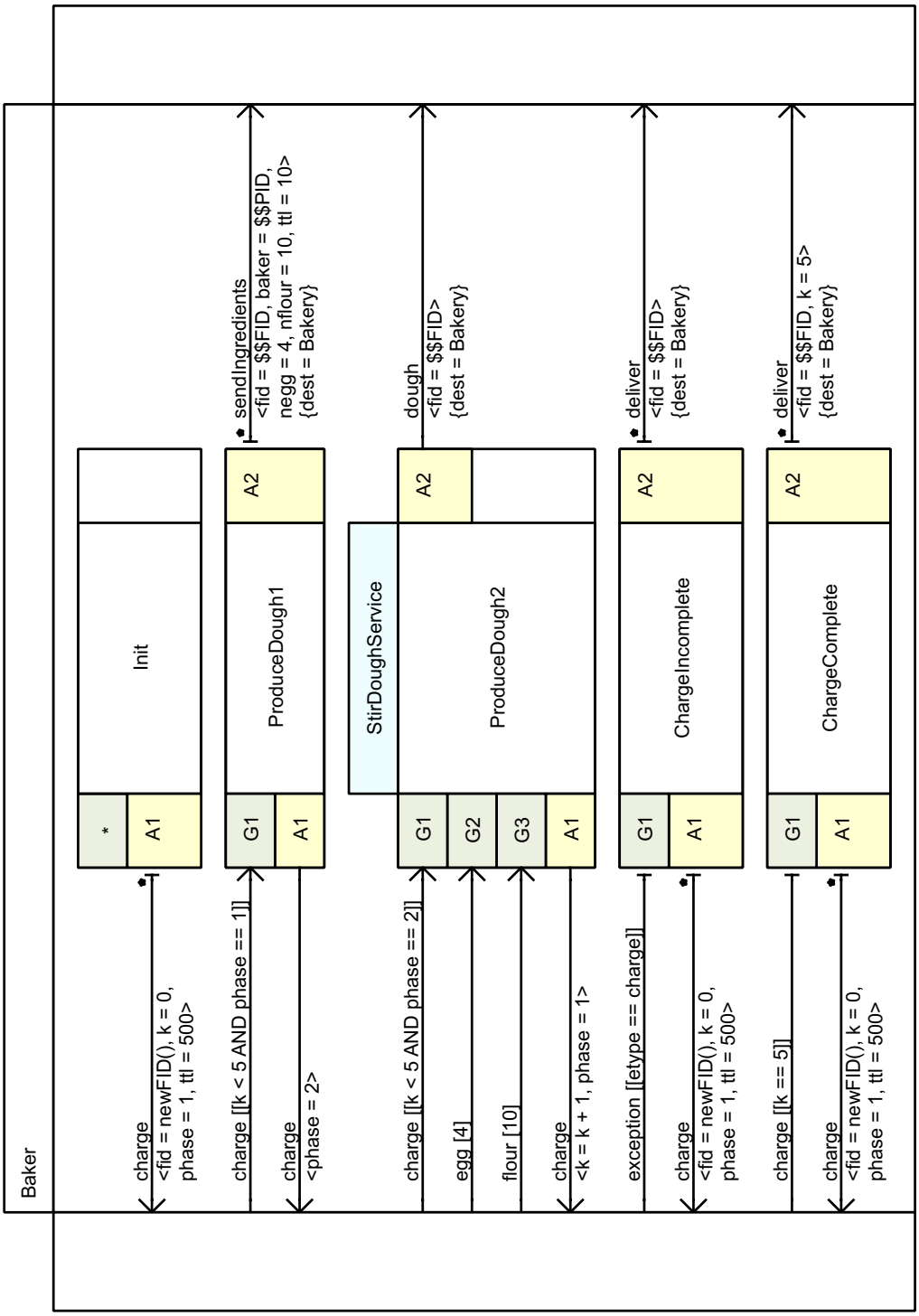


Figure 29: Baker peer (from the example in [50])

CHAPTER 4

Requirement Analysis

This chapter analyzes the features of related modelers and develop requirements for the PMMT based on the discussion in Section 2.8. A focus shall be put on features that facilitate usability to encourage the usage of the PMMT when working with PM models.

Since it would be presumptuous to create a full-fledged integrated and visual development environment on the first try, the focus of this thesis shall be to create a *minimal viable product (MVP)*. This again aims to encourage the usage of the PMMT by first providing a smaller, more basic, but coherent feature set that can iteratively be extended with new features and integrated with existing PM tools. Several features that were deemed out of scope for an MVP will be discussed in Chapter 8.

The requirements are split into functional (**FR**) and non-functional (**NFR**) requirements [21]. The difference between these two requirement types is that FRs describe what a system can *do*, while NFRs are a description of a system's property, characteristic, or quality (e.g. performance, maintainability, usability, or efficiency).

4.1 Functional Requirements

The following functional requirements have been identified for the PMMT:

Graphical PM Editor (FR01)

The actual core feature shall be a graphical editor for the PM supporting users in designing models on a drawing canvas. This includes an underlying semantical model of the PM that corresponds to the shown graphics. It shall be possible to rearrange the graphics via mouse interaction (e.g. *drag and drop*, or *point and click*), without losing the semantic connections between containers and wirings represented by links.

In light of the fact that we want to achieve an MVP, the first version of the PMMT shall support the PM building blocks described in Chapter 3 with the following restrictions:

a) It suffices to model containers as PIC or POC. The details of the underlying XVSM containers can be skipped.

b) A link's selector, assignments, and properties do not need to be mapped to an underlying semantic model. For the first PMMT version these properties can be stored as simple strings and it is left to the users to provide them in an appropriate form for the intended use.

c) Only a single service per wiring needs to be supported, because in the first PMMT version this can be a wrapper service, which abstracts the calls to other services.

These restrictions should be addressed and removed in future work.

Routing of Links (FR02)

The lines of links connecting e.g. containers and wirings shall be routed automatically. The links should also cross each other as little as possible. Other rectangular obstacles like wirings, subpeers, and services should also be taken into account when routing links. Their areas should be avoided as long as this is possible and thus, the links should be routed around them.

Entry Type Editor (FR03)

An interface to define the contents of entries shall be provided, where properties can be added, changed, or removed. Each property shall be assigned a type (like integer, floating point, or boolean value), a default value according to the type, and a description. System properties shall also be displayed, but must not be removed from entries.

Inline Text Editing (FR04)

The user interface shall provide interactions to edit texts in place whenever this is possible. For example see the following steps:

1. double click on name of peer or wiring.
2. display a cursor directly where the name is shown.
3. change the name.
4. finish editing when enter is pressed.

The idea is to not force the user to change focus to another user interface, but instead keep the attention where it already is. In CPN Tools this is the case, while in the BPMN2 Modeler texts can be edited either in the properties pane or in the properties window, which opens after a double click.

Context-Sensitive Menus (FR05)

All actions that involve an element in a model shall be provided near the element itself, e.g. show controls when hovering the mouse pointer over an element, or show a menu after a (right) click. That way, when a user looks at an element and wants to change its properties, focus can stay on the element while doing so. When on the other hand the user wants to inspect a related element, that element can be opened, shown, or selected after clicking on a control or menu.

Toolbox (FR06)

A toolbox with all available model elements shall be available where the user can drag items and drop them in an open model to create that item where it is dropped. Users can have a look at all available model elements and easily place them in their models.

Structural Overview (FR07)

An overview of the current model shall be provided to help users in locating specific elements within the model. An example for this is the outline view in the BPMN2 Modeler. All the elements within the model are shown in a tree view.

Plausibility Checks (FR08)

Since a model built in a modeler has not just a syntactic but also a semantic meaning attached to its elements (in contrast to a drawing in a graphics program), there shall be provided a mechanism to detect problems in the model. These inconsistencies or errors should be shown to the users in a way such that they can easily change focus to the causal element in the model. The user's focus should be guided from the detected inconsistency to the actual element in the model, enabling the user to efficiently fix the error.

Support for Runnable Model (FR09)

A common use case for the PMMT is to design coordination of concurrent processes, which makes it very important to support a runnable model of the actual design as this greatly reduces the turnaround times in the development process. Therefore the PMMT shall be able to generate code for the various implementations of the PM and thus help to reduce the time from the design to a testable software product. Furthermore, this can be used to generate the platform-independent PM-DSL [53] from the designed model as well, which makes the integration of already existing PM tools much easier and faster.

Graphic Export (FR10)

The PMMT shall be able to export the designed models as graphic files, which is not only useful for documentation purposes, but also supports a consistent look of the PM in future research publications, since the PM is an active research project.

4.2 Non-Functional Requirements

There are also non-functional requirements that shall be met by the PMMT:

Extensibility (NFR01)

The PMMT shall provide a plugin system for plausibility checks and code generation, to enable the support of many different implementations of the Peer Model. Furthermore, since they can read the complete model and produce files, plugins could be used as an export interface, e.g. to create another graphical representation. Also transformations on the model could be implemented in such a plugin, for example to transform the model to an output format that allows for model verification. This creates a lot of possibilities for plugin developers.

Usability (NFR02)

Users shall be able to easily find their way around the PMMT, without long introductions and tutorials. The aim is to provide a tool for working with the PM and keeping the users' focus on the designed model, not on the tool that is meant to support them. This means the user interface shall be intuitive and provide functionality where it is anticipated.

Maintainability (NFR03)

In the past, software was delivered on floppy disks and later CD-ROMs, some even on USB drives, which meant that once the medium was created, there was little to no chance of fixing errors later. Then, with the rise of the Internet it suddenly was possible to deliver patches or hotfixes that modify a software installation and fix errors. Since today most of the software is downloaded as a whole from the Internet, it also has gotten much easier to fix errors after the software's initial release. This increased pace of software release cycles makes it even more important to keep the future maintainability of the software in mind when designing it in the first place.

For that reason, the software architecture of the PMMT should be modular to enable the integration of new features without having to change too much of the existing code. This also means that the PMMT should be designed in a way to keep the needed boilerplate code for new features as minimal as possible.

Portability (NFR04)

The PMMT's code should be portable, which means it should be possible to create a running executable of the PMMT for different platforms (e.g. Apple macOS, GNU/Linux, Microsoft Windows) without much effort.

Data Compatibility (NFR05)

The PMMT shall store its project data in a way that can be read in updated versions. In other words, newer versions of the PMMT shall be able to read project data saved with older versions. This shall ensure that PM projects are not lost by updating to a newer version of the PMMT. For the first version the format shall be text-based for easier human understanding of the saved data. A change to a binary format for smaller project file size shall be taken into account when designing the serialization of project data.

Performance (NFR06)

The performance of the PMMT is very important, because users typically do have a very low threshold for unresponsiveness of graphical user interfaces, which can be as low as 0.7 seconds according to [27]. Therefore, user actions in the PMMT shall have low latency, such that users immediately get feedback of the application. Long running actions must not freeze up the application user interface, but rather run in the background with according user feedback. This aims most importantly to not scare off users with a slow, sluggish user interface, but also to contribute to good usability.

CHAPTER 5

Design & Implementation

This chapter first introduces the principal design decisions. Next the implementation details of the PMMT are discussed. These include the presentation of possible programming languages and graphical user interfaces, the architectural choices, which are influenced by the underlying technology, and finally the different possibilities for existing third party libraries, which might prove useful for specific tasks.

5.1 Design Decisions

The following design decisions are based in part on the takeaways of the related modelers presented in Chapter 2 as well as on the found requirements of the previous chapter.

The user interface of the PMMT shall be designed as a *Multiple-Document Interface (MDI)*, which has a main application window that contains the other documents as child windows or MDI windows. All related tools, except the PM Monitoring Tool, are designed as MDI applications, where every model diagram is edited within its own MDI window. This has the advantage of providing a common user interface in the main window that controls the respectively active child window. The common user interface includes, e.g., tool bars, status bars, or tool windows. The user interfaces themselves are made up of control elements called widgets. They can be simple widgets with a single task, like buttons, check boxes, and text boxes, or they can be more complex widgets that visualize data in form of a list or tree view. Additionally, custom widgets can be arbitrarily nested and display any other widgets.

Since virtually all diagram editors are operated with a pointing device (e.g. a mouse), it makes sense to use a drag and drop workflow wherever this is useful. This workflow is also used by all related tools. For example, users click model items in the editor and drag them to the desired position or they click a palette item in a tool window and drag it into the editor to create it at the dropped position. Such a palette window shall also

be provided, which shows the available model items, like wirings, links, services, and subpeers. From there users can drag a desired item and drop them into the editor to create a new item.

Each peer shall be edited in its own MDI window within the application window. Firstly, this clearly separates the peers and forces users to design clear interfaces between peers, because the entries are sent to other peers by properties. Secondly, this is also common practice among the related tools, except the PM Monitoring Tool, which shows the whole model in a single view to enable users in following the flow of entries. The peer is edited on a so-called canvas that can, in theory, become infinitely large and the user can change the zoom factor at will and scroll within the window. Subpeers hide their contents when they are displayed in their parent peer, but users have a simple possibility to open the subpeer in their own editor window and also to open the parent peer editor window from a subpeer. As stated in requirement FR01, only one service per wiring will be supported in the MVP implementation of the PMMT.

The PMMT aligns all its items in the graphical editor on a grid, to ensure that there is enough space for the items to display their information and keep them neatly arranged.

Links connect to containers of either their own enclosing peer or its subpeers. Users can drag them around freely within the peer and when they are connected to a container, the links will dock to the container. This behavior distinguishes two cases, namely whether the link is connected to a container or not. If connected, the links try to keep their vertical position relative to the wiring. If not connected on the other hand, the links keep their relative position to the wiring. This enables designers to draw the model and move wirings around later to, e.g., increase model clarity without changing the graphical representation of the moved wiring.

Newly created links get the next free number for their type (guard or action) within the wiring. This guarantees that the numbers are unique. They can be rearranged within the wiring as it best fits the graphical model representation. However, since it is not expected that they will be created exactly in the order that they shall be executed, there will be user actions to change the number of individual links as well as renumbering all links within the wiring according to their positional order.

According to requirement FR01, the selector, assignments, and properties of links are internally not modeled as abstract data structures but as plain strings. This enables users to design models with all necessary information, but they are required to take their target code generator into account, because they can only use features by their target Peer Model runtime.

Entry types are not edited within the graphical editor but rather in a separate window, which provides a tabular view of all entry types in the current project and their respective contents. A configuration file specifies the available system properties.

5.2 Technologies

The choice of programming language and GUI toolkit to use for the implementation of the PMMT needs to consider the following factors, which are partly based on the requirements in Chapter 4:

Extensibility The chosen programming language needs to provide a way to extend the final product to fulfill NFR01. This means that either it is possible to deliver compatible executable extensions for the already deliverd product later on, or that extension points for such a matter need to be developed explicitly if such a system is not provided by the programming language.

Maintainability NFR03 requires the chosen technologies to enable the software developers and architects to design the program in a modular way, which will support its later maintainability.

Performance The implemented program needs to be responsive at all times to fulfill NFR06. This means that the GUI must provide feedback to users as fast as possible, because of user's typically very low threshold for unresponsiveness.

Portability The programming language and GUI toolkit need to be available on multiple platforms and should support the developers by keeping the code explicitly written for each platform at a minimum. The focus should be on a single code base maintained by the developers that results in executable programs on multiple platforms to fulfill NFR04.

For the development of the Peer Model Monitor (cf. Section 2.7) an evaluation of suitable technologies for a graphical program has been conducted in [26]. The same choices for programming languages (C#, C++, and Java) and GUI toolkits (.Net, Qt, and JavaFX) are also viable for this thesis and will be considered for the implementation of the PMMT. Javascript and Python will be considered as additional possible programming languages. Since the choice of GUI toolkit often dictates the programming language to use [27], one cannot be chosen without considering the other, which leads to the evaluation of the combinations in the following sections.

5.2.1 C# / .Net

Although .Net[1] is available cross-platform, the native user interface only supports Microsoft Windows and Apple macOS. As such it is not yet a viable alternative for this thesis, because GNU/Linux users would not be able to use the PMMT without considerable effort (e.g. using a virtual machine, which in turn would influence the application's performance for the worse).

[1] `https://dotnet.microsoft.com/`

5.2.2 C++ / Qt

C++ is a compiled language with static typing and compilers available for all major platforms. Together with Qt[2] this programming language and GUI framework support a lot of platforms from a single code base. The approach can be summarized as *write once, compile anywhere*. Qt provides a lot of rich widgets and user interface components, as well as abstractions to operating system and hardware, which means that it is quite easy to have a single code base with very little code for explicitly supporting different platforms. This means the code compiles for every platform and produces an executable application specifically for that platform, and thus fulfills the portability requirement (NFR04). There is comprehensive and detailed documentation available for all its individual modules. Although most of the code is standard C++ code in Qt, there are some nonstandard extensions to support a more concise and decoupled way for events, packaging of resources directly in the executable, and the user interface designer of Qt Creator[3]. These extensions are the Meta Object Compiler (MOC), Resource Compiler (RCC), and User Interface Compiler (UIC), which themselves create standard C++ header and source files that need to be compiled and linked together with the other files they resulted from. In C++ the programmers are responsible for the memory management. That means that memory is explicitly allocated and also freed. This might seem a bit tedious at first, but with the C++ programming technique called *Resource Acquisition Is Initialization (RAII)*[4], the lifetime of resources like memory, open files, open sockets, etc. is bound to the lifetime of an object, which in turn frees the resource in its destructor. Additionally, this explicit resource handling results in better memory and runtime efficiency [27], because resources are freed as soon as they are not needed anymore, and there is no additional runtime needed for the garbage collector.

5.2.3 Java / JavaFX

Java is also a compiled language with static typing, but in contrast to C++ it does not compile to machine code. Instead it is compiled to Java bytecode, which is itself interpreted by the Java Virtual Machine (JVM). This means that all platforms that have an implementation of the JVM can execute the compiled program, which results in a *compile once, run anywhere* approach. JavaFX 12 supports all major platforms and also contains a lot of features and widgets. Since Java abstracts the hardware with its JVM, it is quite easy to support multiple platforms and operating systems. On the other hand the additional layer of the JVM between compiled bytecode and machine code naturally takes a toll on the runtime efficiency, because the JVM has to interpret the bytecode [11, 27]. Since Java uses garbage collection for the disposal of allocated memory that is not used anymore, this does not only have an impact on the runtime efficiency, but

[2] `https://www.qt.io/`

[3] `https://www.qt.io/development-tools`

[4] Resources are acquired in the constructor of an object and freed in its destructor. Each object encapsulates one resource and as such is the handle of this resource. For example, smart pointers use this to prevent memory leaks.

also on memory efficiency, because the unused memory is only freed in the next garbage collection run. Until then, the program still owns the unused memory, which continues to count as consumed memory.

