FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Developing an Interactive, Visual Monitoring Software for the Peer Model Approach

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Maximilian Alexander Csuk
Matrikelnummer 0625909

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: A.o. Univ. Prof. Dr. Dipl.-Ing. eva Kühn
Mitwirkung: Projektass. Dipl.-Ing. Stefan Craß

Wien, 01.12.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Developing an Interactive, Visual Monitoring Software for the Peer Model Approach

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Maximilian Alexander Csuk
Registration Number 0625909

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: A.o. Univ. Prof. Dr. Dipl.-Ing. eva Kühn
Assistance: Projektass. Dipl.-Ing. Stefan Craß

Vienna, 01.12.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Maximilian Alexander Csuk
Herklotzgasse 28/39, 1150 Wien

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.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

First of all, I would like to thank eva Kühn for her continued support as my supervisor throughout the duration of this Master's thesis. Secondly, I am grateful for the constructive criticism and recommendations given by Stefan Craß and Thomas Hamböck. Special thanks are directed to Markus Hof, Renate Hof and Matthias Schwayer for proofreading the thesis. I would also like to express my deepest gratitude to my beloved girlfriend Karo for her everlasting faith in me. And last but not least, my mother, who has always been incredibly supportive of me and my undertakings.

Abstract

The Peer-Based Programming Model (short: *Peer Model*) is an approach for designing and implementing distributed systems, developed by the Space-Based Computing group at the Vienna University of Technology. Its structure is inspired by asynchronous message queues, tuple space communication, staged event-driven architectures and data-driven workflows.
The Peer Model approach is relatively new and no tools that aid in the monitoring and debugging process exist yet. Developers need to resort to manual methods in order to gain insight into the workings of a running Peer Model based system, which is cumbersome and time consuming. This Master's thesis fills the gap by developing a visual monitoring software for the Peer Model approach, whose purpose is to help developers analyse and debug Peer Models.
A requirements analysis is performed to explore the needs and problems developers currently face when working with the peer model by conducting personal interviews with them. The analysis is strongly linked to the identification of what data is required to enable proper monitoring and debugging, as well as the development of suitable, standardised data formats. Suitable visualisations can facilitate the understanding and increase the productivity of developers better than textual representations. Fortunately, the Peer Model is a good fit for representing it visually. A graphical notation for representing Peer Models has been established prior to this thesis, but it only captures the static structure of models. Consequently, illustrations for the dynamic events occurring within a running Peer Model system are developed and implemented in the monitoring tool.
The visual peer model notation resembles nested graph structures. Automatic layouting of these structures is a crucial part of this Master's thesis as it spares developers from having to manually arrange elements in order to obtain an informative and visually pleasing representation. Thus, a methodology for layouting Peer Models is developed. Afterwards, the implementation of the monitoring tool is discussed and finally, the developed tool is critically evaluated and potential areas for future works are presented.

Kurzfassung

Das Peer-basierte Programmiermodell (kurz: Peer Model) ist ein Ansatz zur Spezifikation und Umsetzung von verteilten Systemen, entwickelt von der Space-Based Computing Gruppe an der Technischen Universität Wien [50]. Der Aufbau des Peer Models ist inspiriert von asynchronen Message-Queues, Tupelraum-Kommunikation, Event-basierten Architekturen und datengesteuerten Workflows [51].
Der Peer Model Ansatz ist relativ neu. Aus diesem Grund existieren noch keine Tools, die bei Monitoring- und Debugging-Aufgaben unterstützen. EntwicklerInnen müssen auf manuelle Methoden zurückgreifen welche umständlich und zeitaufwändig sind, um Einblicke in die Abläufe eines Peer Model basierten Systems zu erlangen. Die vorliegende Masterarbeit schließt diese Lücke, indem eine visuelle Monitoring-Software für den Peer Model Ansatz entwickelt wird, deren Zweck es ist, EntwicklerInnen bei der Analyse und Fehlerbehebung zu unterstützen.
Um die Probleme zu identifizieren, vor denen Peer-Model EntwicklerInnen regelmäßig stehen, werden persönliche Interviews im Rahmen einer Anforderungsanalyse durchgeführt. In engem Zusammenhang damit steht die Erforschung der Datenbasis, die für angemessenes Monitoring und Debugging notwendig ist, sowie die Entwicklung von geeigneten, standardisierten Daten-Formaten. Geeignete Visualisierungen verbessern das Verständnis von EntwicklerInnen und steigern deren Produktivität besser als textbasierte Repräsentationen. Glücklicherweise eignet sich das Peer Model sehr gut für eine visuelle Darstellung. Vor dieser Arbeit wurde bereits eine visuelle Notation entwickelt. Diese befasst sich jedoch nur mit der statischen Peer Modell-Struktur, nicht mit den dynamischen Prozessen innerhalb eines laufenden Systems. Im Zuge dieser Masterarbeit werden deshalb Visualisierungen für die Vorgänge in Peer Modellen entworfen und innerhalb des Monitoring-Tools umgesetzt. Die entwickelte Notation ähnelt der von verschachtelten Graphen. Das automatisierte Zeichnen der Graphen-Strukturen ist ein wichtiger Bestandteil dieser Masterarbeit, da es EntwicklerInnen die manuelle Positionierung von Elementen erspart und für eine informative Darstellung sorgt. Aus diesem Grund wird ein Layouting-Framework für Peer Modelle entwickelt. Des Weiteren wird auf die Implementierung des Monitoring-Tools eingegangen, wobei Technologie-Entscheidungen und Software-Architektur erörtert werden. Abschließend wird das entwickelte Tool evaluiert und ein Ausblick auf mögliche zukünftige Arbeiten gegeben.

Contents

CHAPTER 1

Introduction

In many distributed software systems, a multitude of autonomous agents interact to fulfil specific goals. The interaction is often complex and requires coordination mechanisms to ensure the proper flow of information between agents. The communication is inherently concurrent, complicating the design of a distributed system even more.
The Peer-Based Programming Model (short: *Peer Model*) is a new approach developed by the Space-Based Computing group at the Vienna University of Technology [50]. The Peer Model proposes a novel way to bridge the gap between the design and implementation of distributed systems. Its structure is inspired by asynchronous message queues, tuple space communication, staged event-driven architectures and data-driven workflows [51].
Distributed systems are very hard to debug and monitor. Due to the concurrent and distributed nature, it is difficult for developers to spot bugs and errors in reasoning. This also applies to the Peer Model. The current procedure to debug problems and gain insight into a running Peer Model system is very rudimentary. Most developers resort to manual debugging means. They interact with the model and perceive the resulting events afterwards by manually checking certain parts that are of interest. However, depending on the complexity of the model, it is generally not possible to keep track of all that is happening as the amount of events may simply be too high and their interaction too complex. Also, events are happening concurrently and in a distributed fashion, increasing the burden for the developer even more. This results in a cumbersome debugging process. Especially if the developer is not familiar with the Peer Model in question, bugs and semantic errors are very hard to spot.
Efforts have been made by the group of eva Kühn to improve the debugging process by developing Peer Model simulations. These simulations generate logging data in the form of textual listings of the occurring events. While this provides a complete picture of the proceedings, the produced logs tend to get big very fast and are hard and

time-consuming to process manually. While stepping through the trace line-by-line, the developer has to keep a picture of the model in his/her mind and mentally perform the actions one after another. During that time, developers often sketch snapshots of parts of the Peer Model on paper to aid their understanding. Unfortunately, the developed simulations and the logging data they produce are tailored to a specific Peer Model use case. No agreed upon general purpose data format for logging exists yet. Subsequently, no tools exist to further process the generated logging data. This Master's thesis aims to alleviate the just presented issues by designing and developing an interactive, visual monitoring software that helps developers accomplish debugging and monitoring tasks regarding the Peer Model. A properly designed monitoring tool helps analysing a distributed system in action and enables developers to make correct assumptions and take appropriate actions. When it comes to monitoring, visual approaches have several advantages over traditional, textual strategies [85]. Rather than wading through lots of logging data, suitable visual representations make it possible to condense the information and offer a much better insight into what is going on.

The process of gathering the necessary data from distributed systems is a difficult task. The task involves the ordering of events that happen on multiple, physically independent parts of the distributed system into a single, logically correct temporal order. Previous works discuss the issues pertaining to the topic of repeatability, determinism and ordering of events in the context of multitasking systems [2, 21, 39, 53]. The task of logging, extracting and ordering event data from a running Peer Model system is not within the scope of this thesis. For this work, it is assumed that all the necessary monitoring information is available. However, the thesis establishes what data is necessary to create a monitoring and debugging tool that is capable of letting the user get proper insights into the inner workings of a Peer Model system. Furthermore, a suitable file format to manage and persist said data is developed.

The proposed way in which Peer Models are to be monitored and debugged is similar to the approach of *Post Mortem Debugging*. The running Peer Model system is put into a state in which it generates logging information of the events happening within. During this time, the Peer Model itself continues to function just as before. The developer may also actively manipulate the model with the goal of debugging the chain of events spawned by his/her actions. After an arbitrary timeframe, the logging procedure is stopped again by the developer and the generated data is consolidated. It is processed in order to obtain a series of replicable events that happened between the start and the stop of the logging procedure. Such a chain of events is subsequently called *Trace*. The trace is then loaded into the monitoring tool and serves as the input data for the monitoring and debugging process.

In general, it is neither possible nor desirable to present a single complete picture of a Peer Model system in action. The amount of event data that a Peer Model produces grows with the timeframe in which the logging process is enabled, with the complexity

of the model and with the level of activity it exhibits during that time. Proper representations and interaction techniques are required to present the information in a meaningful way and give the users the ability to actively explore and make sense of the available data. The *Visual Information-Seeking Mantra* provides an excellent framework for designing interactive visualisation software [79]. It states that users should be able to obtain an overview of the situation first, then apply filter criteria to hide uninteresting information and request details-on-demand on parts that are of interest. Craft and Cairns write:

> Overview provides a general context for understanding the data set; it paints a "picture" of the whole data entity that the information visualisation represents. Patterns and themes in the data that may be helpful can often be seen only from a vantage point that comprises the whole view. From this perspective, major components and their relationships to one another are made evident. Simply the overall shape of the data itself can provide assistance in understanding the information that is encoded. Significant features can be discerned and selected for further examination. Such features might not be readily viewable from another part of the data representation or might be obscured from certain vantage points. Revealing these features at the outset can aid the user in filtering the extraneous information so that they can complete their task more efficiently by excluding unimportant aspects of the representation. [18]

The Visual Information-Seeking Mantra will serve as a guideline as it provides an excellent basis for the development of the monitoring software. Finally, the developed monitoring software is critically evaluated in terms of the decisions regarding visualisation, technology and file formats, by comparing it to similar tools for other distributed system approaches as well as its applicability as a support tool for Peer Model developers.

This thesis is structured as follows: in chapter 2, we present related work on distributed system approaches and their tools, as well as works addressing graph layouting and visualisation. Chapter 3 provides an introduction to the Peer Model. Chapter 4 discusses the requirement analysis preceding the development of the monitoring tool. After that, the visual notation for the static Peer Model structures as well as their dynamic counterparts are introduced in chapter 5. Chapter 6 discusses the development of the framework capable of layouting Peer Model visualisations. Issues and considerations that emerged during the implementation of the tool are presented in chapter 7. Chapter 8 evaluates the developed monitoring software and compares it to similar tools based on other distributed system approaches. Lastly, chapter 9 presents potential areas for future work and chapter 10 concludes the thesis.

CHAPTER 2

Related Work

2.1 Coordination Models

Kühn, Craß and Schermann compared modelling concepts in the domain of distributed and concurrent systems [52]. Schermann also evaluated the following coordination models in his Master's thesis [78]: *Reo* [4], the *Actor Model* [36], *Petri Nets* [43], *UPPAAL* [55], *WS-BPEL* [3], *BPMN* [93] and the Peer Model itself. The criteria for comparison included flexibility, scalability and the ability to reuse components, among other features. Another area of classification was the availability of tools for designing, simulating and analysing systems developed with the respective concept.
This thesis extends the comparison by establishing criteria for an interactive, visual monitoring tool. In the following sections, the evaluated coordination approaches are introduced. In chapter 8, criteria are defined and used to evaluate tools for these approaches in comparison to the developed monitoring tool for the Peer Model approach. The works of Kühn, Craß and Schermann [52, 78] are the basis for which coordination models to include in the evaluation process. A discussion on the considered coordination models follows. A discussion on the comparable tools can be found in chapter 8.

Petri Nets

Petri Nets are a prominent concept for modelling distributed systems. In this case, the term distributed system is meant in a broader sense as Petri nets are also applicable in the domains of business process modelling, workflow management systems, chemistry and biology. After its initial implementation developed in the 1960s [61], Petri Nets continually evolved and exist in many forms today. Various extensions to the base concept

were proposed, many of which are grouped into so-called *High-Level Petri Nets*, including *Coloured Petri Nets* [43], *Timed Petri Nets* [94] and *Hierarchical Petri Nets* [25] as well as hybrid forms. These extensions add features to the base concept to increase its expressiveness and applicability.
The elements making up a Petri Net are based on an exact mathematical definition in terms of their semantics. A basic Petri Net consists of *Places*, *Transitions* and *Arcs*. A place may contain a number of *Tokens* which, in their entirety, encode the system's state. A full set of tokens within all the places in the system is called a *Marking*. A transition is the structural definition of a potential exchange of tokens. Arcs connect places and transitions and are divided into *Input Arcs* that run from a place to a transition and *Output Arcs* that connect transitions to places. Each arc is given a weight in form of a positive, non-zero integer. A transition is said to be *enabled* or *firing* once all of its input arcs have equal or more tokens in their corresponding places than their weight. Once enabled, the transition removes these tokens from the input places and creates tokens in the output arcs' places in the amount of their respective weights. The complete process happens in a single step (atomic). No predefined order for the execution of multiple transitions exists. That means that two or more transitions that could fire at the same time do so in no determined order (non-deterministic).
Basic Petri Nets offer a graphical notation for the description of distributed processes. A Petri Net can be represented as a directed, bipartite graph, where its places and transitions correspond to nodes and arcs correspond to directed edges. Figure 2.1 shows a small example Petri Net.

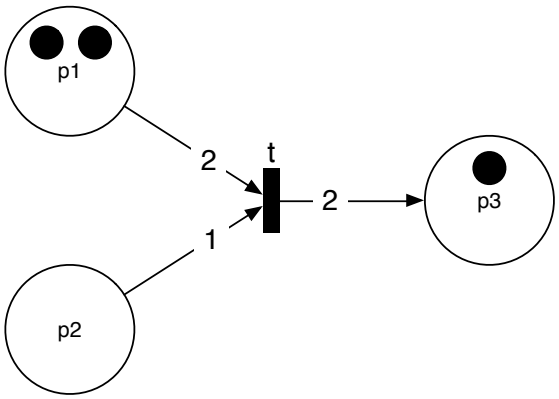


Figure 2.1: Small Petri Net example with three places (*p1, p2 and p3*) and one transition (*t*). *p1* contains two tokens, *p3* one token.

Reo

Reo presents a paradigm for composition of distributed software components and services based on the notion of mobile channels [4]. Reo understands itself as the "glue" between components or services in a distributed system. Similar to the Peer Model, it focusses on the coordination logic between components, rather than the business logic. Reo introduces a concept called *Connectors*, which serve as the basis for communication. The most simple form of connectors are called *channels*. A typical channel has one *source end* and one *sink end* and connects two components. However, channels with multiple sources and sinks are also possible. Data is written into source ends and read from sink ends. Reo also supports different semantics for channels. For example, a *FIFO channel* employs a queue and allows for asynchronous communication between two components. To support sophisticated communication semantics, Reo allows the composition of connectors and channels to form more complex connectors. Figure 2.2 shows a simple example model. For an in-depth discussion on Reo, refer to [4].

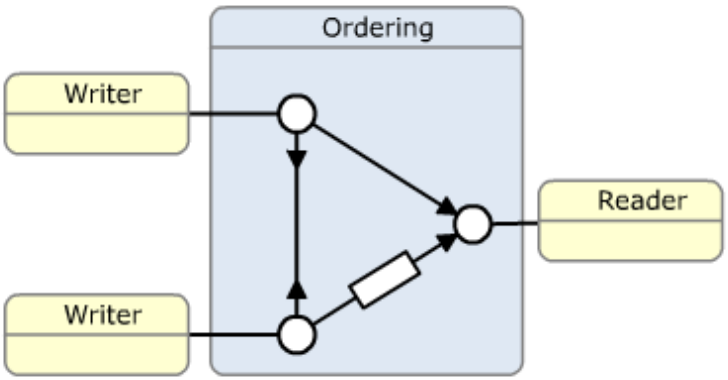


Figure 2.2: Example of a Reo model, showing how the outputs of two independent writers can be brought into order and delivered to a reader sequentially. The *Ordering* connector is the composition of three basic channels. Image taken from [74].

Actor Model

The *Actor Model* is a mathematical model of concurrent computation and was initially developed by Carl Hewitt in 1973 [36]. It follows the philosophy that everything is a so-called *Actor*. Actors communicate with each other by sending *Messages*. When an actor receives a message, it may perform the following actions:

- Send additional messages to other actors (or itself)

- create other actors
- modify its own state to affect how future messages it receives are treated

The actor model has been implemented as the concept for concurrency in several languages, including Erlang [6] and Scala [35]. Other languages, such as Java, add the missing support through libraries [46].
The Actor Model is heavily rooted in mathematical theory, but no generally accepted notation for actor models exists. Several works propose their own notation and/or implement Actor Model functionality [11,77,80]. Due to its highly dynamic nature, visual notations for the actor model are even more difficult to develop. Actors may spawn other actors, which has the potential to dramatically change the state of the system at any time. While previous works proposed approaches to describe Actor Models visually [58, 81], no tools that are comparable to the tool presented in this Master's thesis exist. Thus, the Actor Model is excluded from the evaluation.

UPPAAL

UPPAAL is an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata, extended with data types [55]. A system is structured as a network of several such timed automata running in parallel. An automaton is specified using *Locations*, which represent the state of the model and *Edges* between locations, representing state changes. Each automaton may also define *clocks*, which are used for specifying time constraints on edges. Clocks are real-valued and progress synchronously. Synchronisation between automata however, which are otherwise independent can be realised with the use of *channels*. While earlier versions of UPPAAL only supported binary synchronisation between two automata, later versions introduced *broadcast synchronisation*. In addition, variables may be specified. They behave like variables in programming languages and can be read, modified and used in calculations. The complete state of a model is defined by the locations of its automata, the current clock values and the values of the variables. Figure 2.3 shows a small UPPAAL example modelling the interaction between a user and a lamp.

BPMN

Business Process Model and Notation (short: *BPMN*) is a standard for modelling business processes [93]. It provides a graphical notation in the form of *Business Process Diagrams*, which are similar to activity diagrams from *UML* [54]. BPMN is designed to provide all stakeholders, including developers, business managers and business analysts, with a unified language for the development of business processes and bridges the communication gap between implementation and specification.

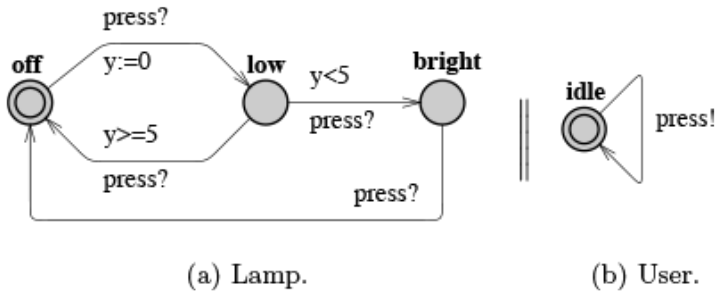


Figure 2.3: Small UPPAAL example showing the interaction between a lamp (a) and a user (b). The lamp can be in one of three states, which correspond to the three locations (off, low and bright). The user might *press* a button at anytime, which triggers an action in the lamp automaton through the use of channels. The clock y is used to distinguish between a double press and a single press. Image taken from [7].

The graphical notation of BPMN consists of four element types: *flow objects*, *connecting objects*, *swim lanes* and *artifacts*. Three elements make up the group of flow objects: *events*, *activities* and *gateways*. These flow objects are interconnected by connecting objects, which are of one of three types: *sequences*, *messages* and *associations*. Swim lanes are used to visually group and separate elements and element groups, providing a means for organising more complex diagrams. Lastly, artifacts are used to enhance the diagram with additional information for the reader. Figure 2.4 shows a small business process modelled using BPMN. For a detailed discussion on the BPMN standard, refer to [9].

WS-BPEL

The *Web Services Business Process Execution Language* (short: *WS-BPEL*) is used to specify business processes focussed on the coordination of web services [3]. There are two possibilities for describing a business process in WS-BPEL: *abstract* and *executable*. WS-BPEL aims to distinguish between *programming in the large* and *programming in the small*. Abstract business processes are used to specify systems at a high-level (programming in the large). Executable business processes on the other hand describe actual behaviour of business interactions (programming in the small).
WS-BPEL is tightly coupled to web services and the associated technologies, such as WSDL and SOAP [19]. Every interaction between participants in the business process is performed using web services.
Every BPEL process has one main *activity*. A *Basic Activity* describes the elemental steps of a process, while a *Structured Activity* describes the overall flow of a process

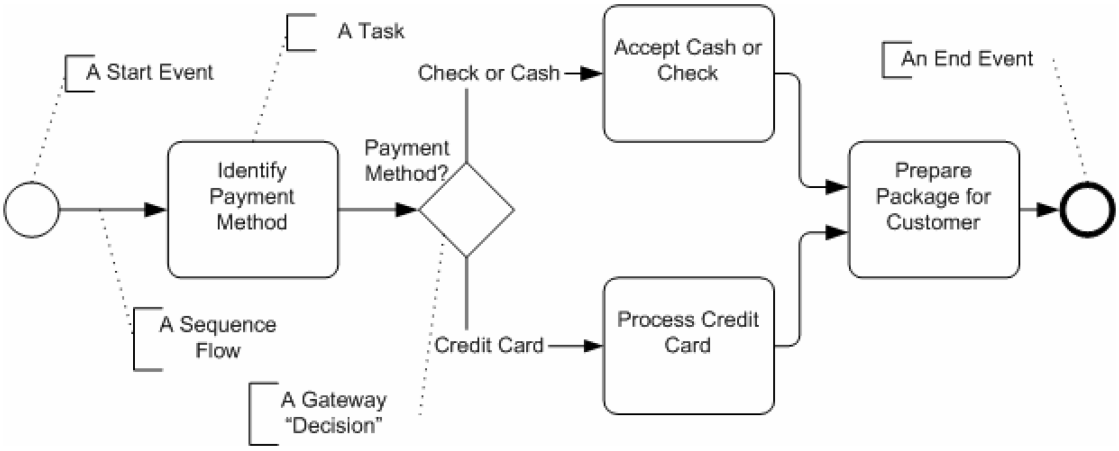


Figure 2.4: A simple business process modelled in BPMN using events, activities and a gateway. Sequence flows connect the flow objects and annotations are used to provide further information. Image taken from [93].

and may contain other activities. Figure 2.5 shows a typical business process modelled in WS-BPEL. For a detailed specification of WS-BPEL, refer to [3].

2.2 Distributed System Visualisation

Using information visualisation techniques to process data produced by distributed systems is a topic that has been explored before. Moc and Carr used visualisation to find system configuration and efficiency problems in a distributed system that had been operational for over three years [59]. However, they focussed more on data retrieval and pre-processing and treated the visualisation of the data only as the final part of a three-step method. Nevertheless, their focus is very similar to this Master's thesis in the sense that their goal is to provide developers with better insight into how the distributed system in question behaves and what improvements can be made. Moc and Carr pursue a more statistical approach to data retrieval and visualisation, using the *Spotfire.net* visualisation tool [82] for rapidly generating and iterating on scatter plot data that they extract from the running system.
Lipski, Berger and Magnor developed *vIsage*, a visualisation and debugging framework for distributed system applications [57]. They present a visualisation and debugging concept environment that is used for the development of an autonomous car, where several computers are controlling a single vehicle. A lot of the produced data has a geometrical context and thus, bird-eye views of the area where the car is operating are used. The obtained data, for example position, orientation and velocity of the car, is superimposed onto the images. Developers responsible for debugging the system can

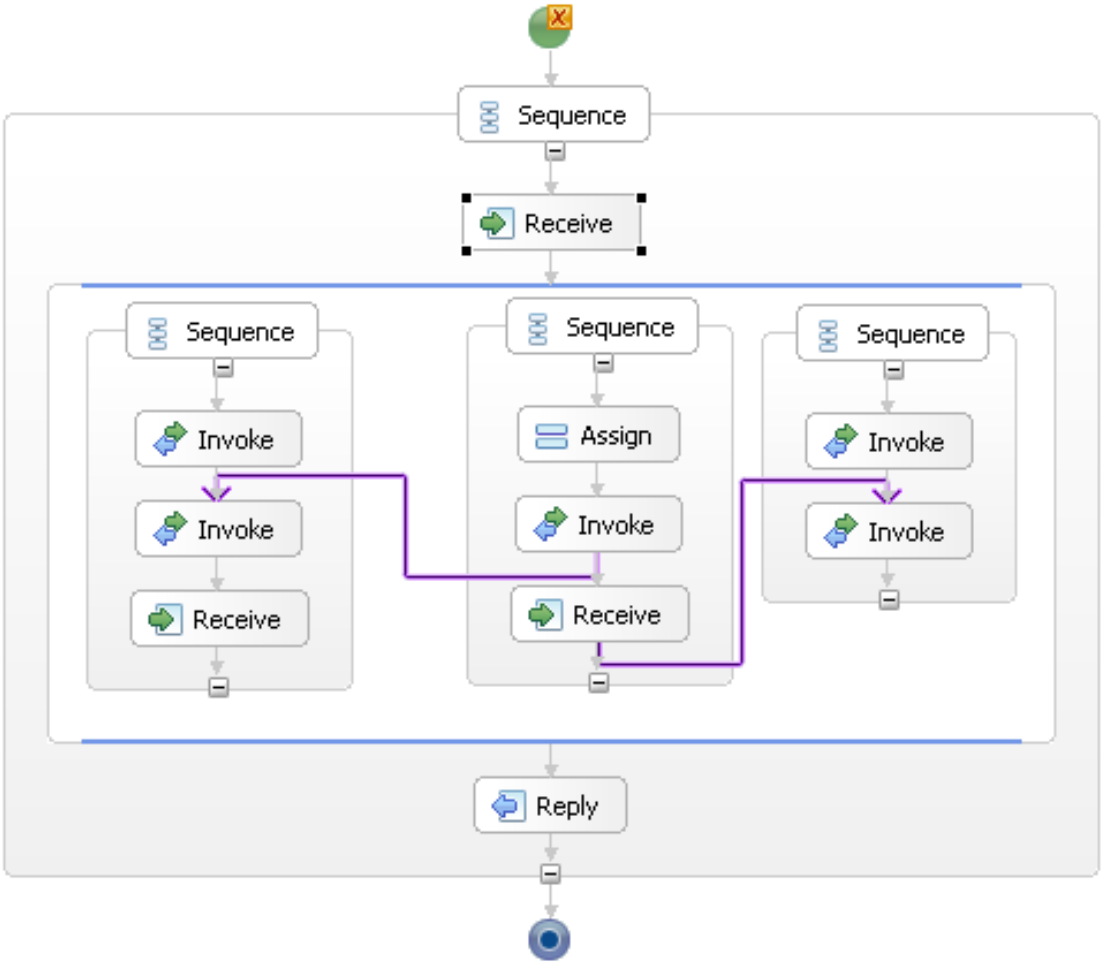


Figure 2.5: BPEL model created with the *BPEL Designer Project* [8].

make use of the augmented birds-eye videos to reach better conclusions and spot errors faster. *vIsage* also supports simulation and the playback of previously retrieved data.
Brumbulli and Fischer developed a tool for the simulation of formally described distributed communication systems [10]. They focused on visualising a series of traces that are made available after the system in question has finished. To enable the user to actively explore the data, Brumbulli and Fischer developed an interface resembling a playback device for controlling the time dimension. Figure 2.6 shows the tool in action.
Sambasivan, Shafer, Mazurek and Ganger addressed the problem of visualising performance problems in distributed systems [76]. Three possible visualisation techniques were implemented and a user study was conducted to find strengths and weaknesses of each approach. The reason for performance problems could be hidden in any part of a potentially large system or the result of bad interactions between parts. Thus, a tech-

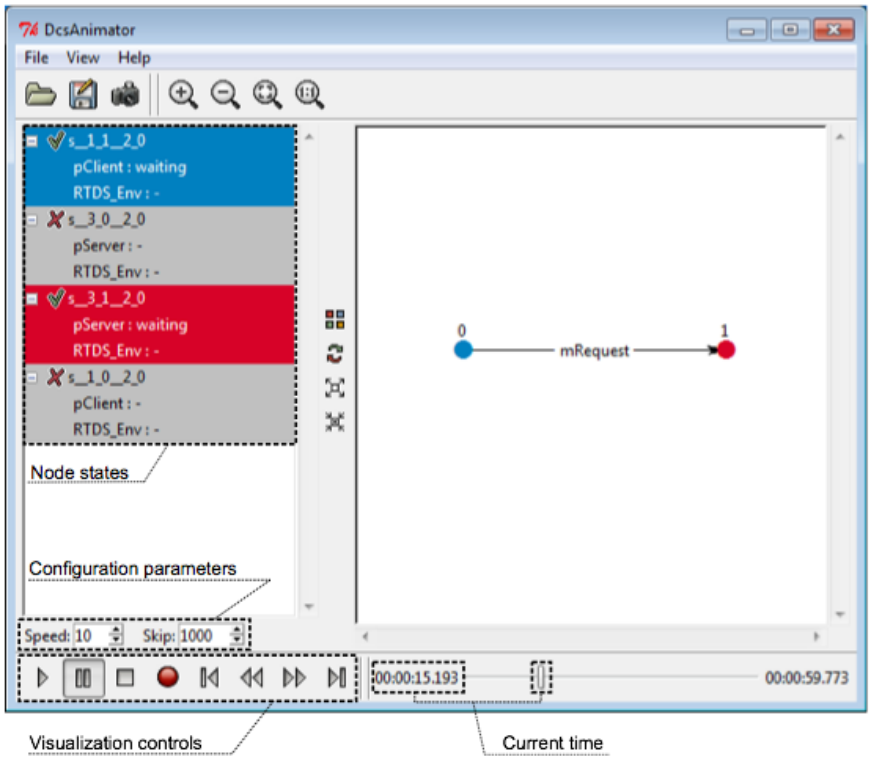


Figure 2.6: Screenshots of the visualisation tool developed by Brumbulli and Fischer. Note the interface for controlling the time dimension in the bottom part of the screenshot. Image taken from [10].

nique called *request-flow comparison* is used to narrow down the likely causes for the performance decrease. The conducted user study included experts, who were familiar with the particular distributed system as well as non-experts, who were unfamiliar with the system, but had general knowledge of the subject.

2.3 Layouting

Interestingly, many of the tools that visualise coordination models do not offer automatic layouting capabilities. The users are tasked with manually positioning elements themselves. In this thesis, automatic layouting of Peer Model representations is employed to relieve users from this tedious task.
At its core, the proposed Peer Model visualisation resembles a graph and papers about graph layouting are closely related to this thesis. Many approaches and algorithms exist for the automatic layouting of all variants of graph structures. The two main approaches are *Force-Directed Graph Drawing* and *Layered Graph Drawing*. In force-directed

graph drawing methods, an initial placement of nodes is continuously altered according to forces defined through the use of concepts adopted from physics, such as springs. Both repulsive and attractive forces may be applied between single nodes or groups of nodes. The problem is either solved through the minimisation of an energy function or through iterative adaptation of the intermediate graph structure. An in-depth introduction to the algorithms used in force-based graph drawing methods can be found at [28, 37].

The most prominent implementation of a layered graph drawing method is the *Sugiyama framework* [83], named after Kozo Sugiyama. It works by assigning layers for each node and structuring the graph in a tree-like fashion. Layered graph drawing very closely resembles how a typical Peer Model is structured and will serve as a basis for the layouting process developed in this Master's thesis.

Closely related to graph layouting is the topic of graph aesthetics. Several papers exist on what makes graph layouts aesthetically pleasing [16, 65–67, 91]. These papers often conduct empirical studies to validate their claims. Many of them define aesthetic criteria a graph should fulfil in order to be clearly understandable. Unfortunately, it is not always possible to fulfil all criteria equally well. Sometimes, criteria conflict with each other and a decision has to be made which one to favour and to what extent. For example, in order to avoid edge crossings, edges might have to be routed in a circuitous fashion which in turn increases the total edge length and potentially introduces additional edge bends. The decision which criteria to favour and more generally, which decisions lead to the aesthetically more pleasing graph, is not trivial. Purchase et al. [65] tried to objectify the matter by introducing a metric to gauge the readability of general, undirected graphs. Ware et al. [91] introduce a methodology for evaluating the cognitive cost of graph aesthetics and empirically test their findings by letting users detect the shortest paths in spring layout graphs. A paper by Diaz et al. [20] formulates an objective cost function based on a number of graph layout criteria and compares algorithms according to this metric. Ham and Rogowitz [88] tackle the problem from a different standpoint by letting users manually alter graphs by repositioning graph nodes to create layouts which they feel capture the data relationships best. Afterwards, they analyse the results and provide concrete recommendations for graph layout algorithms.

In this Master's thesis, path finding algorithms are used on multiple occasions, including the orthogonal routing of links between elements and during the optimisation step responsible for reducing the layout size. In particular, the *A-Star* algorithm is used for path finding [17, 56, 75].

Layouting a Peer Model is in many parts very similar to printed circuit board layouting. The problem of efficiently placing spatial elements onto a confined space, whilst following certain rules and constraints is common for both fields. There exist a variety of software packages capable of automatically and semi-automatically placing and routing elements on a circuit board [26,41,70]. An overview of the topic of printed circuit board

layouting is given by Abboud et al. [1]. Mathematical concepts are presented for solving the two major issues *component placement* and *wire routing* which, interestingly, is very similar to how the Peer Model layouting process is structured.