5.2.4 Javascript / CSS / CSS

Javascript (JS) is, as the name predicts, a scripting language that, together with Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS), is a core technology of the World Wide Web. JSON was used primarily in the browser, until, with the release of Node.js[5], a JSON runtime environment outside the browser was available. This was the beginning of the so-called *Javascript everywhere* paradigm, which enabled the development of a complete web application, frontend and backend, in JSON. In the beginning Node.js started with Google's open source JSON engine V8, which together with an event loop and an I/O library are the foundation of the runtime environment. Since Node.js interprets the JSON code directly, this approach could be summarized as *write once, run anywhere*. For the development of desktop GUI applications there are at least two possibilities: Electron[6] or NW.js[7]. Both of them are GUI frameworks and use the rendering engine of the Chromium project to display CSS, CSS, and JSON websites as a graphical desktop application. This also reveals a drawback of this approach, namely that this application ships a full browser, which is composed of Chromium's rendering engine and the V8 JSON engine. Not only does this increase the size of the resulting application, but also the startup time is much worse compared to a similar application written in C++ or Java. Additionally, the interpretation of the JSON code is significantly slower compared to a compiled language like C++. Another drawback of essentially shipping a web browser is that Chromium has a rather large memory footprint. The simple Electron demo application[8] that displays a window showing supported features uses around 100 MB of memory and 150 MB of disk space. In comparison the final implementation of the PMMT with the bakery model from [50] consumes about 60 MB of memory with the Bakery Peer open and about 12 MB of disk space.

5.2.5 Python / PyQt

Python is an interpreted language with dynamic typing. The reference implementation CPython is available on all major platforms. CPython is itself written in the C programming language, and compiles Python code to an intermediary bytecode similar to Java, that is then executed by CPython's virtual machine. Additionally, there are other implementations like PyPy, which complies with Python 2.7 and 3.5, but features a just-in-time compiler (JIT) that gives it a significant performance improvement over CPython [78]. Qt is natively developed in C++, but also offers language bindings for Python, which provides all the features described before. In Python developers also

[5] https://nodejs.org
[6] https://electronjs.org/
[7] https://nwjs.io/
[8] https://github.com/electron/electron-api-demos

do not need to concern themselves with explicit memory management, because it is done automatically and transparently. CPython uses reference counting and garbage collection internally and frees memory as soon as it is not referenced anymore, which is similar to using RAII in C++. Since Python is not compiled to a single executable file, distribution of a finished program is not as simple as passing along the compiled program in C++. There are various ways to distribute Python programs. One would be the Python package manager *pip*, which is platform independent, but requires an already present and configured Python installation on the target computer. Another is to create individual setup packages for each targeted platform, which can be quite complicated, since the setup frameworks are different for the major platforms.

5.2.6 Conclusion

Since C# / .Net was not considered due to the fact that GNU/Linux support is not yet possible, and all the remaining candidate pairs support the extension of already delivered applications, they have been compared in terms of runtime efficiency and memory efficiency. Dalheimer specifically compares Qt and Java in [27] and Back and Westman compare programming languages used in Google Code Jam in [11]. The result of the candidate's comparison is shown in Table 3. Based on these results and the previous experience and preference of this thesis' author C++ / Qt has been chosen as the programming language and GUI framework to build the PMMT.

	runtime efficiency	memory efficiency	executable size
C++ / Qt	$*$	$*$	$+$
Java / JavaFX	$+$	$\sim$	$*$
Javascript / CSS / CSS	$\sim$	$-$	$-$
Python / Qt	$+$	$+$	$*$

$*$ great $+$ good $\sim$ medium $-$ bad

Table 3: Comparison of programming languages and GUI frameworks

5.3 Architecture

The PMMT is implemented with a layered architecture to make future changes in the software as straightforward as possible and thus also satisfy NFR03. The idea behind the layered approach is that each layer can only access the layers below, but not the ones above. Additionally, the code is partitioned into modules to separate their concerns. This modular structure of the PMMT's implementation is shown in Figure 30. The individual modules are represented by boxes with their name in it. Some of the modules are divided further into submodules, to also separate the concerns within the module.

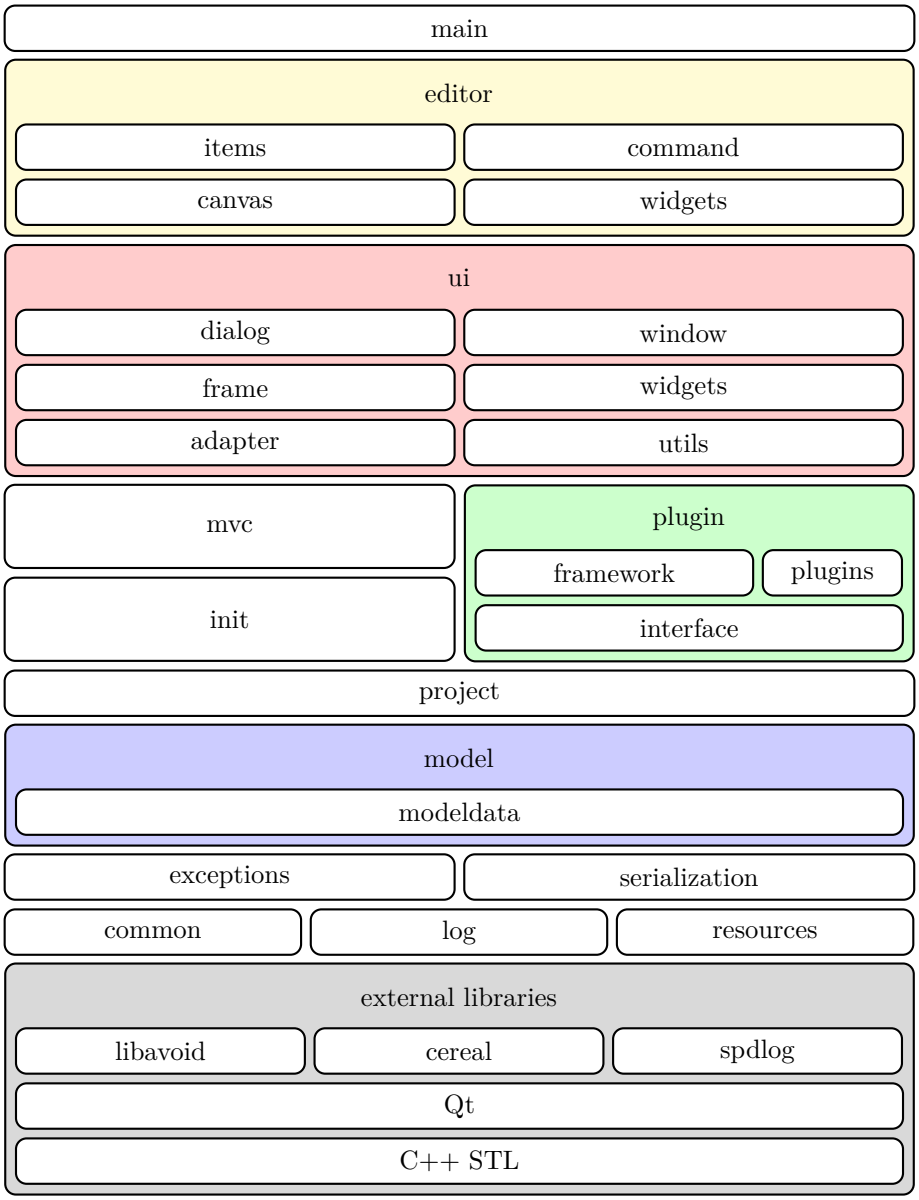


Figure 30: Architecture and module structure of PMMT

The modules have been separated with C++ namespaces in code and by building individual static or dynamic libraries in the build process. This has also been done where feasible for the external libraries, which are at the bottom of the architecture stack and on which the rest of the PMMT implementation may depend.

The choice of C++, its *Standard Template Library (STL)*, and the Qt framework have already been discussed in Section 5.2, and the choice of other external libraries (libavoid, cereal, and spdlog) will be discussed in detail in Section 5.4. The remainder of this

chapter focuses on the modules implemented in PMMT and their respective concerns.

5.3.1 Modular Structure

In Figure 30, starting from the bottom right on top of the external libraries are the modules common, log, and resources. The first contains common utility constructs that can be used all over the rest of the project, e.g. a RAII helper class for freeing resources on destruction, fuzzy floating point comparison functions, or a wrapper class for used file system paths. The log module contains the logging utilities and acts as a wrapper around the external logging library, which can therefore easily be exchanged with another one. At the moment the resources module only contains a wrapper class for the icon files, which are compiled into the executable. That class also is responsible for instantiating the icon objects exactly once for the running executable.

On the next layer is the exceptions module, which contains the class hierarchy of exceptions to be thrown if unexpected error cases are encountered during runtime. These exception classes also use the logging facilities to record all errors in the log. The serialization module contains classes for reading and writing data in a persistent format that is used to load data from and save data to disk. This format is versioned, thus supports data compatibility and ensures that newer software versions can read files saved in older versions. Additionally, there is a base interface for scanning all written data, which can be used to search for specific values.

The following layer contains the model module, which comprises all classes that encapsulate the data elements (cf. Figure 31). The **Element** class is instantiated for every single item in the designed model, e.g., for every peer, wiring, service, link, and subpeer. Additionally, every entry type and system property are stored as single Element instances as well as one instance for the project settings. The generic data available for every type is stored directly in the Element class. Each item has a unique **Key** within the project, which is stored in the field **key** and contains its type and unique number within the type. The available types are mapped by enumeration **Type**. The key is used for references between items and thus represent the model hierarchy via the fields **container** (top-level container element), **parent** (direct parent element), and **reference** (generic referenced element). The **changesCounter** and **deleted** fields store meta-information about the element for the undo support. The **uuid** field stores an universally unique identifier, which can be used in the future to match an element across project boundaries where it might have different keys in different projects. The **instanceNumber** is generally used for the numbering of elements within a parent, which currently means the guards and actions of a wiring. The **attribute** field is used to set a specific attribute of an element, e.g., whether a container is a PIC or a POC. The **name** field stores a user-defined name of an element, while the **id** field stores a programmatically generated identifier. The interface **IElementDetail** is used by the element to store the **detail** data for its specific element type. This detail data is contained in the submodule modeldata. There is one implementation class for each element type that needs detail data, which cannot be represented by the general parts of the element class. There are also container classes for

the model elements and keys that enable organizing multiple model elements, like lists, sets, or hash maps, and a class for building the display names of single elements.

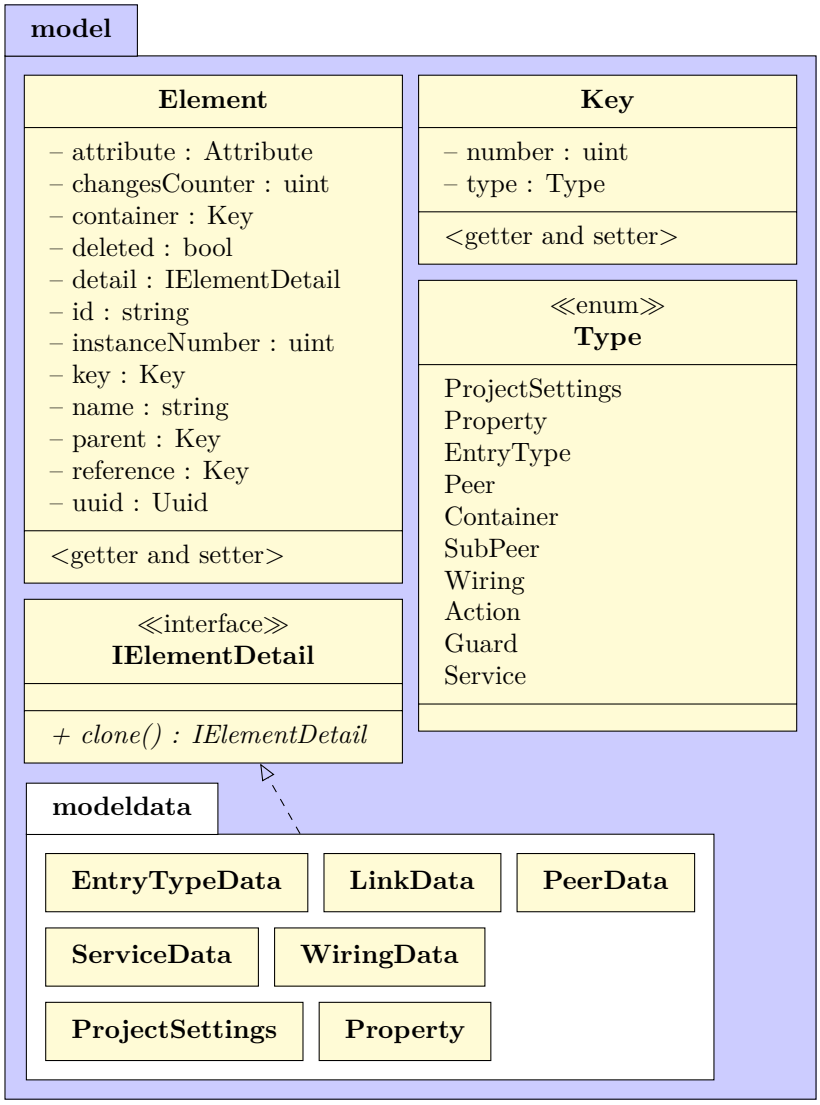


Figure 31: Class diagram for modules model and modeldata

On top of the model sits the project module, which contains the classes responsible for organizing the data of a whole project. Figure 32 shows the class diagram for the module. The **Instance** class is a handle for the stored project that is used throughout the code to read from and write to the project. The **Tx** class and its specializations, **ReadOnlyTx** and **UpdateTx**, are used together with a project instance to manage read and update transactions. The **Query** class is used to read specific elements by their keys or query all elements from the project that match specified filter criteria. The last group of classes in this module is concerned with delivering notifications about data changes in the project.

It is comprised of the **Notifier** class, which is part of every internal project. It uses instances of the **Notification** class, which are sent to widgets that are connected to the notifier. The **Receiver** is a helper class that is used by the widgets to handle the connection to a project's notifier. Finally, the **IObserver** interface is implemented by the widgets that want to receive notifications through the receiver. The **internal** package contains the classes responsible for the in-memory storage of the model elements of the current project. The **Element** class stores a model element in the state of a specific transaction and the needed information like its transaction identifier. The next layer, the **Entry** class, contains one element in all historical states that are still needed by open transactions, which are used, e.g., for handing over a consistent state of the project to another function. The **Type** class contains all entries of a specific element type and the **Project** class stores all elements of one project partitioned into their respective types.

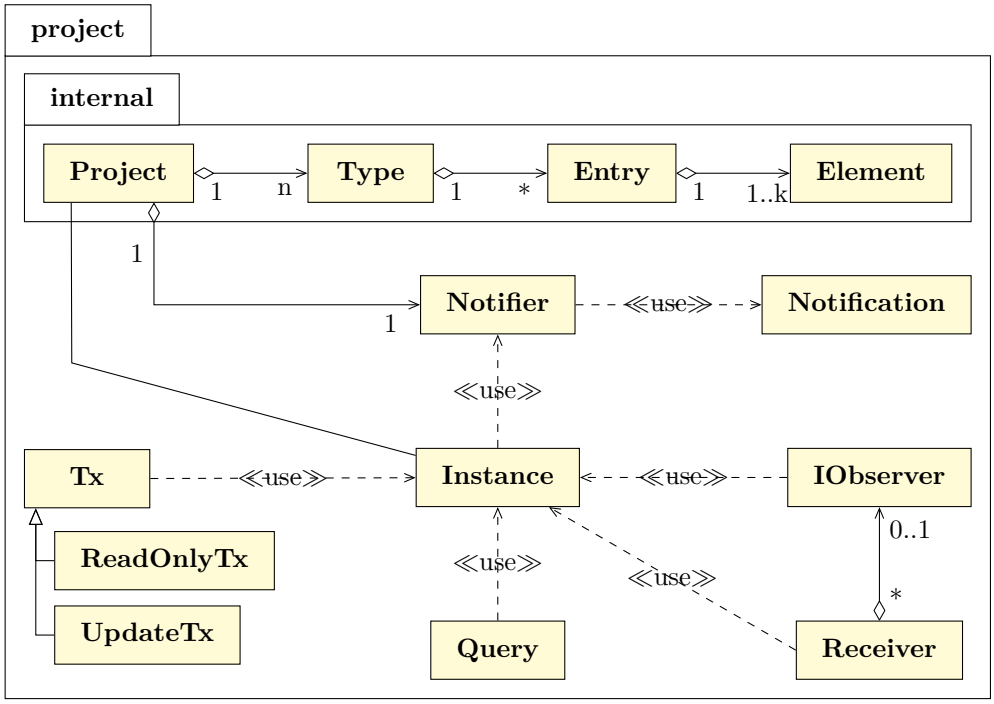


Figure 32: Class diagram for the project module

The plugin module provides the PMMT's plugin interface for extensions in the interface submodule, which is explained in more detail in Section 5.3.2. It also contains the implementation of the plugin framework that consumes the implemented plugins through their interface. This includes the plugin manager responsible for loading the plugins into memory from disk and providing access to them for the rest of the PMMT. Further, the Peer Model factory is implemented in this module, which creates a copy of a PMMT project in the form that is specified by the plugin interface and then passed to the plugins. This is to ensure that plugins do not have direct access to the internal project data of the PMMT.

The init module is responsible for initializing newly created projects. This means, creating the default data within the empty new project required by a fully initialized and running PMMT application. At the moment this includes creating a project settings element and all the properties specified in the property configuration file of the installation directory. An example of this configuration file is shown in Table 4. The file is a plain text tab-separated file, where the heading is a comment line starting with a # sign and with the following column contents:

Type comma-separated list of item types for which this property is available

ID unique property identifier

Short identifier abbreviation (intended for most often used ID's)

Description free text describing the property's semantics

Data Type this property's data type

Value the default value, if no other value is explicitly set

Type	ID	Short	Description	Data Type	Value
Entry,Link	dest	d	Destination	Address	
Entry,Link	dest.exc		Destination Exception	Address	
Entry,Link	dest.iop		Destination IOPeer	Address	
Entry,Link	ttl	tl	Time to Live	Duration	0
Entry,Link	tts	ts	Time to Start	Duration	0
Link	mandatory	mn	Mandatory	Bool	true

Table 4: Example configuration file for Peer Model properties

The next layer contains the module mvc, which stands for the MVC-Pattern [31]. It comprises classes implementing Qt's model/view architecture framework[9] depicted in Figure 33. The Qt abstract item model is an abstract base class for the model part of this framework. The model classes implemented in this module provide the model part of the framework for the PMMT project data. This ensures simple rendering in the Qt standard view classes like list views, tables, or tree views, as well as editing the data via Qt standard delegates like check boxes, text edit fields, or combo boxes.

The ui module is the first layer having a dependency on Qt's graphics classes, as it is also responsible for creating the PMMT's graphical user interface. This module contains the base classes for all widgets and windows used in the PMMT. This module is further divided into submodules to keep their respective concerns separated and their interfaces clear. The utils submodule contains helper classes abstracting drag and drop data,

[9] https://doc.qt.io/qt-5/model-view-programming.html

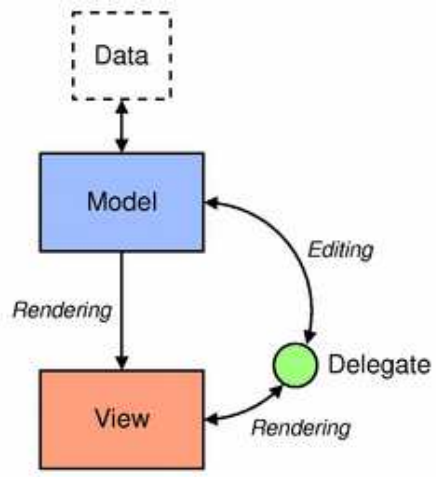


Figure 33: Overview of Qt's model/view architecture (taken from [9])

keyboard shortcuts, or zooming data, to be used by the other ui classes. The adapter submodule has classes for the abstraction of communication between widgets in the same window. For example, the buttons in a window-specific toolbar execute adapter commands that have been registered by other widgets before. The frame module contains the more specific base classes for the different window types that make up the application frame, like MDI windows, docking windows, or window toolbars. Finally, the dialog, widget, and window submodules contain the classes for the specific implementations of the respective items. This includes the main window, all the menus, toolbars, docking and MDI windows, and dialogs.

On the next layer is the editor module, which contains the implementation of the actual graphical Peer Model editor widget. It comprises the widget itself, the wrapper classes for Qt's graphics scene and view, the editor toolbar, a customized scroll bar implementation, and clipboard and grid handling functions. The editor's widgets submodule contains implementations for editor-specific widgets to be shown in windows of the ui module. The canvas submodule comprises classes implementing a scrollable canvas within the graphics scene, which enable the scrolling of a graphic item's contents. This was implemented as a trial whether it is useful to scroll the contents of a peer, but was ultimately disabled. It remains in the code base, because a peer's content still is a canvas on which users place the contained items. The support for undo and redo actions is based on Qt's undo framework[10], which provides classes for undo commands, stacks, groups, and views. The command submodule contains the implementations of specific undo commands for each action in the graphical editor. Every MDI window has its own undo stack, where the command implementations are stored as long as it is open. The undo group provides a single pair of undo/redo actions for all grouped undo stacks, and thus for the whole application. The undo view is currently unused, but could be used in the future to

[10] `https://doc.qt.io/qt-5/qundo.html`

display the undo actions in an undo stack. Finally, the graphic items that represent the individual elements of a Peer Model with which users interact in the graphical editor are implemented in the items submodule.

On the topmost layer is the main module, which contains the main function and is responsible for implementing the startup function and preparing the application, which includes preparing the application fonts and the default font size, setting application information, parsing command line arguments, loading available plugins, and finally opening the main window.

5.3.2 Plugin Interface

The plugin mechanism in PMMT is built on top of Qt's plugin mechanism[11] for extending Qt applications. This mechanism uses a shared library for each implementation of a single plugin. These shared libraries are different for each platform (as are the application executables themselves, cf. Section 5.2.2), e.g. dynamic link library (dll) on Microsoft Windows, dynamic library (dylib) on Apple macOS, or shared object (so) on GNU/Linux. These shared libraries are then loaded during runtime of the PMMT, and can thus be easily installed even after the initial installation of the application.

Since the shared libraries contain executable code specific to each platform, they have to be compiled for each platform individually. Consequently, a different shared library needs to be distributed depending on the platform the PMMT is installed on. For the future, a mechanism to distribute a file containing shared libraries for all supported platforms should be considered, to make plugin distribution easier for end users.

The infrastructure within PMMT consists of several parts that work together:

- a plugin manager that discovers and loads shared libraries in the PMMT installation during application startup,
- an interface that defines the possible functionality of a plugin implementation,
- an interface for a plugin collection, to support distributing multiple plugins in a single shared library, and
- a data transfer model, which contains the available information the plugin implementations can work with.

Figure 34 shows the interfaces that make up the plugin interface to be implemented by a developer. The base interface **IPlugin** describes the basic features of the plugin, which are its name and identifier. Additionally, there are methods for editing and storing preferences specific to this plugin. Then, there are two derived plugin interfaces with specific features for each use case: **IModelChecker** and **ICodeGenerator**. The model

[11] `https://doc.qt.io/qt-5/plugins-howto.html`

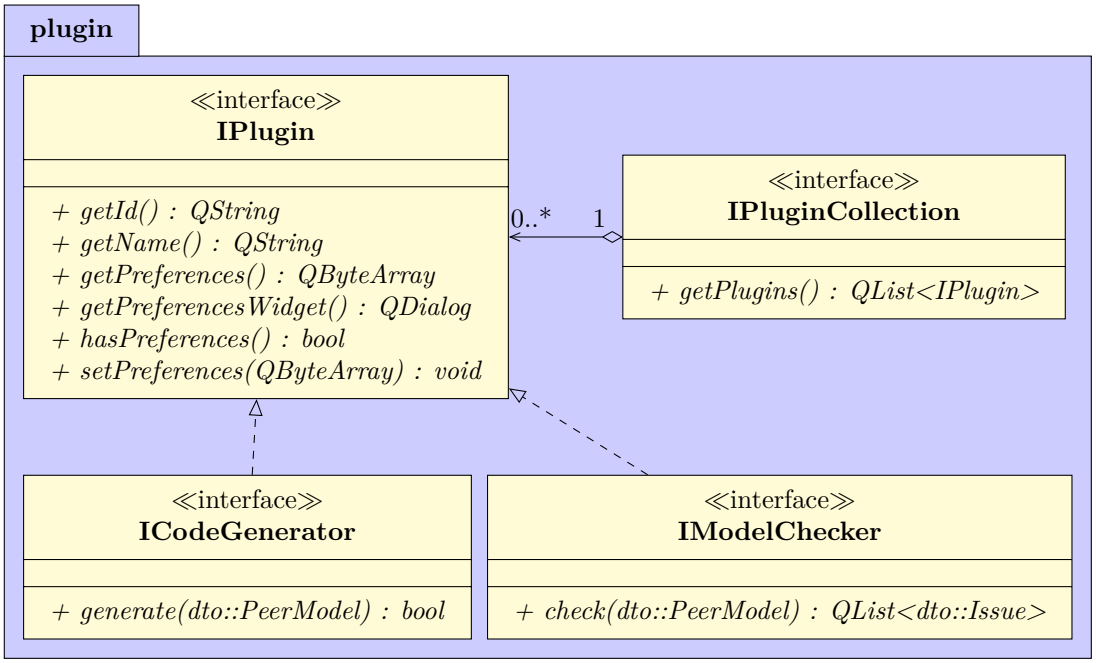


Figure 34: Class diagram of the plugin interface

checker plugin has a single method for checking the passed data transfer model for e.g. errors, design inconsistencies, or unsupported features, which returns a collection of discovered issues to be reported back to the user. The code generator plugin on the other hand currently has a single method for creating the code from the passed data transfer model. The plugin is responsible to store the generated code on disk.

The **IPluginCollection** interface is a workaround for Qt's plugin mechanism limitation of only one top level plugin per shared library. It essentially just returns a list of base plugins that are contained within this shared library. Each of them has to fully implement either the model checker or code generator plugin. This enables plugin developers to ship related plugins in a single shared library, for example the code generator for a specific PM implementation and the according plausibility checker, which ensures that only PM features are used in the model that are supported by the PM implementation.

Currently there is one configuration file for the PMMT, which contains the PM system properties and has already been explained in the description of the init module.

5.4 Third Party Libraries

For certain parts of PMMT implementation, third party libraries have been incorporated, because there is no need to reinvent the wheel in areas that are non-essential to this thesis' research. Therefore, the following sections will each explain specific parts of PMMT that have been implemented based on a third party library.

5.4.1 Logging

Log files are usually the first place to look for errors, when an application does not do what is expected of it. Therefore having log files and writing meaningful messages in them is an important part of supporting users, when there is no way to access their machine, let alone debug the application.

There are multiple possibilities for the implementation of the logging mechanism in PMMT. The first would be the naive approach to implement it from the ground up only depending on facilities provided by C++ itself. Another one is to use logging classes of the Qt framework, which have basic support for logging to the application's console, but all handling of log files, their rotation, and customized configuration of where the logs are written to would also have to be developed from the ground up. A third approach is to use an existing logging library that already supports all the necessary features.

The naive C++ only and the Qt logging approach were rejected, due to the fact that there are available logging libraries that have been rigorously tested, are fast, and easy to use. Therefore, spdlog[12] was chosen as logging facility. To minimize the dependency on a specific library, spdlog is not directly used by the rest of PMMT's code. Instead a wrapper class is used throughout PMMT's code base to make an exchange of spdlog against another library as easy as possible, should the need arise.

Spdlog can be used as a header-only library, which means it only consists of C++ header files and will be compiled completely into the resulting application binary file. Another possibility is to link spdlog as a static library, which enables faster compile times, but makes the setup more complicated. In both cases spdlog depends on a compiler supporting the C++11 standard, which enables customizable, feature-rich, and type-safe formatting based on C++ template parameter packs. Further, spdlog provides support for different log targets (e.g. to rotated log files, to the console, to syslog) as well as custom log targets that can be tailored to the application's exact needs. Finally, the log messages can be filtered based on their severity, and the thresholds can be modified during compile time as well as runtime.

5.4.2 Serialization

To enable users to save their designed models and resume working on them at a later time, the PMMT's project data needs to be saved to and loaded from disk. This transformation of C++ objects in memory to a format that can be portably saved to disk or transmitted over a network, and back to equivalent C++ objects from that portable format is called serialization.

Like in the previous chapter with logging, there are again multiple possibilities to support serialization in the PMMT. Two approaches have been rejected early on: the naive implementation from the ground up based only on bare C++ features, and an implementation based solely on Qt's datastream, XML stream, or JSON objects. In

[12] https://github.com/gabime/spdlog

comparison to using a library, these could make it harder to change the serialization format in the future, or to provide portability across platforms with, e.g., different endianness.

Since the approaches to implement from the ground up were rejected, several libraries were considered as basis for PMMT's serialization:

- Apache Avro[13]
- Apache Thrift[14]
- Boost.serialization[15]
- cereal[16]
- MessagePack[17]
- Protocol Buffers[18]

Apache Avro, Apache Thrift and Protocol Buffers require an additional build step with their own compiler to create C++ header and source files from their own CTL, which is used to describe the objects to be serialized. Since this leads to own serialization classes that contain the data to be serialized, there are two options of using them. Either keep them as they are and require an additional step to copy data to and from the classes used in the PMMT project, or incorporate the PMMT project classes into the respective serialization classes and lose the clearly separated interface. Both options are suboptimal and therefore these libraries have been rejected.

MessagePack is a binary data serialization format that is supported by over 50 implementations in different programming languages and environments. It does not need a CTL file to describe the serialized objects, but instead the data to be serialized is written into a message pack stream one field, member, or variable at a time. There is support for serializing custom classes at once, but since MessagePack is a serialization format, most C++ implementations do not explicitly support versioning of the serialized data and instead leave this to the user.

Boost.serialization and cereal are similar in the way these libraries are used to serialize structures. They both do not need CTL descriptions of the objects to be serialized and can serialize structures to a binary format and XML. Further, they support versioning of the output to ensure data compatibility with future application versions. They are able to serialize single fields as well as custom structures or classes at once, even for classes that cannot be altered, by providing a way to extend the serialization non-invasively. Additionally, cereal is a header-only library like spdlog and supports serializing to a portable binary format and to JSON.

Based on the comparison in Table 5, cereal is used to serve as base library for the serialization of PMMT's project data. The decisive features are that there is no need for

[13] https://avro.apache.org/
[14] https://thrift.apache.org/
[15] https://www.boost.org/doc/libs/1_71_0/libs/serialization/doc/index.html
[16] https://github.com/USCiLab/cereal
[17] https://msgpack.org/
[18] https://developers.google.com/protocol-buffers/

a CTL and thus an extra build step, it can be used non-invasively with existing classes, it supports portable binary, JSON, and XML formats, it supports versioning for the individual serialized classes, and it is a header-only library that is compiled directly into the binary, without the need to link another static library. Listing 5.1 shows how a model element, in this case a guard link, is stored in JSON format using cereal. The detail data specific for each element type shows how cereal handles polymorphic pointers. A *polymorphic_id* together with a *ptr_wrapper* are stored, and the latter then contains the actual *data*.

```
{
   "key": { "type": "Guard", "nr": 4 },
   "changesCounter": 58,
   "deleted": false,
   "id": "",
   "name": "",
   "instanceNr": 3,
   "uuid": "{1065da92-9967-445e-939e-0d8d3e57df8a}",
   "container": { "type": "Peer", "nr": 1 },
   "reference": { "type": "Container", "nr": 1 },
   "parent": { "type": "Wiring", "nr": 3 },
   "attribute": "Undefined",
   "hasDetail": true,
   "detail": {
      "polymorphic_id": 6,
      "ptr_wrapper": {
         "valid": 1,
         "data": {
            "bLeft": true,
            "entryType": { "type": "EntryType", "nr": 4 },
            "count": 10,
            "selector": "",
            "assignments": "",
            "properties": "",
            "sourceType": "LinkSource_Take",
            "destinationType": "LinkDestination_Write",
            "position": { "x": 0.0, "y": 80.0 },
            "gridHeight": 2,
            "canvasPositionOtherEnd": { "x": 0.0, "y": 380.0 },
            "shapeEndGridYOffset": 1
         }
      }
   }
}
```

Listing 5.1: Serialization in JSON format of a model element

	Apache Avro	Apache Thrift	Boost serialization	cereal	MessagePack	Protocol Buffers
No CTL			✓	✓	✓	
Non-invasive				✓	✓	✓
Formats						
Binary	✓	✓	✓	✓	✓	✓
JSON	✓	✓		✓		✓
XML			✓	✓		
Portable	✓	✓	∼	✓		
Versioning	✓	✓	✓	✓		✓
Header-only				✓		

✓fully supported ∼ partially supported

Table 5: Comparison of third party serialization libraries

5.4.3 Link Routing

An integral part of the Peer Model designs are the wirings and their links, which are connected to containers of either their parent peer or direct subpeers of them. As they are the most important part of the logic within a peer, the links should have as little crossings as possible to enable a clear layout and reduce complexity. This can either be achieved by letting designers place the links themselves, or by providing an algorithm that takes care of the link routing. The manual approach implies that users will have to manually correct the links every time they rearrange peer contents. The algorithm approach on the other hand will take care of routing the links to their connected point. It further ensures that links and other peer contents do not overlap each other. Therefore, the whole peer logic is visible to the designers.

Such an approach to automatic line routing while avoiding obstacles is presented by Wybrow in his doctoral thesis [90]. The resulting library, called libavoid, together with libcola for constraint-based graph layouting and two other libraries form the Adaptagrams[19] library of tools for adaptive diagrams. According to the website, the libraries are also used for, e.g., Inkscape[20] (a vector graphics drawing tool) and Graphviz[21] (a graph visualization software), both of which are free and open source software.

In the PMMT libavoid is used for routing the links between wirings and their respectively connected containers. To that end, each peer has an associated libavoid router that is concerned with finding the best connection from the link port to its connected container. Routed links avoid the area of wirings and subpeers by adding these areas as obstacles to the router. This means the areas of wiring bodies, which are comprised of the wiring's

[19] http://www.adaptagrams.org/
[20] https://inkscape.org/
[21] https://www.graphviz.org/

rectangles with their name, the link ports with their identifiers, and the service rectangles, as well as the areas of subpeer items. Link connections are added to the router as connectors, which are then routed in their shortest possible, orthogonal path to their respective destination point, while avoiding overlaps with other links or obstacles. This makes it possible to reroute all links when obstacles or links are moved within the peer, and thus enables links to take better routes if they are not obstructed anymore.

The routing algorithm also has a feature called nudging, which moves overlapping connectors apart by a small distance. This feature is not as useful in the PMMT as it is in other drawing tools, because the Peer Model displays the link parameters along the connector. In an attempt to get around the nudging and still retain the automatic routing feature, while keeping the connectors from overlapping, all the coordinates in the graphic editor are converted from Qt graphic scene to grid coordinates. The grid coordinates are visualized as dots in Figure 35.

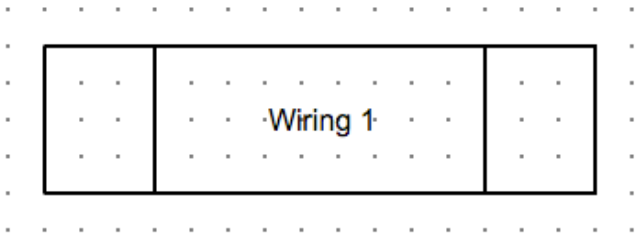


Figure 35: Dotted grid coordinates in the editor

The dots represent the points in the grid, which are 20 pixels apart in each direction. So if a wiring has a height of 80 pixels in the graphic scene, it is 4 grid units in height and added to the libavoid routing algorithm with the grid unit height. This results in links always being apart at least 1 grid unit, because links are only routed directly on the grid coordinates, and thus ensuring enough space for the routed link's information.

CHAPTER 6

Visual Modeling of Peer Model Applications

This chapter first presents changes to the existing Peer Model visualization of links and then the visual modeling workflows for creating, editing, checking, and generating code from Peer Model projects. The main point of consideration is the sequence of actions and events of these workflows in the Peer Model Modeling Tool. Additionally, this chapter also discusses alternative approaches to the ones actually implemented in the PMMT, which have been considered, but have ultimately been discarded or postponed to future work due to the aim of creating an MVP.

6.1 Visualization

Most of the Peer Model visualization in the PMMT conforms to previously presented graphical notations in [52, 53, 54]. One essential thing that has changed is how the different link operations are distinguished. Earlier publications modeled them with different arrow heads and additional textual operation types along the link connector (see Figure 36).

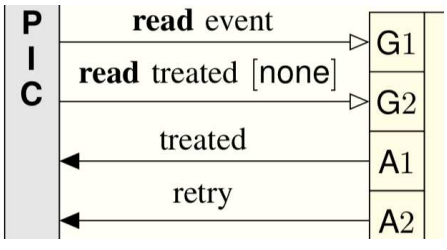


Figure 36: Visualization of links in [53]

In comparison to this, the textual operation type was removed in an attempt to clear up space along the connector for the link parameters. Additionally, the link operation was split into source and target operations, such that now there are three different operations each that can be arbitrarily combined. They are each visualized at the according end of the link connector. Guards have the source operation at the container side and target operation at the wiring side of the connector, and actions vice versa. Figure 37 shows how the PMMT displays links with the resulting changes and Table 6 lists the link operations of each link in the picture.