CHAPTER 3

An Introduction to the Peer Model

The Peer Model is targeted at modelling coordination patterns in distributed environments. It is inherently component-based and encourages the reuse of predefined elements. New parts of a system can potentially be developed by combining existing components. The Peer Model also supports nesting of components within each other, further promoting the reuse of components. A profound distinction is drawn between coordination logic and business logic. The Peer Model deliberately forbids the development of arbitrary business logic within the model and focuses solely on providing the necessary coordination logic facilities. It incorporates a high-level programming and design model as well as its own Domain Specific Language (*PM-DSL*). Developers can design and deploy the complete coordination logic of distributed applications by specifying it with the PM-DSL. First steps towards a toolchain exist to aid developers in this task [49].
A typical Peer Model is made up of different building blocks and each of them is explained in the following sections. Afterwards, a complete example of a Peer Model specification is given with the aid of the PM-DSL. Note that the Peer Model is still in active development and additional features not discussed here may be added in the future.

3.1 Entries

Entries are the moving parts of a Peer Model. They reside inside containers or are transported through the Peer Model. *Wirings* take or read entries from containers and move them around. *Services* create, modify and delete entries according to their predefined implementation.
Each entry has a distinct type, which is specified by a unique name. Typically, the type

is used to denote what the entry is used for. The Peer Model developer may define whatever entry types he/she desires and also decides on their semantics. Among other criteria, wirings select which entries they act upon based on the type.
Entries are both used as a means to transport application data but also to control the Peer Model itself. Subsequently, the data within entries is divided into two categories. The first category represents the *Coordination Data* or *Co-Data*. It contains meta-data utilised by the Peer Model and may influence the movement of the entry through the system. As such, the co-data of an entry is actively queried and also modified by the Peer Model. Wirings may select entries based on the co-data and services may change parts of the co-data to their liking.
The second category of data within an entry is the *Application Data* or *App-Data*. It contains business-logic related data that is neither queried nor modified by the Peer Model itself but is only transported through the system. External components may use this data in whichever way they desire. The app-data part is ascribed to being a business logic concern and is therefore not handled by the Peer Model.
Many developers utilise entries to model a state machine, encoding the (sub-)system's state with entries that are currently present inside the model. Depending on which entries exist, the other parts of a Peer Model act accordingly. Looking at it from this perspective, entries are used to define a Peer Model's state.

3.2 Peers

A construct called *Peer* is introduced. It presents a structured, addressable and potentially persistent coordination component. A peer can be uniquely addressed through its fully qualified name. Each peer incorporates dedicated areas for input and output of entries. These areas are aptly named *Peer In Container (PIC)* and *Peer Out Container (POC)*. The umbrella term for both is *Peer Container (PC)*. Generally, a peer container may contain any number of entries.
Typically, a peer also contains inner coordination logic. This logic is essentially encapsulated within the peer and hidden from the outside. A peer's inner workings are comprised of *Wirings* and *Sub Peers*. In this context, wirings and sub peers may also be summarised into the term *Elements* and referred to as child elements of this peer. The distinction between a peer and a sub peer is entirely based on the point of view. When speaking of a sub peer, it is generally referred to as such because of its affiliation with its surrounding peer. For the course of this thesis, the terms peer and sub peer are mostly interchangeable and are only used to emphasise their relative location. Additionally, the term *Parent Peer* is used to denote the peer that encapsulates the currently referenced peer or wiring. Lastly, the notions of a *Sibling Peer* and a *Sibling Wiring* are used to refer to a peer or wiring in relation to another where both share the same parent peer. More general, such an element may also be referred to as *Sibling Element*.

An important characteristic of a peer is its general passivity. Peers provide the setting and general structure within which the Peer Model operates, but do not perform any actions themselves.
Each peer is assigned to a so-called *Processor*. A processor represents the physical machine a peer is located on. Multiple peers may reside on a single processor. Just as peers, a processor is named. Within a Peer Model, each processor has a unique name and can be referenced by it.
For the purpose of this thesis and the developed monitoring software, a naming scheme for peers is used to uniquely reference them within a Peer Model. The syntax for referencing a peer in a fully qualified manner is

⟨*PeerReference*⟩ ::= ⟨*ProcessorName*⟩{:⟨*PeerName*⟩}:⟨*PeerName*⟩

A peer's reference is defined by the processor it resides in followed by a series of peers, moving down the peer tree and finishing with the actually referenced peer. All elements are separated by colons. Additionally, PIC and POC of a peer may also be directly referenced using

⟨*PICReference*⟩ ::= ⟨*PeerReference*⟩.PIC

⟨*POCReference*⟩ ::= ⟨*PeerReference*⟩.POC

Note the use of a dot instead of a colon to separate the peer reference from the PIC/POC token. This naming scheme is used throughout the course of this thesis as well as within the monitoring application. The file formats developed for logging the events in a Peer Model are such a field of application. See section 7.2 for more information on this topic.

3.3 Wirings

Wirings are the active part of a Peer Model. They are responsible for transporting entries through the system. A wiring is comprised of three sections: *Guard Links*, *Action Links* and *Services*:

Guard Links

The guard section consists of any number of *Guard Links*. Guard links are responsible for feeding the wiring and its potential services with entries. The guards of a wiring are numbered which in turn defines the order of their execution and when they test for appropriate entries. Each guard link is connected to a peer container. Consecutively, each guard queries its container for entries according to its specification. If the specification is met, the next guard starts querying. If a guard cannot meet its specification, the guard is referred to as *blocking* as it prohibits the wiring to continue its execution. Once all guards of a wiring meet their

specifications, they are said to be *firing* and the next step in a wiring's activation begins. A guard's specification consists of an entry type, an operation, a count and optionally a query:

Entry Type
: The entry type defines which entries the guard is looking for in terms of its type. Only a single type may be specified. Entries of other types are ignored.

Operation
: The operation specifies what the guard should do with the entries it acts upon. The options are **read** and **take**. Specifying a **take** operation means that the entries the guard processes are removed from the queried container. By setting a guard's operation to **read**, the entries are duplicated and the original entries are kept in the container while the copies are further processed by the wiring.

Count
: The count sets boundaries on how many entries may be processed in one firing. It may be specified as a number, preceded by a relational operator ($=, <, >, \leq$ *or* $\geq$). Alternatively, it may be set to one of the two special values *[all]* or *[none]*. *[all]* will result in the guard taking or reading all present entries that fit the specification. Setting the count to *[none]* turns the guard into a so-called *none-check* guard. Rather than looking for entries, such a guard checks for the nonexistence of entries that meet the specification. While other guards fire as soon as entries in the queried container meet their specification, the none-check guard fires whenever this is **not** the case.
In general, a guard tries to read or take as many entries as possible within the boundaries set by the count. For example, if the count of a guard is specified as ≥ 2 and there are three suitable entries present in the container, the guard will take all three.

Query
: The query is the only optional part of the specification. The developer may specify certain criteria that an entry must fulfil in addition to its type. A query generally tests co-data properties, can be arbitrarily complex and may include comparison and logical operators.

Each guard link is specified separately. However, there are limits to which container a guard link can target. Only containers of the parent peer as well as sibling peers may be queried. Deeper nested peers are not targetable, just as peers higher up in the nesting may not be targeted.
Once all guards have fired, they put their affected entries into the wiring's *Entry Collection* (short: *EC*). The EC is a distinct container residing within each wiring. Just as a peer container, an EC may contain any number of entries. The

EC is the temporary storage location for entries that are read or taken by guards and is accessed by action links and services, which are discussed next.

Services

A wiring might specify any number of services. Services contain application specific code that operates on the wiring's EC. Just as guards, the services within a wiring are ordered and run in succession. After all the guards of a wiring have fired, the first service executes. Once it is finished, the next service starts and so on.

The logic of a service is written in a language specifically developed for the Peer Model (*Peer Model DSL*). This language is capable of querying, modifying and deleting entries, as well as calling external facilities to communicate with the outside world. For an in-depth discussion on the service specification, refer to [49]. The typical structure of a service is to take or read a number of entries from the EC, modify these entries and/or call external processes based on the data, then write back entries into the EC. However, within a service, the developer is relatively free to employ different setups.

A service does not access the entries in the EC directly, but through the use of *service input links*. These links are very similar to guard links in their structure. A service input link is also determined by a specification that is identical to a guard link specification. It specifies an entry type, an operation, a count and optionally a query. The difference to guard links lies in the fact that a service input link always targets the wiring's EC and does not *block*.

A service also specifies *service output links*. Mirrored to service input links, service output links are specified in the same manner as action links, with the difference being that they deliver entries back to the EC. Action links are discussed next. Once all services within the wiring have executed, the *Action Links* section begins.

Action Links

Action links are responsible for delivering entries to their destination containers. After the guards have fired and the services of a wiring have executed, the first action link activates. Just as the guard links, the action links within a wiring are ordered and run in succession.

Action links operate on the EC and query it for entries. They have a specified entry type and a count. These two settings have the same semantics as their counterparts in guard links. The type specifies which entries to operate on and the count limits their quantity. However, specifying a count of *[none]*, as is possible with guard links, is not allowed. The action link's operation is implicitly set to **take** and therefore does not need to be specified.

The semantic of execution for action links is different to guard links. While action

links also run one after the other, their behaviour can be reasonably described as *best effort*. An action link tries to fulfil the requirements posed by its specification as best as it can. However, if it is not able to, the action link silently finishes without affecting any entries and the next link takes over. An action link does not *block* like a guard link does.
Each action link targets a peer container. Upon querying the EC, it delivers the applicable entries to this container. The rules for what permitted target containers are is the same as for guard links.
After all action links have executed, whether or not they were able to deliver any entries, all remaining entries within the EC are deleted.

Each new iteration of the three sections of a wiring is called *Wiring Activation*. An important characteristic of wirings is that multiple activations may run concurrently. A single wiring can exhibit multiple wiring activations at the same time. These activations are fully autonomous from each other and may be in different stages of their execution. This is a key feature of the Peer Model and makes it possible to have multiple business processes running in a concurrent and autonomous manner. An important fact to consider in the case of multiple concurrent wiring activations is that the EC is not shared across them. Each activation manages its own EC, which gets cleared after the activation finishes.
Similar to peers, wirings may also be referenced in a fully qualified way by means of

⟨*WiringReference*⟩ ::= ⟨*PeerReference*⟩:⟨*WiringName*⟩

Note that the peer reference in the example refers to the wiring's parent peer.
A noteworthy case many developers come across is the need for very simple wirings that solely move or copy a single entry type from one peer container to another. Such a wiring only has a single guard and a single action, both with the same entry type. Typically, no services are applied. As this type of wiring is relatively common, it is explicitly referred to as a *Move Wiring*. As the name implies, a move wiring transports entries from one container to the other. The guard's operation is set to **take**.

3.4 Advanced Features

DEST property

Within an entries' co-data, a *Destination Property* (short: *DEST Property*) can be set. This property may contain the fully qualified name of a PIC. An entry that resides inside a POC and has the DEST-property set is automatically transported to the destination set within the property.
The typical use case for this feature is to transport entries directly across processor

boundaries and without relying on wirings. The DEST-property of an entry can be dynamically set inside a service based on its inner logic. This facilitates the development of very flexible distributed architectures. On the other hand, it circumvents the static rules imposed by the wirings and allows entries to freely travel through the whole model.

Entry TTL and Entry TTS

Entries may also be attributed with the *Time-To-Live* (short: *TTL*) and the *Time-To-Start* (short: *TTS*) properties. Both properties are generally set with a timestamp located in the future. An entry marked with a TTS is considered to be non-existent by guards. Only after the timestamp set in the property has passed, the entry loses this characteristic and may be taken into consideration again by wirings. A popular use of this feature is the implementation of timers: a wiring can be designed to have a guard and an action of the same type both targeting the same peer container. Within a service, an entry of said type with a TTS property is created and sent back through the action. The wiring will not be triggering again immediately but only after the entry has reached its time-to-start. This pattern can be used to implement regular polling functionality, for example.
The TTL property behaves similar in the sense that once the set timestamp is reached, the entry changes its state. But instead of becoming visible again after being disregarded, an entry with a set TTL property is deleted as soon as the timestamp expires. Before this occurs, the entry is treated normally however.
Note that the events pertaining to TTL and TTS timestamps being reached are only executed when the associated entries are located within a peer container. Entries that are currently being transported by a wiring or a DEST move or resident within an EC are not checked for these properties. However, once an entry with an expired TTL or TTS property is put into a peer container, the corresponding event will be carried out immediately.

Flows

Flows are a means to manage the distinction between business processes. An Entry can be attributed with a *Flow ID*, which associates it with a specific flow. This ID is usually set within service implementations. Guard links of wirings have an attribute called *Flow Dependence*, which can be on or off. When a guard link with enabled flow dependence encounters an entry with a set flow ID, it sets the wiring activation's internal flow to this ID. Subsequent guards may then only act on entries that have an equal flow ID or no flow ID at all. Entries associated with a different flow are ignored. This restriction is only applicable for the duration of the corresponding wiring activation. Subsequent wiring activations, even those running concurrently, are not affected and may set their

own flow IDs. This mechanism enables an explicit modelling of business processes and gives more expressive power to Peer Model developers during the design phase.

Explicit Commits

In the default case, the guards of a wiring only fire if all their specifications are met and they do so simultaneously. In some cases, this might not be the desired behaviour. For this reason, a developer can specify *explicit commits*. By placing a commit after a guard link, the guard section of the wiring is effectively cut into two *commit groups*. The guard links of a commit group are treated like a complete, distinct guard section. Once all the guard links in the group have met their specification, the group fires and the affected entries are put into the EC. Commit groups are run in succession according to their guard's order in the wiring. Once a group has fired, the next group starts querying. After all groups have fired, the service part of the wiring is executed.

3.5 Peer Model DSL

The Peer Model DSL (short: PM-DSL) is used to specify Peer Model systems [49]. The PM-DSL file's format is specifically designed to fit the Peer Model's architecture. The code generator located within the PM toolchain can read PM-DSL files and produce language-specific code for various software platforms, including *.Net* [72] and *Embedded C* [49]. Ideally, developers only need to care about the PM-DSL file and should neither need to inspect nor alter the generated code. The resulting codebase may then be deployed onto development, test or production machines where it is executed.
As mentioned before, the Peer Model offers the reuse of components. It does so by allowing peers, wirings and services to be used in multiple places within a system. This is reflected within the PM-DSL by splitting the specification of these components into two distinct steps, a definition and a instantiation step. Uniquely naming the components enables referencing them at a later point in the file, potentially multiple times.
A PM-DSL file itself is split into multiple consecutive parts, each defining its corresponding components. A separate discussion of each part follows:

Entry Types
: The possible entry types that may be present within the system are specified. As each entry type is uniquely named, it is referenced through this name later within the file.

Services
: For each service, its input and output links are specified, as well as its implementation logic. Each service is given a unique name to be addressable later in the file. However, the services are **not** bound to specific wirings. The services are

merely defined but no specific instances of them are created.

Wirings

A wiring is defined by specifying its name, its guard and action links and its services. As services are already defined in the previous part, referencing them by their name is enough. A service referenced in that way is said to be instantiated by the wiring. Multiple wirings may use the same service definition independently. Even multiple instantiations of a single service within a wiring is allowed, albeit unusual. In order for the wiring definition to remain generic, no specific targets for the guard and action links are set. Instead, placeholders in the form of strings are used. Later, during a wiring's instantiation, the placeholders are replaced by actual peer container references. For a more detailed discussion on this functionality, refer to section 7.2.

Peers

Next in the sequence are peer definitions. For each peer, its sub peers and wirings are specified. To define a peer with sub peers, they need to be defined first so that the parent peer can reference them during its own definition. Similar to how wirings instantiate services, peers instantiate wirings by referencing them. But as wiring definitions include placeholders, the peer needs to specify how these placeholders should be replaced. This is done by specifying a *linking map* together with the wiring reference. Again, for an in-depth discussion of this feature, refer to 7.2.

Topology

The topology definition is the last section in the file. In this section, the processors are specified. Each processor is defined by a name and the peers that run on it.

Listing 3.1 shows a small exemplary PM-DSL file that defines one wiring with one service, two peers and one processor. For a thorough discussion on PM-DSL, refer to [49].

```
--------------------------------
--- Entries
--------------------------------
entry X is
    coordination data is
        empty
end;

entry Y is
    coordination data is
        empty
end;
--------------------------------
--- Services
--------------------------------
service Service1 is
    take X;
    emit Y;
begin
    create(Y)
        .set(tts => now + 1 s)
        .emit;
end;
--------------------------------
--- Wirings
--------------------------------
wiring W1 is
begin
    read X [all] from C1;
    running service Service1;
    deliver Y [all] to C2;
    commit;
end;
--------------------------------
--- Peers
--------------------------------
peer Subpeer1 is
begin
end;

peer SmallPeer is
begin
    instantiate wiring W1 linking (C2 => Subpeer1.PIC);
    instantiate peer Subpeer1;
end;
--------------------------------
--- Topology
--------------------------------
topology TestTopology is
begin
    processor SmallProcessor
        runs peer SmallPeer
end;
```

Listing 3.1: PM-DSL file that specifies a small Peer Model. The resulting model contains one processor with one peer, which in turn contains one sub peer and a wiring with a service. Additionally, the DSL file defines two entry types (*X* and *Y*).

CHAPTER 4

Requirement Analysis

4.1 Interviews

In order to develop a suitable tool that helps developers in their tasks, it was essential to find out what their problems and issues are when working with the Peer Model. In order to recognise and document requirements, interviews were conducted with ten developers. The interviewed developers all work with the Peer Model on a regular basis and hence could provide valuable information and input. To get the most out of each participant, the interviews were conducted as one-on-one conversations. One of the goals was to find out about the developer's method of operating in order to gain insight into the differences between developers and their tasks. Especially their way of debugging when problems arise is of interest as it offers hints on how to improve this process. Using the Peer Model, it is possible to create a large range of different projects in terms of size and complexity. Talking about the developers' current projects was helpful in putting their opinions and ideas into context. It also helped to develop a sense of what typical Peer Models look like. Another goal was to find out what problems the developers face on a daily basis and how a monitoring and debugging tool may help them in their tasks.
The interviews were not strictly pre-planned as the intent was to let the interviewee share his/her ideas in a very unconstrained way. However, a set of questions was prepared and asked in order to support the interview flow. The following questions were posed:

1. How does your modelling process look like when working with the Peer Model?
2. In which ways do you debug when you encounter problems?
3. Are there certain types of problems and/or bugs you encounter regularly during development?

4. What size do the projects you work with have? Approximately how many peers, wirings and entry types does a typical Peer Model contain?

5. Can you give a rough estimate on the number of entries that are active inside your Peer Models during development and in production?

6. How do you deal with the problems arising through concurrency during development?

7. Do you think a visualisation tool can help during the development process and in what way?

8. Do you think a visualisation of the events inside a Peer Model can help developers to understand previously unfamiliar models?

The interviews lasted from about 30 minutes to one hour. As the interviews were spread out over the course of about two weeks, we tried to incorporate the intermediate results from the previous interviews into the following ones.

4.2 Found Requirements

A list of found requirements follows. Many of these requirements were obtained as a direct result of the conducted interviews by comparing the received answers and ideas from the developers and interrelating them to each other. However, additional meetings with Peer Model developers were held throughout the entire subsequent development process. Topics of these meetings included certain Peer Model functionalities and their visualisation, the development of data formats as well as intermediate presentations regarding the state of the monitoring tool itself. The meetings revealed additional requirements or defined existing ones more precisely, which were incorporated into the ongoing development process.

Better Debugging Capabilities

As the Peer Model approach is relatively novel, the toolchain and more specifically, the debugging facilities are still in their infancy. Developers resort to manual debugging means or develop their own tools to facilitate this task. Throughout the interviews, developers expressed the wish for better debugging capabilities and tools.

Overview

The interviews showed that developers are missing an overview. Frequently, they need to get into the details of certain sub parts of the system. While doing so, they lack a

bird's eye view that shows an overall picture. An overview also helps new developers get familiar with a specific Peer Model and learn it in a top-down manner rather than having to study each sub part individually and piece them together later.

Detailed Wirings

Wirings represent the most complex part of a Peer Model. Developers spend most of their time designing and adjusting them. As such, it is of outmost importance to get a detailed view of a wiring and the actions it performs. A typical question many developers often pose is why a certain wiring did not activate in certain circumstances when the developer wanted it to do so. A detailed view of a wiring and its changing state allows developers to assess the situation efficiently and find out what is wrong.

Interaction of Subsystems

A Peer Model is made up of multiple parts or subsystems. Developers often try to pursue a modular approach and design parts of the system separately, while specifying interfaces that define how the parts interact with each other. This partition can be made at the processor or at the peer level. The interaction between the subsystems is often the place where bugs and inconsistencies crop up. A visualisation of how these systems interact would be a valuable feature and could help developers in their understanding.

Detection of Nonsensical Structures

Just as any other software, a distributed system built using the Peer Model is subject to change during the course of its lifetime. New parts are added to the system, while other parts change or become obsolete. Peers, wirings or parts of wirings, services or entry types might become unnecessary during the lifetime of the system as requirements and features evolve. A visual representation of the model can help in identifying such structures and aid in their proper extraction.

Race Conditions

In a distributed system, *race conditions* [63] are an ubiquitous issue. Their cause and effect are often hard to detect without the proper tools. Race conditions are especially hard to debug because of their nondeterministic behaviour and volatile occurrence. A proper visualisation of the events happening within a Peer Model can drastically help to reduce the efforts necessary to identify and eliminate race conditions.

Detection of Left-behind Entries

The use of entries as the dominant form of communication leads to the issue of dealing with obsolete entries. An entry needs to be cleaned up once it fulfilled its intended purpose. This can happen within a service, through the purge of the EC contents of a wiring after its activation or by assigning entries a TTL property. However, an entry that is left behind in a peer container will stay there indefinitely if it is not taken or assigned a reasonable TTL. A Peer Model that erroneously does this will start to pile up entries, taking more and more memory while it is handling business processes. Memory leaks like these are hard to spot because the model itself appears to be working correctly.

Control over the Time Dimension

As established before, a running Peer Model system produces a lot of logging data. While the tools developed for monitoring a Peer Model so far offer a chronological listing of the events that occurred, going through this raw data manually is cumbersome and error-prone. Developers need to step through the trace one-by-one and can not easily jump to different places in time. The interviews revealed that a monitoring tool that offers proper control over the time dimension can greatly help in making sense of the given data.

Conservative Use of Animation

The interviewed developers expressed a big affinity towards an efficient way of working. The tools they use should perform their intended tasks as fast as possible. Some of them expressed the desire for a monitoring tool that does not use animations and that they prefer static representations. While animations offer a nice way to visualise changes, this wish was taken into account.

Retention of the Established Visual Notation

During the interviews, the interviewees were presented with preliminary designs and visualisation attempts for the application in the monitoring software. These were also adapted after each interview to incorporate the previously received suggestions. However, the presented designs were not accepted very well. The developers were familiar with the already established visual notations that had been developed in the years before. The differing designs required a lot of additional explanations as they were generally not self-explanatory and seemed to confuse developers more than help them.
While not explicitly requested, the reaction of the developers made it clear that introducing different visual representations was counterproductive. The established notation had evolved organically and already incorporates a lot of conceptual expertise. An im-

portant insight the interviews gave was that leaving the developed visual notation intact was a beneficial decision.

Cross Platform Support

The software platforms and operations systems used by Peer Model developers are diverse. Implementations of the Peer Model exist for various languages and software platforms, including *.Net* [72] and *Embedded C* [49]. Only supporting a single platform or operating system with the monitoring tool would lock out a lot of developers of being able to benefit from the tool. Therefore, it is highly beneficial if the tool is developed in a cross platform way.

Stand-Alone

Closely related to the requirement of cross platform support is the wish for a tool that is stand-alone and does not require a specific software foundation. Prior to the interviews, developing the tool as an Eclipse [23] plugin presented a warrantable option. However, the interviews made clear that many developers have different setups and even though Eclipse is available for all major platforms, it was not found desirable to force developers into using it.

4.3 Conclusion

After the interviews were concluded, the results were gathered and processed further to yield a list of requirements. The interviews helped a lot in establishing the requirements for developing a suitable and useful monitoring tool. They gave insight into how the developers work with a Peer Model throughout its development, from modelling and implementing to debugging and testing. Furthermore, it helped to develop a sense for what typical Peer Models entail and what challenges the developers face.
The interviews also consolidated the idea that a monitoring tool, specifically designed and developed for the Peer Model can help in various problem areas during development. Instead of resorting to manual debugging means, working with proper visual representations of the events facilitates the debugging process of Peer Models and supports developers in their tasks more efficiently.

CHAPTER 5

Peer Model Representation

The DSL describing a Peer Model does not contain any visual information and more specifically, no hints on how to layout it. It contains structural information about the Peer Model, but no explicit data about how to visualise it. The first goal of the Peer Model representation (and the later layouting process) is to take the structure of a Peer Model, given by a DSL file, and visualise it in a way that is meaningful and valuable to the developer. Subsequently, the visualisation of trace data acquired during the logging procedure is the second main goal.

The team around eva Kühn has already developed a graphical notation for representing the static Peer Model. The conducted interviews (chapter 4) showed that the notation had gone through many iterations until that point in time. It was apparent that the developed notation incorporated a lot of conceptual expertise. Not only is it used to visualise Peer Models in scientific papers and journals, the interviews showed that the representation is also an accepted way for developers to design and communicate Peer Models. Developers who are concerned with a specific Peer Model often create scribbles on paper or whiteboard to visualise their ideas and issues. They generally do this with the aforementioned representation. Forcing yet another visual notation change seemed counterproductive. Therefore, it was agreed upon to keep the current notation intact whenever possible. Fortunately, it proved to be a good fit for the visualisation within the monitoring tool as well as a good foundation for further extensions.

The dynamic visualisation of the events in the Peer Model still proposes a difficult problem however. At the time of writing, monitoring support for the Peer Model was at its infancy and therefore, no agreed upon representation for the dynamic behaviour of a model existed. One goal of this thesis is to come up with suitable representations for the dynamic nature of the Peer Model while not moving away too far from the already established representations for the static Peer Model structure. The following section discusses the representations for the Peer Model developed by the group of eva

Kühn. Afterwards, suitable visualisations for representing the dynamic events happening within a Peer Model are proposed in section 5.2.

5.1 Static Peer Model Representation

The elements that make up a static Peer Model were discussed in chapter 3. This section establishes the status quo in Peer Model visualisation that is currently used by developers. The visual notation is used for documentation and in various published papers [50, 51]. Each element is discussed separately and subsequently, a complete example is shown.

Wirings

Wirings are highly customisable and therefore, their representation needs to be equally flexible. Wirings may have multiple guards, actions and services or may be just simple move wirings. A typical wiring is divided into a left and right *link block*, containing the guard and action links, a *service area* and a *center area* that displays the wiring's name. Before a complete wiring is presented, its parts are discussed separately.

Guard Links

Guard links target containers of sibling peers or the parent peer. They are therefore aptly represented as an arrow line. The tail is connected to the target peer container and the tip of the arrow points to the so-called *base* in which the index of the guard is displayed preceded by a *G* (short for guard). The arrow conveniently illustrates the potential movement of entries. The guard link's specification is shown as a text label next to the arrow. Figure 5.1 shows a typical guard link representation. The arrow itself is drawn fully black if the guard's operation is **take** and drawn white with a black border if the operation is **read**. The specification is displayed as a text label. Its format is

⟨*specification*⟩ ::= ⟨*operation*⟩ ⟨*entrytype*⟩ [⟨*count*⟩]

In previously established notations, any potential queries were shown below the specification line. However, only relatively short queries were shown that way. In general, queries can become arbitrarily large, making this location not a good fit for the general purpose as a query might overlap a lot of other elements. For the purpose of the monitoring tool, the query is not shown immediately. Instead, the user can hover over the link with the mouse and a tooltip pops up, displaying the full query.
It is important to note that guards are always located at the left side of a wiring. As such, their arrow always ends horizontally and offers a good

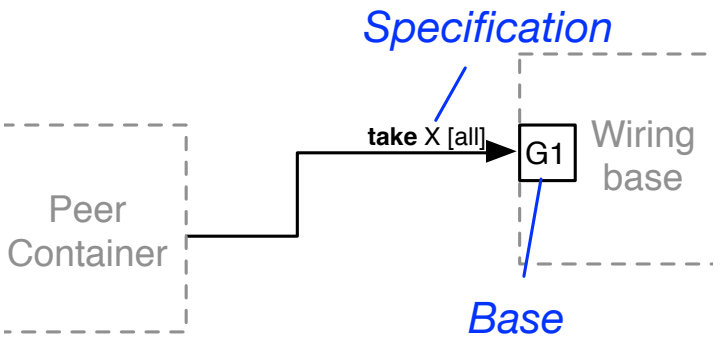


Figure 5.1: Visualisation of a typical guard link within a wiring.

place for the specification text right above it. Multiple guard links are put one below the other according to their activation order.
The link itself is drawn as an arc between the peer container and the wiring. Orthogonal means that the individual edges of a link are either horizontal or vertical. This is a very important characteristic with regards to layouting a Peer Model, which is thoroughly discussed in chapter 6.

Action Links

Action links are depicted very similar to guard links. However, action links may connect on both the left or right side of their corresponding wiring. An action link will connect on the side that is closer to its target peer container in order to avoid long paths. At the same time, the action link's ordering must be respected. Within each wiring's side, the order of action links is maintained. Action links that are activated later are positioned farther down. The tip of an action link's arrow is located at the end targeting the peer container to visualise the direction of movement of entries. Also contrary to guard links, their specification leaves out the operation part as it is always implicitly set to **take**. Figure 5.2 shows an exemplary action link.

Services

Each service is visualised separately and comprised of service input and output links as well as a body that states its name. Service input links are represented very similar to guard links, which is obvious given their similar structure. Similarly, service output links resemble action links very closely. Services sit on top of the wiring and their links are arranged vertically. A typical service is shown in figure 5.3. Wirings without services simply omit the service area altogether.

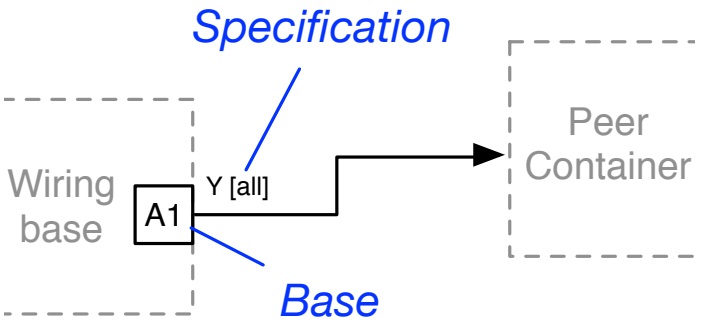


Figure 5.2: Visualisation of a typical action link within a wiring.

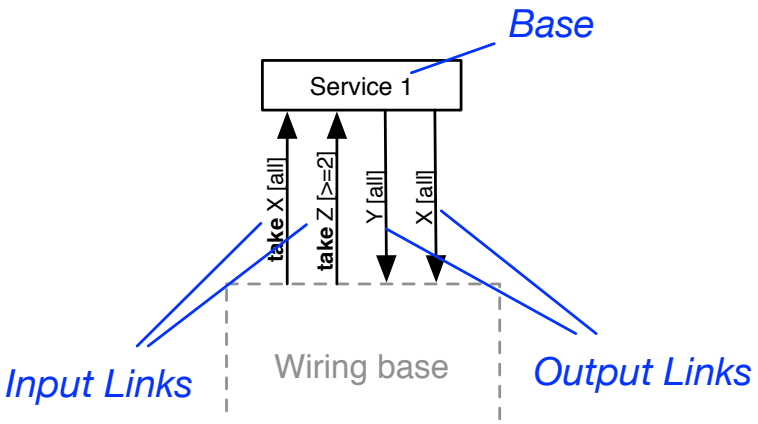


Figure 5.3: A typical service visualised.

The just discussed parts make up a wiring. The wiring's base that connects the parts is a simple rectangle, displaying the wiring's name in its center. Figure 5.4 shows a wiring with two guards, two actions and a service. While in this example, each link targets a separate peer container, hinted at in grey, this does not have to be the case. Multiple links may target the same container. An interesting characteristic is the fact that, while the EC is an integral part of a wiring, it is not explicitly visualised. As this is the representation of the static structure of a wiring, it is only implicitly assumed. Along the same lines is the finding that entries themselves are not visualised explicitly in the static structure of a Peer Model at all. Only references to entry types within the label texts of links are used. No accepted graphical notation for actual entries exists yet.

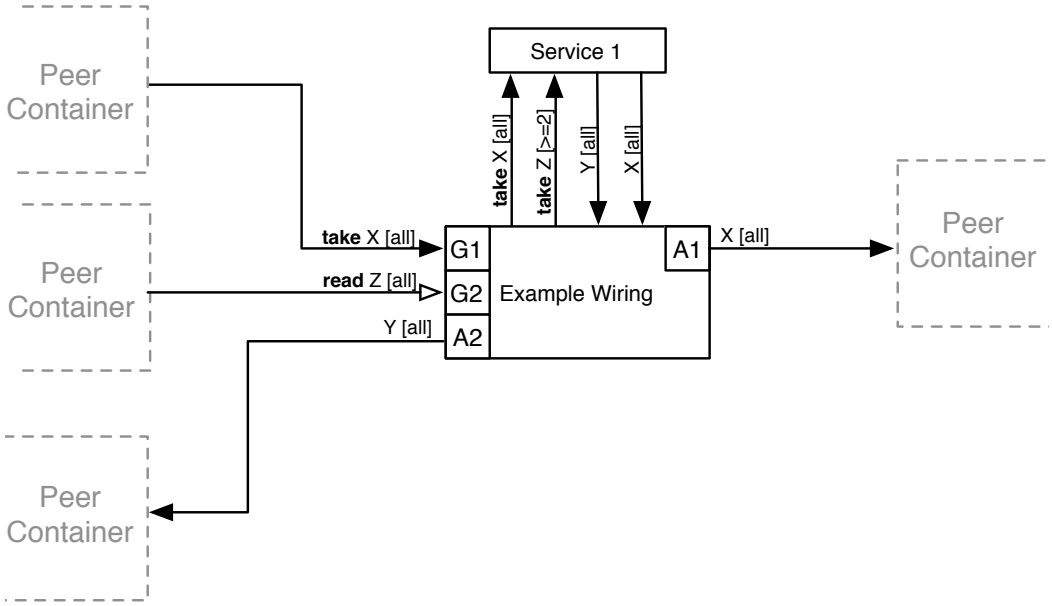


Figure 5.4: A complete wiring named "Example Wiring" visualised. The wiring incorporates two guards, two wirings and a service called "Service 1". Note that multiple services would be placed from left to right in the order of their activation.

Peers

Peers are visualised relatively simple. A peer is comprised of separate areas for its PIC, POC, a center area for its inner logic as well as a title bar for its name. Figure 5.5 depicts a simple peer. Depending on the current needs, the peer can be visualised in collapsed or expanded fashion. Visualising an expanded peer gives insights into its internal structure by displaying the sub peers and wirings within the peer's center area. A collapsed peer on the other hand offers a very compact representation. Its center area remains empty. Refer to figure 5.6 for a visualisation of an expanded peer.

The just discussed wiring and peer representation together form the static Peer Model as it is currently used and understood by developers. Figure 5.6 shows a typical peer representation. An important observation is that wiring links can attach onto peer containers at any position along its sides. This facilitates better layouts and avoids cluttered links. Simpler peer contents can often be visualised without the need for sophisticated link routing. However, this is not always possible, especially for more complex models. In general, the way a peer and its children are visualised is not predetermined and can be implemented in many different ways. Some of these visualisations are arguably better than others. Chapter 6 tries to establish a guideline for what constitutes a *good*

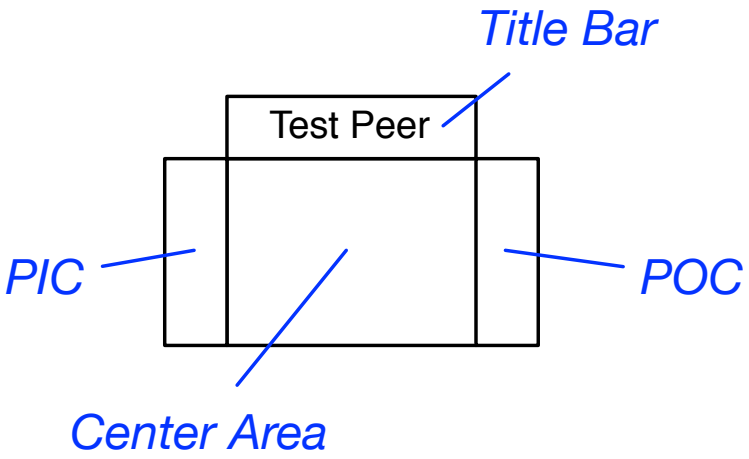


Figure 5.5: A simple peer called “Test Peer“ visualised. It is shown collapsed and therefore incorporates neither wirings nor sub peers.

peer representation and discusses the implementation of a set of algorithms capable of producing such layouts for all possible Peer Models.
Unlike peers and wirings, there is no agreed upon visual notation for a processor. Most of the time, developers do not concern themselves with the physical location of peers and therefore, processors are rarely visualised explicitly. However, for the monitoring tool it is important to develop a visual notation for processors as well. To emphasise the relation between a processor and its contained peers, the processor is drawn as a dashed rectangle that encloses its peers. Figure 5.7 shows an example.

5.2 Dynamic Peer Model Exploration

As stated before, no agreed upon representation of the dynamic state and events in a Peer Model exists. This thesis tries to capture the essence of how a Peer Model behaves and how this behaviour can be described visually. Subsequently, the visual representation supports the goal of developing a Peer Model monitoring software that aids developers in their tasks.
To approach the problem, it can be helpful to think in terms of questions a developer might have concerning the Peer Model he/she is working with. Typical questions are diverse, ranging from the whereabouts of single, specific entries to an overview on the general state of the model. The motivations behind these questions may be equally manifold and span from finding bugs through debugging to spotting optimisation potential to learning the behaviour of previously unfamiliar models.

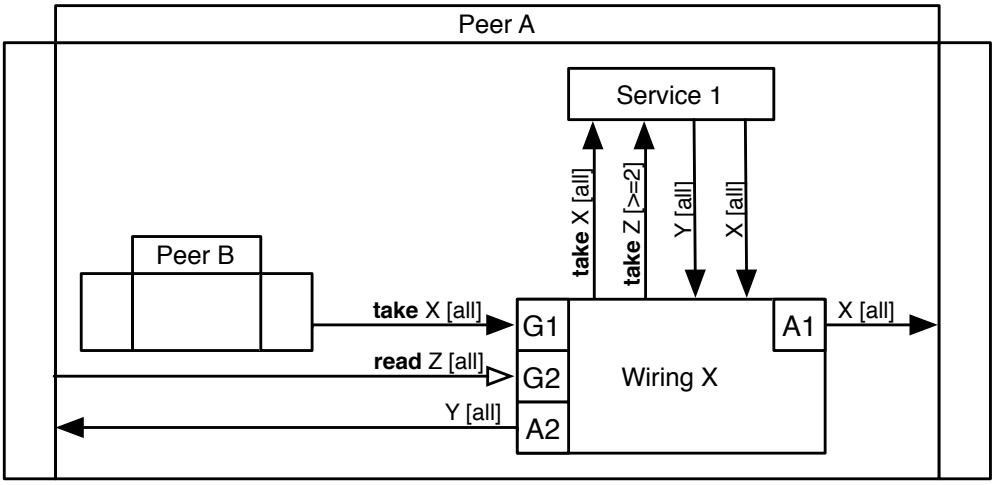


Figure 5.6: A typical peer representation as it is currently used by Peer Model developers. *Peer A* is expanded and its inner wirings and sub peers are visible. Its sub peer *Peer B* however is displayed collapsed. It may also contain logic on its own which is simply not visualised. A small gap between the outer peer and its inner elements helps maintaining readability.

Subsequently, a single visualisation can not cover the entire space of potential questions. Nevertheless, a suitable visualisation aims to answer a predetermined subset in the best way possible. To find out what issues developers working with Peer Models regularly face and what tasks they need help with the most, personal interviews were conducted, which are discussed in chapter 4.
Before looking at ways to represent the dynamic nature of a Peer Model, the underlying data is examined and important characteristics are discussed.

Discrete Changes

Looking at a typical Peer Model trace, it becomes evident that there are only discrete points in time when the state of the model changes. Between these spikes of movement, the Peer Model stays static. This is underlined by the fact that all possible events can be reduced to a single point in time. For example, while a complete wiring activation spans a particular timespan, looking at the events that constitute the activation in separation, they all can be attributed to a singular moment.

Subsequently, these moments can be ordered temporally. Depending on the precision of the datatype used to record the time, it is possible, if unlikely, that two or more moments are indistinguishable by their timestamp. Nevertheless, it is possible to sort the events into an ordered sequence of time steps. Potentially coincident events are resolved by forcing an arbitrary order. The resulting sequence

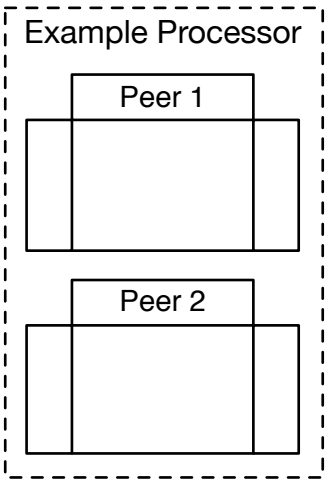


Figure 5.7: Visualisation of an example processor. The processor's name is stated in the top center and its associated peers are arranged below.

enables stepping through a Peer Model trace in a completely linear fashion, even in the presence of concurrency.

Moments

Developers often re-enact Peer Model behaviour by going through the events as they should unfold one after the other. This sometimes happens by outlining the model on paper or by envisioning it mentally. However, critical steps in this event chain are often explicitly visualised by picking the point in time where an event happens and sketching it out. Such a sketch combines the state of the model at the point in time before the event with the event itself that causes the state to change. From that discovery arose the requirement to visualise critical moments of a Peer Model located at a single point in time. By focussing on singular moments in which an events happens, it is possible to visualise the Peer Model behaviour in great detail.

Between the events, the Peer Model assumes specific states. Visualisation of these stationary conditions is equally important in aiding the understanding. The state before and after an event, together with the event itself, form an essential information unit.

Event Chains

One of the most crucial insights is the role time plays in the Peer Model. As established previously, all events happen in a specific moment in time. Events happen before or after each other. Strongly connected to this is the notion of dependency. All Peer Models work by chains of events being set off, with one initiating an-

other, in a strict temporal and causal ordering. Such chains may spawn, merge, be divided, pause and terminate within a Peer Model trace. In practice, multiple chains may run independently of each other within the same model and even within the same elements such as wirings through autonomous wiring activations. Chains may also affect each other indirectly through the presence or absence of entries within peer containers. With time and the dependency of events being such an important aspect of the nature of Peer Models, it was logical to look into ways to visualise these aspects in a suitable manner. Many of the events happening are not isolated, but are in fact part of a bigger process. This is especially true for wirings and their individual actions which together form a wiring activation. Grouping such events into a coherent chunk helps the developer in obtaining a clearer picture of the proceedings within a Peer Model.

Overview + Detail

A very important concept in information visualisation is the idea of *Focus + Context*. To quote Card et al.:

> Focus+Context starts from three premises: First, the user needs both overview (context) and detail information (focus) simultaneously. Second, information needed in the overview may be different from that needed in detail. Third, these two types of information can be combined within a single (dynamic) display, much as in human vision. [13]

While the first two premises seem attainable and a good fit for representing the dynamic nature of the Peer Model, the third premise established by Card et al. is very hard to fulfil. Integrating the time dimension directly into the established static Peer Model representation did not seem like an attainable and worthwhile goal. Fortunately, a strongly related information visualisation concept called *Overview + Detail* exists. Both have in common the distinction between focus/detail and context/overview. But rather than trying to integrate one within the other, Overview + Detail embraces the incoherence of the two visualisation parts by spatially separating them. Cockburn, Karlson and Bederson describe overview + detail in the following way:

> An overview+detail interface design is characterised by the simultaneous display of both an overview and detailed view of an information space, each in a distinct presentation space. [15]

The two sub-views are strongly linked and a change in one view often results in a change in the other. The idea of overview + detail perfectly describes the relation between the detailed view of a single point in time and the overview provided by representing temporally related chains of events. Displaying a single moment in time in all detail represents

the detail, while the temporal overview serves as the necessary context and embeds the detail, temporally relating it to the rest of the data. Implementing the idea of overview + detail was deemed a suitable dynamic Peer Model representation that is capable of answering many questions a developer potentially has.
The visualisation representing the detail is subsequently referred to as *Main View*. The overview's foundation is the temporal aspect of a Peer Model, therefore it is labelled *Timeline* to emphasise this fact. The utilisation of the overview + detail concept combines the level of detail that the main view offers with the outline capabilities of the timeline.
This thesis tries to combine the two approaches of Peer Model visualisation and form a suitable software tool that is capable of answering a lot of questions the developer might pose. Interaction is the missing piece to the puzzle that forms the connection between the two visualisations. Through selecting specific points in time in the timeline, the developer can explore the current state of the Peer Model in detail. By moving between moments, the user gains a sense of what happened through active exploration. Main view and timeline are displayed side by side in a split view fashion.
While the general motivation behind the two forms of visualisation and what information they should convey is determined, it remains to be discussed how these goals can be achieved best within the scope of the monitoring software. A discussion of the two visualisation approaches follows.

Main View

The static Peer Model representation that is already developed does not deal with the concept of time in any way. It does not tackle this aspect of a Peer Model as it is only concerned with the general structure of the system. In a sense, it controls the rules of the model but not how these rules are enforced within any particular materialisation of the model. However, it provides a very detailed view of the Peer Model and is well recognised by all developers. Furthermore, during the design phase it became apparent that the static representation offers the possibility to superimpose information onto it in order to display the dynamic state of the model for a particular moment in time. Because of that, it is used as the basis for the main view of the dynamic representation of the monitoring software. A beneficial side effect is that developers do not need to learn a complete additional set of visual notations as the general form is adopted from the static representation. Only small adjustments are necessary to enable the superimposition of static and dynamic Peer Model information.

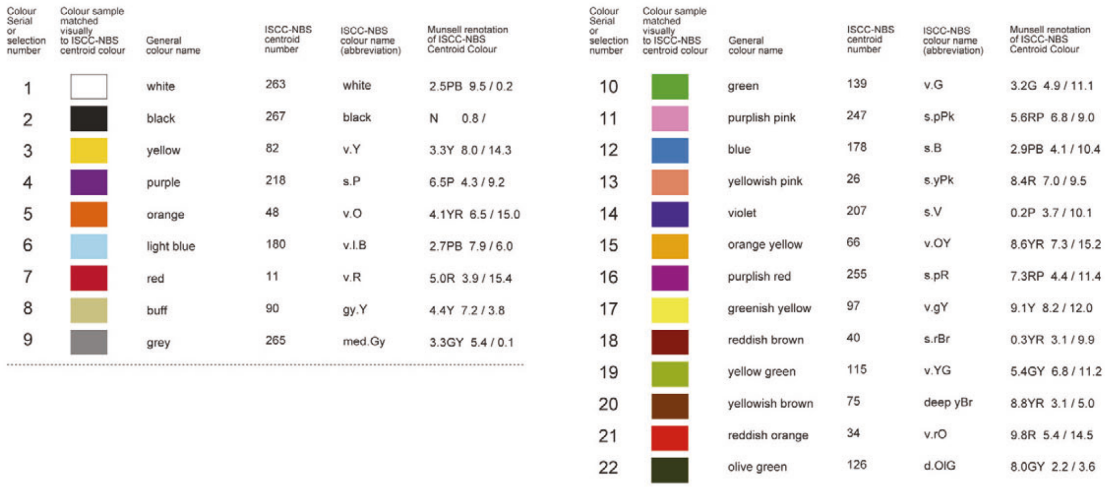
Colour

The static Peer Model representation did not make use of a very important aspect of visualisation: colour. It should be noted that it has benefits to reduce or completely refrain from the use of colour. For example, sketching Peer Models on paper or whiteboard is definitely easier if only a single type of pen is required. However, for the purpose of the monitoring software, there is no reason not to use colour as a means to enhance the visualisation and increase the readability. Many works on information visualisation deal with the topic of colour [13,90]. And while the general use of colour is easily justifiable, how to use it is not trivial. To quote Edward Tufte:

> [. . .] avoiding catastrophe becomes the first principle in bringing colour to information: Above all, do no harm. [86]

It was decided upon that the information of entry types is a very crucial one and that it is justified to use colour to differentiate between entry types. The type of an entry is arguably its most important characteristic. Likewise, wirings first and foremost select entries based on their type. In this light it makes sense to discern entries of different type by colour.
Using the same colour for multiple things gives them a sense of cohesiveness. Likewise, different colours are used to emphasise the disparity between things. With the amount of different groups the need for different colours also rises. However, there are limits on the number of colours a human can easily distinguish between [64]. The use of a good colour palette is essential for this task.
While there is no theoretical upper limit on the number of different types of entries, the Peer Models currently seen in practice typically do not use more than a handful of types. Nevertheless, it remains important that a good colour palette is chosen. Coming up with a sensible amount of easily distinguishable colours was an important requirement. The technical term for this task is *Nominal Information Coding* [90]. Choosing colours is not straight forward. In many encodings, colours that appear very distinct not necessarily reflect that in their values and vice versa. For example, equally distributing colours based on their hue does not result in visually equidistant colours. Many works exist that propose guidelines and algorithms for generating unique colour palettes [12, 14, 29, 33]. Kenneth L. Kelly proposed a list of 22 colours of maximum contrast [48]. The list is tailored to be extensible in the sense that if a subset in the form of the first n colours is chosen, they exhibit the maximum contrast between each other. This is a beneficial property as smaller Peer Models that do not need all colours can work with the most distinct colours while larger Peer Models subsequently add additional colours. The correspondence of type to colour is done by sorting the occurring entry types alphabetically and mapping each to the next available colour. Furthermore, Kelly keeps in mind the various types of colour blindness in the human population and builds the list in accordance to these limitations. Kelly's list serves as a good starting point for choosing a

suitable palette. Figure 5.8 shows the list as proposed by Kenneth L. Kelly. However,



Colour Serial or selection number	Colour sample matched visually to ISCC-NBS centroid colour	General colour name	ISCC-NBS centroid number	ISCC-NBS colour name (abbreviation)	Munsell renotation of ISCC-NBS Centroid Colour
1		white	263	white	2.5PB 9.5 / 0.2
2		black	267	black	N 0.8 /
3		yellow	82	v.Y	3.3Y 8.0 / 14.3
4		purple	218	s.P	6.5P 4.3 / 9.2
5		orange	48	v.O	4.1YR 6.5 / 15.0
6		light blue	180	v.l.B	2.7PB 7.9 / 6.0
7		red	11	v.R	5.0R 3.9 / 15.4
8		buff	90	gy.Y	4.4Y 7.2 / 3.8
9		grey	265	med.Gy	3.3GY 5.4 / 0.1
10		green	139	v.G	3.2G 4.9 / 11.1
11		purplish pink	247	s.pPk	5.6RP 6.8 / 9.0
12		blue	178	s.B	2.9PB 4.1 / 10.4
13		yellowish pink	26	s.yPk	8.4R 7.0 / 9.5
14		violet	207	s.V	0.2P 3.7 / 10.1
15		orange yellow	66	v.OY	8.6YR 7.3 / 15.2
16		purplish red	255	s.pR	7.3RP 4.4 / 11.4
17		greenish yellow	97	v.gY	9.1Y 8.2 / 12.0
18		reddish brown	40	s.rBr	0.3YR 3.1 / 9.9
19		yellow green	115	v.YG	5.4GY 6.8 / 11.2
20		yellowish brown	75	deep yBr	8.8YR 3.1 / 5.0
21		reddish orange	34	v.rO	9.8R 5.4 / 14.5
22		olive green	126	d.OlG	8.0GY 2.2 / 3.6

Figure 5.8: List of 22 colours for maximum contrast proposed by Kenneth L. Kelly [48]. Figure taken from [33].

adjustments are needed to make it work for the Peer Model visualisation. The first two colours of Kelly's list are understandably simply black and white. While they are indeed the two colours with the most contrast, they can not be used for labelling entry types. As the various structures in the Peer Model are already visualised in black on white background, using black and white for entries as well would not work. In the same vein, colours that are relatively bright are not a good fit because they do not stand out well against a white background.
Using the list of Kelly as a starting point, 10 colours have been chosen based on their visual distinctiveness. This number presents a good compromise that should work for most use cases. In the rare cases where it does not suffice, the colours are reused in a round-robin fashion. Figure 5.9 shows the final 10 chosen colours.
Additionally, the final monitoring tool provides a key table that displays the correspondence between entry types and colours. This helps to reinforce the affiliation and supports the developer in associating colour-entry type pairs.

Entries

Unlike the static representation, the main view of the dynamic representation requires the explicit visualisation of entries. Otherwise it is not possible to properly represent the model's state. While entries are very basic in their nature, they can potentially contain a lot of information. Apart from the type, an entry may include - at least in theory - arbitrary quantities of co-data and app-data. Particular parts of co-data have an intrin-

sic meaning, like the TTL, TTS and the DEST-properties. Displaying these properties explicitly enriches the visualisation and discloses more information about the entry. In certain circumstances it is also advantageous to group entries of the same type together to save space and provide a quick overview, rather than displaying each entry separately. Developers who have been working with the Peer Model are using scribbles to aid their understanding. It is not surprising that they sometimes need to draw entries explicitly. Evidenced by the interviews, developers often tend to use circles to represent entries. Circles are very easy to recognise and conveniently look completely different to all other elements in the already developed visualisation. Using circles to represent entries is therefore a logical step. Throughout the whole dynamic representation, entries are always displayed in the same way. This enhances the recognition value drastically and spares developers from learning multiple graphical notations for entries. The circle representing an entry is coloured accordingly to reflect its type. As discussed previously, figure 5.9 shows a list of 10 entries, coloured according to the developed palette.
The grouping of entries (later referred to as *entry groups*) of the same type (and colour)

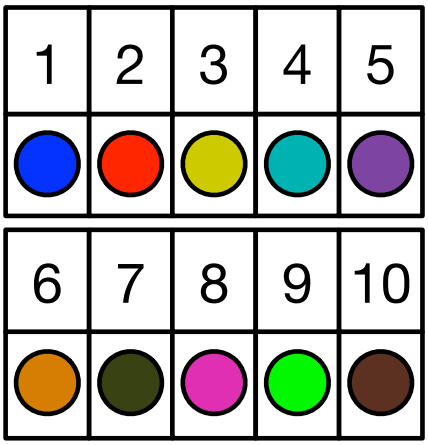


Figure 5.9: Final 10 colours used as a colour palette for distinguishing entry types. The palette is based on the list of 22 colours for maximum contrast proposed by Kenneth L. Kelly [48].

is used to save space and provide a more compact representation. The difference between a single entry and a group of entries is hinted at by sketching a stack of entries put on top of each other. Additionally, the number of entries in the group is displayed. Figure 5.10 shows how groups of entries are visualised. To get information about the individual entries, the developer may click on the group to open a list where each entry is displayed individually.
The TTL and TTS co-data properties of entries play a special role as they can greatly influence an entry's movement and subsequently, the Peer Model. For this reason, it

is important to visualise their presence explicitly. While not common, both properties might be set for a single entry simultaneously. In both cases, an absolute timestamp is set that specifies the time when the entry is deleted or respectively made active. In order to avoid a potential gap between the real events and the re-enactment within the monitoring tool, TTL and TTS events are explicitly stored within the trace data. That means that the monitoring software does not have to check for these events itself and can rely on the data it is being provided with. On the other hand, this means that the timestamp set in the properties does not necessarily coincide with the actual event time. During the design phase of the software, considerations were given about whether and how to display the TTL and TTS properties of entries. In the end it was decided against explicitly displaying the time left until an entry's TTL or TTS was reached. Overloading an entry visualisation with a detailed timer representation was found to be too much for the little available space. Furthermore, as just discussed, calculating the remaining time is not unambiguous. Instead, the entry representation is only slightly modified to advert the user to the presence of a TTL or TTS property. The exact timestamp can still be revealed by clicking on the entry and bringing up the context menu, as discussed later. The presence of a TTL property is visualised by drawing the entry itself semi-transparently, emphasising its impending removal. An entry with a TTS property is represented by a down-scaled circle. This hints at the fact that such an entry is not (yet) taken into consideration by wirings and is not yet *fully grown*. Entries that have the DEST property set are not visualised differently from entries without it. The reasoning behind this is that entries with this property are supposed to be sent on their way by the corresponding event in the trace. How this is visualised is discussed shortly in its own subsection. An entry with a flow ID is marked by a white triangle and the ID in the middle. Note that actual flow IDs from the trace are generally not easily readable. Thus, the monitoring tool maps them to incrementing integer values starting at zero and displays these mapped IDs. Additional information, like other co-data properties and

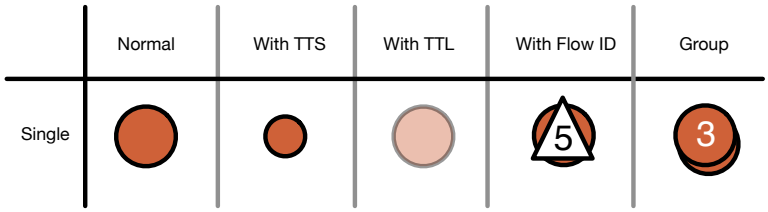


Figure 5.10: Table showing different entry representation based on which property is set. Certain properties may also occur in combination with each other. For this reason, the representations are combinable as well. For example, an entry that has both TTL and flow ID set is visualised semi-transparently in addition to the white triangle with the flow ID.

the app-data are not visualised directly as they are generally not as important for the coordination logic. However, the developer might still want to enquire about this data. By clicking on an entry, a separate window opens that displays the complete set of information that this entry holds. The data is presented in a nested tree view, by which the developer can easily navigate through the complete information. Figure 5.11 shows this view.

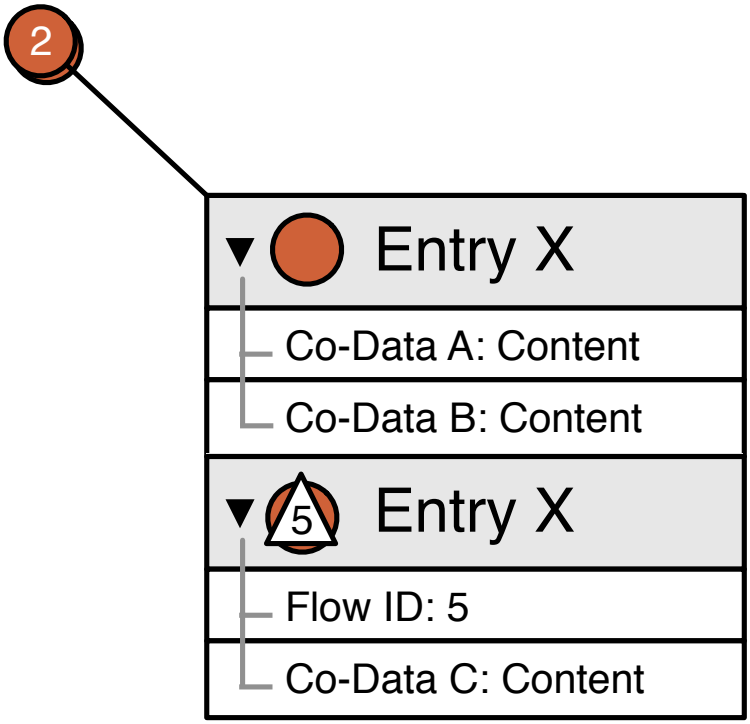


Figure 5.11: Context menu that opens when clicking on an entry or an entry group. For each entry, all the associated co-data is shown as key-value pairs.

Wirings

Guard, action and service links all have in common that they select entries based on their type. To emphasise this information, the links are colour-coded based on the type of entry they select. This makes it much easier for developers to quickly gauge what kind of entries a wiring acts upon and and quickly compare this information with the contents of the respective peer container.
The dynamic data concerning wirings are its activations. To overlay dynamic information on top of the static structure, a wiring needs to display the state of its currently ongoing activations. A wiring passes through multiple states during an activation. Each

of these states corresponds to an event, such as a guard or action link activation. Subsequently, each event presents a distinct moment in the trace that can be visualised. During an activation, each link is either in the *waiting state*, the *active state* or in the *finished state*. The waiting state means that the link has not yet performed its duty. The active state indicates that it is just performing its task and is currently moving the selected entries. After that, the link goes into the finished state where it stays until the activation is finished.
Apart from the link's state, the entries that are affected are the most critical piece of information. A developer wants to know exactly which entries were affected by each link, including information about the entries themselves. For these reasons, it is sensible to design distinct representations for each of the three states. In the following, guard, service and action links are each discussed with respect to their waiting, active and finished representations.

Guard Links

A suitable metaphor to the three possible states are the states of a traffic light. Red means stop (=waiting state), yellow indicates a change (=active state) and green means go (=finished state). Subsequently, a guard's base is coloured to reflect this, making it easy to quickly gauge its state. An active guard link representation should properly depict the entries it acts upon. The static representation features an arrow that points in the direction of entry movement. This can be used to emphasise the guard's activity. Also, each guard has a guard base. By slightly moving its label and enlarging the base, it provides enough space to place an entry or an entry group. Figure 5.12 shows a guard link in its waiting, active and finished state. In terms of affected entries, none check guards represent a special case. As a none check guard activates if there are no entries matching its specification, it does not deliver any entries to its wiring. In this case, there are simply no entries displayed, even in the active state.

Action Links

Action links are enhanced very similar to how guard links are. An action link is either waiting, finished or active, in which case it moves entries that need to be visualised. But rather than displaying the affected entries next to the wiring, they are displayed at the other end of the link where the peer container is. This emphasises that the entries will be placed inside the container. Other than that, it closely corresponds to the way guard links are displayed. Figure 5.13 displays an action link in the three possible states. Similar to how a none check guard does not move any entries, an action link might not move any entries either because none of the entries present in the EC fit its specification. No entries are displayed in this case.

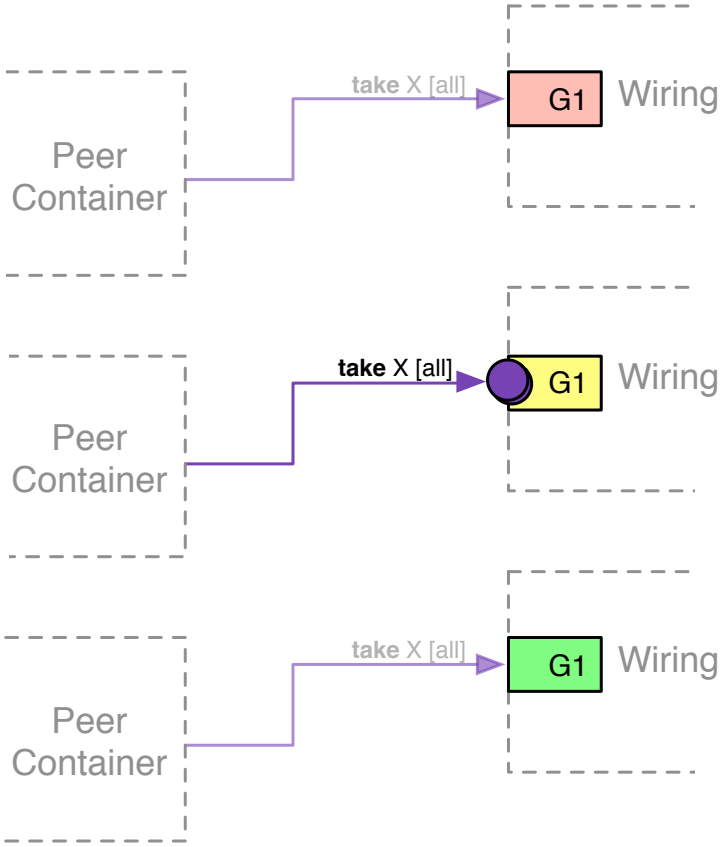


Figure 5.12: A guard link in its three possible states: not yet fired (top), active (middle) and finished (bottom) state.The link is always coloured to reflect the guard specification's entry type. The three colours are specifically chosen to not clash with the colours for entry types. In the active state, the moved entries are displayed within the link's base with the arrow directly pointing at them. If not active, the link and the specification label are displayed semi-transparently.

Services

Just like in the static case, service input links closely resemble guard links, while service output links are very similar to action links in their appearance. A service link does not have a distinct base. Instead, the service area itself is coloured depending on its state. A service may also be in one of three states: waiting, active or finished. Figure 5.14 shows a wiring with a service during the course of an activation.

The entry collection (EC) is a very critical part of every wiring. During an activation, the EC is in constant flux and changes with almost every event. As such, it is vital

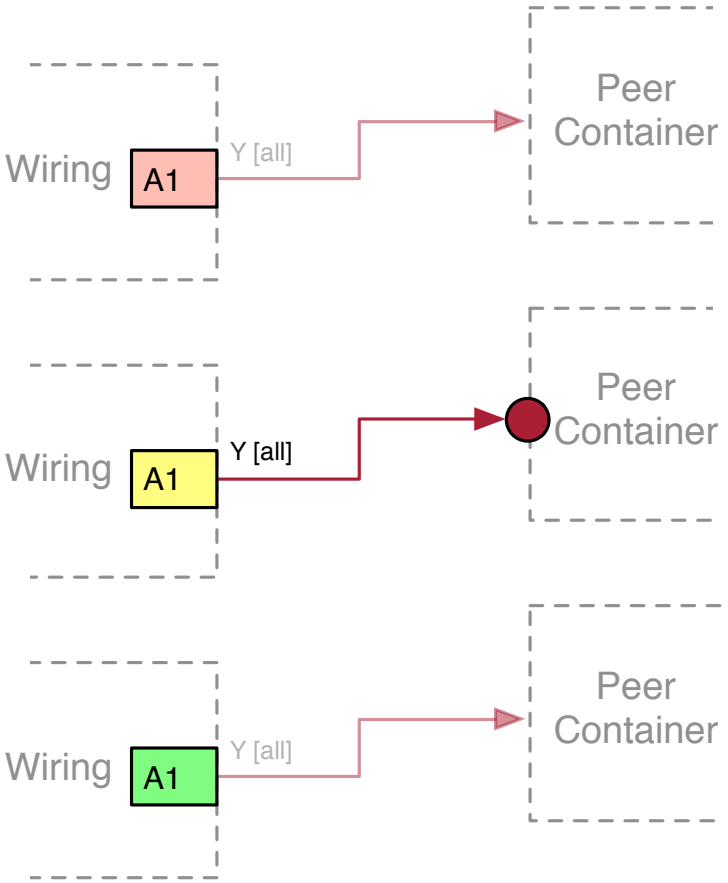


Figure 5.13: An action link in its three possible states: not yet fired (top), active (middle) and finished (bottom) state.The link is always coloured to reflect the action specification's entry type. In the active state, the moved entries are displayed at the end of the link with the arrow directly pointing at them. If not active, the link and the specification label are displayed semi-transparent.

to display its contents for the duration of an activation. Visualising the EC however presents a problem. Unfortunately, the static representation does not reserve any space for the EC as it does not visualise it at all. To properly display the EC and its contents, each entry or entry-group needs a separate space. Fortunately, the number of entries (or entry-groups) in an EC is limited. In the most extreme case, each guard link and service output adds entries of a distinct type to the EC. Therefore, an upper bound for the number of distinct entry-groups is the number of guards plus the number of service output links. That means that the center area of the wiring always provides enough space for the EC contents. In the static representation of wirings, the center area of a wiring is solely occupied by the wiring's label. Especially for larger wirings with many guards,

services and actions, this results in a lot of wasted space. In order to use the available space better, the EC is positioned in the center area of its wiring. The label is moved down to the border of the wiring. This small adjustment is a compromise that makes visualising the EC possible while only slightly changing the existing visual notation.
The EC container is visualised very similar to peer containers (discussed in the following section), which also emphasises their similar nature. But instead of layouting the entry(-groups) in a vertical fashion, they are organised horizontally. If an entry does not fit in the same line anymore, a new line below the previous one is begun. The entries themselves are displayed in the already established style.
A wiring's activation lasts from the first activated guard to the last activated action. When visualising moments within that timespan, the wiring's body is displayed with a bold border and its label text also is displayed in bold to indicate its activity. Depending on which state the activation is in, different parts of the wiring are shown as waiting, active or finished, respectively.
Figure 5.14 shows a complete iteration of a wiring activation. It consists of six moments in time that can be selected by the user. In the state, the guard links fire (I), after that, the service is called (II-IV) and finally, the two action links move their respective entries (V-VI). At any moment in time, the visible entries may be inspected to find out more about their co-data and app-data. Note how the wiring becomes more and more "green" and the red declines as the activation continues, which emphasises the notion of progress very well. An important issue is the possibility of multiple concurrent activations of a single wiring. While it is important to have a complete picture of the wiring's full state, multiple activations generally tend to belong to different chains of events. It is therefore expedient to not merge activations in a single representation. Instead, when encountering multiple activations, the developer is given the ability to switch between activations at will. The proposed dynamic wiring visualisation only displays one activation at a time. A counter in the middle of the wiring shows the currently displayed activation and how many there are altogether. Buttons allow the developer to switch back and forth between them, if desirable.
It is important to note that an EC is bound to a single wiring activation. That means that multiple concurrent activations each have their own EC and therefore do not share any entries. In terms of visualisation, that means when switching between concurrent activations, the appropriate EC needs to be shown. Figure 5.15 shows a dynamic wiring representation that highlights such a case.