Link	Source Operation	Target Operation
G1	take	delete
G2	read	write
G3	create	shift
A1	read	shift
A2	create	write
A3	take	delete

Table 6: Link operations in Figure 37

Figure 37: Visualization of links' source and destination operations

6.2 General User Interface

6.2.1 Starting the PMMT

Initially, users have to choose if they want to start a new Peer Model project, or continue working with an existing one. For that purpose the project dialog shown in Figure 38 is displayed when opening PMMT. If the user chooses to create a new project, the main window will be displayed immediately. If an existing project is opened, first the native operating system's file open dialog is displayed, where the desired PMMT project file is selected. After that also the main window is displayed.

Other possible approaches of greeting users after application start include showing the main window with a welcome page or showing just a welcome window. This page or window should offer more options than just open or create, for example opening the last

recently used projects, help topics, or a news feed of recently added application features. For now the project dialog is sufficient, but this should be considered as a future feature.

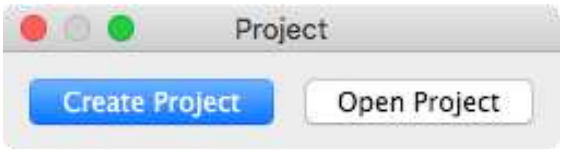


Figure 38: Create or open project dialog

6.2.2 Main Window

The main window is shown in Figure 39, with the integral parts of the PMMT: a) the menu bar, b) the tool bar, c) two docking windows, and d) the MDI area.

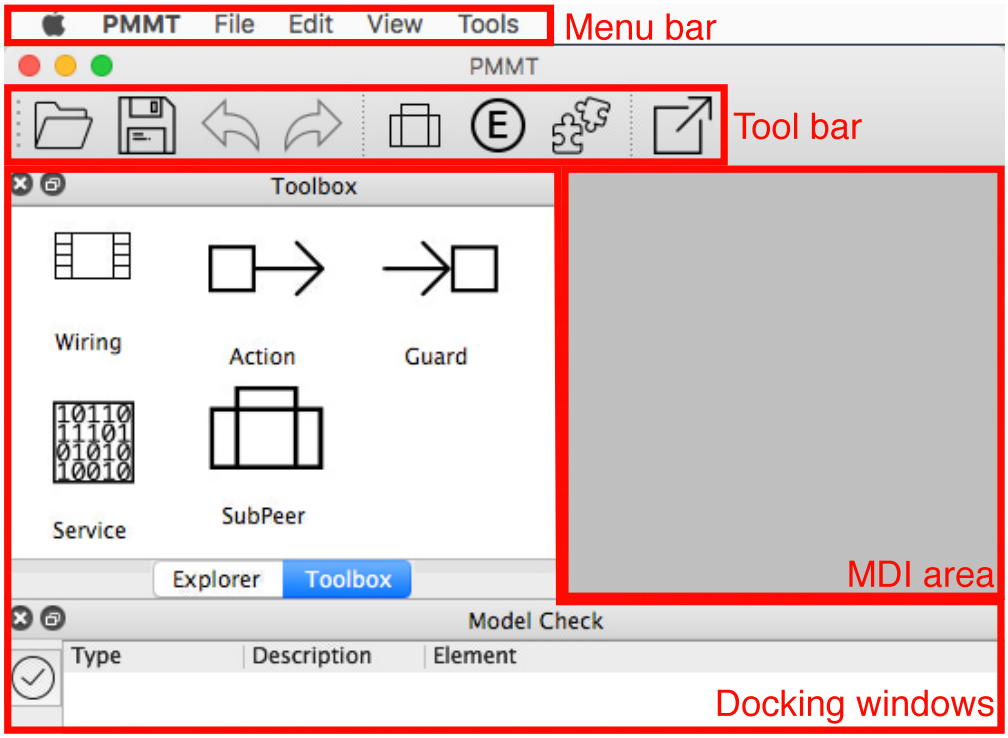


Figure 39: PMMT main window

6.2.3 Menu Bar

The menu bar at the top in this case is not part of the main window itself, because in Apple macOS and some Linux desktop environments a global menu bar is used rather than an individual menu bar in each window like in Microsoft Windows. The menu bar contains entries for saving the current project and for controlling the currently shown docking windows (see Figure 40).

Figure 40: File and view menus

6.2.4 Tool Bar

A tool bar (cf. Figure 39) with the most commonly used actions in Peer Model editing is displayed either right below the menu bar or at the top of the main window, depending on the operating system's use of a global or individual menu bar. The icons from left to right trigger the following actions:

1. open an existing project
2. save the current project
3. undo the last action
4. redo the last undone action
5. create a new peer and open the editor
6. open the entry type editor
7. open the plugin manager
8. start code generation

6.2.5 Tool Windows

Figure 39 also shows two tool windows docked on the left and at the bottom of the main window. All the tool windows can be closed and opened via the previously described view menu in the menu bar. They can also be docked at all sides of the MDI area. If there is not enough space on the display (e.g. small notebooks), the tool windows can also be stacked on top of each other. This is shown in the *Toolbox* window of the figure, where at the bottom of the window a tab bar with the window titles is displayed. The desired window can be displayed by clicking on the according tab. It is also possible to undock the tool windows and move them around freely on all available screens.

6.2.6 MDI Area

In the MDI area the editor and plugin manager windows will be displayed. MDI stands for *Multiple Document Interface* and it implies that each peer is displayed and edited in its own MDI subwindow, which is displayed in the MDI area. Usually the MDI area will take up the most display space when working with Peer Models. Currently MDI windows can only be displayed in the MDI area of the main window. Future work on the PMMT should enable displaying MDI windows in separate MDI areas and thus providing better support for multiple screens.

6.3 Hierarchical Model Overview

The model explorer gives the user a hierarchical overview of the currently open project (see Figure 41). The tree view shows a parent node for peers and one for entries. For each peer that exists in the project, a node is created below the peer parent node. All child items of the peer (wirings, links, services, and subpeers) are again displayed at their hierarchical position in the model. Users can jump from the tree item to the actual item in the editor by double clicking on it. This will open the editor window of the according peer, select the item, and make sure it is visible in the editor. Further, items that allow a user-defined name can also be renamed in the model explorer by clicking its name once when already selected.

The model explorer could be extended in future work to increase its usability. For example, when working in the editor, the model explorer could mirror the current selection and also select the item that is currently selected in the editor. Another possibility would be to extend the drag and drop features by enabling to drag an entry type onto a link to set its entry type property, or onto a wiring to directly create a link with the dropped entry type. Additionally, a dragged wiring could be copied and created in the peer it is dropped into.

Figure 41: Model explorer window

6.4 Managing Entry Types

The entry editor window (shown in Figure 42) can be opened via the corresponding tool bar button. The window shows two tables next to each other, the one on the left lists the currently available entry types, the one on the right lists the properties of the selected entry type on the left.

The tool bar on the left side of the window has an add button and a delete button. They are dependent on the currently active context, which means they add a new entry type when the left table is active, and a new user-defined property for the currently selected entry type when the right table is active. The delete button works analogously for the currently selected entry type or user-defined property.

Properties in the Peer Model can have subproperties and they are represented with their parent properties ID and a dot as prefix, e.g. “dest.exc”. Another approach would be to change the right table to a tree view with multiple columns, where the subproperties are represented as child nodes of their parents.

The entry editor window is designed as a master-slave view, where the left table showing the entry types is the master table and the right table is the slave table, which shows the properties of the entry type that is currently selected in the master table. The columns in the slave table correspond to the system property configuration file explained in Section 5.3.1. The column *Sys* shows if this property is a system property, which is a read-only information for users. The identifier of a property in the *ID* column can be nested by using a “.” as separator and a list entry for parent properties is not necessary to create sub-properties. Columns *ID*, *Value*, and *Description* are edited via text fields similar to spread sheet applications. In the *Datatype* column users can choose from a list of available data types from a drop-down box. An alternative to that design would be to only use a single tree view with columns, where each entry type is a parent node in the tree, and all its properties are child nodes.

Both of the described alternatives could be implemented in future work, possibly with an option for users to select their preferred view. Additionally, future work should consider adding support for further data types like user-defined enumerations, lists and sets.

6.5 Creating and Opening Peers

New peer items can be created by clicking the icon with the peer symbol in the tool bar of the main window (see Figure 39). The editor for the newly created peer will then be displayed in the MDI area. Existing peer items are opened by double clicking them or one of their content items in the model explorer.

A future extension could provide a search field that provides a full-text search within the current project, from where the existing peers could also be opened.

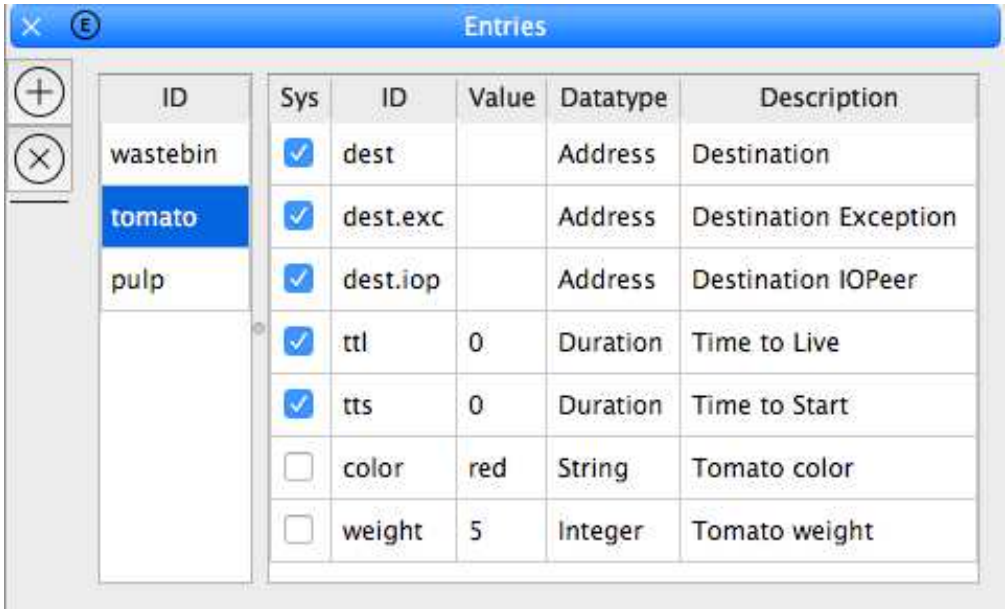


Figure 42: Entry editor window

6.6 Creating and Editing Peer Contents

Chapter 2 already discussed some different workflows of creating new model items in the related modelers and Section 5.1 presented the drag and drop workflow as a design choice for the PMMT. The *Toolbox* shown in Figure 43 serves the purpose of a palette view, from which new items can be created.

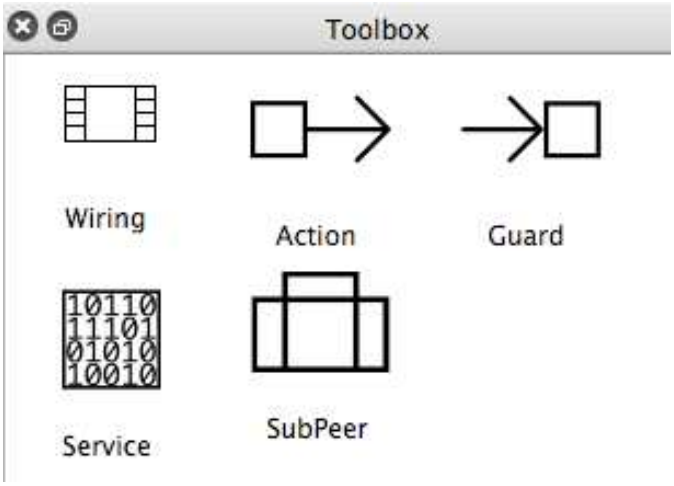


Figure 43: Toolbox with peer content items

Additionally, the context menu of a peer's or subpeer's name field provides an action for

replacing this peer's contents with the contents of another peer (shown in Figure 44). This is a first approach to using peers as templates. For future work it should be considered to implement a pattern-based approach, like the one described in [55]. Another possibility could be to implement an approach based on a type system for peers and wirings, where all peers and wirings are instances of underlying types, like objects and classes in the classical object-oriented programming paradigm.

Figure 44: Peer's editor window with context menu

Figure 44 also shows the peer editor window tool bar, which is located at the left window frame. The top button deletes the currently selected editor items, while the next two buttons export the whole peer as a graphics file for, e.g., documentation purposes. There is one button for each currently supported format, namely Portable Document Format (PDF) and Scalable Vector Graphics (SVG).

In the beginning of the implementation of the PMMT, also the *select and click* workflows was implemented for creating wirings, where users first click the item in the palette and thereby change the mode of the mouse cursor to creating that selected item. A subsequent click in the editor now creates the selected item. However, with the implementation of further item types only the drag and drop workflow was updated accordingly. Nonetheless, it would easily be possible to also revive the select and click workflow, and improve the combination and use of both workflows in the editor in future work.

The following sections will describe the similarities and differences between the creation workflows of the different item types.

6.6.1 Creating Wirings

Figure 45 shows the steps for creating a new wiring. The starting point is an open editor with an empty peer, in which the wiring will be created.

1. The wiring is dragged from the toolbox on the left.
2. The mouse cursor shows the wiring icon while dragging.
3. A preview wiring is shown as soon as it could be dropped.

After dropping the wiring at the desired position in the peer, it is created like it was shown in the preview. The only difference is that during the preview its name just says "Wiring" and after creation the wiring's data in the model is populated with default

data. This results in displaying an instance number after “Wiring” that is unique for this wiring instance. When creating multiple wirings, they all have a unique name, which can be changed in the text edit widget that is displayed in place after clicking a selected wiring’s name. Every wiring can be changed into an init wiring via its context menu.

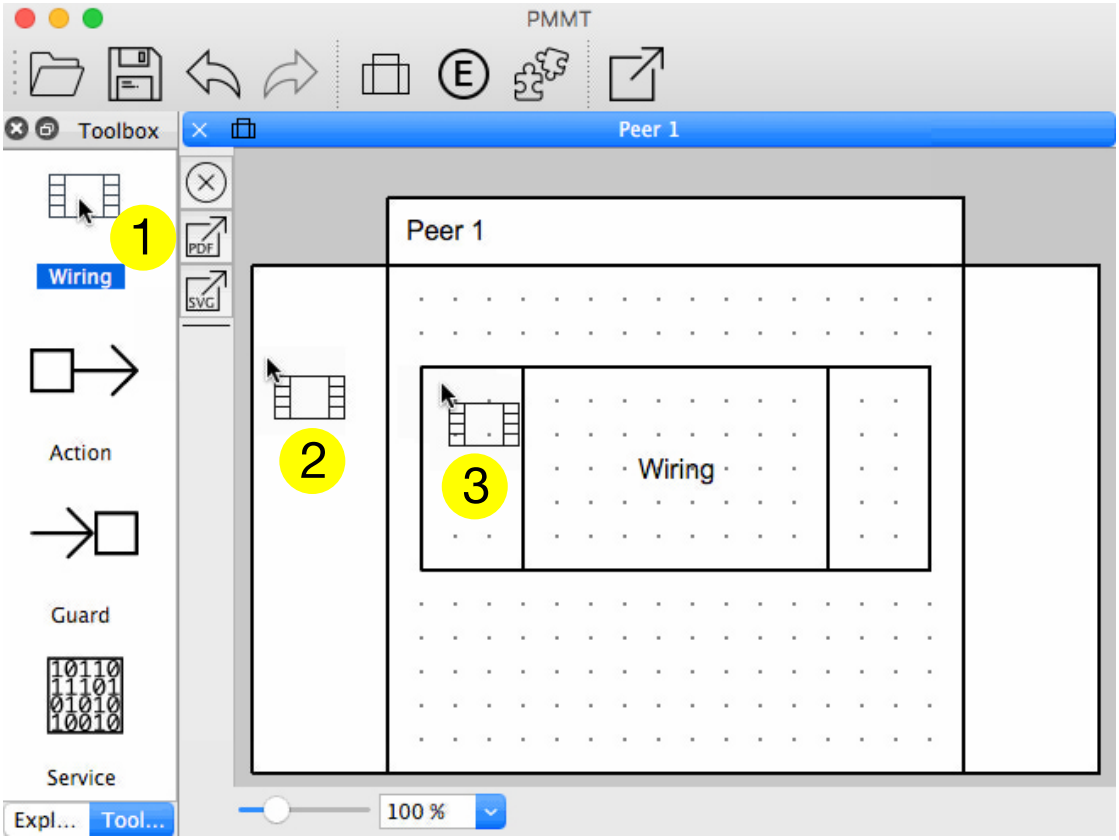


Figure 45: Creating a wiring

6.6.2 Creating Subpeers

Since subpeers are within the hierarchy of their parent peer on the same level as wirings, they can be created analogously to wirings. The internal logic of a subpeer is not visibile in the parent peer editor. For that, the subpeer has to be opened in its own editor window (cf. Section 6.7).

6.6.3 Creating Services

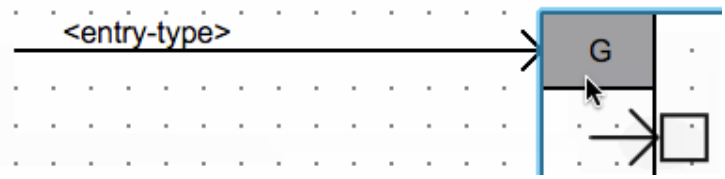
The drag and drop workflow for creating services is again similar to creating wirings and subpeers, but since they are not direct children of a peer it is slightly different. The main difference is that the service preview is only displayed when the mouse cursor is dragged into a wiring item, which is shown in Figure 46a. The name also changes when the service is created, due to the same reasons as with the wiring.

(a) Preview when creating a service (b) Created service after drop

Figure 46: Creating a service item

6.6.4 Creating Links

The workflow for creating links is quite similar to creating services, as links are also direct child items of wirings. The difference is that links can be placed on the left and right sides of wirings rather than at the top. The two types of links, guards and actions, behave equally when creating them. They also show a preview of the link to be created (see Figure 47a). Links again have the same behavior described for all previous items with populating the item's data after creation, and thus displaying the correct name as shown in Figure 47b. The rules for numbering have already been explained in Section 5.1. As the position of links within wirings is important for their execution sequence, they can also be placed before or between already existing links. This does not result in automatic renumbering of existing links, which can be achieved via the wiring's context menu. Alternatively, the numbering of links can be corrected manually via the link's context menu.