Move Wirings

Move wirings are a common occurrence in many Peer Models [50]. While there is nothing special about move wirings in terms of functionality, their simple structure combined with their commonness justifies the design of a distinct visualisation. The main goal of this distinction is the potential to save space and obtain a more compact, yet equally

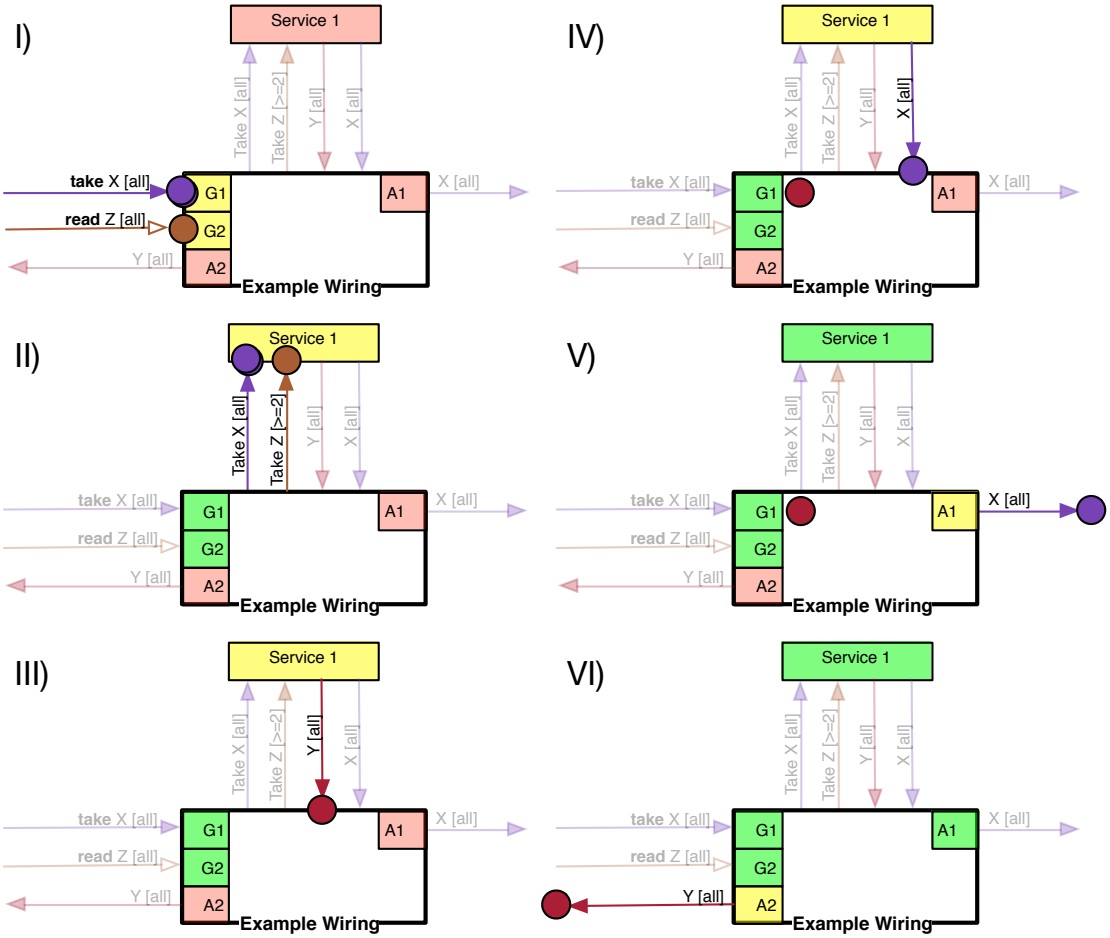


Figure 5.14: Typical procedure of a wiring activation. Successively, more and more parts of the wiring are marked in green. First the guard links fire (I), then the service (II-IV) and finally the action links (V-VI). The EC also changes repeatedly throughout the duration of the activation. Peer containers are omitted for clarity.

expressive representation. Note that this also applies to the static Peer Model representation as no suitable visual distinction has been proposed yet. While [50] proposed a distinct notation for move wirings, the notation has since been deprecated.
As a move wiring contains exactly one guard and one action, which furthermore have very similar specifications, a number of simplifications can be applied. The specifications can be consolidated and displayed only once. Also, the EC only has to be big enough for a single entry or entry-group at most. Figure 5.16 displays a wiring in two representations. Once as in the normal way and once in the simplified fashion. The guard and action links' operations are combined under the term **move**. The wiring's base is shaped to resemble an arrow in order to emphasise the movement mechanism.

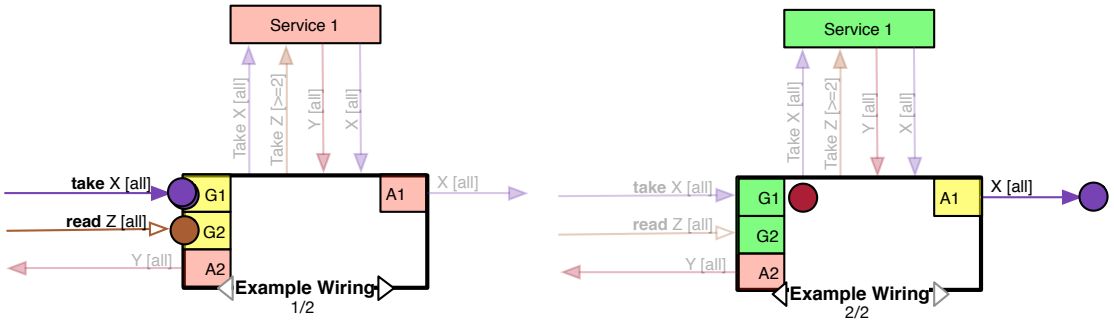


Figure 5.15: Visualisation of multiple concurrent wiring activations. Note how the two visualisations display different states and different ECs. The user can switch between the activations be clicking on the arrows or by selecting them in the timeline.

Just as normal wirings, move wirings can be activated multiple times concurrently. In this case, arrows and an index are shown similar to multiple activations in the normal representation.

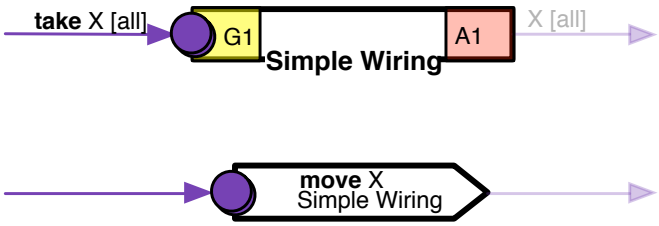


Figure 5.16: Wiring displayed in the normal fashion (top) and in the simplified form (bottom). Note how the link specifications for guard and action link are combined and moved into the wiring. Left of the labels is the EC area that is sized to fit one entry-group. Colouring in the traffic light scheme is omitted as well.

Flow Aware Wiring Activations

The visual representation of wiring activations is an important matter as it changes the way a wiring works temporarily. A guard with an activated flow dependence flag is marked by a small zigzag line. As soon as a wiring with a flow dependent guard encounters one or more fitting entries with a set flow ID, subsequent guard may only take entries with the same or no flow ID. If not made explicitly clear, this can lead to situations in which a wiring does not proceed with its activation even though it seems to have fitting entries for the currently blocking guard. To minimise user confusion, the wiring

activation's internally set flow ID is displayed at the base of the guard responsible for it. Figure 5.17 shows a small example of a wiring activation with flow dependence.

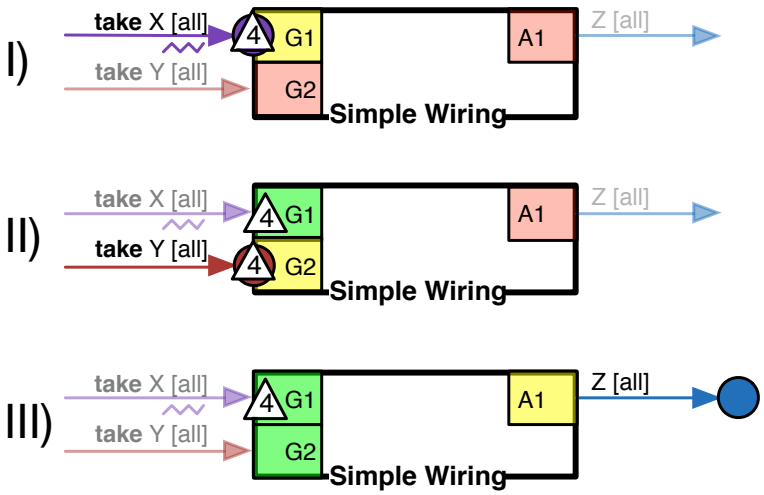


Figure 5.17: Simple wiring activation with a guard with flow dependence. After the guard has successfully fired, the corresponding flow ID is shown at the guard's base for the remaining duration of the wiring activation. This is also an example for explicit commits on guards. The two guards fire separately from each other and are therefore visualised in two separate events.

Compact Service Representation

Services, especially their input and output links, take a lot of space. Developers already familiar with the Peer Model at hand can make use of a more compact representation. Therefore, the service link labels can be hidden and the links themselves shortened. This reduces the size of most wirings drastically. Figure 5.18 shows the difference between the normal and the compact service representation. While developers new to the Peer Model need the service link specifications to familiarise themselves with the details of the model, a developer who is working with a specific Peer Model for a longer period of time generally does not need to see this information. Hiding it creates space for other important Peer Model elements and visually condenses the representation.

Peers

Contrary to the static representation, a peer within an active Peer Model needs to display the contents of its PIC and POC. Fortunately, the static representation reserves a dedicated space for both containers, each represented as a rectangular area. The contents of a container can be superimposed onto the static representation in a relatively straight

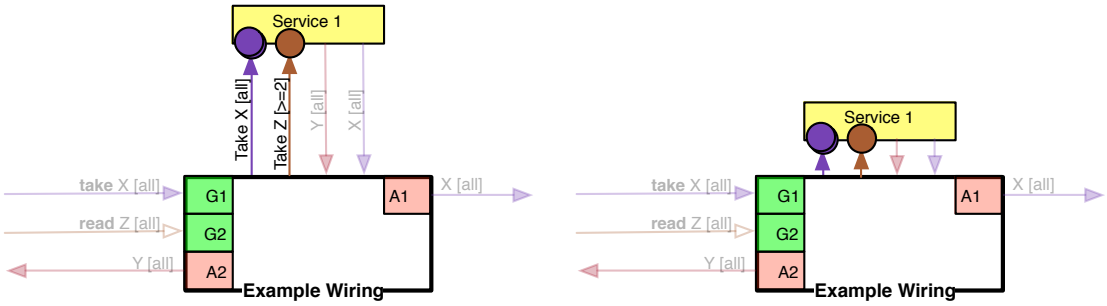


Figure 5.18: Comparison of the normal service representation (left) and its compact counterpart (right). Hiding the service link specifications enables a much more compressed representation. The user may switch between the two representations at any time.

forward way. The container areas only need to be slightly widened compared to the static representation to work for all use cases.
Entries that are located in the container are displayed one below the other in a vertical strip in the center of the rectangle. To save space, entries of the same type are grouped together, as explained previously. Figure 5.19 shows a collapsed dynamic peer representation with entries present in its containers. Note that the rules for expanded and collapsed peers are still the same as for the static case. Some peer visualisations can be-

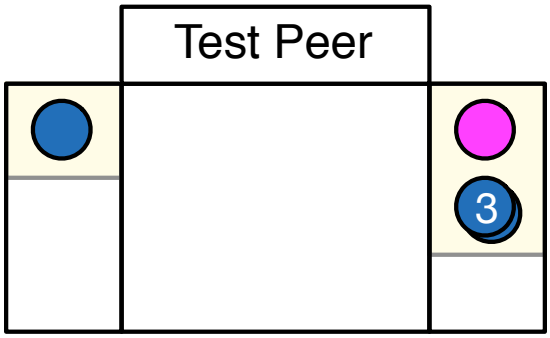


Figure 5.19: A collapsed peer with the contents of its peer containers visualised.

come very large, depending on the number of sub elements and their layout . Vertically, the PIC and POC representations also shrink and grow with their associated peers. If the entries in the containers would be placed at fixed positions, they could potentially be off screen and the user could not see them even when the peer itself is visible. Especially when a peer becomes larger than the main view's viewport (which governs what the user sees), the contents of its PIC and POC might not be shown and could deceive the user into thinking they are empty. To counteract this, a container's content is not fixed.

Depending on the position of the viewport, the contents are moved dynamically along the vertical axis in order to stay visible as long as the corresponding peer is. Figure 5.20 explains this behaviour in detail. Conversely, there can be situations where the contents

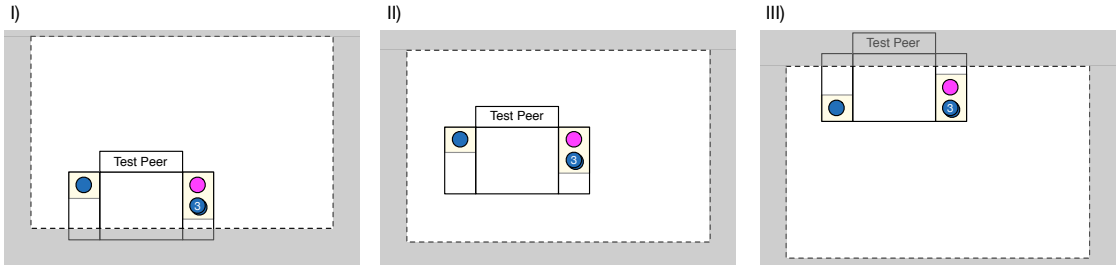


Figure 5.20: A peer representation and its container contents in three situations. The user's viewport is displayed as the white rectangle with the dashed black line. Depending on where the viewport is positioned, the peer container contents move in order to stay visible whenever possible.

of a peer container do not fully fit into the provided space, mostly when the associated peer is collapsed. In these situations, the entries that do not fit anymore are omitted and instead, an icon indicates this fact. The icon can be clicked to reveal the additional entries in a context window, similar to the window that opens when clicking on an entry group (shown in figure 5.21).

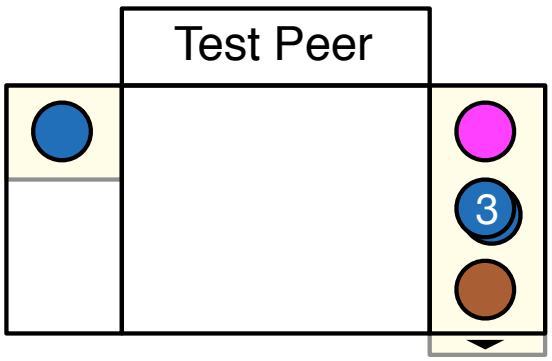


Figure 5.21: Peer container where not all entries fit into the provided container space. An icon at the bottom indicates this fact and can be clicked to reveal the additional entries.

DEST property moves

The most common way for entries to move around a Peer Model is by being transported by wirings. However, the *DEST property* mechanic (discussed in chapter 3) allows for an alternative transportation method. Unfortunately, visualising entries that are being moved in this way is not straightforward. If not explicitly visualised, entries seemingly vanish from one peer container and appear at another at a later point in time.
Moving entries using the DEST property bypasses the predefined composition and enables the movement of entries without being constrained by the static model. A service may set the DEST property on entries at runtime and can send them to any peer container within the model, disregarding the rules enforced by wirings and their valid targets for guard and action links. This presents a visualisation challenge as entries can be sent between any two peer containers.
In order to represent this movement, the source and destination container are connected by an arrow line. Proper orthogonal routing of this connection would be complex and not produce good paths in many cases as it can not be taken into consideration during the layout process. Instead, the connecting line is straight and simply cuts through any elements along its way. While this is not ideal as it may obstruct elements positioned between the source and the destination, the visualisation of the entry movement is definitely the important information in this situation and is therefore given priority. The line itself is drawn dashed and coloured like the entry it moves. An arrow similar to the ones on guard and action links is drawn at the destination to illustrate the direction of movement.
An entry being moved by setting the DEST property consists of two distinct events that happen at different points in time. As such, both events are visualised separately. The first event occurs when the entry is sent on its way from the source container. The other event happens once the entry arrives at its destination container. Figure 5.22 visualises the two events in a simple Peer Model. The endpoints for the visualisation are located at the top right corner for POCs and the top left corner for PICs. These points also move to stay visible when the viewport changes, as discussed earlier.

Entry TTL and TTS events

It has been established how entries that have a TTS or TTL property are represented. But the actual events that correspond to these features have yet to be discussed. During an TTL event, the entry is removed from the peer container it is currently in. This is visualised by putting the entry in question to the top outer corner of the container, similarly to the DEST property move case. Additionally, the entry is crossed out to properly represent that it is now being removed. In the case of a TTS event, the entry in question is displayed with a halo around it to emphasise that it is now in a usable state. Figure 5.23 shows the representation of both events.

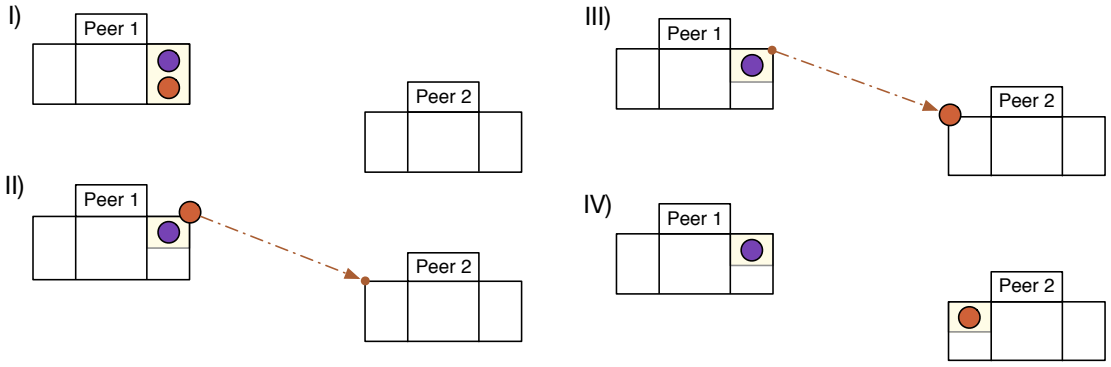


Figure 5.22: Visualisation of a DEST property move. I shows the moment before the move. Then the entry is sent on its way (II). III represents the moment when the entry arrives at its destination and IV shows the model after the move is complete.

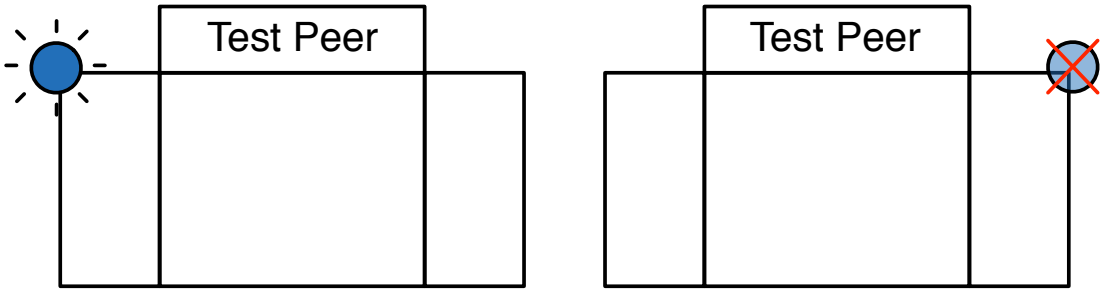


Figure 5.23: Visualisation of an entry TTS event (left) and an entry TTL event (right) in the the main view.

Timeline

The timeline represents the context view and has multiple functions. The first is to help the user make sense of the temporal structure of the Peer Model trace. The events are plotted according to their moment in time in order to give the user an overview. Secondly, the user is able to move through the trace and jump to different points in time by clicking on event representations in the timeline.
The temporal aspect of the events are laid out on the horizontal axis. Vertically, the timeline is split into rows, subsequently called *Time Rows*. Each processor active in the trace is given its own time row. Every event can be assigned to the processor it occurs on and hence, to a time row and is displayed in the time row's *Event Area*. Multiple time rows are positioned below each other. Using a slider, the user is able to adjust the granularity of the time dimension. Decreasing it causes events to move closer together while increasing the time granularity positions them further apart. Depending on the

trace specifics, the user might want to zoom out to see more events simultaneously. In other traces, he/she might want to zoom in to decrease the overlap of events and increase the legibility of the labels. In most traces and depending on the adjusted zoom level, the timeline will not fully fit onto the screen. The timeline is therefore scrollable in both dimensions.
The correlation between the main view and the timeline is given by the *Indicator*. The indicator is positioned to represent the active moment and the currently visualised Peer Model state in the main view.
Figure 5.24 sketches its general structure. The timeline view closely resembles similar views in video editing software packages. Within an event area, different types of events

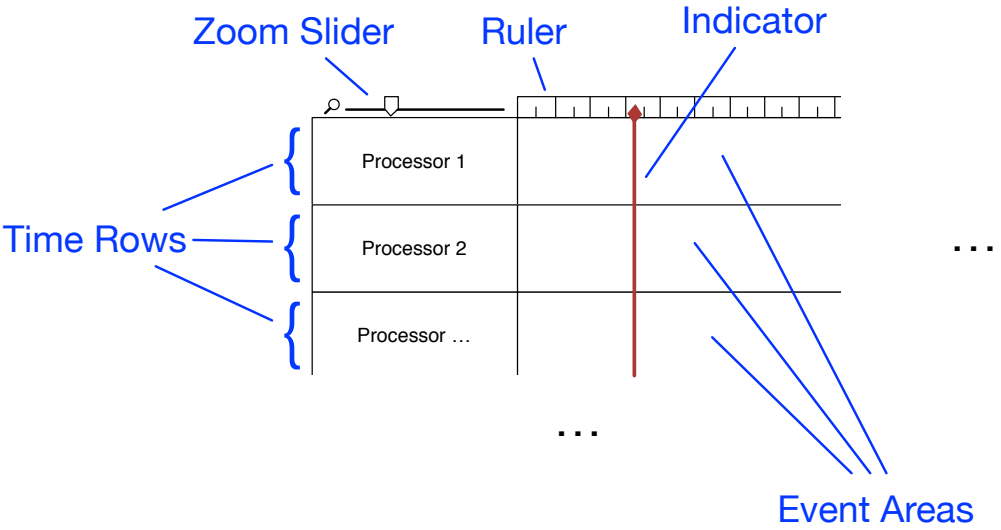


Figure 5.24: Overview over the structure of the timeline view. The temporal dimension is laid out horizontally. The events are positioned according to their time and within the processor they occur in. An indicator is used to show the currently selected point in time. The zoom slider can be used to adjust the zoom level in the temporal dimension.

are visualised. Each event can be selected by the user which in turn visualises this event in the main view. A discussion for each event type follows:

Initial State

The first event in each processor's event area is the initial state. For each processor, the initial state is simply depicted by a vertical line that is located at the far left. The line can be selected in order to jump to this moment and see the state of the Peer Model at the beginning of the trace.

Wiring Activations

A wiring activation actually consists of multiple events. These are the events

corresponding to action and guard link activations as well as service input and output activations. To emphasise the affiliation of events belonging to a single wiring activation, a bar that spans the time from the first guard link to the last action link activation is drawn. The bar is subdivided into areas for the guards, actions and services. Each of these areas is coloured slightly differently and a linear gradient between them helps to emphasise the transition. The timespan of services is also denoted explicitly by a bar with the service's name.
For every event, a vertical line represents the exact moment and it also roughly defines the clickable area to jump to this moment. For guard and action link activations, a small label denotes the index (or indices, in the case of multiple guard firings in a single event) to clarify the order and emphasise the correlation. Guard and action links have differently shaped polygon labels. The labels for the guard links resemble the back part of an arrow, while the action links are shaped like the front part. Service events are shaped like arrows pointing up for service input links and arrows pointing down for service output links respectively. The backgrounds are always coloured according to the corresponding entry type that the event acts upon. In the case of guard link events with multiple guards firing, the background is coloured in a linear gradient that incorporates all the corresponding entry type colours. Figure 5.25 shows a typical wiring activation representation in the timeline.
The representation of a wiring activation in the timeline provides an overview and shows the important aspects of a complete activation at a glance. Details like what entries are processed specifically and the contents of the EC are omitted in favour of providing an easily processible outline.

Entry TTL and TTS

The events pertaining to expiring TTLs and TTSs of entries are visualised similarly. A simple vertical line that denotes the exact point in time, overlaid by a label stating "TTS"or "TTL" respectively. The label is coloured according to the affected entry's type.

DEST Property Moves

As discussed earlier, a DEST property move consists of two events. One event when the affected entry leaves the peer container and another when it arrives at its target. An interesting point about these two corresponding events is that they can happen at different processors. This means that the two events can not be connected by a simple bar as it is possible with wiring activations. Events on different processors are inherently positioned below each other. For this reason, a thin line connects the events that belong to a DEST property move. Similar to the visualisation of these events in the main view, the line potentially cuts through other event representations in the event areas between but is thin enough

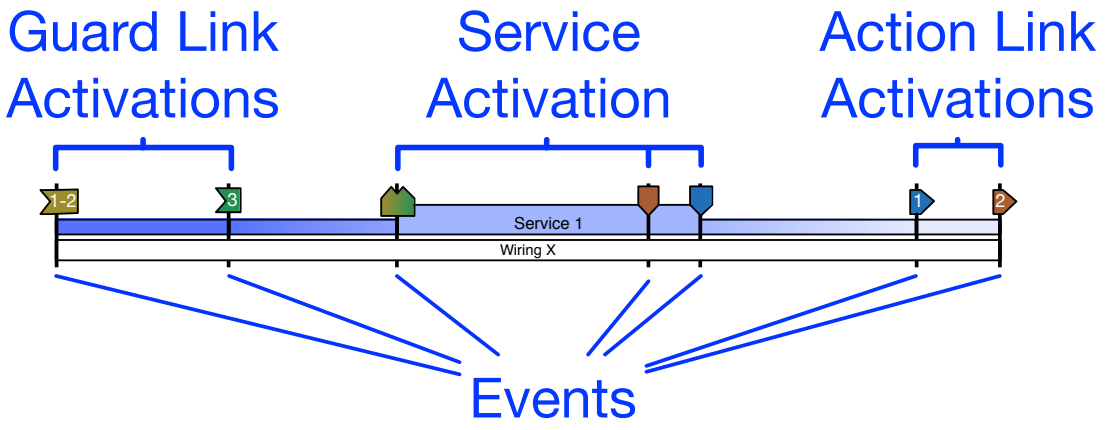


Figure 5.25: Typical wiring activation representation in the timeline. Guard and action link events as well as the service events are denoted by a vertical bar and a coloured label. The service's name is explicitly shown within the service' timespan bar. The bottom bar shows the full temporal extend of the activation and the associated wiring's name. Note how two of the three guard events as well as the two service input events are merged together.

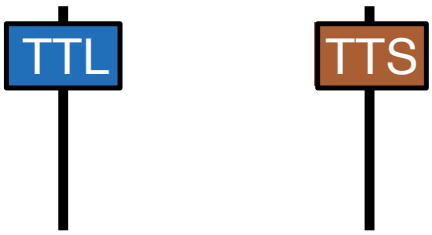


Figure 5.26: Timeline entry TTS and entry TTL visualisations. Similar to wiring activation events, the label is coloured according to the corresponding entry's type.

to not affect readability much. Figure 5.27 displays how a DEST property move is visualised in the timeline view.

An important feature of the timeline is to show multiple events and event-chains happening at the same time. This is done by displaying concurrent events organised along the vertical dimension. This can be the case with events happening on different processors, but also within a single processor. In this case, the corresponding time bar's height is increased to accommodate concurrent events. For example, two wiring activations might overlap in terms of their start and end time and therefore need two separate *lanes* within the time row. The monitoring software tries to fit the event representations as

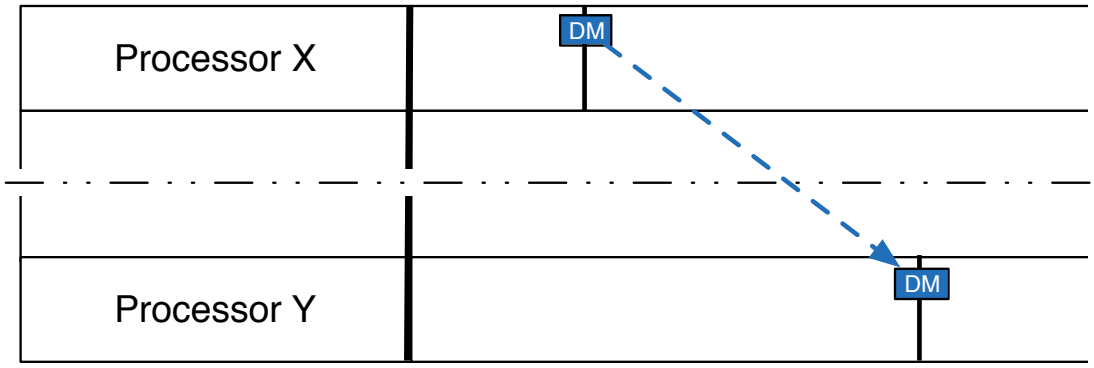


Figure 5.27: Visualisation of a DEST property move in the timeline across different processors.

tightly as possible not to waste precious space. Figure 5.28 shows a complete example of multiple events visualised in the timeline, some of which are occurring concurrently.

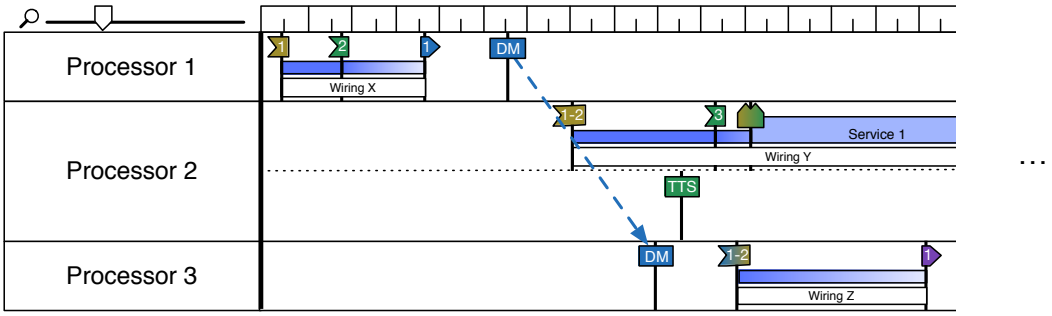


Figure 5.28: Complete example of the timeline view. A developer familiar with the Peer Model's mechanics can easily get an overview of the occurred events. "Processor 1" activates "Wiring X", which emits an entry (encoded in blue) with a set DEST property. Afterwards, the entry is sent across processor boundaries to "Processor 3", where "Wiring Z" picks it up shortly after. In the meantime, "Processor 2" executes "Wiring Y" which can fulfil its third guard after an entry's TTS has expired (encoded in green). As the events happen concurrently, "Processor 2" has two lanes.

CHAPTER 6

Layouting

One of the main challenges when developing the monitoring software is the layouting of the examined Peer Model. Users should not be tasked with manually positioning wirings and sub peers and routing guard and action links before starting their intended task. Thus, automatic layouting was a vital part of the software and is given its own chapter. Well layouted models help the user in completing his/her tasks faster and in a more efficient manner. Poorly constructed layouts do not offer a good representation of the Peer Model and require manual repositioning of elements, thereby hindering the user.
As established in chapter 5, both the static and the dynamic representation of the Peer Model use the same layout. While they may display different information depending on the context, the underlying layout is the same for both.
The devised Peer Model representation is that of a graph at its core. However, several requirements, such as the ordering of guard and action links within a wiring present additional challenges. Several papers exist on what makes good graph layouts [16, 65–67]. These papers often conduct empirical studies to validate their claims. Many of them define aesthetic criteria a graph should fulfil in order to aid the understanding. These criteria include, but are not limited to:

1. Minimised number of edge crossings
2. Minimised number of edge bends
3. Maximised symmetry
4. Edge orthogonality
5. Node orthogonality

6. Consistent flow direction
7. Minimised edge lengths
8. Unoccluded nodes
9. Minimised size
10. Suitable aspect ratio

The produced layouts should follow these criteria in order to make it pleasant to work with. Unfortunately, it is not always possible to fulfil all criteria equally well. Sometimes, criteria conflict with each other and a decision has to be made on which one to favour and to what extent. For example, in order to avoid edge crossings, edges might have to be routed in a circuitous fashion which in turn increases the total edge length and potentially introduces additional edge bends. The decision on which criteria to favour and more generally, which decisions lead to the aesthetically more pleasing graph is not trivial. Purchase et al. [65] tried to objectify the matter by introduced a metric to gauge the readability of general, undirected graphs. Ware et al. [91] introduce a methodology for evaluating the cognitive cost of graph aesthetics and empirically test their findings by tasking users with finding the shortest paths in spring layout graphs. Another work formulates an objective cost function based on a number of graph layout criteria and compares algorithms according to this metric [20]. While the visual Peer Model notation is not exactly a graph, it can relatively easily be transformed into one (as discussed later). Subsequently, the established criteria can be used as a guideline for layouting Peer Models as well.