(a) Preview when creating a link

(b) Created link after drop

Figure 47: Creating a link item

6.6.5 Connecting Links to Containers

Links can be connected to containers in two ways. The first one is to grab the link end on the far side of the wiring with the mouse button (shown in Figure 48a) and drag it to the edge of the container it should be connected to. There the mouse button is released and the link is connected to the container as shown in Figure 48b.

(a) Grab link for moving

(b) Connected link

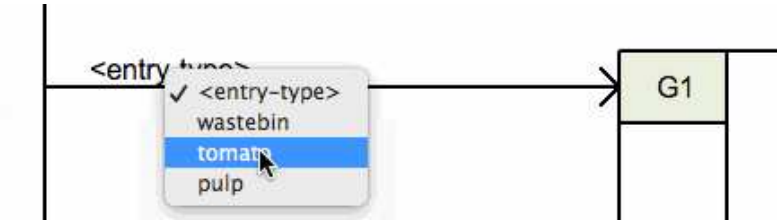
Figure 48: Connecting a link to a container

The second way to connect links to containers also supports connecting all links on the same side of a wiring. The link ends will stay at their relative position to their parent wiring when it is moved. This can be used to grab and move the wiring such that the link ends connect to the container and drop the wiring at that position. Now the links are connected to the container and will stay connected to it, even if the wiring is moved again.

6.6.6 Edit Link Parameters

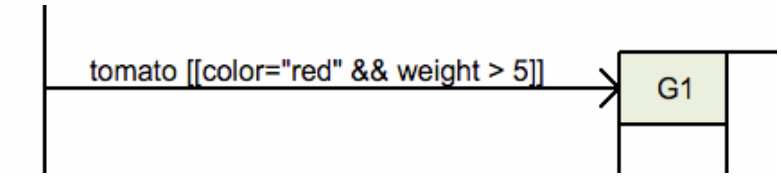
Links show by default only those parameters that have meaningful data (i.e. non-default data set by the user) except for the entry type, which is always visible. Each parameter has a placeholder text, to show which link parameters are available. Since users will want to change a link's parameters, all of them are displayed when hovering with the mouse cursor above the area of a link in the editor (see Figure 48a).

The entry type can be selected from the menu shown in Figure 49a, which is displayed after clicking on the placeholder. In accordance with requirement FR01 all other parameters are text-based and can be edited by double clicking the placeholders, which will display a text editor widget right in the position of the placeholder. Figure 49b shows this text editor widget with a text cursor for the selector, and the populated text of the selector is shown in Figure 49c.



(a) Edit link's entry type

(b) Start editing link's selector



(c) Populated link's selector

Figure 49: Editing a link's parameters

The idea behind showing only link parameters with meaningful data and hiding all others, except when editing them, is to provide a view of the model, that is:

> *as simple as possible, but not simpler*
>
> — Roger Sessions paraphrasing Albert Einstein[1]

This means that all the information of the model's behavior is visible, but all unnecessary information for understanding the model, like parameter placeholders, are hidden.

The difference between showing all information fields of a link and showing only the ones with meaningful data can be seen in Figures 49b and 49c. In the latter only the entry type and the selector are displayed, because count 1 is the default value and assignments and link properties are empty.

Since currently the selector, assignments, and link properties are text-based parameters, there is only basic text input support for these fields. Future work should definitely include improvements on this situation, e.g., by providing context-sensitive automatic text completion on known properties or variables, or by offering a dialog for creating these link parameters.

[1] According to `https://quoteinvestigator.com/2011/05/13/einstein-simple/` this quote stems from Roger Sessions, when paraphrasing an aphorism of Albert Einstein.

6.7 Working With Subpeers

When interacting with subpeers of a peer, they are in the hierarchy similar to wirings, namely direct children of their parent peer. Links from wirings can connect to a subpeer's PIC or POC the same way as they connect to their parent peer's containers. However, since subpeers themselves have contents as any other peers, there are some other criteria to consider that are unique to subpeers.

Firstly, in contrast to standard top level peers their fully qualified names represent the actual hierarchy, when editing the contents of a subpeer. This means that the complete hierarchy of parent peers is prefixed to the subpeer's name. Let's examine the displayed names with the following example: the model has a top level peer **P1**, which contains a subpeer **SP2**, which itself contains a subpeer **SP4**. The names of the items are displayed as follows when editing the contents of

- **P1:**
  - P1's own name and fully qualified name are the same and always displayed as "P1", because it is the top level peer.
  - SP2's name is displayed as "SP2", because in the contents of P1 the hierarchy is obvious.
  - SP4 is not yet visible, as the contents of subpeers are not visible from the parent peer.
- **SP2:**
  - SP2's fully qualified name is displayed as "P1 / SP2", to accurately represent its hierarchy and that it is not a top level peer. If SP2's own name is edited, only the part after the last "/" sign, namely "SP2" can be edited.
  - SP4's name is displayed as "SP4", analogously to "SP2" when editing P1.
- **SP4:**
  - SP4's fully qualified name is displayed as "P1 / SP2 / SP4". Analogously to SP2, only the last part "SP4" is editable.

Apart from the naming scheme, the second thing to be considered is the workflow for users when working with subpeers. The creation has already been discussed, but once the subpeer is created, users need to edit its contents. Since after its creation the subpeer is also displayed in the model explorer, users can open them from there to start editing its contents. Another possibility is to open the subpeer's context menu via right mouse click and use the menu entry to open the subpeer's editor window. On the other hand, the workflow from editing the subpeer's contents to getting to the parent peer's contents should also be considered. This can be achieved via the context menu available in the subpeer's editor by right clicking the subpeer's name field (as shown in Figure 50). The other menu action, which replaces this subpeer's contents with those of another peer, has already been explained in Section 6.6.

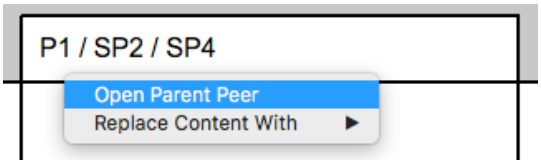


Figure 50: Subpeer's context menu

6.8 Plugin Configuration

The available plugins can be configured in the plugins window (shown in Figure 51), which can be opened either via the tools entry in the menu bar or the tool bar button. This window gives an overview of the available plugins, shows and controls their current status (enabled or disabled), shows whether they are a model check or code generation plugin, and also provides access to the preferences of each plugin. The preferences are specific to each plugin and transparent to the PMMT, which means they currently are implemented by the plugin developers themselves. Future work could include work on a common framework for plugin preferences, which further supports plugin developers in creating customizable plugins.

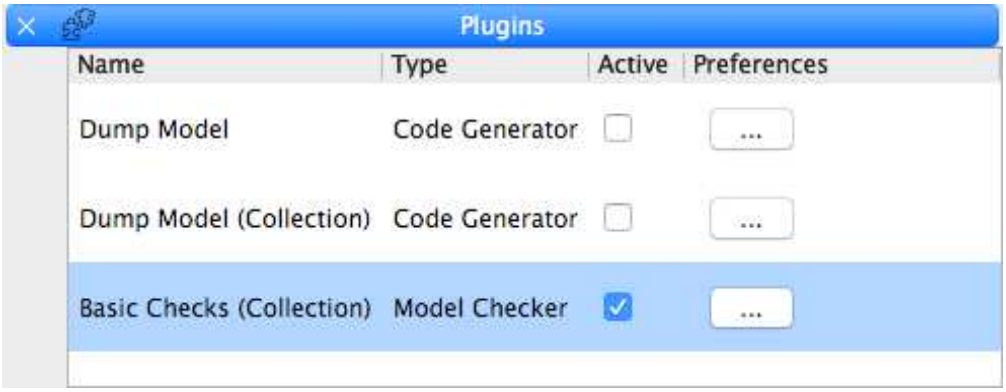


Figure 51: Plugin manager window

6.9 Using Model Checks

Model check plugins help users to identify possible problems within their models, for example if there are any links without a connected container that need a connected container to read entries from, or if there are any links that do not have a valid entry type selected. The results of a model check run are shown in Figure 52.

The workflow of running a model check currently relies on an explicit start by the user. This can be done by first opening the model check window via the view menu and then clicking the button on the left with the checkmark icon in the model check window (see Figure 52). This will run all enabled model check plugins on the current model. After all plugins finished, the combined results are displayed in the table.

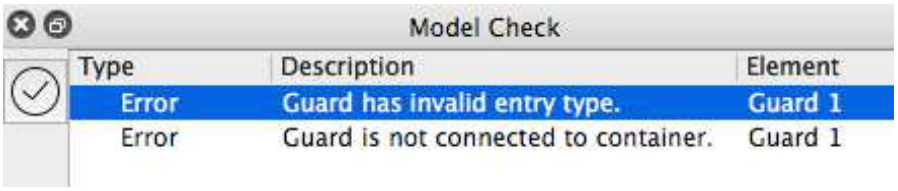


Figure 52: Model check result window

Users can then use the check results in the table to investigate erroneous model items by double clicking the respective line. This will open and display the editor of the peer that contains the causal model item, select it, and change e.g. the link's background to red criss-cross lines (cf. Figure 53).

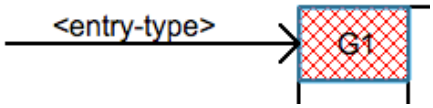


Figure 53: Erroneous item shown in editor

Another approach to running the model checks would be to continuously run them as a background task instead of requiring the user to explicitly start them. Additionally, the error markings could be displayed as soon as the errors are discovered and as long as they are not corrected, instead of just showing them once the erroneous item is put into focus from the results window. Together these two features could be implemented in a future extension.

6.10 Code Generation

The ability to create code from the designed model is provided via code generation plugins. A code generation run with all enabled plugins can be started by users via the export button in the tool bar (the rightmost one in Figure 39).

The PMMT currently provides two code generation plugins, which write a text-based tab-separated dump of the complete model to the application's standard output. The first one uses the single plugin implementation per shared library and the second one provides a collection of plugins in one shared library. These are meant to provide an example for future plugin developers, as they show how to use the information provided by the individual element types.

In the future, the approach of explicitly invoking code generation could be extended to be executed regularly in a background job, such that there always is code in a configured directory on disk, which represents the current status of the model.

CHAPTER 7

Evaluation

After the Peer Model Modeling Tool has been implemented, this chapter presents its evaluation by comparing it to the related tools of Chapter 2, and checking the fulfillment of the found requirements from Chapter 4. Additionally, a first user study was conducted to get some insight into the usability of the graphical user interface. Finally, a retrospective view of the implementation choices described in Chapter 5 concludes this chapter.

7.1 Comparison with Related Tools

In Chapter 2 several related tools have been explored and compared regarding their graphical editor features and auxiliary model development features. Since these tools have been used to analyze the necessary features for a graphical model editor and an integrated model development environment, the PMMT implemented for this thesis will now be compared to the related tool's features. The result is shown in Table 7 and will be discussed in the following paragraphs.

7.1.1 Graphical Editor

The graphical editor is the main feature that was implemented in the PMMT, because it also is the main focus point of this thesis. The following features enable efficient and easy usage of the graphical editor.

Automatic Layouting

PMMT does currently not support automatic layouting of items in the graphical editor, but link connectors are routed automatically while still avoiding other editor items. The other presented editors have neither support for automatic layouting nor do they avoid other items with lines connecting editor items. However, it might prove useful to implement the automatic layouting algorithm of the PM Monitoring Tool [26] in the

PMMT and provide users the possibility to use it at their own discretion, e.g. to relayout specific peers or selected items.

Context Menus

In PMMT's graphical editor, context menus are used to provide access to context-sensitive actions, e.g. reordering links within a wiring. Additionally to general context menus that are accessed by a right mouse button click, there are some menus that are accessible once the mouse pointer enters the vicinity of certain model elements. For example, the height of wiring links can be changed by dragging the height handles displayed below the link's identifier, or the properties of a link that still do not have a value are shown once the mouse pointer moves in the vicinity of that link.

Drag and Drop

New items are created in PMMT by dragging them from the toolbox into the editor and dropping them right where they should be created. Existing items can be moved by dragging them to the desired position. Selected items can be resized by dragging their border to the desired new size. To select multiple items in the editor, the mouse pointer can drag a lasso selection around them, and once the mouse button is released, all model elements within the created selection rectangle will be selected.

File Export

PMMT supports exporting designed models as SVG and PDF files. Since these are both scalable graphics, they can be used to print in every size without pixelization. Direct export as pixel graphics, e.g. PNG files, is not supported, which leads to the partial fulfillment rating. However, this can be implemented as a future extension of the PMMT.

Information Overlay

In PMMT's editor all information necessary to understand the model design is already shown directly within the editor itself. This is quite similar to the way CPN Tools provide the information about the designed model and renders any additional information overlay useless.

Inline Text Editing

All the texts in PMMT's models can be edited in the graphical editor, exactly where they are displayed. There is no need for additional dialogs just for editing an element identifier.

Item Palette

The toolbox provides an overview of all available items that can be placed in the model. From there they can be dragged into the model to create new items.

Hierarchical Overview

The model explorer provides a tree view of the current project and thus a hierarchical overview of the designed model. All the peers of the model are shown with their child elements, which assists users in discovering the model top down.

Keyboard Shortcuts

PMMT supports keyboard shortcuts in different application areas, e.g. for deleting selected items in the graphical editor or entry editor, opening an existing project, or saving the current project. The shortcuts are shown directly in the menus and conform to the operating system standard.

7.1.2 Code Generation

Generating code from the designed model is supported by PMMT via code generation plugins, which provide the ability to add new code generators without changing the PMMT application itself. The available code generators are presented in the plugin configuration window described in Section 6.8.

7.1.3 Error Checking

Similar to code generation plugins, there is an interface for model checker plugins. This enables users to provide their own customized error checks in such a plugin without the need to change the PMMT application. The errors found by these plugins are presented in the model check table, which gives an overview of all discovered problems within the model. By double clicking individual issues, the respective erroneous item is opened, focused, and marked as erroneous by highlighting it with a special color. However, the PMMT only highlights the erroneous item itself and not all parent items, as it is done by CPN Tools, which would make discovery of the item even easier. This could be achieved in future development.

7.1.4 Future Work

The following features have not been addressed explicitly in this thesis, but are subject to future extensions of the PMMT.

- Deployment Support (cf. Section 8.11)
- Simulation (cf. Section 8.12)
- Model Checking and Verification (cf. Section 8.13)
- Performance Analysis (cf. Section 8.14)

The application architecture was designed to be modular and extensible, and should thus make it easy to, e.g., provide new plugin interfaces to integrate these features.

Feature	PMMT	AsmEE ASMETA	BPEL	BPMN2	CPN Tools	ECT	Papyrus	Peer Model Monitoring Tool
Graphical Editor	●●	○○	●●	●●	●●	●●	●●	○○
Automatic Layouting	●○		○○	○○	○○	○○	○○	●●
Context Menus	●●		●○	●●	●○	●○	●●	●○
Drag and Drop								
Creating	●●		●○	●○	●●	○○	●○	
Moving	●●		●○	●○	●●	●●	●○	●●
Resizing	●●		●○	●○	●●	●●	●●	
Lasso Selection	●●		●●	●●	○○	●●	●●	
File Export	●○		●○	●○	○○	●●	●○	
Information Overlay	●●		●●	●○	●●	●○	●○	●●
Inline Text Editing	●●		●○	●○	●●	●○	●○	
Item Palette	●●		●●	●●	●●	●●	●●	
Hierarchical Overview	●●		●●	●●	●●	●●	●●	
Keyboard Shortcuts	●●		●●	●●	●●	●●	●●	
Code Generation	●●	●○	○○	○○	●○	●●	●●	○○
Deployment Support	FW	○○	●●	●●	○○	○○	○○	○○
Error Checking	●●	●○	●○	●○	●●	●○	●○	●○
Simulation	FW	●●	○○	○○	●●	●●	○○	●●
Model Checking and Verification	FW	●●	○○	○○	●●	●●	○○	○○
Performance Analysis	FW	○○	○○	○○	●●	●●	○○	○○

Legend: ●● completely fulfilled ●○ partially fulfilled ○○ not fulfilled
FW will be addressed in future work (see also Chapter 8)

Table 7: Graphical editor feature comparison of PMMT with related tools

7.2 Requirement Fulfillment

The following sections revisit the requirements identified in Chapter 4 and check whether the implemented PMMT application fulfills them adequately.

7.2.1 Graphical PM Editor (FR01)

The graphical editor is the core feature of the implemented PMMT and provides a drawing canvas to design PM models. It supports user interactions with the designed model via mouse and keyboard actions, while keeping the underlying semantic model intact and in sync with the displayed model.

7.2.2 Routing of Links (FR02)

Connected links in the editor are routed automatically, e.g., from wirings to the connected containers, while avoiding crossings with other links or overlaps between links and wirings or subpeers as far as algorithmically possible. Rerouting of links is triggered whenever an item is created, moved, resized, or deleted. This is achieved by integrating the libavoid library of the *Adaptagrams library of tools for adaptive diagrams* [90].

7.2.3 Entry Editor (FR03)

PMMT provides a window for managing the entry types of the designed model in a table view (cf. Figure 42). Users can create, edit, and remove entry types as a whole as well as single properties of a specific entry type. The properties have an identifier, a type, a default value, and a description, all of which can be changed by the user.

7.2.4 Inline Text Editing (FR04)

All item identifiers and free text properties (like link properties or assignments) are edited directly in the graphical editor view as demanded in the requirement (cf. Figure 49b). There is no need for dialog windows to enter a new text. Only the properties that are chosen from a list of available values and are not free text are chosen from context menus, like a link's entry type or the link's order within the wiring.

7.2.5 Context-Sensitive Menus (FR05)

PMMT provides model item actions directly in the context menu of that item. This includes for example: changing a link's entry type, changing the numbering of links within a wiring without moving them, renumbering all links of a wiring according to their graphical position, or replacing a subpeer's content with another peer's.

7.2.6 Toolbox (FR06)

The Toolbox window (cf. Figure 43) provides an overview of all available Peer Model items that can be placed in the model. From there users can drag the desired item and drop them in the peer at the position where they want to create the item.

7.2.7 Structural Overview (FR07)

The Model Explorer window (shown in Figure 41) keeps users informed about the current state of the designed PM by showing the project contents in a hierarchical way. The model items are visualized in a tree view with every peer as a parent node for their respective child items, such that users can inspect every peer top down.

7.2.8 Plausibility Checks (FR08)

The PMMT features a model checker mechanism based on the plugin system, so that new checks can be created without the need of changing the PMMT application itself. Results of the executed checks are shown to the user in the model check window (shown in Figure 52). The erroneous items involved in the error can be highlighted (cf. Figure 53) and put into focus by double clicking the error entry in the model check result table.