Generally speaking, different approaches to tackle the issue of graph drawing exist. The two most prominent ones are *Force-Based Graph Drawing* and *Layered Graph Drawing* (also called *Hierarchical Graph Drawing*). In force-directed graph drawing methods, an initial placement of nodes is continuously altered according to forces defined through the use of concepts adopted from physics, such as springs. Both repulsive and attractive forces may be applied between single nodes or groups of nodes. The problem is either solved through the minimisation of an energy function or through iterative adaptation of the intermediate graph structure. An in-depth introduction to the algorithms used in force-based graph drawing methods can be found at [37].

The most prominent implementation of a layered graph drawing method is the *Sugiyama framework* [83], named after Kozo Sugiyama. It works by assigning each node to a layer and structuring the graph in a tree-like fashion. Layered graph drawing very closely resembles how a typical Peer Model is structured. Subsequently, certain concepts will be adopted for the Peer Model layouting process. A detailed discussion of the Sugiyama framework follows shortly.

Layouting a Peer Model is in many aspects similar to printed circuit board layouting

(subsequently abbreviated *CBL*). The problem of efficient placement of spatial elements onto a confined space, whilst following certain rules and constraints is common for both fields. There exists a variety of software packages capable of automatically and semi-automatically placing and routing elements on a circuit board [26, 41, 70]. An overview of the topic of CBL is given by Abboud et al. [1]. Mathematical concepts are presented for solving the two major issues *component placement* and *wire routing*. Interestingly, this distinction is also similar to how the hereafter presented layouting process for the Peer Model is structured. While the available space when performing CBL is often predetermined due to standard sizes and constraints, the layouting process for the Peer Model for the purpose of the monitoring tool theoretically has an unlimited amount of space to work with. However, as one of the previously defined criteria states, it is preferable to keep the layout's size as small as possible. The crossing of conducting paths in a circuit board is not allowed and must not exist in the layout. On the other hand, edge crossings in the Peer Model layout are, while discouraged, possible and often unavoidable. An important feature of Peer Model layouting and difference to CBL is the Peer Model's recursive nature. Peers and wirings may be nested inside parent peers, potentially multiple layers deep. The layouting process must therefore be able to recursively layout sub peers, as discussed in section 6.4.
In order to justify the motivation and necessity for a proper layouting algorithm, consider figure 6.1. The same Peer Model is layouted in two different ways. In subfigure I, the elements were placed randomly within the parent peer and then connected in a random order without considering edge crossings or bends. The only enforced rule was to avoid having links move over elements. Otherwise, the layout would be even less readable. Subfigure II is the result of the layouting process developed during the thesis. This layout is arguably much easier to read and understand than subfigure I. It features much less edge crossings and bends and lays out the elements in a more pleasant way. Also, it has a more consistent flow direction. While the naive layout has a lot of links going up, down and backwards, most of the links in subfigure II move to the right, promoting a natural *reading direction*. Lastly, it uses the space more efficiently and is therefore smaller.
This chapter addresses the issues and details of the layouting algorithms used within the monitoring software. A detailed discussion on the applied algorithms and techniques for layouting Peer Models follows.

6.1 Node Layouting / Sugiyama Framework

Each peer belongs to a processor and within, each peer is layouted separately. The sub peers and wirings along with their links are subject to the layouting. The implemented process for a peer is comprised of three distinct steps. In the first step, a variation of the *Sugiyama Framework* [83] is employed to roughly position the sub peers and wirings

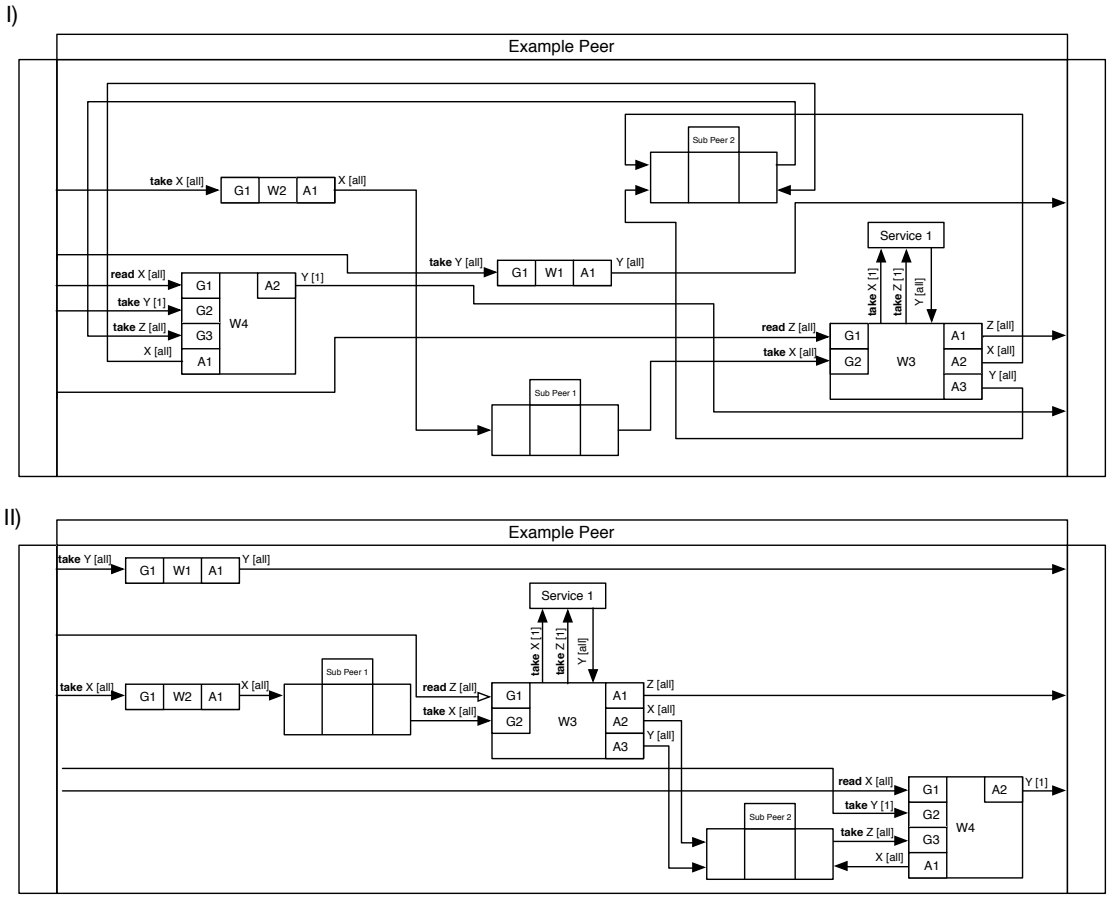


Figure 6.1: Comparison of two layouting results. Subfigure I shows randomly placed elements, after which a naive orthogonal edge routing was performed to place the links. In contrast, subfigure II is the result of the layouting process developed during this thesis. Objectively, II is the much better layout as it fares better in all previously established criteria for graph layouts. Also note that even in the naive approach resulting in I, the link routing is not trivially implemented as it avoids cutting through elements and tries to minimise path lengths at the very least.

inside the peer. As the framework only works on simple, directed graphs, the peer that is being layouted needs to be transformed. For each sub peer and wiring, a graph node is created. Every wiring link is modelled as an edge between two nodes. Guard link edges go from the container to the wiring, action link edges start at the wiring and go to their respective container. PIC and POC of the surrounding peer itself are also added as a graph node each.
It is possible for wirings to have no guards and sub peers to have no link connected to

them, which can lead to multiple roots in the resulting graph (a root is defined as a node that has only outgoing edges). To make sure that the PIC container is the single root and is positioned as the left-most element by the Sugiyama framework, temporary edges are added that connect the PIC's graph node to these *faux roots*.
By way of example, figure 6.2 shows how a Peer Model is transformed into such a directed graph. The resulting graph can be fed into the Sugiyama framework. The frame-

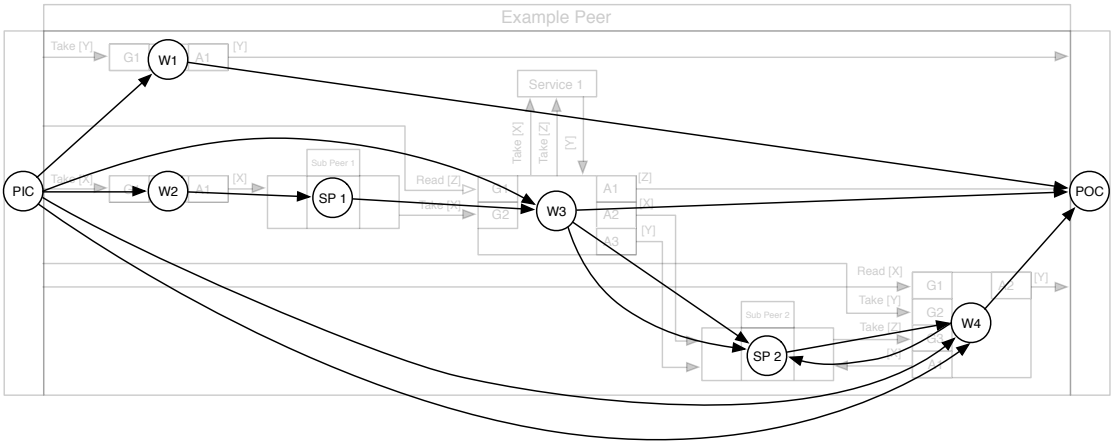


Figure 6.2: Generating the input graph for the Sugiyama framework. Note that in this example, the Peer Models are already shown fully layouted to improve the illustration.

work is actually a collection of several algorithms run in succession, including cycle removal, distribution of nodes into layers, insertion of dummy nodes and permutation of nodes inside layers to reduce edge crossings. After that, the elements are properly sized and placed inside the surrounding peer. The following section describes each step of the Sugiyama framework and also highlights the characteristics of this implementation.

Cycle Removal

Potential cycles in the graph need to be removed as the subsequent parts of the Sugiyama algorithm only work with acyclic graphs, or more specifically, *Directed Acyclic Graphs (DAG)* . A graph is considered acyclic if it contains no cycles [84]. There are different approaches on how to remove cycles from a graph while retaining as many edges as possible. One is referred to as the *Minimum Feedback Arc Set* [22]. It is described as finding the minimum number of edges in the graph that need to be removed in order to yield an acyclic graph. The problem is proven to be NP-hard for general, directed graphs [44, 47].
For the purpose of this software, the input graphs are small. Peers generally contain

only a handful of sub peers and wirings. Furthermore, it is not strictly necessary to find the optimal solution. A suboptimal, yet reasonable solution often results in the same layout as the optimal solution. For these reasons, a simple algorithm for cycle removal is applied. The greedy algorithm introduced by Eades et al. [22] is implemented. The algorithm is not optimal but offers a fairly good heuristic and has a lower computational complexity than exact algorithms. Figure 6.3 shows the algorithm in action on a small graph. It builds on the observation that source and sink nodes can not be part of a cycle. An explanation of the algorithm follows.

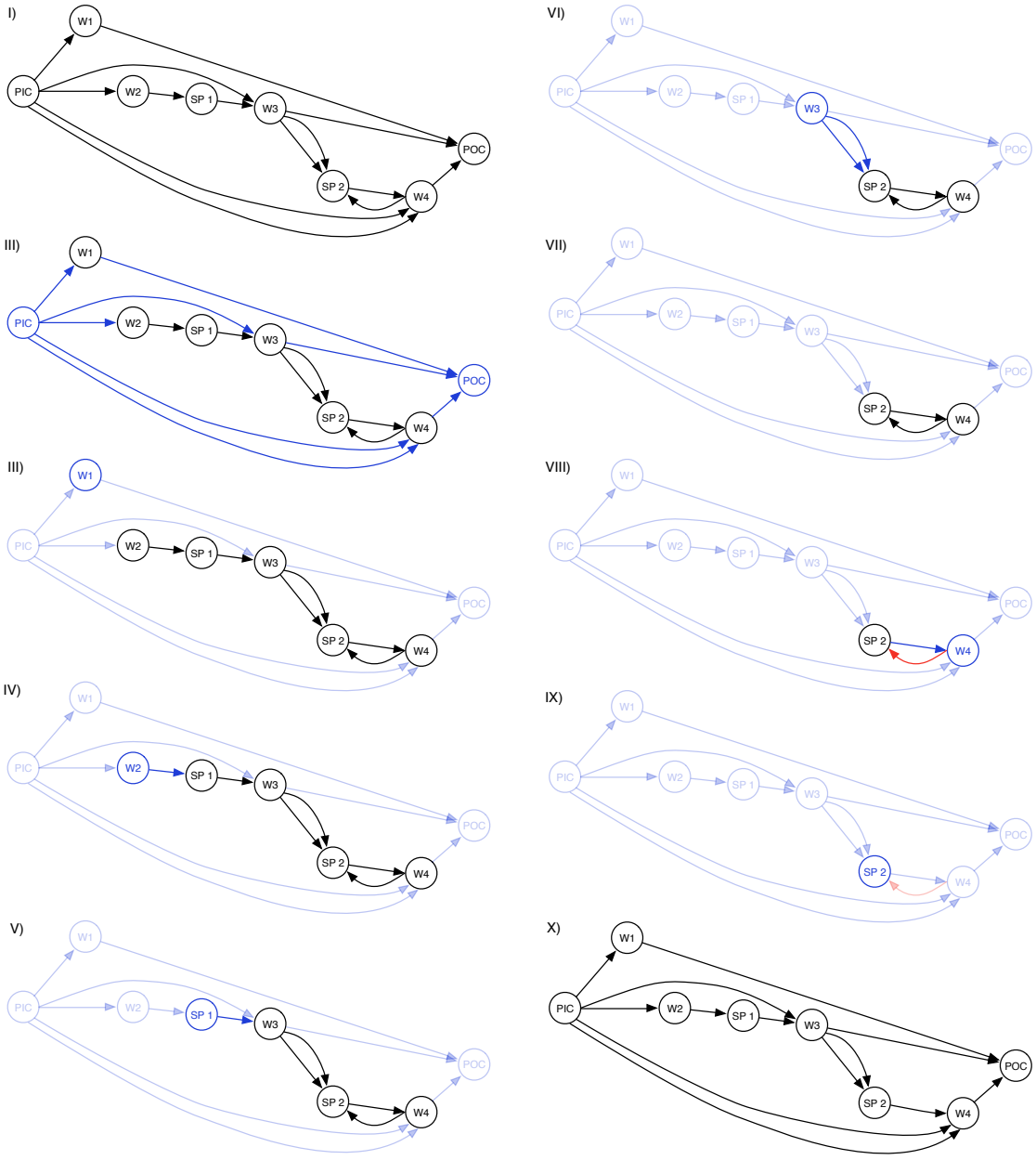


Figure 6.3: Performing the cycle removal algorithm by Eades et al. on the example graph. I shows the initial graph containing a cycle. Consecutively, sources, sinks and isolated nodes are removed from the input graph along with their edges and added to separate sets (shown in blue). The process continues until, at VII, no more sources, sinks or isolated nodes exist in the graph. Therefore, at VIII, a node is chosen and its outgoing edge is removed (shown in red). The algorithm continues until the graph is empty. The retained vertex and edge sets represent the final acyclic graph (X).

Let $G = \{V, E\}$ denote the input graph, where V is the vertex set and E the edge set. Furthermore, let $e^+(v)$ denote the set of outgoing edges of a vertex $v \in V$ and $e^-(v)$ the set of incoming edges. Finally, define the *out-degree* of a vertex as $d^+(v) = |e^+(v)|$ and the *in-degree* as $d^+(v) = |e^-(v)|$.
v is a source node if $d^-(v) = 0$ and a sink node if $d^+(v) = 0$. At each step, the algorithm greedily removes all sources and sinks from the graph. These nodes, along with their in- and outgoing edges are added to the sets K^v and K^e, respectively. Fully isolated vertices, where $d^+(v) = d^-(v) = 0$ are also removed from the graph and added to K^v. After that, the node x where $d^+(x) - d^-(x) = max$ is found. $e^+(x)$ is added to K^e, whereas $e^-(x)$ is simply removed from the graph. The iteration continues until G is empty. The resulting graph $G^n = \{K^v, K^e\}$ is guaranteed to be acyclic. For an in-depth analysis of the algorithm, refer to [22].

Node Layering

The graph is now guaranteed to be acyclic and connected. In this step of the Sugiyama framework, each node in the graph is assigned a layer. For the purpose of this software, the graph is best visualised with the node corresponding to the PIC on the left and its children spreading to the right. Using this representation, the graph's nodes can be laid out on distinct, horizontally placed layers / columns. Let $L(x)$ denote the layer of node x and $p(x) = \{v \in V | (v, x) \in E\}$ the set of parent nodes of x. The layer is decided as $L(x) = \max_{y \in p(x)}(L(y)) + 1$, except for the root node, where $L(x) = 0$. In other words, the layer of a node is the maximum layer of its parent nodes plus 1. A node is defined as a parent node p of node x if there exists an edge going from p to x. This recursive assignment makes it clear why the graph must not contain any cycles as that would lead to infinite loops. Figure 6.4 shows the graph after the layer assignment step.

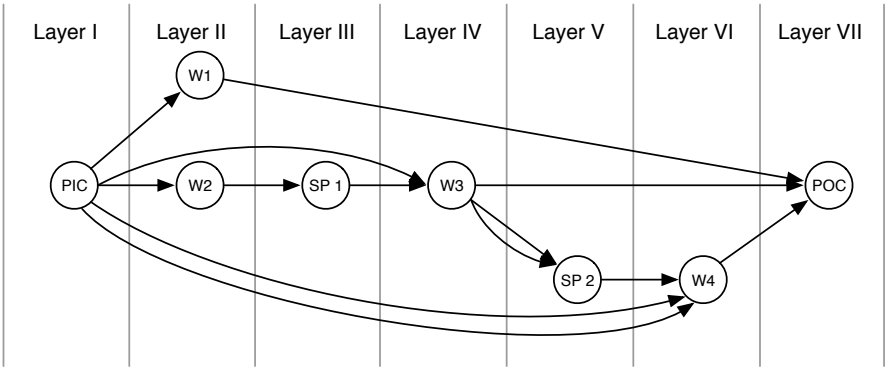


Figure 6.4: Example graph after the node layering is completed. Note how the PIC and POC are assigned to the first and last layer exclusively.

Dummy Node Insertion

The last step in the Sugiyama framework, which is responsible for minimising the number of edge crossings can only work when all edges only connect nodes on neighbouring layers. As can be seen in figure 6.4, this is generally not the case after the node layering step. Thus, temporary dummy nodes need to be introduced into the graph. Each edge that connects distant nodes is replaced by a sequence of temporary nodes and edges that form a chain. The number of nodes in a chain is set to the difference in layers between start and end node minus 1. Figure 6.5 shows the result of that process for the example graph.

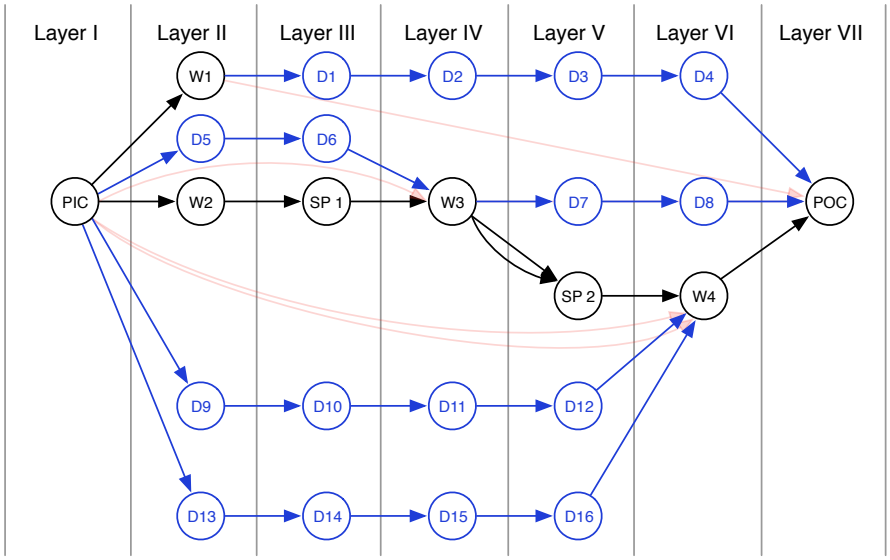


Figure 6.5: Insertion of chains of dummy nodes into the graph. Every edge connecting distant nodes is replaced by such a chain. After this step, edges in the graph only connect nodes on adjacent layers.

Edge Crossing Minimisation

While a node's horizontal position is determined by its layer, its vertical positioning was not discussed until now. Until this point, all figures showed the nodes already placed in vertical positions that led to no edge crossings. However, this is not guaranteed during the actual algorithm. Furthermore, some graph nodes correspond to wirings in the Peer Model. For wirings, the vertical order of in- and outgoing edges matters. While the order of edges does not lead to a different number of crossings in the graph itself, the resulting Peer Model layout suffers from suboptimal ordering. Consider figure 6.6 for an example how a different order in the graph leads to different results in the layout.

In order to minimise the number of edge crossings, two passes of node reordering take

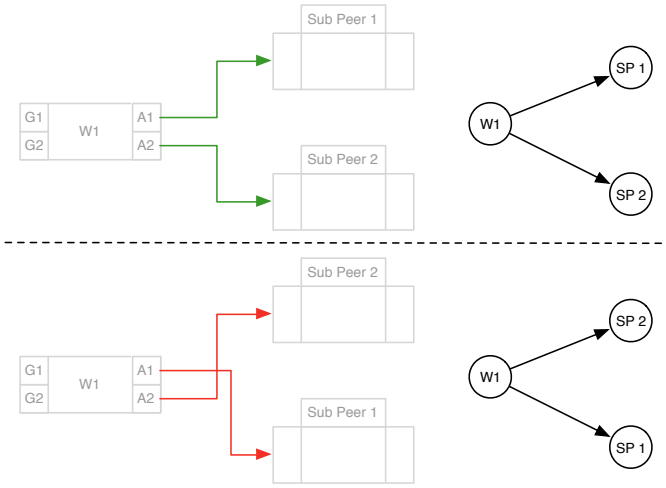


Figure 6.6: Two variations of a simple node graph and the resulting layouts. While the graphs themselves exhibit no crossings in either case, II represents a suboptimal layout as the links have to be routed in a way that creates a crossing. The node placement of variation I is preferred.

place. In both passes crossings are minimised layer by layer. In the first pass, layers are processed left to right, in the second right to left. In a layer sub-step, the current layer itself stays fixed, while the nodes in the next layer are permuted in order to find the best solution. As the graphs are relatively small, it is feasible to look at all permutations of node orderings. For each ordering, the number of edge crossings is calculated and the permutation with the smallest number is chosen. As stated before, it is not enough to only look at the graph's edge crossings. When wirings are involved, the ordering of the edges plays a crucial role. To introduce an order, the outgoing edges of each node are indexed. Edges with lower indices are considered to start at a higher vertical position in the corresponding Peer Model layout and vice versa. Figure 6.7 illustrates a typical problem in the edge crossing minimisation step.

The process to determine how many edge crossings a certain permutation would produce is calculated according to algorithm 6.1. The algorithm looks at each outgoing edge in a layer in succession. It inspects nodes that have a higher vertical index than the current edge's source node, but have edges targeting a node with a lower index than the current edge's target. Such a case corresponds to an edge crossing. For nodes representing wirings, the edges below the current one in the same node also need to be considered and checked for potential crossings.

When all permutations for a layer are explored, the one with the minimum number of crossings is chosen and the graph is updated accordingly. Then, the next layer is pro-

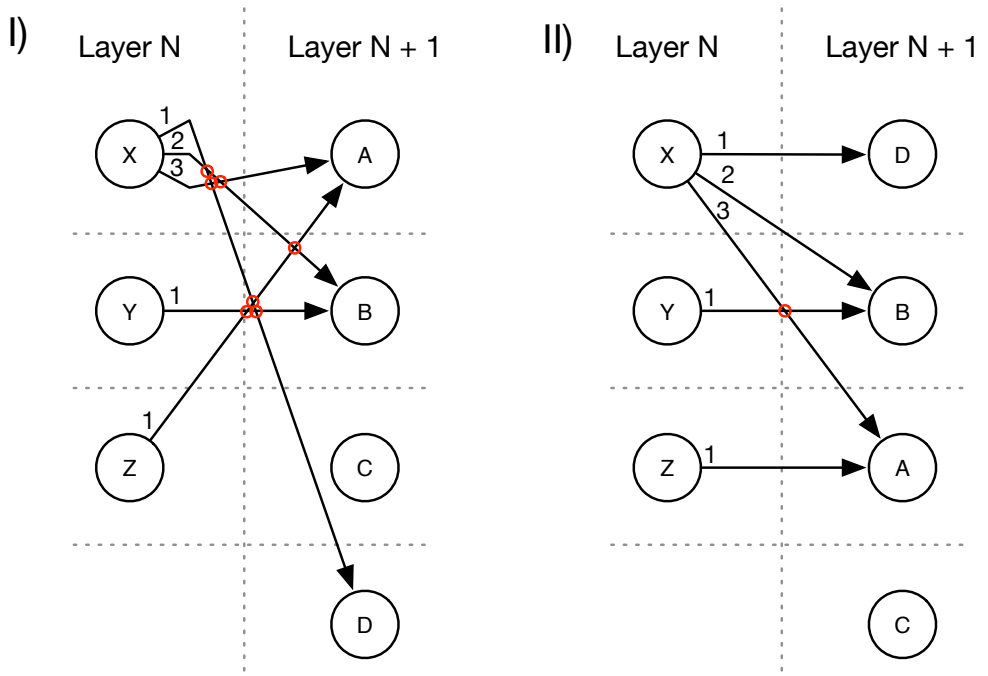


Figure 6.7: Typical setup in the edge crossing minimisation step. Layer N is considered fixed, while layer $N + 1$ is permuted. Subfigure I shows a suboptimal permutation that results in seven edge crossings (marked red). Subfigure II on the other hand features an improved permutation of the nodes in layer $N + 1$ and has only one crossing. Note that the outgoing edges of nodes in layer N are ordered (particularly node X) and how this affects the results.

cessed. If a permutation without crossings is encountered, the loop is exited early as this represents an optimal solution and further exploration of the problem space is redundant.
After the left-to-right pass has finished, the right-to-left pass is executed. In this pass, the algorithm looks at incoming edges rather than outgoing ones and permutes the nodes in the previous layer. Looking at figure 6.7, layer $N + 1$ would be the fixed layer while layer N is permuted. Once the pass has reached layer 1, the edge crossing minimisation step is complete.

Element Sizing

A two dimensional grid is established along which the elements are aligned as this makes exact placement easier. The grid also helps with the later layouting steps, especially edge routing. The smaller the grid's cells, the more freely the elements can be

```
1  Function calculateEdgeCrossings(L)
2      num ← 0;
3      for i ← 0 to L.size do
4          for j ← 0 to L[i].outgoing.size do
               // store the index of the current edge's
                  target
5              targetIndex ← L[i].outgoing[j].node.index;
6              if L[i].correspondsToWiring then
                   // current node is actually a wiring, so the
                      order of the links matters and subsequent
                      links of this node are checked for
                      crossings
7                  for k ← j + 1 to L[i].outgoing.size do
8                      if L[i].outgoing[k].node.index < targetIndex then
9                          num ← num + 1 ; // found a crossing
10                     end
11                 end
12             end
               // check edges of subsequent nodes for
                  crossings
13             for k ← i + 1 to L.size do
14                 for l ← 0 to L[k].outgoing.size do
15                     if L[k].outgoing[l].node.index < targetIndex then
16                         num ← num + 1 ; // found a crossing
17                     end
18                 end
19             end
20         end
21     end
22     return num;
23 end
```

Algorithm 6.1: Calculating the number of edge crossings for a layer permutation.

placed. Elements can also be manually repositioned by the user after the initial layouting (see section 7.4 for a detailed discussion on this topic). The grid is respected in this operation as well and elements are snapped to the grid in all situations. Hence, having a fine-meshed grid also gives the user more freedom in his/her layouting choices. On the other hand, a coarse grid makes it easier to align elements with each other. A smaller

cell size negatively affects the performance of the layouting algorithm. Both the edge routing and packing stages rely on the grid and take longer when using a finer-grained mesh. For all the presented reasons, a cell size of 15 pixels was chosen as it represents a good compromise. However, the layouting framework is implemented to work with other cell sizes as well, should the need arise.
Before the elements can be placed, their exact dimensions need do be determined. This sizing process happens in grid units. As described in chapter 3, (sub) peers can be nested inside each other. While working with the monitoring tool, the user can choose if he/she wants to view a sub peer in full detail (*expanded*) or wants a compact, simplified representation that saves screen space (*collapsed*). Therefore, two representations of a sub peer exist, depending on the user's choice. Figure 6.22 shows the difference in visualisation of a peer, first expanded, then collapsed. An expanded sub peer is layouted by applying the layouting process in a recursive fashion. Its size is therefore dependent on the results of the layouting procedure responsible for this sub peer. For a detailed discussion on recursive layouting of sub peers, refer to section 6.4. On the other hand, a collapsed sub peer is layouted in a more simple manner. Its inner workings are hidden from the user, which is another example of the *Visual Information-Seeking Mantra* [79]. A collapsed peer should be visualised as small as possible while still offering the most important information like the peer's name as well as its containers. As wiring links can be connected to the PIC and POC of a peer, it should be sized vertically so that each link can connect without overlapping others. Figure 6.8 shows a typical collapsed peer and its dimensions.
A wiring's height is determined by the number of guards and actions and by the max-

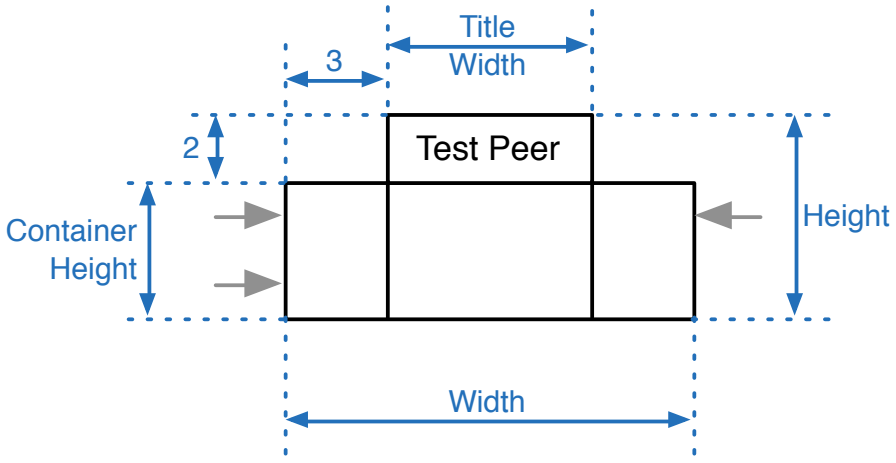


Figure 6.8: Dimensions of a collapsed peer. The container height is dependent on the number of incoming links while the title width is set in accordance with the peer's label width.

imum height of its services. Guard links are always on the left side of the wiring, even in cases when the guard link targets containers to the right, such as the POC of the parent peer. On the other hand, action links can be positioned on either side, depending on where their target container is within the layout. As the coarse positioning is known from the previous steps, the action link's side can be deferred from the target container's associated layer. If the target's layer is smaller than the wiring's layer itself, the action link is positioned on the left side and vice versa. While this can lead to suboptimal layouts in theory, using this heuristic for determining the side of the action link has served well in practice.
A wiring's width is defined by its services, the link bases on either side and the label stating its name. Services are positioned in a way that their service links do not interfere with the guard and action link bases. Also, a gap of 1 unit between services is maintained for better readability. Figure 6.9 shows a typical wiring and its dimensions.
Each sub peer and wiring has a layer and an index assigned to it. Using this informa-

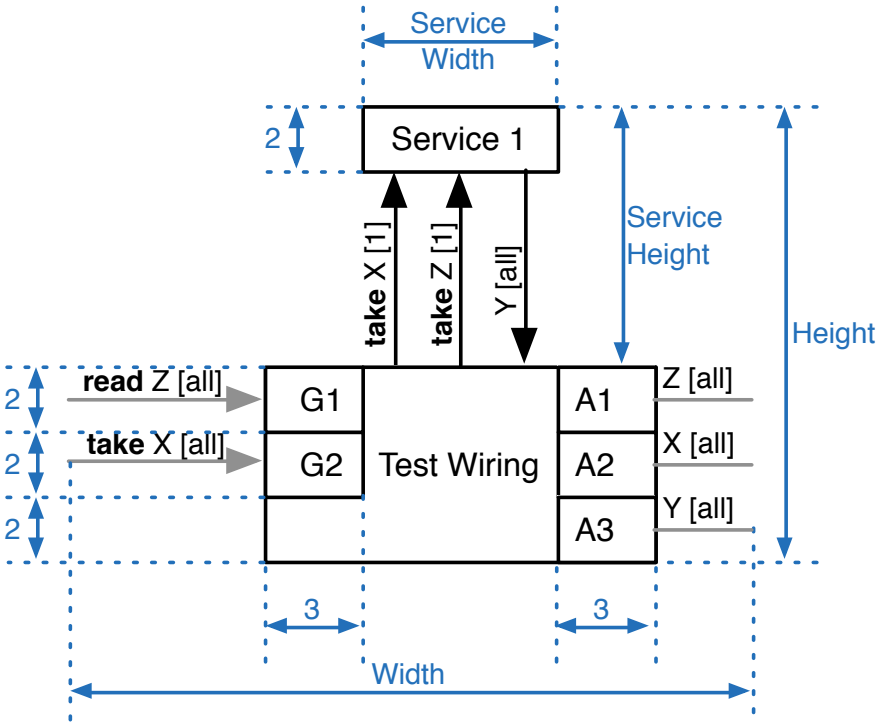


Figure 6.9: Typical wiring and its dimensions visualised. A wiring's width is dependent on the width of its services and the link bases of its guard and action links. Its height is deferred from the maximum height of its services and the number of link bases on either side.

tion along with their sizes, the elements are placed inside the surrounding peer. To find

the exact positions on the grid, the maximum width for each layer and the maximum height for each index is determined from the element's sizes. This spans a coarse grid, with each cell containing at most one element. Additionally, gaps are inserted between cells both horizontally and vertically. To facilitate better edge routing, the elements of horizontally neighbouring cells are vertically aligned so that their areas where edges connect start at the same vertical position. Consider figure 6.10 for an example on how this helps simplifying the overall layout. Horizontally, the elements are left aligned within their cell.
It is important to leave enough space between neighbouring cell columns so that the links can be routed in an efficient manner. A crucial observation is that a single link can only have at most one vertical edge in the area between two columns. This leads to the rationale that the number of links crossing between two columns is an upper bound for the needed space in order to allow proper routing. Employing this bound guarantees a reasonable setup for the subsequent link routing procedure. It should be noted that while this approach is wasteful in many circumstances, the packing step which will be discussed later is capable of remedying this issue. Figure 6.11 shows the example peer after the node layouting step.