7.2.9 Support for Runnable Model (FR09)

The second implemented plugin interface in the PMMT integrates code generation plugins. They are used to generate code for Peer Model runtimes, which in turn can then execute the designed models. This achieves support for integrating multiple PM implementations as desired in the requirement.

7.2.10 Graphic Export (FR10)

The PMMT supports the export of designed peers as SVG and as PDF files. Both are scalable graphics and thus can be printed in different sizes without becoming blurred or pixelated. The exported images contain only the link properties that have meaningful content, which improves the readabilty of the exported peer while still containing all necessary information to understand the peer's functionality.

7.2.11 Extensibility (NFR01)

The PMMT application's plugin framework loads all available plugins during startup and users can choose which ones should be used in the plugin manager window (cf. Figure 51). Additionally, it is relatively easy to provide new plugin interfaces and integrate them into the PMMT, because they are clearly separated in the application's architecture.

7.2.12 Usability (NFR02)

The PMMT application aims to provide a simple workflow and keep actions close to the items acted upon. Most of the features described in the functional requirements and in Chapter 6 are aimed to support the fulfillment of this requirement. A usability study has been conducted for evaluation, which showed very positive overall feedback and is explained in more detail in Section 7.3.

7.2.13 Maintainability (NFR03)

The architecture of the PMMT, described in detail in Section 5.3, is designed modularly and built up in layers, which helps to integrate future extensions at the necessary layer and only requires changes to the modules directly interacting with the extension. The needed boilerplate code for, e.g., new model items is also kept to a minimum, because all current model items are based on a base class that contains all common functionality.

7.2.14 Portability (NFR04)

Since the PMMT is developed using C++ and the Qt framework, and all used third party libraries are also available in source code, it can be compiled for every platform that provides a C++17 standard conforming compiler toolchain and standard library. This includes Apple macOS, GNU/Linux, and Microsoft Windows. For the development of the PMMT application a Docker[1] image has been used to enable development on all platforms for which a Docker engine is available. Docker itself is based on light-weight virtual machines, named containers, and keeps the development environment isolated inside the docker container without affecting the host operating system.

7.2.15 Data Compatibility (NFR05)

The project data of the PMMT is saved to disk using the cereal third party library (explained in Section 5.4). The data on disk contains the version it was written with, and thus the PMMT is able to open files written by older versions. This ensures that saved projects can be opened with future versions of the PMMT application and are not lost by updating to a newer application version. On the other hand, this also implies that future application versions need to support all previously supported features or explicitly take care of such incompatibilities. If needed, a future extension of the PMMT could also support saving in an older project format, which is a minor extension for the serialization interfaces, but introduces other problems, e.g. what to do with new features not yet available in older formats.

7.2.16 Performance (NFR06)

A special attention was paid to the PMMT's performance throughout the development process: design decisions, technology choices, software architecture design, and finally

[1] `https://www.docker.com/`

the implementation. The functionality tests in the development phase as well as the user study showed no application freezes or other performance problems. Since both, the functionality tests and the user study, worked with relatively small models, which are comparable in size with the example in [50], a more comprehensive performance and scalability evaluation of the PMMT should be conducted as future work.

7.2.17 Overview

All of the functional and non-functional requirements identified in Chapter 4 have been fulfilled completely, which is shown in Table 8 below.

ID	Requirement	Fulfillment
FR01	Graphical PM editor	✓
FR02	Routing of links	✓
FR03	Entry editor	✓
FR04	Inline text editing	✓
FR05	Context sensitive menus	✓
FR06	Toolbox	✓
FR07	Structural overview	✓
FR08	Plausibility checks	✓
FR09	Support for runnable model	✓
FR10	Graphic export	✓
NFR01	Extensibility	✓
NFR02	Usability	✓
NFR03	Maintainability	✓
NFR04	Portability	✓
NFR05	Data Compatibility	✓
NFR06	Performance	✓
Legend:	completely fulfilled	✓

Table 8: Fulfillment of the identified requirements

7.3 Usability Study

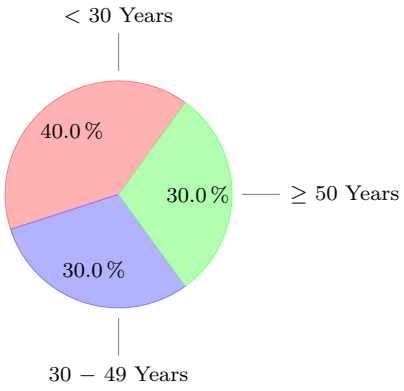
This section explains the details of the user study, that has been conducted to evaluate the PMMT's usability.

7.3.1 Study Setting

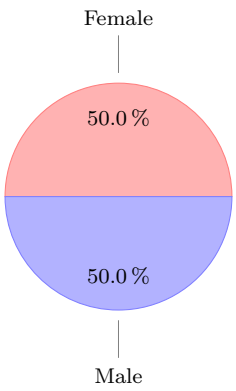
The study was conducted by teleconference and screen sharing, due to the physical distancing measures involved in the fight against the COVID-19 pandemic. The study's setup was the same for all participants. It started with an introduction to the Peer Model

that explained all the individual parts based on Chapter 3. After that the PMMT's user interface was explained in a quick hands-on overview. Finally, the participants had to complete the modeling assignments, which are explained in detail in the next section. Every participant's result was documented using a questionnaire (cf. appendix Chapter A).

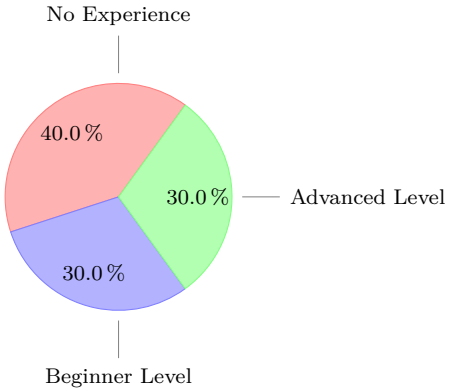
The group of 10 study participants was structured as shown in Figure 54. 4 participants were younger than 30 years, 3 were between 30 and 40 years old, and 3 were 50 years or older. 5 were female and 5 male. 3 had an advanced level of previous modeling knowledge, 3 had beginner level, and 4 had no previous knowledge. 5 had a highest education level of the Austrian matura (general university entrance qualification) and 5 are currently studying at a university or already had their degrees. 6 of the participants work in software engineering, either development or quality assurance, the others are a public servant, a scientific researcher, a pharmaceutical drug safety officer, and a student office clerk.



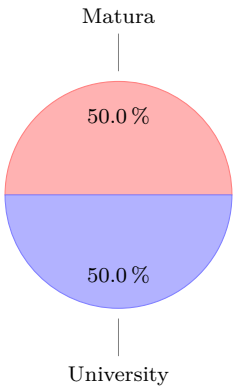
(a) by age



(b) by gender



(c) by previous modeling knowledge



(d) by highest level of education

Figure 54: Group structure of the study participants

7.3.2 Modeling Assignments

At first the participants were presented the **TomatoFactory** peer (shown in Figure 55, which was modeled after the peer in [51]) printed on paper and asked to model it in the PMMT, to familiarize themselves with the editor.

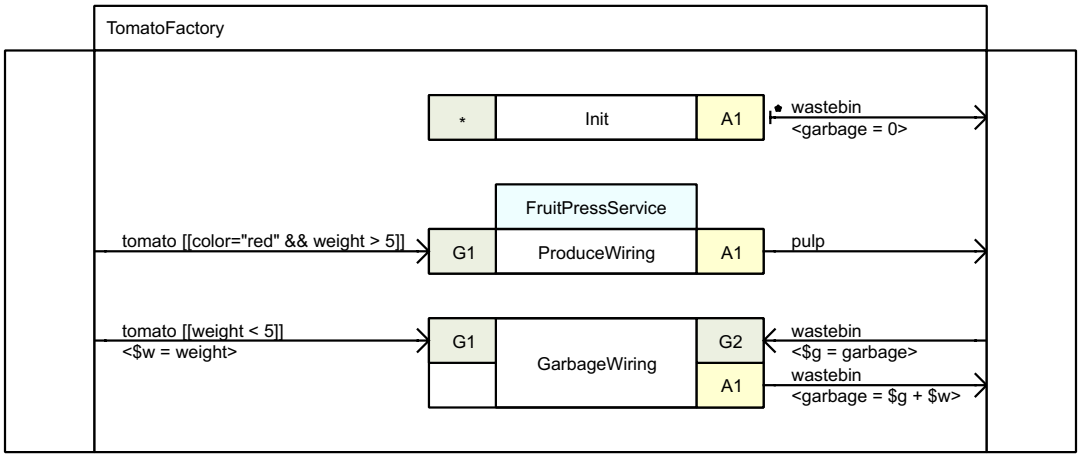


Figure 55: TomatoFactory peer (modeled after [51])

Following that, participants were asked to extend their tomato factory model according to the following instructions:

1. Create new peer **KetchupBottleFactory**.
2. Send produced **pulp** to the **KetchupBottleFactory** peer using the action's **DEST** property.
3. Create new entry types: a) **ketchup** b) **salt** c) **sugar** d) **vinegar**
4. Create new wiring **CookWiring** with following content:
   a) Guard that takes 20 **pulp** entries.
   b) Guard that takes 4 **sugar** entries.
   c) Guard that takes 2 **vinegar** entries.
   d) Guard that takes 1 **salt** entry.
   e) Service **CookingService** that combines and pasteurizes the ingredients.
   f) Action that delivers all created **ketchup** entries to the POC.

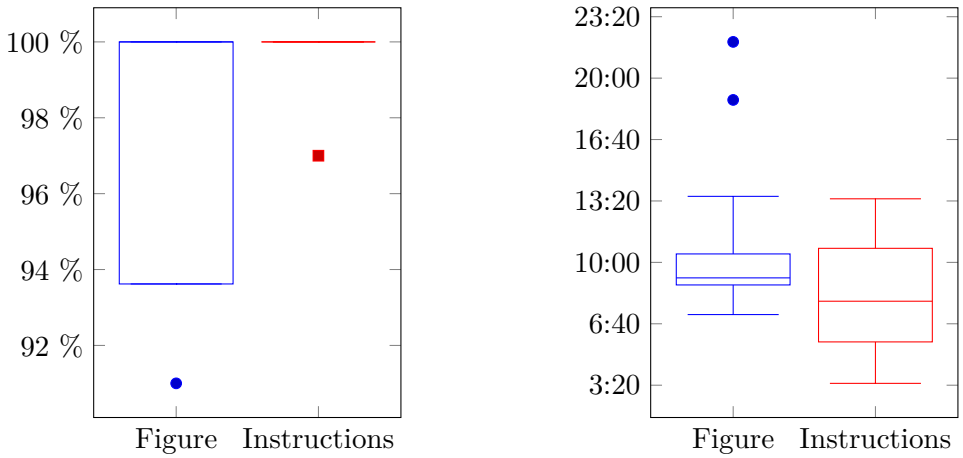
The study coordinator observed the participants while they worked on the modeling assignment. The participants were allowed to ask any questions arising during the assignment for them to complete the tasks. All the asked questions and the provided answers and hints were documented, which are to be evaluated in the study results.

7.3.3 Results

Assignments

Figure 56a shows to what percentage the participants completed the study assignments. In the first modeling assignment the participants had to use the PMMT to model a Peer Model presented as a figure, which was completed 100 % by 5 participants. Another 3 finished the presented model completely in the graphical editor, but did not create the used properties in the entry type editor window. The models of the 2 remaining participants each contained one mistake, e.g., a wrong link source operation or a wiring with only one action link A2 instead of A1. Only one of those 2 results is shown as an outlier in the figure, because the model also missed the properties in the entry type editor window. The second modeling assignment with the textual instructions was completed 100 % by 9 participants. Only 1 participant had a wrong count specifier on a created guard link.

The time needed for completing the individual study assignments is presented in Figure 56b. In the first assignment were two outliers with just below 22 and 19 minutes, both of the participants do not have a background in software engineering. The others range from 7 minutes 10 seconds to 13 minutes 35 seconds with a median of 9 minutes 10 seconds. 6 of the participants finished below 10 minutes, 5 of them with a software engineering background. This leaves only 1 participant with a software engineering background that took longer than 10 minutes. The second assignment has a median finish time of 7 minutes 54 seconds and only 2 participants took longer than the upper quartile of 10 minutes 46 seconds. 3 participants were faster than the lower quartile of 5 minutes 41 seconds and 1 other paricipant finished below 6 minutes, which makes a total of 4 participants faster than 6 minutes.



(a) Completion rate

(b) Completion time (in min:sec)

Figure 56: Results of the modeling assignments

The participants asked for help regarding the following problems, which arised in the assignments. The most common questions, which were asked by 5 participants each in the second assignment, were how the DEST property is to be specified on the action link and how the count is specified on the guard links. 4 participants asked where the link source operation can be changed. 3 participants had a problem with the creation of new services and links, which need to be dragged into the wiring. 2 participants each had problems with:

- an issue in the PMMT regarding the peer content edge, where links could be dragged outside the peer content,
- finding the button to open the entry type editor,
- the visual similarity between subpeers and wirings with a service, and
- starting and committing text field edits, which is done via double click and the return key, respectively.

1 participant each had a question

- how to create a new peer,
- how to reorder the links within a wiring,
- regarding the usability of the entry type editor, and
- how to assign a value to an entry property on a link.

Feedback

An overview of the participants' questionnaire ratings of the PMMT is shown in Figure 57. The general design of the graphical user interface was rated clear by 9 participants and rather clear by 1 participant, while all 10 participants think the readability and font size is suitable.

The next questionnaire section, regarding the graphical editor, concerned modeling PM models with the PMMT. The handling was rated simple by 3 participants, rather simple by 6 participants, and rather complicated by 1 participant. The intuitiveness of the editor interface was rated rather simple by 4 participants and simple by 6 participants. 9 participants rated the modeling workflow practical and 1 participant rated it rather practical.

In the questionnaire section regarding the PMMT's plausibility check the overall handling was rated rather simple by 6 participants and simple by 4 participants. All 10 participants rated the user guidance from the presented error message in the table to the actual error within the model with simple. The error messages of the plausibility checks were rated rather comprehensible by 2 participants and comprehensible by 8 participants.

The last section in the questionnaire concernes the handling of PMMT's code generator, which is rated rather simple by 1 participant and simple by 9 participants.

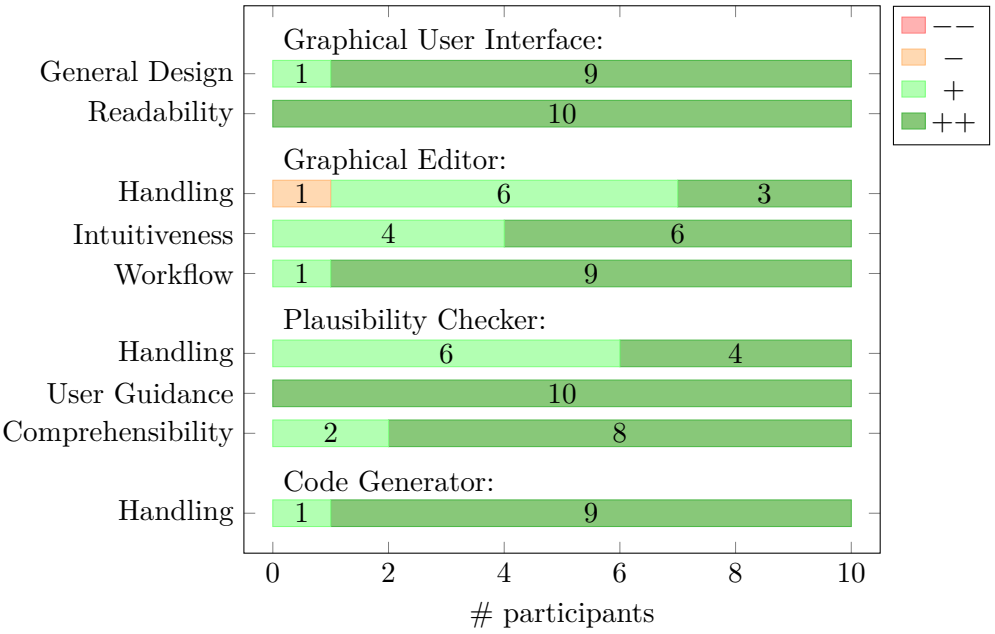


Figure 57: Questionnaire ratings for the PMMT

The textual feedback from the questionnaire was generally very positive. The participants liked the very intuitive workflow, the clear and simple editor interface, and that the handling of the editor seemed very familiar. This also resulted in the opinion that there is no need of prior knowledge to finding one's way around the modeler. One participant highlighted the good performance when moving items within the editor. Automatically hiding the detail information of links that do not contain meaningful data was also positively received. When this information is displayed, the different formatting with single or double square brackets, angle brackets, or curly braces helped the users to identify the contained information. One participant found it very convenient that links are automatically connected to containers when the wiring is moved and the links collide with the container border.

The questionnaire also revealed some known bugs in the code that handles rearranging links within a wiring and keeping the relative connector position accordingly. The problem that was mentioned most often in the questionnaire is that text changes in the editor are discarded if not commited with the return key. This should be changed to only discard on pressing the escape key and commit the changes in any other case. Only accepting *ALL* and *NONE* in the count specifier if they are written in uppercase was also mentioned as improvable by accepting regardless of the case. One participant also noted the missing visual feedback when pressing the tool bar buttons (only in Apple macOS) or that there is no success message, when no problems were found in a plausibility check. Access to

the plausibility check window was also mentioned as being too hidden in a menu, which should also be accessible through a tool bar button.

Apart from fixing the known bugs and remedying these shortcomings, the study participants also suggested some features to further improve the usability. One participant suggested to automatically change a wiring's height to the minimal used height by the links when double-clicking the lower border, which is similar to the automatic width calculation of columns in almost all spreadsheet applications. A link's currently not edited text fields should be hidden while editing another text field to ensure that they do not overlap or otherwise impede editing the contents and thus always keep the currently edited text readable. Another participant suggested to connect links automatically to the container on the corresponding side of the wiring. A further suggestion was to run the model checks automatically after changes in the model, but also keep the feedback non-intrusive to keep the users focus on their tasks.

Conclusion

The usability study's sample size is admittedly quite small, but nevertheless a positive trend can be recognized by the results. Although the study revealed some minor problems in the graphical user interface that should be addressed in the continued development of the PMMT, the overall feedback and ratings of the application were very positive in nature. While most of the problems that have arisen can easily be fixed in the continued development of the modeler, others must be addressed by the PM Technical Board, e.g., the visual similarity between the graphical representations of a subpeer and a wiring with a service. The improvements suggested by the study participants should also be considered for implementation as most of them seem quite practical. Finally, another usability study in a larger setting should be conducted in the future to further evaluate the PMMT and probably compare it directly to other graphical editors.