6.2 Link Routing

The second step in the layouting process is link routing. While the wirings and sub peers themselves have been placed, wiring links were mostly ignored until now. Whereas guard and action links demand proper routing, services are always placed on top of their associated wiring. Hence their links require no routing and are simply connected in a straight vertical line. The subsequent discussion is therefore solely concerned with routing action and guard links.
For better readability of the peer, only orthogonal edges are allowed. This means that edges can only be either vertical or horizontal, but not oblique. Orthogonal edges give a more tidy look, while graphs with slanted edges can look disorganised.
The grid established in the previous step is used again to route all guard and action links. Edges can only move along grid points. This both simplifies the routing process and results in a consistent look. As the elements themselves are also aligned, the start and end points of links always lie on the grid. Links are always routed from a wiring to a (sub) peer's container. This seems counter-intuitive at first because guard links are actually depicted as arrows moving from a container to the wiring. However, during layouting, it is beneficial to treat them as a path going in the other direction. This makes it possible to treat guard links the same as action links. While all links have a single starting point at their respective wiring, their target is generally not a single point. A link can end at a container at one of multiple places and the position that results in the best path needs to be found in the course of the link routing process.

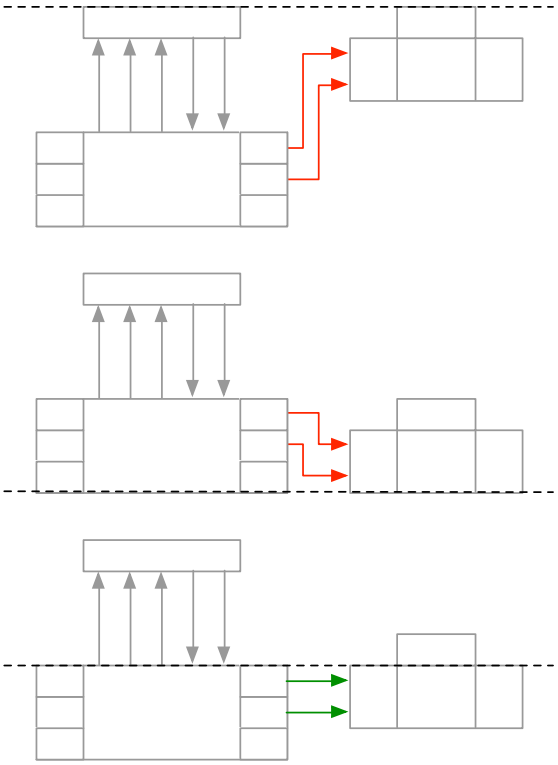


Figure 6.10: Example how proper vertical alignment of neighbouring elements helps creating shorter links. As the upper two subfigures show, aligning the elements at their top or bottom often leads to suboptimal links. Aligning them along their first common link produces better results. Note that the link labels are omitted for clarity.

Links are routed one after the other, but make use of the information from previously routed links. Because of this, the order in which this happens greatly affects the outcome of the process. As a general heuristic, shorter links are routed first. For each link, the *Manhattan Distance* between the start and each potential target is calculated and the shortest one is chosen. The manhattan distance between two points is calculated as

$$d(a,b) = abs(a_x - b_x) + abs(a_y - b_y)$$

This constitutes a heuristic for the length of the final path. Routing shorter links first intuitively makes sense, because these are mostly those that connect elements from neighbouring layers, possibly even on the same vertical index. If long links were to be routed first, they could potentially cut through the short links' ideal paths, possibly forcing them to find different, much longer ones. At the same time, links with long paths generally do not benefit as much from this prioritisation. It should be noted that this heuristic does not always lead to an optimal solution but produces fairly good results in virtually

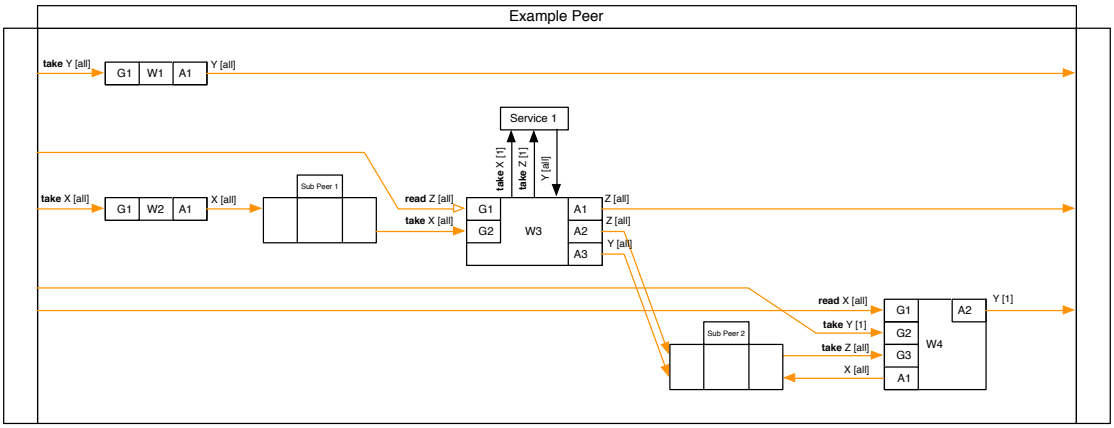


Figure 6.11: Layout after the node layouting step for the example peer. Note that the links, with exception of the service links, are not yet routed (depicted in orange). Proper routing is the goal of the next step in the layouting process.

all circumstances.

While the approach works well in most use cases, there is a prominent edge case in which it leads to suboptimal layouts. If two or more links share the same start wiring and target peer container, applying the just discussed metric often leads to unnecessary crossings. Refer to figure 6.12 for an in-depth explanation of this issue. As this use case is very prevalent in typical Peer Models, an exception to the rule is applied: if two or more links share the same start wiring and target container, their relative order of routing is reversed.

To route a path, the *A* pathfinding* algorithm [17, 56, 75] is applied. In its primal form, A* finds the shortest path between two nodes in a directed graph. The graph also has to have a metric, which is established by giving each edge a weight that represents the cost associated with traveling along that edge. Furthermore, all weights need to be positive. The grid used in the previous steps serves as the basis for the pathfinding graph. As one of the goals when routing a link is to minimise the number of corners, this is ensured by setting up the graph in the following manner: for each point in the original grid, four graph nodes are created. While they all occupy the same space, they also encode directional information. Each of the four nodes represents the current direction (up, down, left or right). Each graph node has at most three outgoing edges, which correspond to the movements of turning left, turning right and continuing in the same direction. The two edges that represent a turn are given a higher cost, which enables the A* algorithm to avoid unnecessary turns. Figure 6.13 shows a cutout of the resulting graph. For a detailed explanation of the A* algorithm itself, including performance and optimality, refer to [17, 56, 75].

For the algorithm to create sensible paths, obstacles are inserted into the graph to mark

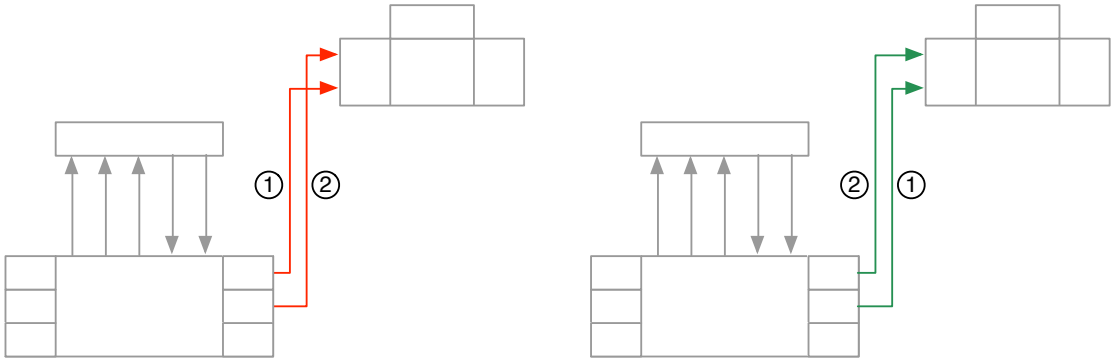


Figure 6.12: Example case where the rule of routing shorter paths first leads to suboptimal layouts. In the left case, the shorter (as calculated by the heuristic) path is routed first. As it tries to find the shortest path, it connects to the peer at the lowest possible position. However, this requires that the subsequent link crosses the first in order to connect to the peer as well. In the right case, the order of routing is reversed which leads to a better layout. Note that the link labels are omitted for clarity.

areas that should be avoided. Because it is crucial that the links are fully routed in all cases, rather than completely disallowing moving through such areas, a high movement cost for the associated edges is applied instead. This makes sure that the algorithm avoids such regions, but still cuts through them if absolutely necessary.
Obstacles are inserted at wiring and sub peer locations into the pathfinding graph. While sub peers are simply represented by a rectangular area, obstacles for wirings are a bit more complex. If a wiring has services, the areas to the right and left of the service block are not penalised to allow paths to freely move through. Each link going out from a wiring also has a link label attached to it. The label is placed right next to the start of the link's path. It is beneficial if the links themselves fan out horizontally until they are past the link labels. Links cutting through labels impair visual quality and readability of the label's text. However, in some circumstances it is still necessary to cut through labels to avoid even worse layout deficiencies. Because of that, edges moving out from the guard link and action link bases in a horizontal fashion are not penalised. That encourages the A* algorithm to route the link horizontally until it is past the labels. Figure 6.14 shows the path grid after obstacle placement for a wiring. One goal while routing the links is to avoid edges crossing over other edges. While the node placement step does its best to minimise crossings, many peer configurations are simply not *planar* and therefore can not be routed without crossings. The term planarity refers to a property of graphs. A graph is considered planar if it can be drawn in a two-dimensional space without any edges crossing [92]. The same observation can be applied to Peer Model layouts. While edge crossings should be avoided, multiple edges occupying the same space, effectively overlapping one another are even worse. Overlapping edges impair

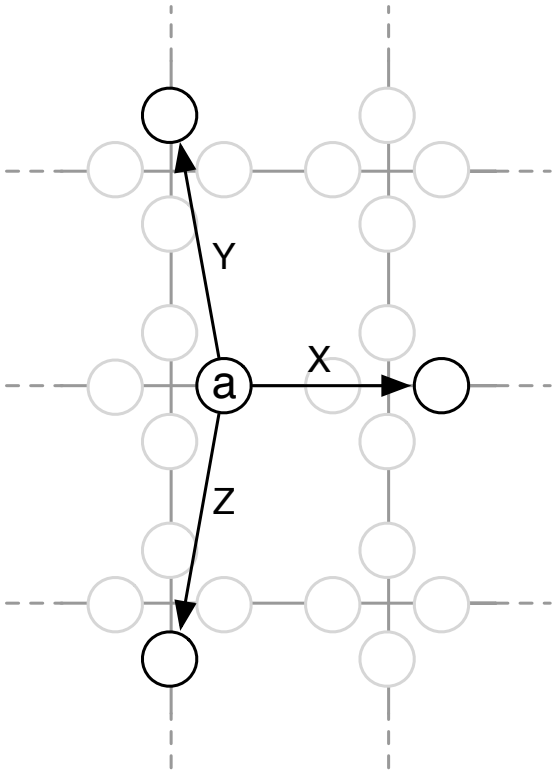


Figure 6.13: Small cutout of a pathfinding graph. If at a, X represents the edge that is taken when continuing moving to the right. Taking Y means turning left and going up, Z turning right and going down. Y and Z are given a higher cost than X to penalise turning. For readability, only edges emanating from a are shown.

readability a lot and make following paths confusing and error-prone for the user.
Both issues need to be considered during the link routing step. After a link is fully placed, the edges it moves along are penalised as high as obstacles to ensure that links routed afterwards avoid using them again. Furthermore, edges that are perpendicular to used edges are also penalised, even though not as highly. Thus, later links are preferably routed around rather than cutting through already placed links. The corner penalty and the crossing penalty are closely related and care has to be exercised when deciding on these numbers. Table 6.1 shows the movement costs and penalties used. Note how a parallel path is penalised as much as edges inside an obstacle.

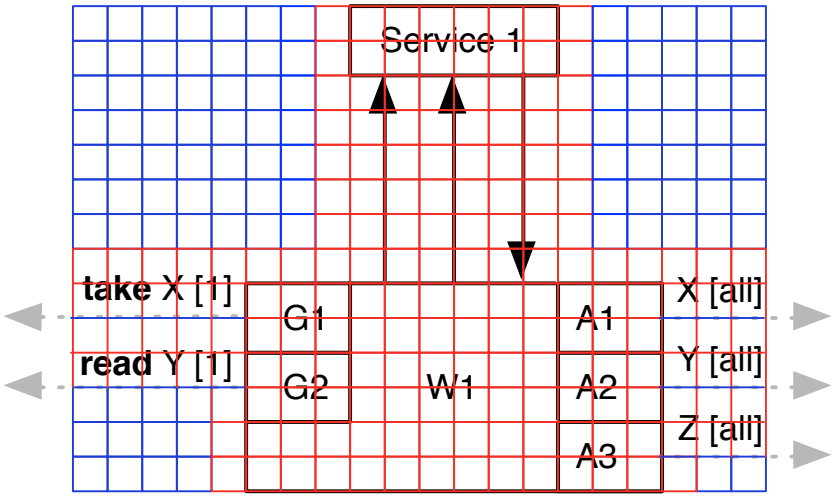


Figure 6.14: Obstacle placement for a wiring before pathfinding. Red denotes highly penalised edges. Blue denotes unmodified edges. The areas to the left and right of the service are freely passable. Note how the areas around labels are penalised, while the edges where the links should ideally go are not (blue). The last link on either side is also allowed to move downwards as there are no labels blocking the way. Also note that turning edges are omitted from the pathfinding graph in this figure to improve readability.

	Cost
Default	1
Corner penalty	3
Crossing path penalty	9
Parallel path penalty	20
Obstacle penalty	20

Table 6.1: Movement costs and penalties for the link routing step. Note that the penalties are cumulative to ensure that even inside obstacles, paths avoid crossings and corners. The values have been determined heuristically in the course of development.

While the start at the wiring is a single node in the pathfinding grid, the target (a peer container) generally consists of an area that spans multiple locations on the grid. The A* implementation is therefore adjusted to work with multiple targets for a single path. The heuristic evaluated at each step in the algorithm is calculated for each target node and the smallest one is considered the current target. Also, the check whether or not the algorithm has terminated is extended to check for all targets instead of only one. Besides the location, the path's direction must also match the target ones to be

considered equal. As stated earlier, the direction is encoded into the grid as well. For the surrounding peer, the PIC must be approached from the right while the POC must be approached from the left. For sub peers, the directions are reversed. Figure 6.15 shows the exemplary peer after the link routing stage.

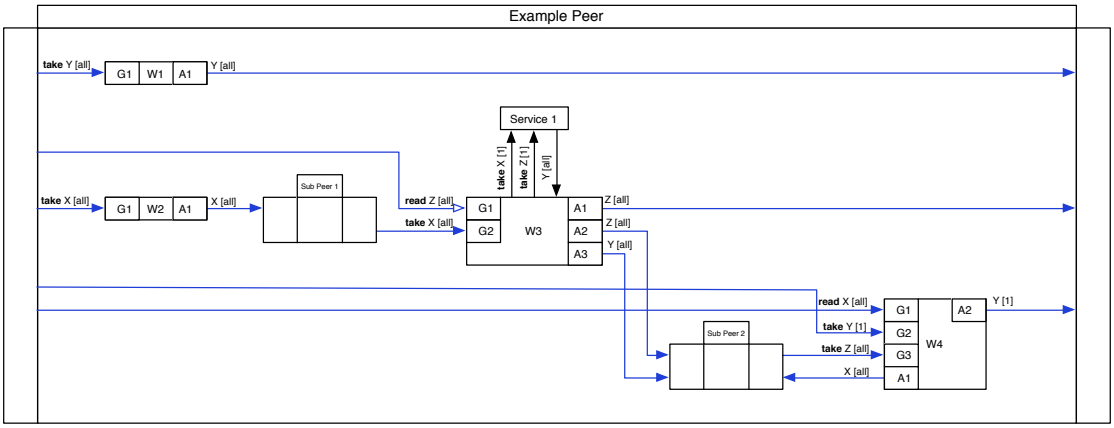


Figure 6.15: The example peer after the link routing stage. Guard and action links are properly routed (shown in blue).

6.3 Packing

Up until this point in the layouting process, minimising the size of the resulting visualisation was not a major concern. While the elements themselves are sized as efficiently as possible, there is a lot of empty space between them. However, as established in the beginning of this chapter, minimising the size of the visualised Peer Model is an important criteria. The *Packing* step in the layouting process aims to reduce the size of the current layout by moving the elements closer together without sacrificing legibility. By minimising the size of all peers, more information can fit onto the screen, reducing the need for scrolling and improving the user interaction.
Having enough space was vital for the link routing step to find efficient paths for the guard and action links of wirings. Therefore, as discussed in the subsection concerned with the placement of elements, the horizontal space between horizontally neighbouring elements was chosen heuristically to make sure the links can be routed in a sensible way. The chosen gap size is a general upper bound to the horizontal space required for proper routing in all cases, but is often times wasteful in terms of space. Additionally, in the *Node Layouting* step, the elements are placed on a coarse grid, whose cell width and height is determined by the widest and highest element respectively. This can also potentially waste a lot of space.

Packing is the process of moving the elements within a peer closer together. This includes the movement of sub peers and wirings along with their guard and action links. However, there are limits to how far these elements can be moved towards each other. The grid established in the previous steps is still of significance. In order to respect the grid, the packing algorithm only moves elements in grid units. All wirings and sub peers should have at least one grid cell of space between each other, so they do not touch or overlap. Similarly, they should not touch the boundaries of their enclosing parent peer either. Parallel links should also have at least one grid cell of space between them. The link routing stage specifically took measures to avoid overlapping links and a packing algorithm that is too aggressive would nullify that effort.
An interesting way to look at potential wasted space is to view it in terms of paths. If there is a path going from the top of the layout to the bottom without moving through an obstacle, there is room for improvement. If such a path exists, it cuts the existing layout into two sets, one on the left of the path and one on the right. The two sets can then be moved towards each other without violating any constraints imposed earlier, thereby reducing the width of the layout. The procedure can also be performed along the other dimension to reduce the size vertically. The problem of finding and eliminating unnecessary empty spaces is essentially transformed into a problem of pathfinding.
The packing procedure is split into two phases, one for each dimension. The first phase is concerned with eliminated unnecessary gaps in the horizontal dimension, the second in the vertical dimension. Apart from a few minor differences, the procedure is equal for both phases:

```
1  while true do
2  |  graph ← setupGraph ();
3  |  path ← findPath (graph);
4  |  if path.valid then
5  |  |  [leftSet, rightSet] ← createSet (path);
6  |  |  moveSet (rightSet);
7  |  else
8  |  |  break;
9  |  end
10 end
```

Algorithm 6.2: Packing Phase Overview

The presented algorithm tries to find a valid path through the layout. If it does, it splits the elements into two sets according to the path and moves the right set towards the left. It continues until it cannot find any valid path. A second packing phase for vertical gaps is run afterwards. A detailed discussion of each step in the packing phase follows. Note that the subsequently described steps refer to the horizontal pass. The steps for the vertical pass are essentially the same, but with swapped dimensions and directions.

Graph Setup

Just as in the pathfinding step, a graph grid is generated to run the pathfinding on. However, while the pathfinding procedure shows similarities to the link routing process, there are important differences. Firstly, this algorithm works on the *weak dual graph* of the original pathfinding graph used for link routing. The *weak dual* to a graph A is the graph that has a vertex corresponding to each bounded face of A, and an edge joining two neighboring faces for each edge in A [27]. Figure 6.16 shows the graph used for packing in blue laid on top of a small test layout.

Secondly, while the link routing algorithm only penalises edges, it still allows traversing them. Conversely, for the packing problem, obstacles are placed by removing certain edges from the graph, effectively prohibiting the traversal of these edges entirely. All traversable edges in the graph have the same movement cost. Thirdly, while the link routing graph contains four nodes per location to account for turning (see figure 6.13), the packing graph only contains one per spatial position. It is to be defined what counts as an obstacle within the graph for the purpose of the packing phase:

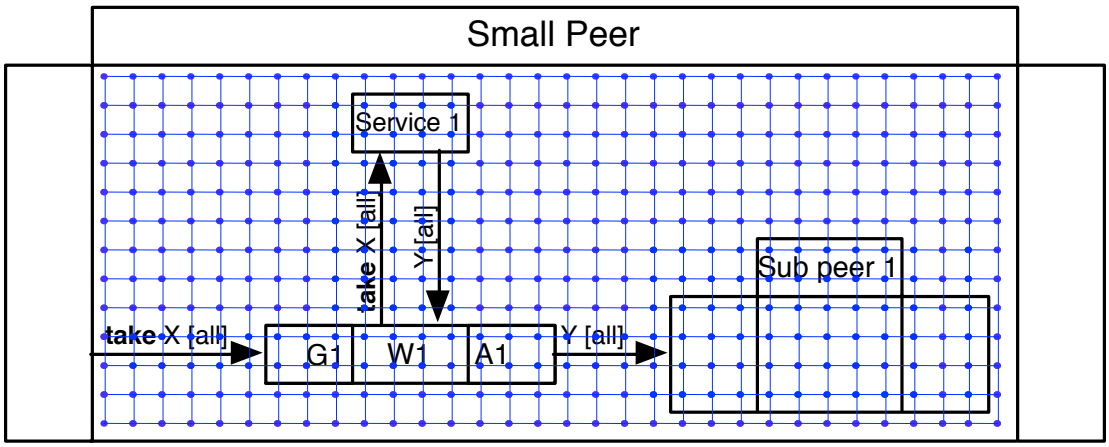


Figure 6.16: Visualisation of the dual pathfinding graph laid on top of a small peer layout. Note how the wiring links move shifted in between the nodes instead of along them.

Sub peers and wirings

Areas where sub peers and wirings are placed are fully impassable. For wirings, this also includes service call blocks along with their service links. Furthermore, the first part of each wiring link (where the link's label is located) is not crossable either.

Vertical link edges

In the case of the horizontal phase, only horizontal guard link and action link

edges may be crossed. Vertical edges of links are not crossable. This behaviour is inverted for the vertical phase.

Left- and rightmost columns
The left- and rightmost columns of nodes in the graph along with their edges are made impassable. This ensures that the pathfinding does not find paths that lead to one of the two resulting sets being empty.

For the purpose of explaining the packing algorithm, a small peer layout is used for visualising the performed actions. Figure 6.17 shows this layout and its packing graph with obstacles placed.

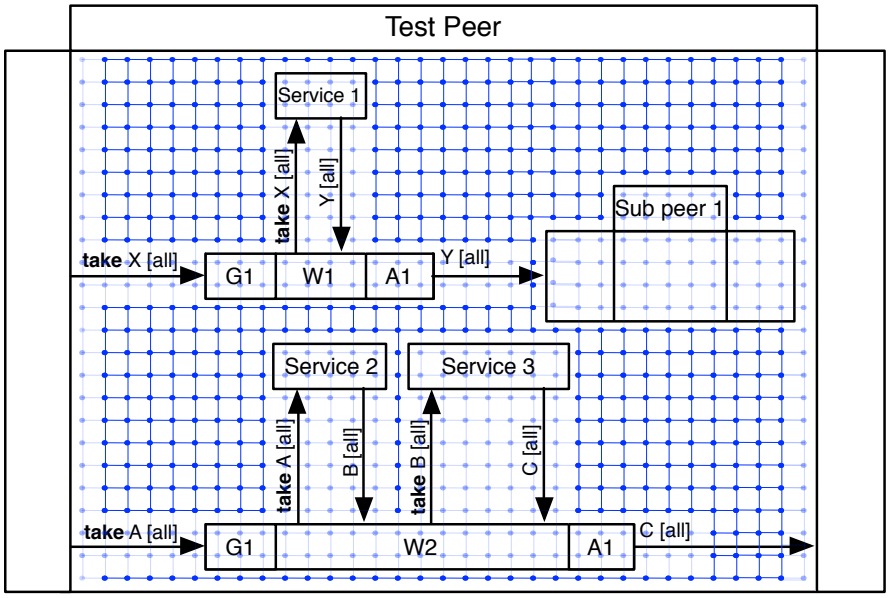


Figure 6.17: Small example peer layout with its corresponding packing graph displayed on top. Note how the areas where sub peers and wirings are placed are explicitly made impassable in the graph.

Pathfinding

The next step in the packing phase is pathfinding. Informally speaking, if a path from the top of the graph to the bottom can be found, the layout can be packed more tightly horizontally. It should be noted that in the horizontal phase, two additional graph nodes have to be inserted at the top and bottom of the graph. They are used as the start and goal of the pathfinding algorithm, respectively. Edges connect the start to the first row of nodes while the bottom row is connected to the goal. In the presented figures, these

nodes are omitted for the sake of better readability. The pathfinding itself is performed by a variation of the previously discussed *A* algorithm*. All traversable edges are given the same weight and the euclidean distance is used as the heuristic.
Figure 6.18 shows a valid path through the test layout. While the wirings and sub peers themselves are not passable, horizontal links can be crossed vertically by the path. Note that the first and last edge of the found path are removed again for the next packing step. If no path can be found, the layout is considered optimally packed with regards to the respective dimension and the step ends.

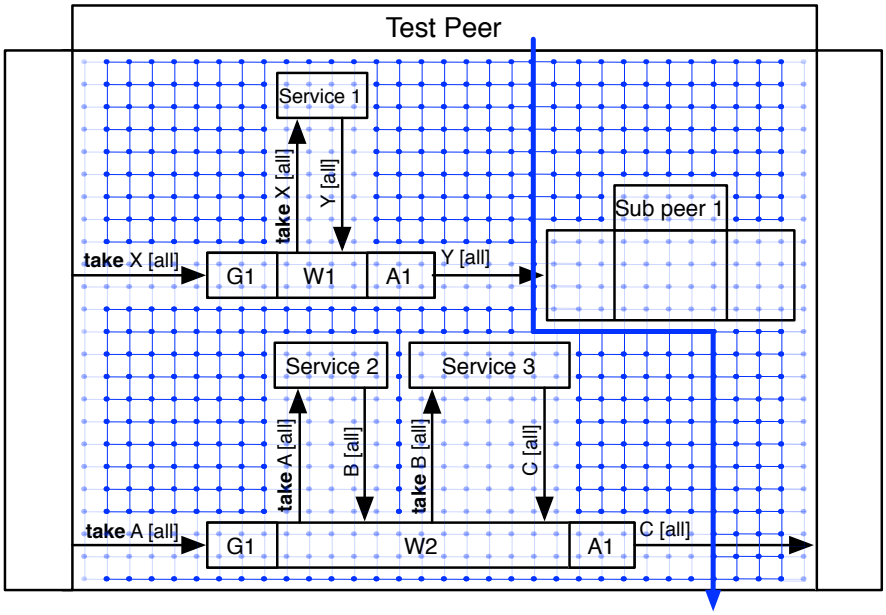


Figure 6.18: Packing graph with a valid top-to-bottom path.

Set Creation

A successful path geometrically splits the layout into two distinct areas. Each sub peer and each wiring are on either side of the path and therefore belong to the left or right area respectively. Links that are not crossed by the found path can also be assigned to one area exclusively. For links that cross the path, a more elaborate process is necessary.
For the purpose of the set creation, guard and action links are split into their control points. Sub peers and wirings are reduced to a single point corresponding to their top left position. This results in an input set S of 2D positions, each being a representative of a sub peer, a wiring or a control point of a link. The elements of S are then divided

into two output sets, L and R. By setting up the pathfinding graph as done in the previous subsections, both L and R are guaranteed to be non-empty if a path has been found. As the used pathfinding grid is the dual graph of the grid used for element placement, a point in S can always be exclusively attributed to either L or R. Mathematically, $\{L, R\}$ is a partition of S.
To determine which point in S belongs into which output set, a polygon P_L is constructed, corresponding to the area to the left of the path. After the elements of L are determined, elements of S belonging to the right set R can afterwards be found by simply setting $R = S \setminus L$. P_L is made up of the vertices of the splitting path, followed by two additional vertices, one located in the far top left of the layout, one in the far bottom left.
For each point $P \in S$, a check whether or not the point lies within the polygon P_L is performed. This is known as the *Point-In-Polygon Problem* (PIP). P_L in general can be concave but is guaranteed to have only straight edges. A found splitting path never self intersects, Therefore, the resulting polygon is also never self intersecting. While it is possible that subsequent edges of the polygon are *collinear*, this can safely be disregarded. The *Even-Odd* rule is used to perform the PIP test. It works by tracing a ray from the point in question into any direction and recording the number of polygon edges it hits on its path. If the number of hit edges is odd, the point is determined to be inside the polygon. If the number is even, the point is considered outside. For a more detailed discussion of the PIP problem, refer to [34]. Figure 6.19 shows the advancement in the packing phase from figure 6.18. The polygon P_L is depicted in green, with its vertices and edges in blue. The green points correspond to the elements of L, while the red points represent the elements of R. Note how P_L is defined big enough to also contain the leftmost points of S.

Set Movement

The previous step was responsible for finding the elements of the layout that should be moved. This corresponds to the set R of points, which in turn represent wirings, sub peers and controls point of links. In the horizontal phase, these layout elements are now shifted to the left by one grid position. It is important to note that the control points of links that dock onto the POC of the surrounding peer now do not fully connect it it anymore. For this reason, the surrounding peer is also reduced in width.
Figure 6.20 continues the packing process for the example peer from figure 6.19. Subfigure I displays the excessive space that can be conserved by moving the elements of the set R to the left. Subfigure II shows the packed layout after the set movement step is completed. This also marks the end of the packing step. No valid path can be found anymore that would cut the layout into two sets, both horizontally and vertically. The layout depicted in subfigure II is considered optimally packed and the packing step is therefore completed. Figure 6.21 shows the example layout used in the previous parts

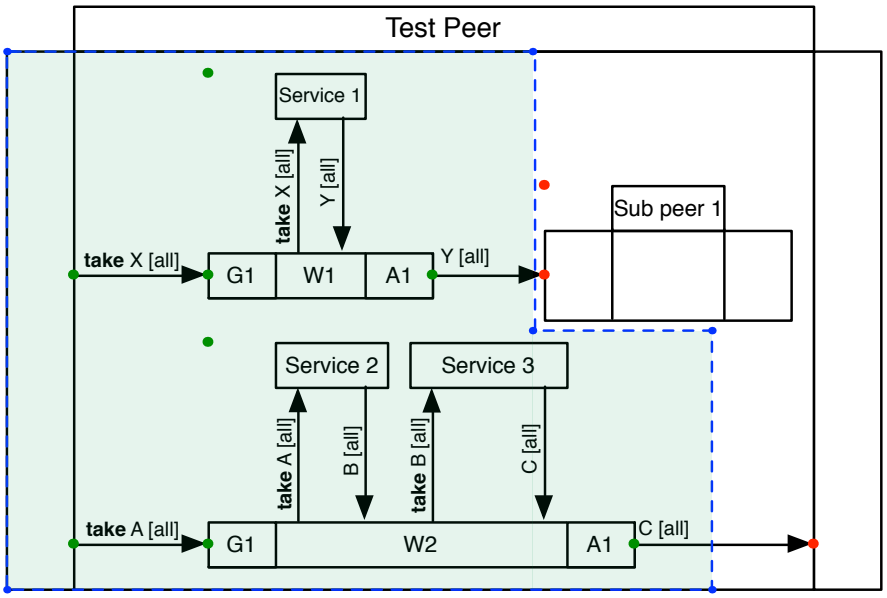


Figure 6.19: Splitting the layout into two distinct polygons. The left polygon (outlined in blue, filled with green) contains the elements corresponding to the green dots. The right polygon contains the other elements corresponding to the red dots.

of the layouting process after the packing step has been applied to it. Compared to figure 6.15, the layout is more compact and the available space is used more efficiently. It should be noted that there are certain edge cases in which the just proposed packing algorithm does not lead to an optimal solution in the sense that it neglects to move elements closer together. However, these cases are very rare in practice and do not impact the general usefulness of the packing process.
A interesting characteristic of the algorithm is that it does not change the general structure of the layout. The rough positions of elements and their relation to each other is not changed. Also, no additional link edges are introduced.

6.4 Nested Peers / Recursive Layouting

The structure of the Peer Model is recursive in nature. This facilitates a component based design and encourages the reuse of peers. The same peer definition can be instantiated and used in multiple places. Specifically, peers can be nested inside each other. Inside a peer there can be wirings and sub peers, which in turn can also contain yet other wirings and sub peers. This is a challenge for the layouting and the visualisation of Peer Models in general.
While working with the monitoring tool, the user has the option to expand and collapse

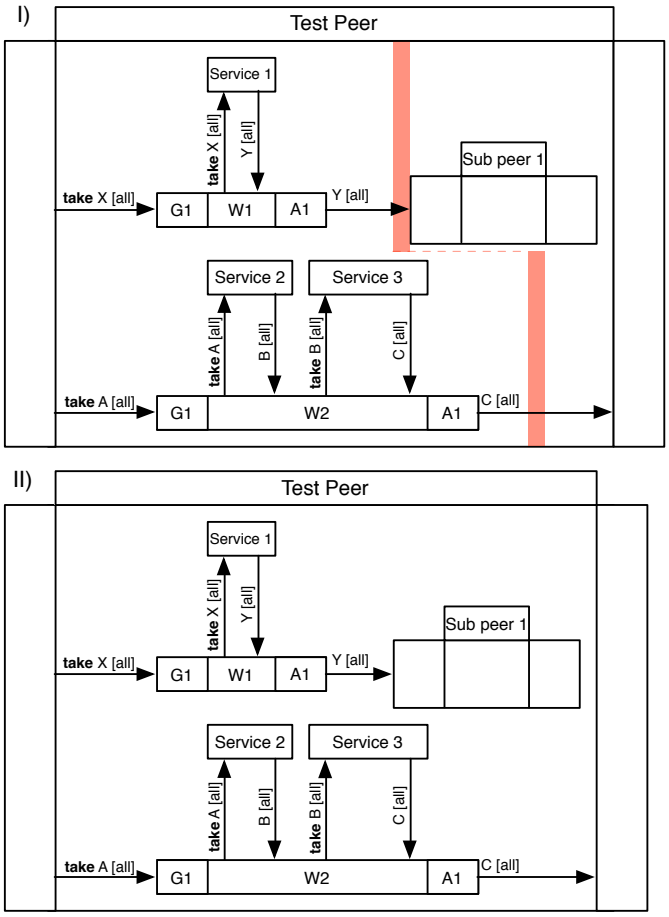


Figure 6.20: Moving the sets closer together. Subfigure I shows the potentially wasted space that can be removed as found in the previous steps. Subfigure II displays the layout after the right set has been moved.

peers at his/her convenience in order to expose information he/she is interested in or to hide currently irrelevant information. The layouting and visualisation of collapsed peers is described in the previous subsections. For expanded peers, the whole layouting process is run again in a recursive fashion. Inner peers need to be layouted first in order to compute their final size. This information can then be used to properly layout the parent peer. It should be noted that even the root peers inside a processor can be collapsed to give the user a very high-level view on the complete Peer Model. Figure 6.22 shows a small peer layout, once with the sub peer collapsed, once expanded.

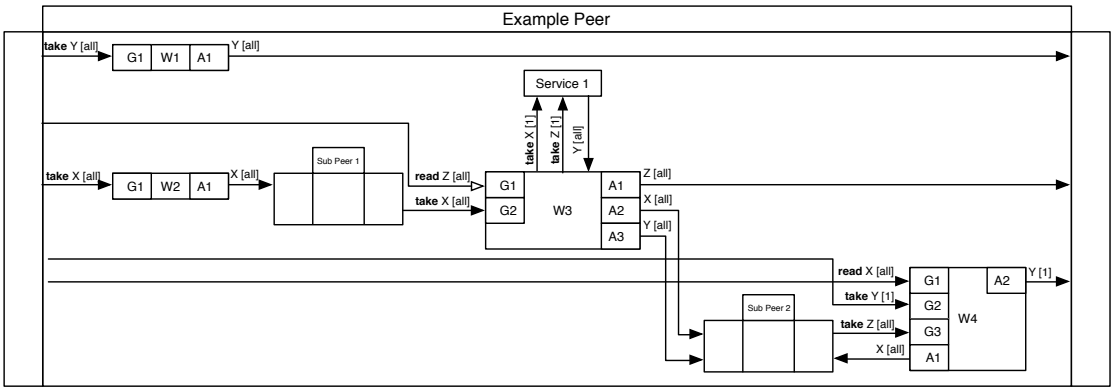


Figure 6.21: The example peer after the packing step. The elements are positioned much closer to each other than before and the available space is used more efficiently. Compare to figure 6.15 to see the difference.

6.5 Processor Layouting

The previous subchapters discussed how a single peer and its child elements are layouted. This subchapter deals with the positioning of readily layouted peers within their respective processor and how the processors themselves are positioned. For all positionings, the previously discussed grid also applies.

Within a processor, the layouting process for each contained peer is run. After that, the full size of a peer is known and it can be properly placed within the processor. Peers are positioned one below the other. The order is determined by the peer's appearance in the static structure definition. See section 7.2 for a detailed discussion of the static Peer Model structure.

The processor's width is determined by the widest peer plus a small border. It's height is calculated as the sum of all contained peers as well as gaps between peers and borders at the top and bottom. At the top, the border is slightly broader to fit a label that states the processor's name.

Multiple processors are aligned horizontally and are placed one after the other. This is to compensate for the vertical ordering of peers within processors and use the screen space efficiently. Just as with peers, the order of the initial horizontal placement is also determined by the appearance of the processors in the VIL. However, a specific Peer Model might have a certain semantic to it that is better explored by the user when a different positioning is used. For this reason, processors can be manually moved in order to get a better visualisation. However, the movement of peers within a processor is not allowed.

Figure 6.23 shows how the general structure of the complete layouting process. It gives an overview over how its parts are interrelated and depicts the general sequence of al-

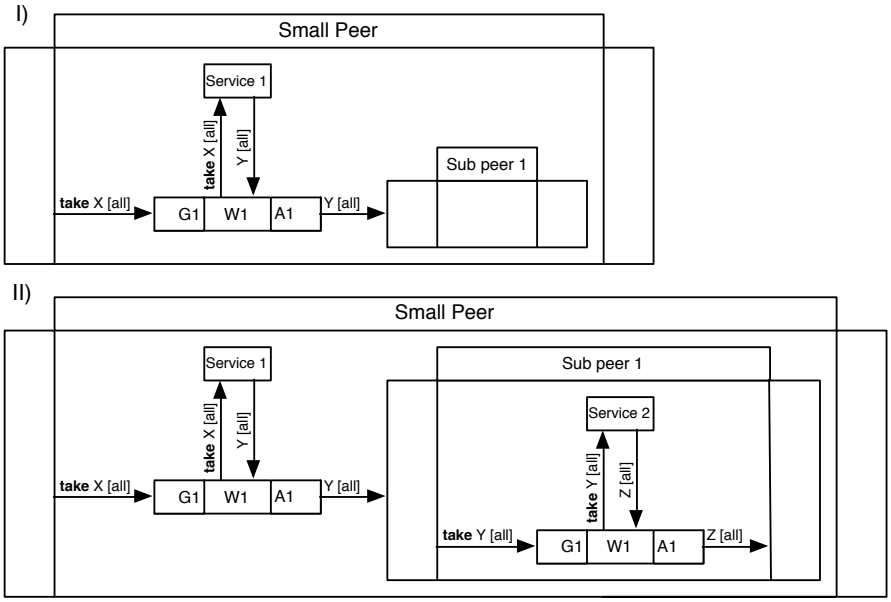


Figure 6.22: Small test peer, once with *sub peer 1* collapsed (I), once expanded (II). Expanding a sub peer leads to a larger overall layout. The layouting process is recursively repeated for expanded sub peers.

gorithms necessary to layout a Peer Model.

6.6 Relayouting

In certain situations the layout needs to be redone. This is primarily the case after the user has manipulated the layout manually. Relayouting can happen in a number of ways. In some cases only a subpart of the layout is restored, in others the complete model is subject to relayouting.

Repositioning of elements

The user has the ability to reposition elements after the initial layouting step. In this case, the corresponding links that connect to the moved element obviously need to be updated as well. Unfortunately, it is generally not possible to only reroute the directly affected links only as other links might be affected indirectly by the new routes. Furthermore, only rerouting a portion of the links negates the previously established routing order. Therefore, all links within the parent peer are rerouted according to the procedure discussed in section 6.2. Rerouting might also change the required space the peer needs. Thus, the update is recursively applied to all parent peers in the chain up until the last

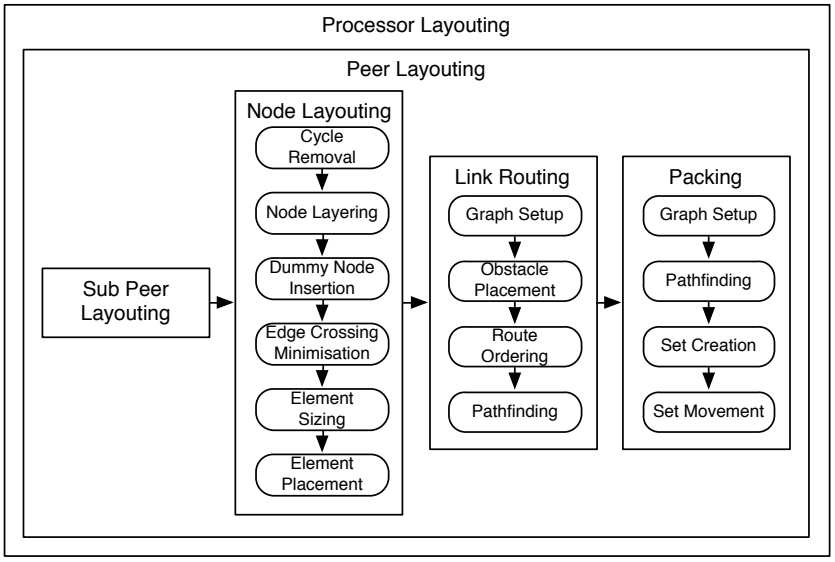


Figure 6.23: Structural overview over the complete layouting process.

peer that is a direct child of the processor. Apart from that, no other layouting steps are performed.

Expanding of a sub peer

The expansion of a sub peer has far reaching consequences. First of all, the sub peer itself needs to be fully layouted. This is done by applying the complete layouting process. Secondly, as the sub peer might have changed its size due to this, its parent peer needs to be partially relayouted as well. It is therefore treated as if the sub peer was moved and a rerouting is applied (see the previous subsection).

Manual relayouting

The user is provided with the functionality to force a full relayouting of a peer or sub peer. This is especially useful when wanting to reset the layout after moving around a lot of elements or after expanding a lot of sub peers. The peer in question is subject to a complete relayout while its parent peers recursively have their links rerouted, just as in the previous cases.

Compact representation of services

Changing the way how services within a peer are represented is a severe change. Thus, the peer in question is fully relayouted like in the case of a manual relayouting.

6.7 Conclusion

This chapter presented the developed framework for layouting Peer Model visualisations. The framework consists of a series of algorithms performed in succession. The combination of the Sugiyama layout algorithm with grid-based path finding and the packing algorithm presents a novel approach for layouting graph structures like the visual Peer Model notation. In particular, the introduced packing algorithm is a novelty.

CHAPTER 7

Implementation

While the last chapter dealt with the layouting process in particular, the focus of this chapter is to describe the general implementation of the monitoring tool. Topics include the motivation behind the choices in technology, such as the software platform, the UI framework and the graph layouting library. The general code architecture and the motivation behind it is presented. Furthermore, the applied interaction techniques as well as the development of the input file formats are discussed. Also, general issues and considerations that came up during the implementation phase are presented.

7.1 Technology Choices

Before developing the monitoring software, research was conducted to decide on which technologies and software solutions to use. The found requirements (see chapter 4) offer a good insight into what the technologies need to be capable of.

Software Platform

Three software platforms were examined: C++, Java and .Net. For C++ and Java, a GUI framework was tested each. A discussion on the three platform follows:

C++ and Qt

Qt [68] is a cross-platform application framework. It features native graphical user interfaces on most operating systems and therefore integrates well into the platform. Instead of implementing its own controls, It uses the native APIs of the underlying operating systems for the rendering and interaction of control elements whenever possible.

Qt is essentially a C++ framework, but offers bindings for other languages, such as

Ada or Python. Additionally, developers may leverage JavaScript and the markup language QML for developing graphical user interfaces [69]. Apart from graphical user interfaces, it may also be used for other software, such as command-line tools.
An important concept of Qt is *Signals* and *Slots*. It is the preferred way for realising communication between objects. GUI widgets (or other objects) may send signals that contain information, such as event data. Signals may then be received by other objects using slots. C++ does not natively support this concept, but Qt's implementation spares the developer from a lot of boilerplate code by leveraging a *Meta Object Compiler* (short: MOC) [87], that automatically generates the necessary C++ code.
For the main view of the monitoring tool, the *QGraphicScene* class together with the *QGraphicsView* class from the *Graphics View Framework* is used. It provides a canvas for drawing and managing 2D graphical items, such as graphic primitives and text. It is also responsible for event propagation, like mouse events and managing the scene graph through the use of nested *QGraphicsItem*s.

Java and JavaFX

JavaFX [42] is a software platform for developing desktop applications and rich internet applications (RIAs). It has been developed by Oracle as a replacement for and successor to Swing as Java's standard library for graphical user interfaces. JavaFX is based on a scene graph and goes much farther with this concept than Qt does. Unlike Qt, in which the scene graph is only prominent within the *QGraphicScene* element, the scene graph in JavaFX is the central element of the GUI and all elements that are displayed are a part of it. However, the drawing of the scene graph is done in so-called *retained mode*. This means that instead of drawing primitives directly, one defines the elements that should be displayed and JavaFX handles the rendering itself. Once an element is added to the scene graph, it is constantly drawn without any more action required. This is in stark contrast to the way Qt handles rendering of elements.
JavaFX exists in two branches, namely version 2.x and its successor version 8.x. JavaFX 8 incorporates many new features compared to its predecessor, such as additional controls and themes as well as improved concurrency handling and CSS styling support. Unfortunately, it requires a Java 8 runtime environment to be present on the system. At the time of writing, dissemination of Java 8 is still very low. While the features JavaFX 8 offers would have been beneficial for the development, requiring Java 8 was considered too excessive. The developed monitoring tool therefore uses JavaFX 2.

.Net

Whereas Qt and Java FX work in a cross-platform setting, .Net solely offers Mi-

crosoft Windows as the officially supported operating system. However, cross-platform support was an obligatory requirement that had to be fulfilled by the monitoring software. While the *Mono Project* [60] might have been a suitable alternative to enable cross-platform support, it was deemed a risk that might have had negative effects on development. Therefore, the Mono Project was not considered and .Net was not deemed suitable for developing the visual monitoring tool.

As .Net drops out due to its missing support for other operating systems than Microsoft Windows, this leaves C++/Qt and Java/JavaFX on the shortlist. One of the most important requirements was the proper drawing of and interaction with the Peer Model main view. This involves drawing of graph structures, including 2D primitives like lines, circles, rectangles and text in order to build the Peer Model visualisation. Also, interaction with these elements is necessary, primarily clicking of elements with the mouse and handling the resulting mouse events.
After evaluating Qt and JavaFX by developing small test applications, both were considered to be a good fit for the development of the monitoring tool. In the end, the decision came down to personal preference and experience. The author is more competent in Java than in C++. While both Qt and JavaFX were unfamiliar frameworks, the author is much more acquainted using Java than any C++ environment. For this reason, Java and JavaFX were chosen as the software platform and the application framework.

Graph Layouting Library

Automatic layouting of the Peer Models is an important aspect of the monitoring tool. Chapter 6 thoroughly discusses the taken approach. A number of libraries for the task of graph layouting and rendering exist, which provide algorithms for commonly used layouting approaches and spare the developer the necessity of implementing them himself/herself. After deciding on the software platform, but prior to starting the implementation, three graph libraries, namely Graphviz [30], JGraphX [31] and Jung [45], were evaluated in order to test their suitability for the monitoring tool and whether or not they can reduce the implementation time. Unfortunately, no feasible graph layouting library based on JavaFX exists at the time of writing. A short introduction to the evaluated libraries follows:

Graphviz

Graphviz [30] is open source graph visualisation software. It may be used in conjunction with the *DOT* graph description language, with which graphs can be specified in human readable text form. The DOT language specifies the structure of a graph, but does not offer a lot of functionality about how the graph should be visualised. Graphviz offers multiple, distinct tools that tackle the issue of

layouting. They take DOT files and produce a graphical representation of the graph in various format, some of which can serve as a basis for further processing. Peer model layouting requires creating layered representations of directed graphs and the Graphviz tool capable of this is called *dot*. It both supports orthogonal routing and a feature called ports, which allows for defining specific connection points for a graph node. However, the combination of the two features is not supported, which is a vital requirement for layouting Peer Models. In general, Graphviz does not offer the amount of adjustability required for the layouting tasks at hand.

JGraphX

JGraphX [31] is a Swing-based library for graph visualisation. It is the open-source fork of the commercial JavaScript library *mxGraph* [62]. JGraphX provides functionality for visualisation and interaction with node-edge graphs. It also offers basic capabilities needed for layouting Peer Models, such as orthogonal routing of edges and hierarchical node positioning. Unfortunately, when evaluating the features practically by layouting actual Peer Models, it did not perform well. While the node placement was feasible, link routing was subpar. Orthogonal edge routing is only possible when specifying two specific endpoints. But the Peer Model notation requires links to be able to connect to peer containers at any location and the layouting procedure is supposed to find the best connection locations by its own. Also, JGraphX offered very little flexibility for adjusting the hierarchical node positioning.

Jung

Jung [45] features different methods for layouting graphs, including tree layouts, radial layouts and balloon layouts. However, its support for hierarchical layouts is limited. Jung appears to focus more on force-based layout algorithms and does not support most of the features needed for properly layouting Peer Models, which is based on hierarchical graph layouting.

While several libraries exist for general purpose graph layouting, no tested library fulfilled the stated requirements. The challenges that layouting Peer Models poses could not be met by either Graphviz, JGraphX or Jung. Among other features, none of the libraries support proper orthogonal routing in combination with the manual placement of connection points.
Some libraries are able to provide algorithms for subparts of the proposed approach in chapter 6, like the Sugiyama layouting algorithm. But including and integrating an external library just for small parts of the full layouting approach did not seem like a feasible solution. Furthermore, modifications to certain base algorithms were necessary for them to be useful. For example, in the case of the Sugiyama framework, changes to the way it performs the edge crossing minimisation step are required (refer to 6.1 for a

detailed discussion). The tested libraries simply do not offer this degree of flexibility. Lastly, each library introduces its own set of dependencies that need to be managed and considered when deploying the tool. For the stated reasons, no third-party graph layouting library was used and instead, the layouting procedure was fully implemented from scratch.

7.2 File Formats

This chapter discusses the development of the two file formats required when working with the monitoring tool. The first format corresponds to the static structure of a Peer Model, and is called *VIL* (*Visualisation Intermediate Language*). The second contains the dynamic events happening within an active Peer Model. A collection of consecutive events is called *Trace* and the associated file is called *TIL* (*Trace Intermediate Language*). Splitting the information in two separate files makes sense, as there is generally a one-to-many relationship between the VIL and the TIL. There can be many different traces for the same Peer Model implementation and developers are encouraged to work with multiple TIL files. As can be seen in the user interface, the monitoring tool allows the separate loading of VIL and TIL. Developers can load different TIL files while keeping the same VIL file loaded.
A separate discussion about VIL and TIL follows. Figure 7.1 gives an overview on the roles TIL and VIL play for the monitoring tool.

VIL

The VIL contains all the necessary information to build the static representation of the Peer Model. It corresponds very closely to the Peer Model's own DSL. But while there exists a parser for the Peer Model DSL [49], we decided that an intermediate format generated from the DSL was a better fit for the monitoring tool because including the DSL parser seemed counterproductive. Developing an intermediate format also opened up the possibility to choose a widely spread and amply supported description language. We agreed upon using JSON as the VIL's format of choice. JSON is a well-known, open standard format for which parsers exist for most languages, Java being no exception. A data-processing tool called Jackson [40] is used for parsing all JSON data in the monitoring tool. The translation of the DSL into the VIL is done within the Peer Model's toolchain [49]. Listing 7.1 shows a small exemplary VIL file and figure 7.2 shows its corresponding representation when loaded into the monitoring software. An important thing to note about the VIL (and also the underlying DSL) is that there is a distinction between definition and instantiation of services, wirings and peers. Each is given a separate section within the VIL where the definitions take place. All definitions are given a name and can be referenced at other places in the file. For example, in listing

```
{
  "entryTypes":[
    { "name":"X" },
    { "name":"Y" }
  ],
  "services":[
    { "name":"Service 1", "implementation":[],
      "input":[
        { "operation":"Take", "type":"X",
          "query":"", "count":"[all]" } ],
      "output":[
        { "operation":"Write", "type":"Y",
          "query":"", "count":"[all]" } ]
    } ],
  "wirings":[
    {
      "name":"W1",
      "guards":[
        { "from":"C1", "operation":"Take", "type":"X",
          "query":"", "count":"[all]" } ],
      "services":["Service 1"],
      "actions":[
        { "to":"C2", "type":"Y", "query":"",
          "count":"[all]", "operation":"Write" } ]
    } ],
  "peers":[
    { "name":"Small Peer", "subpeers":[ "Sub peer 1" ],
      "wirings":[
        { "wiring":"W1",
          "linking":{ "C2":"Sub peer 1.PIC" } } ] },
    { "name":"Sub peer 1", "subpeers":[],
      "wirings":[] } ],
  "processors":[
    {
      "name":"Small Processor", "peers":[ "Small Peer" ],
      "parameters":[]
    }
  ]
}
```

Listing 7.1: Exemplary VIL file. Note that the file is the result of running the PM-DSL file from listing 3.1 through the toolchain. Figure 7.2 shows the resulting Peer Model visualisation.

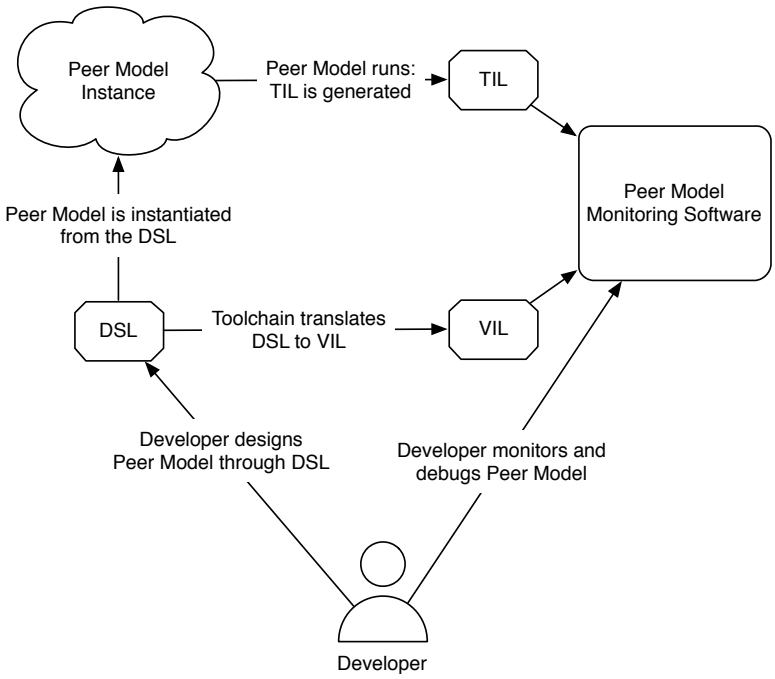


Figure 7.1: Diagram depicting the role of VIL and TIL in the overall workflow of the Peer Model toolchain with regards to the monitoring tool.

7.1, a service named *Service 1* is defined on lines 7 to 14. Later, it is referenced by wiring *W1* on line 21.

A goal of the Peer Model's DSL is to encourage modularisation. Wiring definitions may contain placeholders in place of actual target containers for guard- and action-links. The instantiation of wirings happens when defining the peers. This instantiation can include *linking* information which describes how the wiring's placeholders should be replaced. A single wiring definition can therefore be used multiple times inside multiple peers with different configurations. This setup encourages building of reusable components.

In listing 7.1 on lines 28 to 30, the wiring *W1* is instantiated within peer *Small Peer*, including information on how to connect the wiring's links. In this example, *C2* is replaced by the container reference *Sub peer 1.PIC*. To keep the format concise, default values are used if a placeholder is not specified. By default, guard links are connected to the parent peer's PIC, while action links are connected to the POC, respectively. As *C1* is not specified, *W1*'s guard will be linked to *Small Peer*'s PIC. Referencing of peer containers works in accordance to the notation specified in chapter 3.

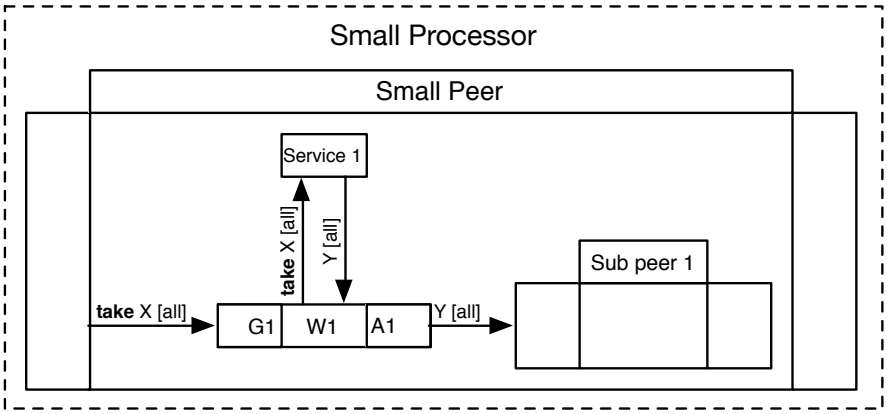


Figure 7.2: Small test processor, resulting from the VIL file in listing 7.1.

TIL

The Peer Model approach is relatively novel. While monitoring and debugging toolchains are currently being developed, many features are still missing. Specifically, no standardised format for representing the events inside a Peer Model exists. For this reason, we developed the TIL format. In conjunction with the VIL, it contains all the information necessary to retrace the events that happened during a timespan and enables developers to gain insight into the workings of the observed Peer Model.
The TIL is created by tracing mechanisms that record the occurring events while the Peer Model is running. This can either happen within a real-world Peer Model instance or inside a simulation. It should be noted that the actual creation process of the TIL is not part of this thesis and is therefore not discussed any further.
Initial drafts of the TIL used a tabular file format with JSON-encoded entries stored inside cells. Each event was stored in a row and the rows were sorted by their timestamp. However, the underlying data was not well suited for such a format. The various events do not share a lot of common data fields but each exhibit distinct information which required the format to contain many contextual columns. Depending on the type of the event, which was specified in a field at the start of the row, subsequent fields contained different data. This was both confusing and error prone. To overcome the shortcomings, a different file format fully based on JSON was developed and is discussed below.
All events that are recorded in a TIL contain information about the entries they work with. Each entry is always fully specified, including all of its co-data and app-data. While this produces significant overhead and duplicated data, it was eventually considered the best option. Before reaching this conclusion, different approaches have been proposed, one of which was to identify each entry with a system-wide unique ID. Unfortunately, the Peer Model implementations generally do not keep entry-IDs that are unique across the whole system. It was not deemed feasible to force all Peer Model

```
{
        "timestamp": "0.0",
        "peers": {
                "testProcessor:testPeer":{
                        "PIC": [
                                { "type": "Y" }]
                        "POC": [
                                { "type":"X" },
                                { "type":"Z" }]
                }
        }
}
```

Listing 7.2: Example of an initial state section in TIL files. The containers of peer "testPeer" are filled with entries of type "X", "Y" and "Z".

implementations to add this feature. The approach would have required Peer Model implementations to keep a consistent map of all present entries in order to correctly reference them when creating events. Instead, within the monitoring software, finding the correct entries is based on generating unique IDs from hashing each entry's contents and comparing IDs for equality. Refer to section 7.3 for a discussion on this topic.
Both an entry's co-data and its app-data are stored inside a JSON key-value map, with all keys and values encoded as strings.

Events

Several types of events can occur in a running Peer Model. An enumeration of the possible events along with an explanation follows:

Initial State Event

In general, it is not always possible or expedient to start the tracing process with an empty model. For this reason, the initial state event captures the condition of the complete model at the time of startup. However, only the contents of peer containers are stored. No wiring activations are captured, which in turn means that no wirings must be active when starting the tracing process. This compromise is made to reduce the complexity of obtaining the initial state while still being able to start the trace while the system is running. The Peer Model implementation is responsible for enforcing this constraint. Every TIL file contains exactly one initial state event, including a timestamp and the contents of each peer container. Listing 7.2 shows an initial state event.

Wiring Event

The term *Wiring Event* is used as an umbrella term for the following events:

1. Guard links event
2. Service inputs event
3. Service output event
4. Action link event

Each wiring event contains a list of entries it acts upon. It also contains a timestamp that represents when the event happened. A wiring event does not hold any information about the underlying structure of the wiring it takes place at but rather references the corresponding wiring in the VIL file. The references are specified using the previously established notation for wirings. Using both the VIL and the TIL, the monitoring software is able to assign wiring events to the actual wirings and their links and services. See section 7.3 for a detailed discussion on the topic. While action link events and service output events only correspond to the activation of a single action link, respectively a single service output, guard link events and service input events are more complex. By default, all guards of a wiring activate at the same time. Correspondingly, all service inputs of a service activate simultaneously. This requires a single event that contains all the affected entries. To correlate the entries with the correct guard link (or service input) that processed them, the event is designed to support this. Additionally, the introduction of explicit commits in the guard section also requires this structured approach. Listing 7.3 shows a typical guard link event within which three guards activate, the first one of which is a none-check guard. Service input events are structured in the same fashion.
Action events and service output events are comparatively simple. Listing 7.4 shows an exemplary action link event. Service output events are structured analogously.

DEST-Send and -Receive

It is possible for entries to be sent to a different peer container without a wiring by means of the DEST property. A service can set an entry's DEST property to any peer container's location. If the entry is then put into a container, the corresponding peer tries to resolve the location and send the entry to the associated target container. This marks the first of two events, namely the *DEST-Send* event. Such an event consists of the affected entry, the peer container it is sent from and a timestamp. When the entry arrives at the target peer container, another event is created: the *DEST-Receive* event, which is structured equally to the DEST-Send event, but stores the destination container instead of the source. Listing 7.5 shows an example DEST-receive event.

```
{
        "type": "guardLinks", "timestamp": "2.5",
        "wiring":"testProcessor:testWiring",
        "guards": [{
                "index":"1",
                "entries": []
        }, {
                "index":"2",
                "entries": [{ "type":"X" }]
        }, {
                "index":"3",
                "entries": [{ "type":"Y" },
                            { "type":"Y" }]
        }]
}
```

Listing 7.3: Event notation for guard link activations in TIL files. The first three guards of the wiring "testWiring" activate simultaneously. The indices denote which entries belong to which guard. Note that the first guard is a none-check guard and the entries array is therefore empty.

```
{
        "type": "actionLink", "timestamp": "3.8",
        "wiring":"testProcessor:testWiring",
        "index":"2", "entries": [{ "type":"Z" }]
}
```

Listing 7.4: Action link activation event as it is listed in TIL files. Note its relative simplicity compared to guard links events.

```
{
        "type": "destMoveReceive", "timestamp": "6.0",
        "destination": "testProcessor2:testPeer.PIC",
        "entry": {
                "type":"event",
                "coData": {
                        "DEST":"testProcessor2:testPeer.PIC"
                }
        }
}
```

Listing 7.5: Example DEST-receive event. Note that the entry co-data still includes the DEST-property itself. This is necessary to correlate the event to an earlier DEST-send event with an analogously structured entry.

TTL/TTS events of entries

Entries can have *TTL* (*Time To Live*) and *TTS* (*Time To Start*) properties attached to them. In both cases, a timestamp is stored. An entry with a TTS property is not considered by wirings until the timestamp is reached, whereas an entry with a TTL is removed once the timestamp is reached. As these properties are also stored within the entry, it would be possible to calculate the corresponding events implicitly and omit explicit event types. But this could lead to discrepancies between the retraced procedure and the actual execution due to timing issues and race conditions that are present in distributed systems. For this reason, TTL and TTS events for entries are explicitly recorded and stored. Such an event contains the affected entry and a timestamp. Listing 7.6 shows an example TTL event. TTS events are structured analogously.

7.3 Software Architecture

The *Model-View-Controller* (short: *MVC*) [73] pattern served as the basis for the monitoring tool's architecture. Extensibility and flexibility were two key goals in terms of code structure and MVC supports a sensible separation of responsibilities. Figure 7.3 gives an overview on the various parts of the software from an architectural perspective and in which part of the MVC structure they fit in. In the following sections, the important parts are discussed in detail.

```
{
        "type": "entryTTL", "timestamp": "17.03",
        "peer":"testProcessor:testPeer.POC",
        "entry": { "type":"X",
                   "coData": { "TTL":"17.0" }
        }
}
```

Listing 7.6: An exemplary entry TTL event specified in a TIL file. It is important that the event still lists the affected entry with the TTL property set. Otherwise, the hashing process would yield a different ID and the corresponding entry in the peer could not be found by the retracer. Also note the difference between the TTL property and the actual event time.

Static And Dynamic Model

For the monitoring tool, it is necessary to have an internal model of the static structure that constitutes the underlying Peer Model. Elements, such as processors, peers and wirings need to be represented within the software. The class structure pertaining to this model resembles that of the relationships in the actual Peer Model. It is also closely related to the structure of the VIL file, which is no surprise, as the internal static model is fully built from this file. Figure 7.4 shows the class structure of the static model in UML notation.
Additionally, the dynamic events stemming from a TIL file also need to be represented in-memory as class structures. This includes elements such as entries, wiring activations and their events as well as other event types. Most dynamic elements maintain references to elements of the static model. For example, a *GuardLinkEvent* object stores a reference to the *GuardLink* object corresponding to the guard link it takes place at. Figure 7.5 gives an overview on the class structure of the dynamic model in UML notation.