7.4 Implementation in Retrospect

The previous sections have already analyzed how the PMMT compares to the related tools and whether the found requirements are fulfilled by the application. This leaves a retrospective view of the implementation of the PMMT to be discussed in this chapter.

7.4.1 Technology Choices

C++ as the chosen programming language and the Qt framework as platform abstraction and graphical user interface library proved to be a good fit for the development of the PMMT. A lot of the provided window and widget features could be used out of the box and were easily integrated in the application. The graphics scene, used for the graphical editor, was different than regular widget-based graphical user interfaces and took a little longer to get used to, but with the excellent documentation provided by Qt this was not too big of a problem. Additionally, Qt's signal-slot mechanism for connecting events and

event handlers was easy to use and helped to keep the concerns of single components clearly separated.

A lot of the implementation was done on Apple macOS, while automatic builds were executed on a Microsoft Windows machine every time the code was pushed to the server repository. Later in the development phase, the build system was also set up as a docker container, such that it could be easily reproduced without the need of manually setting up the build environment.

7.4.2 Architecture

The layered and modular application architecture, described in Section 5.3, turned out to work well during implementation of the PMMT and grouping related features in a module also helped to structure the code. Necessary extensions to the PMMT that emerged during the implementation phase were easily implemented into this clearly structured code base.

Furthermore, the conceived plugin interface architecture (cf. Section 5.3.2) is already being used to create a general code generation plugin that supports a multitude of target programming languages via simple configuration [42]. This work also helped to reveal issues of the plugin interface during the development phase, and thus improve the quality of the interface for future users.

7.4.3 Third Party Libraries

The cereal and spdlog libraries were very useful in the beginning of the development phase, because they contributed a lot to the development speed. They were easily integrated into PMMT's code base, while still keeping a thin abstraction layer around spdlog and keeping the touching points with spdlog at a minimum, such that it could be exchanged for another library without much effort if that should be necessary in the future. Although this was not done in the same fashion for cereal in a trade-off for type safety, the overlaps between PMMT code and cereal's library code are limited to only the modeldata and serialization modules.

The integration of libavoid for automatic routing of links was not quite as straightforward at the start, because overlapping connectors get automatically moved apart from each other by libavoid's routing algorithm, so that they do not overlap anymore. Since the links of PMMT need space around their lines to show the link's properties, this feature of libavoid is not as helpful for PMMT as it is for graphic diagram editors. At the time of implementation it was not possible to disable exactly this feature, so a workaround was implemented, which is described in Section 5.4.3).

CHAPTER 8

Future Work

In this chapter several opportunities for future research around this thesis' results are presented. Since the aim of this thesis was to create an MVP, some of the following opportunities are features that were not deemed important enough to be implemented as part of this thesis. They could be implemented in the future, to close the gap to the related tools of Chapter 2. Other features might present a different modeling approach or view to the same underlying data, which could improve the usability of the PMMT. Finally, since the Peer Model itself evolved over the course of writing this thesis, there are new features that need to be supported in the graphical editor and the underlying model.

8.1 Localizability

The implementation of PMMT already uses the Qt framework's facilities for internationalization of displayed texts. It just does not provide other languages than English at the moment, because this was not deemed important enough for the MVP. This means that extending the application to support other languages than English should be a quite straightforward task, and could also be accomplished by crowd-sourcing it as part of the planned open source development of the PMMT.

8.2 Live Debugging of the Peer Model

An interesting extension to the PMMT, which would make a huge difference in developing an application, is to integrate the possibility of debugging the designed model. This would include single stepping through code, inspecting all the data, or setting break points to halt execution. Additionally to being a useful feature for the developers, this would also bring the PMMT a lot closer to being a full integrated development environment (IDE).

8.3 Universal Plugin Containers

At the moment, plugins for the PMMT have to be compiled separately for each supported platform to create the required binary files. As part of opening the PMMT's development to the open-source community, a build server or service should be implemented that automatically compiles the code for all supported platforms and creates a universal plugin container. This container can then be used by the PMMT application to extract only the binaries matching the running application's platform. This approach would greatly improve the plugin handling for end users, because they do not need to worry about the plugin's application platform.

Based on this universal plugin containers, a plugin server can be created, which hosts the plugins for PMMT applications to download. This server can also distribute digital signatures of the plugins to guarantee that they have not been modified by any third party.

8.4 User Interface Extensions

Some possibilites of future extensions of the user interface have already been discussed in Chapter 6: (a) replacing the project dialog with a welcome page (cf. Section 6.2.1), (b) support for dragging MDI windows out of the main window to improve multi-display support (cf. Section 6.2.6), (c) extending model explorer functionality with mirrored selections and better support for *drag and drop* (cf. Section 6.3), (d) an alternative tree view in the entry type editor as well as additional supported data types (cf. Section 6.4), (e) a project-wide full-text search for finding and opening items (cf. Section 6.5), (f) additional support for the *select and click* workflow (cf. Section 6.6), and (g) improved support for plugin preferences in the plugin framework (cf. Section 6.8). The following sections present possible future extensions that have not yet been discussed.

8.4.1 Open Peers from the Tool Bar

Currently, new peers are created from a tool bar button and existing ones are opened from the model explorer by double clicking the tree item. A tool bar menu with a menu item to create new peers and a list of menu items representing the currently existing peers could be used to unify access to new and existing peers. The same menu item could then seamlessly be included in the menu bar.

8.4.2 Hierarchical Error Highlighting

The CPN Tools hierarchically mark all items as erroneous if they contain at least one erroneous child item. This enables users to see at the first glance if there are any errors in their model. Since that also helps users in finding the errors, this would be a nice feature to integrate in the PMMT.

8.5 Expression Model for Link Definition Parts

In the current implementation of the PMMT a link's definition parts, especially the query selectors, assignments, and properties, are entered and stored as plain text, and thus requires the used model checker and code generator plugins to parse the entered selectors themselves and report errors. In the future, the entered text should be converted to an expression model directly in the editor. This way the user interface can be easily changed to a wizard-based approach, where users can choose the individual parts of these expressions from a list (e.g. like the formula editor in Microsoft Excel), and even if the user interface stays a text box, the storage already is an expression model that is also passed to the different plugins, which then do not have to implement a parser, but rather can just check the expression model for unsupported expressions.

8.5.1 Syntax Highlighting

Based on this expression model, it would be useful to highlight the entered values according to the expected syntax, such that users can more easily see if there is a problem when they are just entering the text. Since the underlying Qt controls already provide support for syntax highlighting, this can easily be integrated in the PMMT.

8.5.2 Semantic Text Completion

The users would also benefit greatly if the text input of these link parameters provided semantic text completion. For example, if the user starts typing an assignment, the editor can provide a list of suggestions matching the already typed beginning, which the user can choose from in order to complete the input.

8.6 Project Search Box

Currently, there is no way to search for the usage of a specific text in the PMMT project. This is usually not too big a problem for small projects, but it might become a problem for developers as the projects grow. To improve developer efficiency, a global search box could be provided (e.g. in the tool bar) that supports searching for occurrences of an entered text either in the whole project, or in a narrower context, e.g., only in link properties or element identifiers.

8.7 Support for Peer Model Extensions

Kühn et al. already introduced the concept of composable design patterns into the Peer Model [55]. They have not been implemented yet, because they were deemed not important enough for the MVP. Nonetheless, they should be supported by the PMMT in the future. Once there is support for the patterns themselves, the PMMT could be further extended to also support libraries of patterns to be stored independently of

the project file. This would enable one user group to focus on the development and implementation of specific patterns to be used by others, and thus bring another level of abstraction. The end users of this pattern libraries can then focus solely on their problem at hand and reuse existing tested coordination patterns.

Craß et al. present a decentralized access control model suitable for the Peer Model in [24]. It uses a PM meta model and additional containers in each peer to store the policy information that controls interactions between autonomous peers. Future work should put a focus on supporting the design of access control within the Peer Model, because security grows more important every day. Additionally, support for the meta model would also enable the full feature set of the PM Java enterprise implementation [20], which makes use of additional containers for storing the current state of the PM and for dynamically allocating new PM items.

The concept of distributed flexible wiring transactions (FWTX) presented by Kühn in [50] was not considered for the PMMT, because its functionality can also be bootstrapped by using other PM features. This should be implemented in future extensions of the PMMT, because it simplifies the visualization of the PM by abstracting the transactional features that otherwise have to be modeled explicitly.

Another extension not implemented in the MVP are assertions in the Peer model, which have been introduced by Kühn et al. in [56]. They present asynchronous runtime assertions, which are closely related to links and use the already available query mechanism of the Peer Model.

8.8 Type System for Peer Model Items

In the current PMMT implementation peers, wirings, and services are specific instances in the designed model, and it is the responsibility of the code generator plugins to transform them into runnable code, which might need further adaption by the developer. Creating a type system for the PM items, like classes and objects in the object-oriented programming paradigm, where the user first designs the type of an item and then instantiates it in a concrete model, could improve the reusability of already designed coordination logic. This approach also needs a new concrete model editor, where the more generally designed item types are instantiated and configured to fit a very specific and concrete use case. It should also be considered how this approach would work together with the composable design patterns. However, it must be considered that changes to a peer type might have far-reaching unpredictable consequences to other peer instances than the one currently edited. This situation might not always be desirable.

8.9 Background Task for Model Checks

At the moment, the model checks have to be started explicitly by the user to calculate the current errors in the model, which are then displayed in the result table. Erroneous items

are only highlighted as such if they are focused upon by double clicking the entry in the result table. For the users it would be helpful if the model checks were run continuously in a background task, and found errors were highlighted in all views of the model on the fly as soon as they were discovered. This would greatly shorten the time between a user action and the visualization of the error directly in the editor. Additionally, the actions would be reduced a user has to take to find errors in the designed model.

8.10 Periodic Code Generation

Similar to the model check plugins, the code generators are only run when explicitly requested by the user clicking the export button in the tool bar. This could also be executed periodically in a background task, to always keep an up-to-date version of the generated code on disk, which already contains the latest changes in the editor. As this might cause a higher performance load depending on the computer, it would be useful to provide options for disabling this feature, such that each user can choose for a specific project whether code generation should be run periodically in the background.

8.11 Deployment Support

The BPEL Designer Project and BPMN2 Modeler provide support for the deployment of a designed business process or workflow to an application server, which then executes the modelled workflow. Creating a Peer Model application server with the ability to host Peer Model applications would prove useful, as such a Peer Model server could be used to coordinate and orchestrate several existing services. This can be seen as an addition to the code generation plugins, which also support creating a runnable instance of the designed model. To accurately model the distributed nature of the Peer Model, special attention must be paid to the deployment to distributed PM runtimes.

8.12 Simulation

The integration of simulation features into the PMMT would greatly help users in understanding their designed models and also to look for logical errors in their designs. This includes providing inspection features that can, e.g., display the values of entries, like the simulation features in CPN Tools. Another interesting aspect would be to provide animation of the involved Peer Model items directly in the graphical editor, as described in Section 2.5.3. This visual feedback could be useful for developers to understand the flow of entries in their model. It might also be useful to create the possibility to choose the executing Peer Model implementation for the simulation, such that the implementations can be compared more easily. It might be useful to support the monitoring features of the Peer Model Monitoring Tool [26] directly within the graphical editor, either based on the same logging format or on a new interface without the need of log files.

8.13 Model Checking and Verification

It would be interesting to see verification tools for the Peer Model in the future, which can be used to prove the correctness of designed models, and integrate them in the PM toolchain (cf. Figure 1). The PMMT could generate PM-DSL, which could then be verified. Additionally, graphical feedback of the verification could then be integrated into the PMMT. This would greatly improve the application range of the PMMT and the Peer Model itself.

8.14 Performance Analysis

Analyzing the performance of a designed model and thus identifying possible bottlenecks in the application is an important feature to finetune an already working application or workflow. As such, the performance analysis of a designed Peer Model should provide statistics for peers, wirings, or links, which tell the developer about entry throughput, waiting times, partial wiring firings (e.g. when not all guards are satisfied), or raised exceptions. This could be integrated into the PM toolchain analogously to the previous section.

CHAPTER 9

Conclusion

This thesis introduced the Peer Model Modeling Tool, the first graphical modeler specifically tailored to the Peer Model. Its main goal is to support users of the Peer Model in designing their models. Starting off with the already established graphical notation of the Peer Model, the visualizations of link operations were adjusted to fit both an interactive editor and an exported Peer Model diagram. The implemented application is intended as a starting point towards a full-fledged graphical development environment for the Peer Model. Thus the implementation focused on a modular design that can be easily extended with new features in the future. Already existing and established graphical tools for other underlying models have been evaluated and compared with regard to their features, and subsequently been used to gather requirements for the PMMT. With the requirements in mind, several technologies have been evaluated for their suitability before C++ and the Qt framework were selected as foundation for the PMMT.

Previously used generic drawing tools for PM designs were not suitable for error checking or code generation, because they were purely graphical diagrams and thus lacked an underlying semantic model. The PMMT allows users to visually design their PM models and have an underlying semantic model that represents their graphic models. This is the basis for all further use of the designed model, like the implemented model checking and code generation features. Since the PM is still evolving and new implementations of the PM runtime are developed, the model checker and code generator interfaces are based on a plugin system. Thus new plugin implementations supporting new PM runtimes can be provided without the need of modifying the application itself. A thesis that is currently in progress takes this a step further by creating a plugin based on a template language [42] that can support many different target formats.

The focus during development of the graphical editor was to create user interface workflows that are as simple as possible and that are intuitive for the users. This resulted in a workflow similar to a self-assembly kit, where new items can be created by dragging them from the tool box and dropping them in the model editor at the desired place.

Rearranging existing items in the graphical editor is also as simple as dragging them to the desired new place and editing them is mostly accomplished via context menus.

Subsequently, the PMMT was evaluated against the already established graphical editors for other tools and the fulfillment of the established requirements was checked. Additionally, an initial user study has been conducted to investigate the usability amongst the target user group. The generally very positive results of this study show that the implementation of the user interface and workflows is going in the right direction. Finally, this thesis concludes with discussing possible future research topics and extensions to the PMMT.

Appendices

APPENDIX A

Usability Study

This chapter describes the details of the questionnaire used in the usability study. Section "Aufgabe" was used to evaluate each participant's performance in the modeling assignments (cf. Section 7.3.2) and, thus, is the only part of the questionnaire that was filled out by the author. After the assignment was completed, the participants filled out the remaining parts of the questionnaire. In Section "Angaben zur Person" the participants provided information about themselves, like age, gender, job description, highest level of education, and whether they had previous modeling knowledge. Next, in Section "Bewertung" they rated the PMMT in the categories (a) general user interface and readability, (b) modeling (handling, intuitiveness, workflow), (c) plausibility checks (handling, user guidance, comprehensibility), and (d) code generator handling. Finally, the participants answered open questions about their impression of the PMMT in Section "Fragen zu PMMT", e.g., what did they miss or what was positive/negative.

Aufgabe

1. Modellieren nach Vorlage

Zeit: _____

Hilfe: _____

Ergebnis: _____

2. Modellieren nach Anleitung

Zeit: _____

Hilfe: _____

Ergebnis: _____

Angaben zur Person

3. Alter: ☐ unter 30 Jahre ☐ 30 - 49 Jahre ☐ 50+ Jahre
4. Geschlecht: ☐ männlich ☐ weiblich ☐ divers
5. Beruf: _____
6. Höchster Bildungsgrad:
   - ☐ Pflichtschule
   - ☐ Fachschule / Lehre
   - ☐ Matura / Meister
   - ☐ Universität
7. Vorkenntnisse Modellierung:
   - ☐ Keine Erfahrung
   - ☐ Anfänger/in
   - ☐ Fortgeschrittene/r
   - ☐ Experte/Expertin

Bewertung

Bewerten Sie folgende Eigenschaften:

Allgemein

8. Gestaltung der Oberfläche: übersichtlich ☐ ☐ ☐ ☐ unüberschaubar

9. Lesbarkeit und Schriftgröße

   ☐ zu klein ☐ passend ☐ zu groß

Modellierung

10. Bedienung kompliziert ☐ ☐ ☐ ☐ einfach

    Wie war die Handhabung des Modelers?

11. Intuitivität schwer/aufwändig ☐ ☐ ☐ ☐ einfach/schnell

    Wie waren die benötigten Aktionen / Bausteine zu finden?

12. Workflow umständlich ☐ ☐ ☐ ☐ praktisch

    Wie waren die Tätigkeitsabläufe?

Plausibilitätsprüfung

13. Bedienung kompliziert ☐ ☐ ☐ ☐ einfach

    Wie war die Bedienung der Plausibilitätsprüfung?
    (zB: Fenster öffnen, Prüfung starten)

14. Benutzerführung kompliziert ☐ ☐ ☐ ☐ einfach

    Wie war die Führung von der Problemanzeige zum verursachenden Objekt?
    (zB: Finden des problematischen Editorelements ausgehend vom Fenster der Plausibilitätsprüfung)

15. Verständlichkeit unverständlich ☐ ☐ ☐ ☐ verständlich

    Wie verständlich waren die Problemmeldungen?

Code Generator

16. Bedienung kompliziert ☐ ☐ ☐ ☐ einfach

    Wie war die Bedienung des Code Generators?

Fragen zu PMMT

17. Welche Funktionen fehlen besonders?

18. Was ist positiv aufgefallen?

19. Warum sind diese Dinge positiv aufgefallen?

20. Was ist negativ aufgefallen?

21. Warum sind diese Dinge negativ aufgefallen?

22. Was wäre eine bessere Lösung?

List of Figures

List of Tables

Listings

Acronyms

API application program interface. 47

ASM Abstract State Machine. 2, 8–12, 31, 45, 127

AsmEE ASM Eclipse Environment. xiii, 8–12, 37–39, 43, 100

ASMETA ASM metamodeling framework. 8, 10, 11, 38, 39, 41, 42, 100

BDD Binary-Decision Diagram. 12, 43

BPEL Business Process Execution Language. xiii, 2, 12–15, 18, 20, 37–41, 100, 117, 127

BPMN Business Process Model Notation. xiii, 2, 15–20, 37–41, 56, 57, 100, 117, 127

CPN Colored Petri Net. xiii, 2, 20–26, 37–43, 56, 98–100, 114, 117, 127

CSS Cascading Style Sheets. 65, 66

CTL Computation Tree Logic. 11, 30, 43

CTL Computation Tree Logic. 3, 76–78

ECT Extensible Coordination Tools. xiii, 2, 27–31, 37–43, 100, 127

FIFO first in, first out. 27, 28

FSM finite-state machine. 8, 9, 127

fUML Foundational UML. 35, 36

GUI graphical user interface. 5, 63–66, 129

HTML Hypertext Markup Language. 65

IDE integrated development environment. 113

ISO International Organization for Standardization. 31

JS Javascript. 65

JSON Javascript Object Notation. 65, 75–78, 131

JVM Java Virtual Machine. 25, 64

LTL Linear Temporal Logic. 11, 43

MB megabyte. 65

MDI Multiple-Document Interface. 61, 62, 72, 83, 84, 86, 114

MVC Model-View-Controller. 71

MVP minimal viable product. 55, 56, 62, 81, 113, 115, 116

ODE Orchestration Director Engine. 14

OMG Object Management Group. 15, 31, 32, 35, 36, 127

PDF Portable Document Format. 88, 98, 102

PIC Peer-In-Container. 47, 48, 52, 53, 56, 68, 93

PM Peer Model. 2–6, 36, 39, 40, 42, 51, 55–59, 61, 62, 74, 97, 101, 102, 104, 108, 110, 116–119

PM-DSL Peer Model domain-specific language. 3, 57, 118

PMMT Peer Model Modeling Tool. 4–6, 55–59, 61–63, 65–68, 70, 71, 73–76, 78, 79, 81–84, 87, 88, 94, 95, 97–111, 113–120, 123, 126, 128, 129

PNG Portable Network Graphics. 98

POC Peer-Out-Container. 47, 48, 53, 56, 68, 93, 106

PSCS Precise Semantics of UML Composite Structures. 35

QIA Quantitative Intentional Automata. 30

RAII Resource Acquisition Is Initialization. 64, 66, 68

SAT propositional satisfiability. 12, 43

SOA service-oriented architecture. 15

SOAP Simple Object Access Protocol. 12

SQL Structured Query Language. 47

SVG Scalable Vector Graphics. 88, 98, 102

SysML Systems Modeling Language. 31–36, 127

UDDI Unified Description, Discovery, and Integration. 12

UML Unified Modeling Language. 31, 32, 35, 36

WS-BPEL Web Service Business Process Execution Language. 2, 12, 14, 17

WSDL Web Service Description Language. 12–14, 16, 31, 127

XML eXtensible Markup Language. 13, 20, 75–78

XVSM eXtensible Virtual Shared Memory. 47, 51, 56

Bibliography

[1] BPEL Designer Project. `https://projects.eclipse.org/projects/soa.bpel`. [online, last accessed on 2018-02-27].

[2] BPMN2 Modeler. `https://www.eclipse.org/bpmn2-modeler/`. [online, last accessed on 2019-02-20].

[3] Extensible Coordination Tools (ECT). `http://reo.project.cwi.nl/reo/wiki/Tools`. [online, last accessed on 2018-11-13].

[4] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[5] Farhad Arbab. What do you mean, coordination? *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, 19:10–21, 1998.

[6] Farhad Arbab. Reo: A Channel-based Coordination Model for Component Composition. *Mathematical structures in computer science*, 14(3):329–366, 2004.

[7] Farhad Arbab, Christian Koehler, Ziyan Maraikar, Young-Joo Moon, and José Proença. Modeling, Testing and Executing Reo Connectors with the Eclipse Coordination Tools. In *5th International Workshop on Formal Aspects of Component Software (FACS)*, volume 8, 2008.

[8] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. AsmetaSMV: A Way to Link High-Level ASM Models to Low-Level NuSMV Specifications. In *Abstract State Machines, Alloy, B and Z (ABZ 2010)*, volume 5977 of *LNCS*, pages 61–74. Springer, 2010.

[9] Paolo Arcaini, Silvia Bonfanti, Angelo Gargantini, and Elvinia Riccobene. Visual Notation and Patterns for Abstract State Machines. In *Software Technologies: Applications and Foundations (STAF 2016)*, volume 9946 of *LNCS*, pages 163–178. Springer, 2016.

[10] François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and Linearity: An Algebra for Discrete Event Systems*. John Wiley & Sons Ltd, 1992.

[11] Alexandra Back and Emma Westman. Comparing Programming Languages in Google Code Jam. Master's thesis, Chalmers University of Technology and University of Gothenburg, 2017.

[12] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61 (2):75–113, 2006.

[13] Silvia Bonfanti, Marco Carissoni, Angelo Gargantini, and Atif Mashkoor. Asm2C++: A Tool for Code Generation from Abstract State Machines to Arduino. In *NASA Formal Methods (NFM 2017)*, volume 10227 of *LNCS*, pages 295–301. Springer, 2017.

[14] Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. AsmetaA: Animator for Abstract State Machines. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2018)*, volume 10817 of *LNCS*, pages 369–373. Springer, 2018.

[15] Egon Börger. The ASM Method for System Design and Analysis. A Tutorial Introduction. In *Frontiers of Combining Systems (FroCoS 2005)*, volume 3717 of *LNCS*, pages 264–283. Springer, 2005.

[16] Egon Börger. The Abstract State Machines Method for High-Level System Design and Analysis. In *Formal Methods: State of the Art and New Directions*, chapter 3, pages 79–116. Springer, 2010.

[17] Egon Börger and James K. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of the EATCS*, 64:105–127, 1998.

[18] Fred D. J. Bowden. A Brief Survey and Synthesis of the Roles of Time in Petri Nets. *Mathematical and Computer Modelling*, 31(10):55–68, 2000.

[19] Alessandro Carioni, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A Scenario-Based Validation Language for ASMs. In *Abstract State Machines, B and Z (ABZ 2008)*, volume 5238 of *LNCS*, pages 71–84. Springer, 2008.

[20] Stephan Cejka. Enabling Scalable Collaboration by Introducing Platform-Independent Communication for the Peer Model. Master's thesis, Technische Universität Wien, 2019.

[21] Lawrence Chung and Julio Cesar Sampaio do Prado Leite. On Non-Functional Requirements in Software Engineering. In *Conceptual modeling: Foundations and applications*, volume 5600 of *LNCS*, pages 363–379. Springer, 2009.

[22] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Computer Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[23] Stefan Craß, Eva Kühn, and Gernot Salzer. Algebraic Foundation of a Data Model for an Extensible Space-Based Collaboration Protocol. In *Proceedings of the 2009 International Database Engineering & Applications Symposium (IDEAS '09)*, pages 301–306. ACM, 2009.

[24] Stefan Craß, Gerson Joskowicz, and Eva Kühn. A Decentralized Access Control Model for Dynamic Collaboration of Autonomous Peers. In *Security and Privacy in Communication Networks (SecureComm 2015)*, volume 164 of *LNICST*, pages 519–537. Springer, 2015.

[25] Kevin Crowley and Robert S. Siegler. Flexible Strategy Use in Young Children's Tic-Tac-Toe. *Cognitive Science*, 17(4):531–561, 1993.

[26] Maximilian Csuk. Developing an Interactive, Visual Monitoring Software for the Peer Model Approach. Master's thesis, Technische Universität Wien, 2014.

[27] Matthias Kalle Dalheimer. Qt vs. Java: A Comparison of Qt and Java for Large-Scale, Industrial-Strenght GUI Development. Technical report, Klarälvdalens Datakonsult AB, 2010.

[28] Michel Diaz and Patrick Sénac. Time Stream Petri Nets a model for timed multimedia information. In *Application and Theory of Petri Nets (ICATPN 1994)*, volume 815 of *LNCS*, pages 219–238. Springer, 1994.

[29] Stephan Diehl. Software Visualization. In *Proceedings of the 27th international conference on Software engineering (ICSE '05)*, pages 718–719. ACM, 2005.

[30] Thomas Friese, Matthew Smith, Bernd Freisleben, Julian Reichwald, Thomas Barth, and Manfred Grauer. Collaborative Grid Process Creation Support in an Engineering Domain. In *High Performance Computing (HiPC 2006)*, volume 4297 of *LNCS*, pages 263–276. Springer, 2006.

[31] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[32] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. AsmEE: an Eclipse plug-in in a metamodel based framework for the Abstract State Machines. In *First International Conference on Eclipse Technologies (ECLIPSE-IT)*, 2007. `https://cs.unibg.it/gargantini/research/papers/asmee_eclipeit07.pdf`. [online, last accessed on 2020-05-29].

[33] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A Metamodel-based Language and a Simulation Engine for Abstract State Machines. *Journal of Universal Computer Science*, 14(12):1949–1983, 2008.

[34] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. Model-Driven Language Engineering: The ASMETA Case Study. In *Proceedings of the Third*

International Conference on Software Engineering Advances (ICSEA), pages 373–378. IEEE, 2008.

[35] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[36] Sébastien Gérard, Cédric Dumoulin, Patrick Tessier, and Bran Selic. Papyrus: A UML2 Tool for Domain-Specific Language Modeling. In *Model-Based Engineering of Embedded Real-Time Systems (MBEERTS 2007)*, volume 6100 of *LNCS*, pages 361–368. Springer, 2007.

[37] Thomas Hamböck. Towards a Toolchain for Asynchronous Embedded Programming based on the Peer-Model. Master's thesis, Technische Universität Wien, 2015.

[38] Bill Haskins, Jonette Stecklein, Brandon Dick, Gregory Moroney, Randy Lovell, and James Dabney. Error Cost Escalation Through the Project Life Cycle. *INCOSE International Symposium*, 14(1):1723–1737, 2004.

[39] Matthew Hause. The SysML Modelling Language. In *15th European Systems Engineering Conference*, volume 9, pages 1–12. INCOSE, 2006.

[40] Holger Hermanns and Joost-Pieter Katoen. The How and Why of Interactive Markov Chains. In *Formal Methods for Components and Objects (FMCO 2009)*, volume 6286 of *LNCS*, pages 311–337. Springer, 2009.

[41] Georg Holasek. Evaluation of the Peer Model Framework on a RCM4300 Evaluation Board. Bachelor's thesis, Technische Universität Wien, 2014.

[42] Maximilian Irlinger. PMTL - A Template Language for the Peer Model. Master's thesis, Technische Universität Wien, in preparation.

[43] ISO/IEC 19514:2017. Information technology – Object management group systems modeling language (OMG SysML). Standard, International Organization for Standardization, Geneva, Switzerland, March 2017.

[44] Sung-Shik T. Q. Jongmans, Francesco Santini, Mahdi Sargolzaei, Farhad Arbab, and Hamideh Afsarmanesh. Automatic Code Generation for the Orchestration of Web Services with Reo. In *Service-Oriented and Cloud Computing (ESOCC 2012)*, volume 7592 of *LNCS*, pages 1–16. Springer, 2012.

[45] Sascha Klüppelholz and Christel Baier. Symbolic model checking for channel-based component connectors. *Science of Computer Programming*, 74(9):688–701, 2009.

[46] Heather Kreger. Fulfilling the Web Services Promise. *Communications of the ACM*, 46(6):29—-34, 2003.

[47] Lars Michael Kristensen, Søren Christensen, and Kurt Jensen. The practitioner's guide to coloured Petri nets. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(2):98–132, 1998.

[48] eva Kühn. Peer Model: Agile Middleware and Programming Model for the Coordination of Parallel and Distributed Flows. Technical report, Institute of Computer Languages, TU Wien, 2012.

[49] Eva Kühn. Reusable Coordination Components: Reliable Development of Cooperative Information Systems. *International Journal of Cooperative Information Systems*, 25(04):1740001, 2016.

[50] Eva Kühn. Flexible Transactional Coordination in the Peer Model. In *Fundamentals of Software Engineering (FSEN 2017)*, volume 10522 of *LNCS*, pages 116–131. Springer, 2017.

[51] Eva Kühn and Sophie Therese Radschek. An Initial User Study Comparing the Readability of a Graphical Coordination Model with Event-B Notation. In *Software Engineering and Formal Methods (SEFM 2017)*, volume 10729 of *LNCS*, pages 574–590. Springer, 2017.

[52] Eva Kühn, Stefan Craß, Gerson Joskowicz, Alexander Marek, and Thomas Scheller. Peer-Based Programming Model for Coordination Patterns. In *Coordination Models and Languages (COORDINATION 2013)*, volume 7890 of *LNCS*, pages 121–135. Springer, 2013.

[53] eva Kühn, Stefan Craß, and Thomas Hamböck. Approaching Coordination in Distributed Embedded Applications with the Peer Model DSL. In *40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 64–68. IEEE, 2014.

[54] eva Kühn, Stefan Craß, Gerson Joskowicz, and Martin Novak. Flexible Modeling of Policy-Driven Upstream Notification Strategies. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC '14)*, pages 1352–1354. ACM, 2014.

[55] eva Kühn, Stefan Craß, and Gerald Schermann. Extending a Peer-based Coordination Model with Composable Design Patterns. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 53–61. IEEE, 2015.

[56] eva Kühn, Sophie Therese Radschek, and Nahla Elaraby. Distributed Coordination Runtime Assertions for the Peer Model. In *Coordination Models and Languages (COORDINATION 2018)*, volume 10852 of *LNCS*, pages 200–219. Springer, 2018.

[57] Olaf Kummer. Introduction to Petri Nets and Reference Nets. *Sozionik Aktuell*, 1: 7–16, 2001.

[58] Gordon Kurtenbach and William Buxton. User Learning and Performance with Marking Menus. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '94)*, pages 258–264. ACM, 1994.

[59] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic Symbolic Model Checker. In *Computer Performance Evaluation: Modelling Techniques and Tools (TOOLS 2002)*, pages 200–204. Springer, 2002.

[60] Charles Lakos. From Coloured Petri Nets to Object Petri Nets. In *Application and Theory of Petri Nets (ICATPN 1995)*, pages 278–297. Springer, 1995.

[61] Christoph Maier and Daniel Moldt. Object Coloured Petri Nets - A Formal Technique for Object Oriented Modelling. In *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *LNCS*, pages 406–427. Springer, 2001.

[62] Philip Merlin. *A study of the Recoverability of Computer Systems*. PhD thesis, University of California, 1974.

[63] Toshiyuki Miyamoto and Sadatoshi Kumagai. A Survey of Object-Oriented Petri Nets and Analysis Methods. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 88(11):2964–2971, 2005.

[64] OASIS wsbpel-v2.0-OS. Web Services Business Process Execution Language Version 2.0. Specification, Organization for the Advancement of Structured Information Standards, 2007. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[65] OMG formal/2008-01-16. MOF Model to Text Transformation Language, v1.0. Specification, Object Management Group, 2008. `https://www.omg.org/spec/MOFM2T/1.0/`.

[66] OMG formal/2013-12-09. Business Process Model and Notation (BPMN). Specification, Object Management Group, 2013. `https://www.omg.org/spec/BPMN/`.

[67] OMG formal/2017-05-01. OMG Systems Modeling Language Version 1.5. Specification, Object Management Group, 2017. `https://www.omg.org/spec/SysML/1.5`.

[68] OMG formal/2017-07-04. Action Language for Foundational UML Version 1.1. Specification, Object Management Group, 2017. `https://www.omg.org/spec/ALF/1.1/`.

[69] OMG formal/2017-12-05. OMG Unified Modeling Language (OMG UML) Version 2.5.1. Specification, Object Management Group, 2017. `https://www.omg.org/spec/UML/2.5.1/`.

[70] OMG formal/2018-12-01. Semantics of a Foundational Subset for Executable UML Models Version 1.4. Specification, Object Management Group, 2018. `https://www.omg.org/spec/FUML/1.4/`.

[71] OMG formal/2019-02-01. Precise Semantics of UML Composite Structures Version 1.2. Specification, Object Management Group, 2019. https://www.omg.org/spec/PSCS/1.2/.

[72] Marian Petre. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of the ACM*, 38(6):33–44, 1995.

[73] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Technische Hochschule Darmstadt, 1962.

[74] Chander Ramchandani. *Analysis of Asynchronous Concurrent Systems by Petri Nets.* PhD thesis, Massachusetts Institute of Technology, Cambridge, 1974.

[75] Anne Vinter Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Søren Christensen, and Kurt Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In *Applications and Theory of Petri Nets (ICATPN 2003)*, volume 2679 of *LNCS*, pages 450–462. Springer, 2003.

[76] Dominik Rauch. PeerSpace.NET : Implementing and Evaluating the Peer Model with Focus on API Usability. Master's thesis, Technische Universität Wien, 2014.

[77] Elvinia Riccobene and Patrizia Scandurra. Towards an Interchange Language for ASMs. In *Abstract State Machines 2004. Advances in Theory and Practice (ASM 2004)*, volume 3052 of *LNCS*, pages 111–126. Springer, 2004.

[78] Armin Rigo and Samuele Pedroni. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA '06)*, pages 944–953. ACM, 2006.

[79] Patrizia Scandurra, Angelo Gargantini, Claudia Genovese, Tiziana Genovese, and Elvinia Riccobene. A Concrete Syntax derived from the Abstract State Machine Metamodel. In *Proceedings of the 12th International Workshop on Abstract State Machines (ASM 2005)*, pages 345–368, 2005.

[80] Gerald Schermann. Extending the Peer Model with Composable Design Patterns. Master's thesis, Technische Universität Wien, 2014.

[81] Joseph Sifakis. Performance evaluation of systems using nets. In *Net Theory and Applications*, volume 84 of *LNCS*, pages 307–319. Springer, 1980.

[82] Kent Inge Fagerland Simonsen. On the use of Pragmatics for Model-based Development of Protocol Software. In *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE)*, volume 723, pages 179–190, 2011.

[83] Kent Inge Fagerland Simonsen. PetriCode: A Tool for Template-Based Code Generation from CPN Models. In *Software Engineering and Formal Methods (SEFM 2013)*, volume 8368 of *LNCS*, pages 151–163. Springer, 2013.

[84] Andrea Sindico, Marco Di Natale, and Gianpiero Panci. Integrating SysML with Simulink using Open-Source Model Transformations. In *Proceedings of 1st International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2011)*, pages 45–56, 2011.

[85] Jérémie Tatibouët, Arnaud Cuccuru, Sébastien Gérard, and François Terrier. Formalizing Execution Semantics of UML Profiles with fUML Models. In *Model-Driven Engineering Languages and Systems (MODELS 2014)*, volume 8767 of *LNCS*, pages 133–148. Springer, 2014.

[86] Rüdiger Valk. Petri Nets as Token Objects. In *Application and Theory of Petri Nets (ICATPN 1998)*, volume 1420 of *LNCS*, pages 1–24. Springer, 1998.

[87] Rüdiger Valk. Object Petri Nets. In *Lectures on Concurrency and Petri Nets (ACPN 2003)*, volume 3098 of *LNCS*, pages 819–848. Springer, 2003.

[88] Bernd Walter. Timed Petri-Nets for Modelling and Analyzing Protocols with Real-Time Characteristics. In *Proceedings of the IFIP WG 6.1 Third International Workshop on Protocol Specification, Testing and Verification (PSTV)*, pages 149–159. IBM Research, 1983.

[89] Lisa Wells. Performance analysis using CPN tools. In *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools (VALUETOOLS '06)*, pages 59–68. ACM, 2006.

[90] Michael Wybrow. *Using semi-automatic layout to improve the usability of diagramming software*. PhD thesis, Monash University, 2008.