Intermediate Data Format

The JSON-library *Jackson* is capable of mapping a file to an object tree consisting of *POJO*-classes and collections. However, both the VIL and the TIL contain internal references, encoded as strings. An example of such cases is the definition of sub peers and wirings within a parent peer in the VIL files. Jackson can not automatically convert these string-based references into in-memory Java references. Thus, an intermediate object tree is developed. The tree is a one-to-one in-memory representation of the input file, where string references are not yet resolved. This approach is used for both VIL and TIL files. The intermediate format is then transformed into the final, separate data

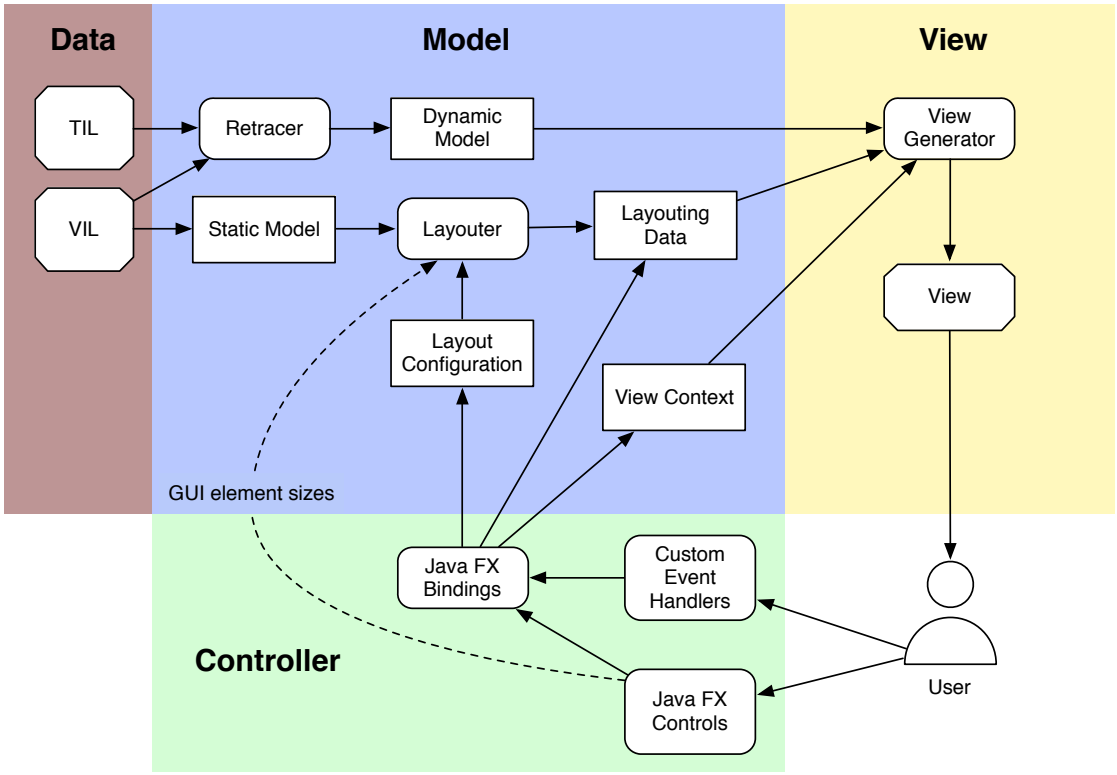


Figure 7.3: Overview diagram showing the general software architecture of the monitoring software. The application is based on the MVC architectural design pattern.

structure by resolving the string references into java object references, resulting in the static and dynamic model discussed in the previous sections. Listing 7.7 exemplarily shows a condensed version of the *ProcessorDefinition* class, which is part of the intermediate data format. Listing 7.8 shows the associated final class for a processor. Note the difference in the peer lists between the two formats.

Layouting

Extensibility of the software was a an important goal. The layouting process is designed to be almost completely independent from the view tier and therefore, JavaFX. The only exception is the size of certain elements within the view. For example, the width of a peer's label can influence the peer's size. Apart from that, the layout can be produced without depending on the used view technology. This helps fulfilling the design principle *Separation Of Concerns (SoC)* [38]. Should the need for a different GUI technology arise, swapping out JavaFX becomes a lot easier. Furthermore, the layouting data can potentially be exported and used for other purposes as well, such as external visualisa-

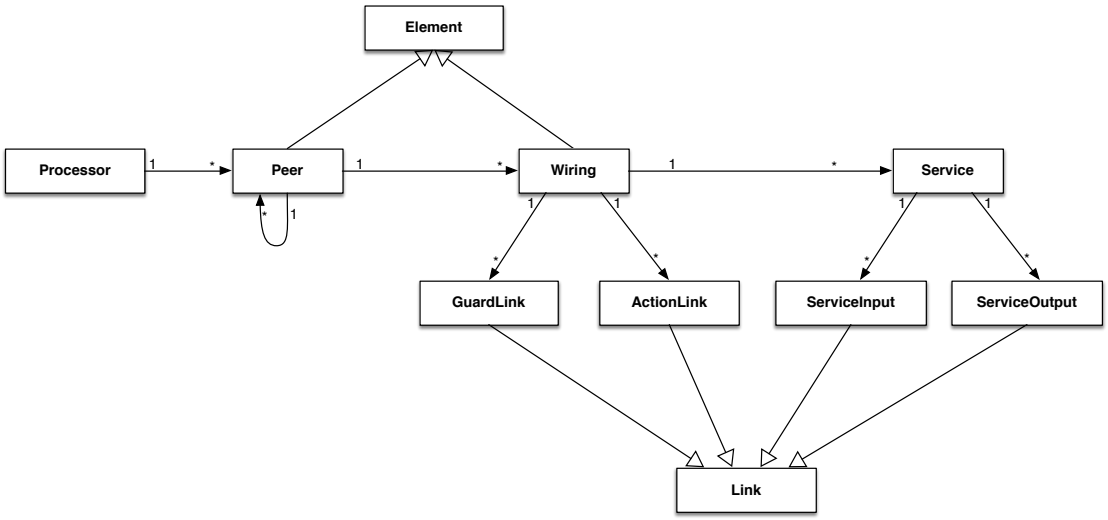


Figure 7.4: Diagram of the static class structure in UML notation.

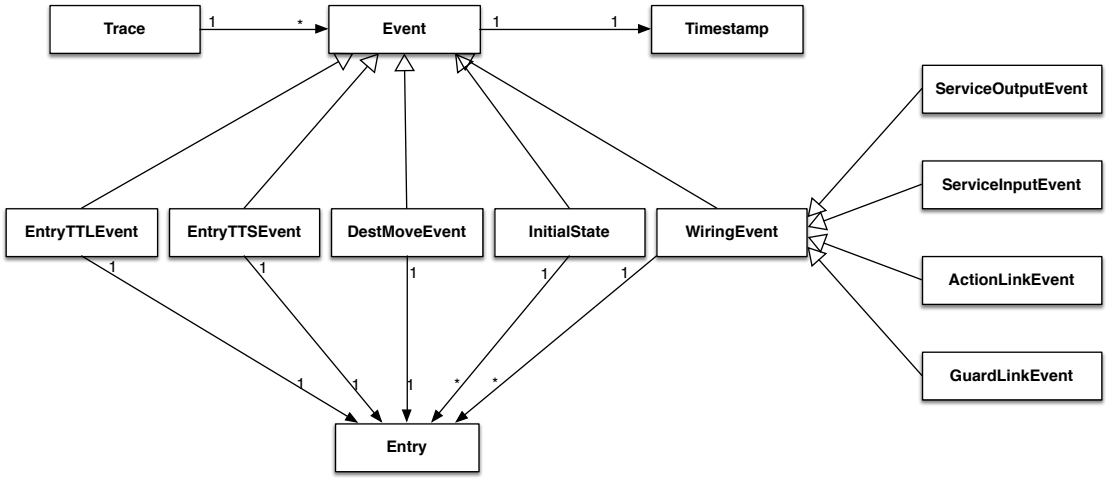


Figure 7.5: Diagram of the dynamic class structure in UML notation. References to the static part of the model are omitted for clarity.

tions.
Data structures to hold the layouting information are needed. They are generated by the layouter, which is discussed in chapter 6. The layouting data is passed to the view generator which is responsible for assembling the scene graph. For a discussion on the view generator, refer to section 7.3.
The layout data itself is an object tree, with a list of *ProcessorLayout* objects at its root (shown in listing 7.9). The layout tree closely correlates with the structure of the Peer

```
public class ProcessorDefinition {
        private final String name;

        private final List<String> peers =
                new ArrayList<String>();

        // rest of class removed for brevity
}
```

Listing 7.7: Class file for the intermediate data format of processors. The *peers* list stores string references to peer definitions.

```
public class Processor {
        private final String name;

        private final List<Peer> peers =
                new ArrayList<Peer>();

        // rest of class removed for brevity
}
```

Listing 7.8: Class file for the final data format of processors. Compare the *peers* members in this listing and in listing 7.7.

```
public class ProcessorLayout {
        private final List<PeerLayout> peers =
                new ArrayList<PeerLayout>();

        private int x;
        private int y;
        private int width;
        private int height;

        private final Processor processor;

        // rest of class removed for brevity
}
```

Listing 7.9: ProcessorLayout class as used within the monitoring tool. The members *x, y, width* and *height* are specified in grid coordinates and are therefore integer based.

Model itself. Every visible element has a layout object, which references its corresponding model object, which in turn represents a part of the actual Peer Model. Figure 7.6 shows a UML class diagram of the layout classes. The Peer-, Wiring- and Processor-

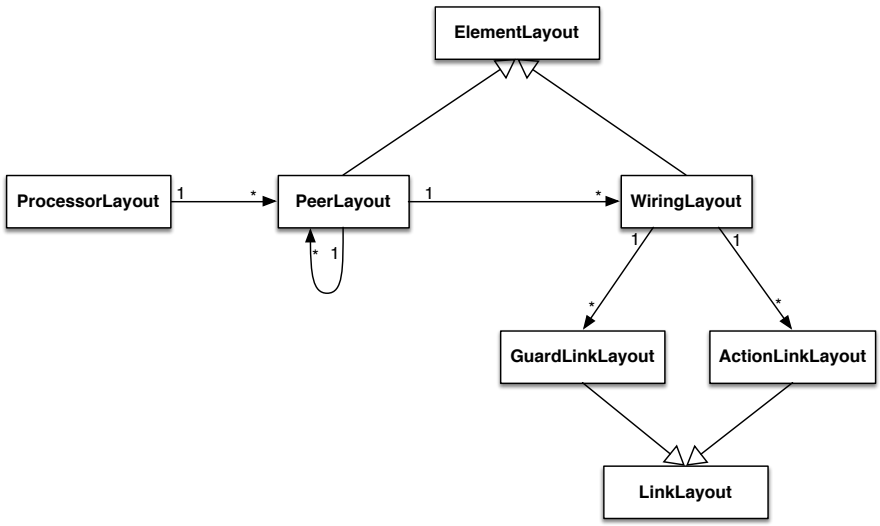


Figure 7.6: UML class diagram depicting the classes used to store the layouting information for the various elements. Note that the service links do not have their own layout classes, but are part of the wiring layout. This is mainly because they are not fully routed like action and guard links are and therefore do not require control points. Note that references to the model classes are omitted for clarity.

Layout classes contain a position and a size, both specified in grid coordinates. In general, most spatial information of the layout classes is specified in that coordinate system. The LinkLayout class specifies a list of control points used for determining the link's arc.
A data structure called *Layout Configuration* is used for specifying parts of the information needed for layouting. Specifically, this incorporates whether a peer should be shown expanded or collapsed and whether or not the services of wirings should be visualised in the compact form. When a model is loaded, the layout configuration is set to its default values: top-level peers are expanded, lower level peers are collapsed and all services are displayed in their full form. The data in the layout configuration may be modified by the user through the user interface, which requires a relayouting to take place (refer to section 6.6 for a detailed discussion).
The *View Context* stores whether the main view should display the static or the dynamic representation of the Peer Model. This setting may also be changed through a button in the user interface. In the dynamic representation, the timeline is shown at the bottom

while in the static representation, the main view takes the full window's height. In the dynamic case, the view context also stores which moment is currently visualised.

View Generator

The *View Generator* is responsible for creating the main view that the user is eventually presented. It uses the layouting data to determine the position and size of elements, such as peers and wirings as well as the route of links according to their control points. As the layouter is retrieving actual GUI element sizes from the view, it is able to generate the final layout with no modification necessary later on. Therefore, the view generator can fully trust the layouting data and use it without adjustment.
Additionally, it relies on the view context, which governs whether the static or the dynamic representation is supposed to be displayed. Depending on this setting, the view generator either creates dynamic elements or their static counterparts. If the dynamic representation is active, the dynamic model is taken into account as well and the elements are visualised in accordance to this data.
Figure 7.7 shows most of the used base view classes and their subclasses for both representation. The view generator and the view classes are the first part in the MVC chain that is heavily intertwined with JavaFX. The view classes derive from JavaFX base classes representing elements in the scene graph. In case of a relayouting or a change in the view context, the view has to be partially or fully recreated. However, full rebuilds are often suboptimal and waste unnecessary amounts of computation time. The implementation is therefore optimised to recreate as little as possible. For example, if the user moves an element in the main view, such as a peer or a wiring, the corresponding view class is updated, as well as the layout class. Furthermore, in accordance to section 6.6, the links in the affected peer are rerouted. This in turn triggers a recreation of the view classes for the links. All other view elements are left intact and do not need to be recreated.

Retracer

While the TIL in conjunction with the VIL contains all the events that occurred in the Peer Model, it does not explicitly include the model's full state at each point in time. More specifically, the information about which peer contains which entries at what time needs to be inferred from the available data. Also, the contents of all wirings' ECs need to be computed for all moments during which the associated activations are occurring. The *Retracer* is responsible for reenacting the events from the trace, starting with the initial state.
A dedicated data structure is implemented that holds the complete state of the initial Peer Model. All events are ordered by their timestamp and executed one after the other according to this order. Each event performs its actions on the data structure, modifying

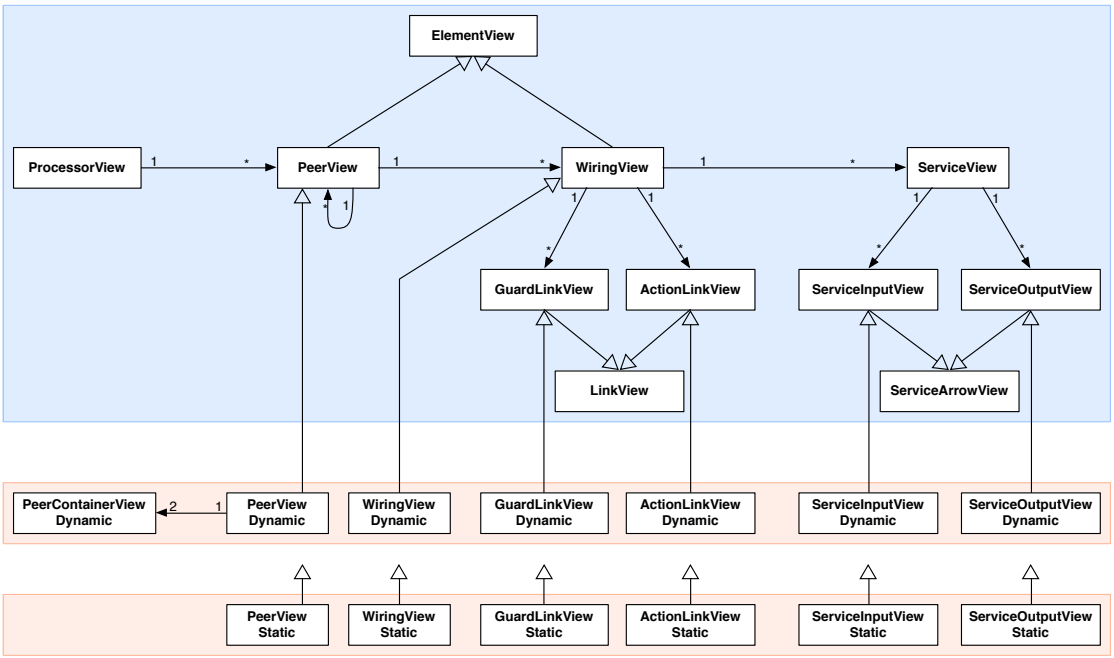


Figure 7.7: UML class diagram presenting an overview of the classes used in the view. The class structure in the base (blue background) corresponds closely to the layout classes (see figure 7.6 for a comparison). Additionally, the view includes separate classes for the service and its input and output links. Deriving from the base classes are the corresponding classes for the dynamic and the static representation, respectively (orange background). Dependencies on JavaFX base classes are omitted for brevity.

it according to its event type. For example, for a *Guard Link Activation*, the associated guard link's operation type is checked. If it is a *Take* operation, the affected entries are removed from the guard link's target container. A *Read* operation does not modify the target container's contents. Conversely, to perform an *Action Link Activation*, the affected entries are added to the associated action link's target container. For each wiring activation, a dedicated EC data structure is maintained and updated according to the events.

The entry IDs are used to identify the correct entries to process, ensuring parity with the actual events. The TIL does not contain any explicit information about the structure of the wiring itself, but only holds a fully-qualified reference to the wiring in the VIL for each wiring activation. Through this reference, the details of the wiring link can be obtained and matched to the events in the TIL. By performing all the events in succession, the real Peer Model processes are reenacted on the intermediate data structure. Combining the information from VIL, TIL and the retracer, a full picture of the Peer Model's state at each point in time is generated. For the retracing to work, it is of utmost im-

portance that the timestamp data is reliable and that the order of events is preserved. Interchanged events can lead to a number of issues. In the best case, events that do not depend on each other are swapped. While this produces an incorrect visualisation of the Peer Model in action, it is not catastrophic. In the worst case however, the retracer is unable to complete its task. For example, if a guard link activation tries to take an entry *before* the action link activation that puts this entry there is executed, the retracer can not find the entry. In this case, the retracing process is stopped and the user is presented with an error message indicating the reason.
As the format specified for TIL files always contains the complete entry, it is necessary to compare entries on this level. The retracer needs to unambiguously decide which entry it is supposed to treat and thus, an equality check between entry objects is mandatory. Two entries that store exactly the same set of data are considered equal. This includes the entries' type as well as their co-data and app-data. Comparing the full data set of two entries for equality is inefficient and error-prone though. Instead, a hash code is generated and stored alongside the entry itself. As the entry class is immutable, the hashing process only needs to be performed once at creation time. Subsequently, entries with differing hash values do not need to be compared further and can safely be regarded as unequal. However, as the implemented hashing is no *perfect hash function*, two entries may have the same hash value even if they are not actually equal. Although this happens very rarely, identical hash values can not serve as a guarantee for equality of the entries. Thus, a full comparison of the entries' data is performed after encountering equal hash values. Only if this check succeeds as well, the entries are considered equal.

Controller

The controller part of the software is heavily coupled to JavaFX. An important feature of JavaFX-based user interfaces are *Properties* and *Bindings*. Properties are generally used instead of simple member variables. Unlike simple Java types, properties allow the attachment of listeners and work in conjunction with bindings. Bindings are used to connect different properties together, either unidirectionally or bidirectionally. If one property changes, the other is automatically updated as well. Bindings may specify arbitrarily complex interactions between properties.
The Peer Model monitoring tool makes heavy use of properties and bindings. For example, a property called *zoomfactor* is specified and bound to the timeline's zoom slider. Additionally, the positions of the event representations in the timeline are in turn bound to the *zoomfactor* property. After the bindings are set up, the slider may be used to change the timeline representation by changing the horizontal scale, effectively zooming in and out. The positions of the events are automatically recalculated whenever the slider value is changed.
However, not all user interaction could be implemented using default widgets, such as buttons and sliders. Fortunately, JavaFX offers full interaction support for all objects

in the scene graph. Mouse and keyboard event listeners can be bound to any element. For example, the view objects of sub peers and wirings are enhanced with mouse drag listeners. Inside a specified callback function that is called whenever the element is dragged, the affected links are rerouted and the corresponding layout data is updated.

7.4 User Interaction

Interaction techniques played an important role in the development of the Peer Model monitoring software. This section discusses what methods of interaction the user is given in order to work with the tool and interactively explore Peer Models and their processes.
The screen is divided into three distinct areas: the *Main View*, the *Sidebar* and the *Timeline*. Figure 7.8 shows an example screenshot where the three areas are highlighted. Each area is discussed from a user interface standpoint in the following sections.

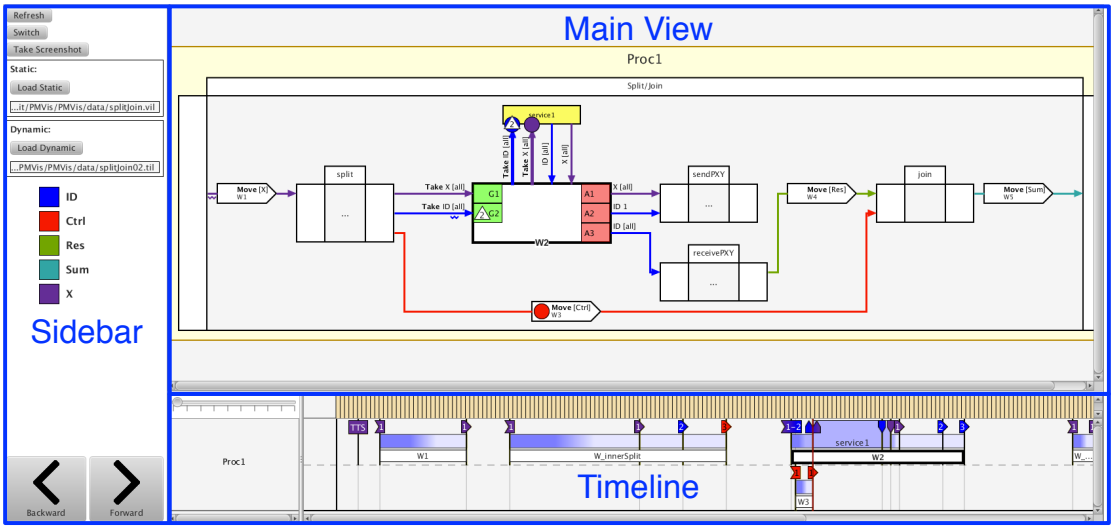


Figure 7.8: Screenshot of the Peer Model monitoring tool in action. The three areas (*Main View*, *Timeline* and *Sidebar*) are highlighted.

Main View

It contains a detailed and customisable representation of the monitored Peer Model. The area that the loaded Peer Model takes up can be arbitrarily large in both vertical and horizontal dimension, depending on its complexity. Thus, the main view is designed like a big canvas. It is pannable by mouse to enable both horizontal and vertical scrolling. The user may use the scrollbars, the mouse scroll wheel or by dragging the mouse while

the right mouse button is pressed.
The main view visualises both the static model and the dynamic events in the Peer Model. Depending on the user's selection, it presents the static structure or overlays it with dynamic events from the TIL file. The two view modes are aptly called *Static View* and *Dynamic View*.
Initially, after loading a static model through a VIL file, the Peer Model is layouted and the static view is presented to the user in the main view. For a detailed discussion on the layouting process itself, refer to chapter 6. If no trace data is loaded, the dynamic view is not available. By default, all peers that reside directly within a processor are displayed expanded. Sub peers are visualised in a collapsed fashion. This ties in well with the Information Visualisation Mantra. After loading, the user is given an overview on the Peer Model, and has the option to expand sub peers at his/her own discretion to explore the model more thoroughly.
After the initial layouting, the user can move around most elements separately at his/her own wish. This includes movement of processors, sub peers and wirings. Repositioning of root peers within processors is not allowed, to keep processors as small as possible. Intuitively, the user can move around elements by clicking and dragging them with the mouse. Necessary layout adjustments, such as rerouting of links are performed automatically afterwards (refer to section 6.6).
When right-clicking on a peer's title bar, an options menu opens from where the user can change settings regarding layouting choices, such as showing/hiding its wirings, expanding/collapsing the peer itself or requesting a full relayout of the peer. In the dynamic view, the user may click on any entry or entry group to open a contextual menu, revealing all details about the entry/entries. Refer to section 5.2 for a detailed discussion on what this feature offers.

Timeline

When the dynamic view is shown, the timeline is visible and gives a temporal overview of the loaded trace file, as previously discussed in section 5.2. Just as the main view, the timeline might grow to any size and is therefore placed inside a scroll pane which allows for horizontal and vertical movement. In addition to using the scrollbars at the sides, the user may use *Mouse Scroll* to scroll vertically and *Shift + Mouse Scroll* to scroll horizontally.
An important characteristic is the fact that the processor labels need to be visible at any time, even when scrolling. Otherwise, the user would easily become lost as he/she would not know which events belong to which processor anymore after scrolling to the right. The same issue arises with the ruler at the top of the timeline. If it would scroll with the rest of the view, it would be disappear as soon as the user scrolled down. To overcome this problem, the processor labels only participate in the vertical scrolling and the ruler only participates in the horizontal scrolling of the timeline. Thus, both the la-

bels and the ruler stay visible and yet are correctly positioned to correspond to the event areas.
To select a specific event, the user simply clicks on the respective event visualisation in the timeline. The indicator is moved to this position and the main view displays the moment in detail. By performing a double click on an event, the main view is additionally panned to focus on the event. For example, when double clicking on a wiring activation event in the timeline, the main view's viewport moves to centre on the wiring where the activation occurred. As an alternative approach, the developer may step through the trace one event after the other by clicking on one of the two buttons in the sidebar (discussed shortly) or by using the keyboard shortcut *Alt* + *Left* or *Alt* + *Right*, respectively. To increase or decrease the zoom factor, the slider at the top left may be used. It offers continuous zooming and the changes are performed interactively while moving the slider.

Sidebar

The sidebar is located at the left edge of the window and provides general settings, such as loading VIL and TIL files, switching between the static and the dynamic view and resetting the complete layout in the main view. The user may load different VIL files at any time, whereas loading TILs is only possible with a VIL in place. Importing a new VIL discards any previously loaded TIL. When trying to import a TIL file that does not correspond to the VIL, the user is presented with an error message. This is also the case if a file could not be loaded because it is malformed. The correspondence between VIL and TIL is done by creating a hash of the VIL file and storing this hash in the TIL file.
A button allows the user to take a screenshot of the full main view in its current state and export it as an image file. The image's filename can be specified in a separate dialog. This feature permits quick sharing of Peer Model visualisations.
For developing the sidebar, the default widgets of JavaFX are leveraged, such as buttons, labels and file choosers. Additionally, the sidebar contains the key table that correlates the occurring entry types to colours (discussed in section 5.2). At the bottom, to the left of the timeline, the two buttons for stepping through the trace are located. They become disabled in the static view or if there is no TIL file loaded.

7.5 Deployment

One of the main goals of the monitoring tool was broad availability. Developers should be able to use it in their preferred development environment. Ideally, it should neither require an installation nor have additional dependencies. Even a suitable tool might not be employed if it is tedious to get going.
In order to keep the friction of setting up low, executable packages for Windows, Mac

and Linux are created. The monitoring tool can be started directly from the package, no installation procedure is required. The only requirement for running the monitoring tool is a Java Runtime Environment (JDK) Version 7+. The package contains a runnable JAR file and an accompanying script. The script is either a batch file (Windows, .bat) or a shell script (Mac and Linux, .sh) and simply runs the JAR file.
Developers intent on using the monitoring tool simply download the package for their platform, unpack it and run the script to start. The frictionless setup facilitates the integration of the monitoring tool into a developer's workflow and encourages its use.
Alternatively, the source itself is packaged and distributed as well. Developers favoring building their application from source may do so with this package. The source code is fully compatible with Eclipse and can easily be imported, as it was the main environment during development. Alternatively, a *Maven* script is provided and may be used for compiling and running the application from the command line.

CHAPTER 8

Evaluation

After developing the monitoring tool, an evaluation was conducted. First, the tool is evaluated based on the requirements established in chapter 4. Afterwards, the developed monitoring tool is evaluated based on a comparison with similar tools for other coordination model approaches. Lastly, various development decisions and their implications are discussed and critically reviewed.

8.1 Evaluation of Requirements

In the following section, each of the requirements established in chapter 4 is evaluated separately to see if the developed monitoring tool fulfils it. Afterwards, a table summarising the results is presented.

Better Debugging Capabilities

Generally speaking, the developed monitoring tool is definitely an asset when it comes to debugging of Peer Models. It helps developers by providing insight into how the Peer Model in question behaves, enabling them to find bugs and act accordingly. The monitoring tool breaks open the requirement of doing *Black Box Debugging* and allows developers to figure out the inner workings of a Peer Model.

Overview

The request for an overview can actually be interpreted in two ways. The first is a structural one: developers want an overview on the general structure of the Peer Model they are working with. While the main view's primary role is to provide a very detailed view, by hiding sub peers and compacting services, it can provide a high-level picture of the model in question. The other way to look at the request

for an overview is a temporal one: the timeline can provide a general picture of the events happening as recorded by the trace. It supplies the developer with an overview on the proceedings along the time dimension. Hence, the developed monitoring tool fulfils the requirement.

Detailed Wirings

The monitoring tool is capable of displaying wirings in great detail. That is the case for both its static structure, including guards, services and actions, and the wiring activations and events pertaining to it.

Interaction of Subsystems

The monitoring tool is capable of visualising Peer Models in their entirety. As such, it can display the subsystems side-by-side and visualise their interaction directly. For example, processors are shown next to each other, and their interaction can be observed by navigating through the trace or by examining the timeline.

Detection of Nonsensical Structures

As a Peer Model evolves, previously needed structures might become obsolete, such as certain wirings or complete entry types. While this generally does not impede the model's functionality, finding and eliminating such structures is beneficial, as it keeps the model concise and easier to maintain. The monitoring tool does not offer any specific features for finding obsolete structures. However, by visualising the system in various degrees of detail, it supports the developer in this task.

Race Conditions

Race conditions present a problem that is hard to overcome. They are difficult to properly detect and visualise. The developed Peer Model visualisation tool does not offer any specific features for dealing with race conditions as that would have gone beyond the scope of this Master's thesis. However, the ability to comprehensibly visualise traces can still help developers figure out what is happening and aid them in detecting race conditions.

Detection of Left-behind Entries

Developers can explore the trace and jump to arbitrary moments in time. By doing so, he/she can easily spot left-behind entries that are not removed automatically and remain in their container indefinitely. While the monitoring tool does not offer any automatic detection capabilities for these use cases, it makes recognising them relatively easy for the developer.

Control Over the Time Dimension

Using the timeline, the developer has full control over what moment should be visualised. The tool offers the possibility to step through the trace forward and

backward, or jump to specific points in time by clicking on the events in the timeline.

Conservative Use of Animation
The interviewed Peer Model developers expressed a dislike for animations that interfere with an efficient workflow. There were definitely possibilities for employing animations, especially regarding the dynamic movements of entries. But to accomplish this requirement, the developed monitoring tool does not contain any animations. Instead, it visualises the events in the form of static *snapshots* that the developer can actively navigate through.

Retention of the Established Visual Notation
The team of eva Kühn had already developed a visual notation for the static Peer Model structure. The interviews showed that this notation should be left intact as much as possible. The monitoring tool tries to follow that advice by superimposing the dynamic events onto the static structure. The established visual notation stays present in all situations and provides the necessary context.

Cross Platform Support
The monitoring tool is developed in Java and JavaFX, which inherently enables deployment onto all major platforms.

Stand Alone
The developed tool has no dependencies other than a working Java Runtime Environment version (JRE) 7 and up. While this does not represent a lot of overhead, as many developers already have that dependency installed beforehand, it still adds setup effort for some developers. For that reason, the requirement is considered not fully met. Alternatively, Java FX provides a tool for packaging fully self-contained applications. This would remove the dependency on an external JRE, as the application would include its own. Unfortunately, this also drastically increases the final application's size and therefore, the deployment option is not utilised.

With some minor exceptions, all posed requirements were satisfied. Table 8.1 summarises how the developed monitoring tool performs regarding the fulfilment of the requirements. The software helps developers in many of their regular tasks regarding debugging and monitoring. However, race conditions still present an issue and no specific tools exist to help Peer Model developers in this task. While the monitoring tool can help with race conditions by visualising the events, the developer still has to detect them himself/herself. No feature for automatic detection exists. Although not as severe of an issue, the detection of nonsensical structures is also left to the developer. Both of these features are considered out of scope of the current monitoring tool and are considered potential future works. The dependency on a JRE presents a small drawback in

reaching the goal of a fully stand-alone tool. However, it is a balanced compromise in order to allow easier cross platform development and deployment. Choosing C++/Qt would not have imposed such a requirement on the user. But it would have raised the complexity for deployment, as dedicated builds for each platform, alongside platform specific testing, would have been necessary.

Requirement	Performance
Better Debugging Capabilities	✓
Overview	✓
Detailed Wirings	✓
Interaction of Subsystems	✓
Detection of Nonsensical Structures	✓–
Race Conditions	✓–
Detection of Left-behind Entries	✓
Control Over the Time Dimension	✓
Conservative Use of Animation	✓
Retention of the Established Visual Notation	✓
Cross Platform Support	✓
Stand Alone	✓–

Table 8.1: Overview on the requirements and how well the developed monitoring tool fulfils them. ✓ represents full compliance, ✓– compliance with some reservations.

8.2 Evaluation by Comparison with Other Tools

In the following section, the developed monitoring tool for the Peer Model is compared to similar tools from other coordination model approaches. In order to compare the tools in a meaningful way, criteria pertaining to specific features are established. Each defines a central question that the tool must answer positively in order to fulfil the criteria. Afterwards, the tools are introduced and evaluated. Lastly, a summary on the results is presented.

Criteria

Graphical Notation - Static

A graphical notation describing the static structure of the underlying system is an important feature for the evaluated tools. Visual representations aid developers in their understanding of the system and do a significantly better job than textual representations. Visual notations based on graphs and flowcharts are a perfect fit

for describing the interaction between components, which is a central property of coordination models. The main question for deciding whether the tool meets this criteria is: “does the tool establish a graphical notation for describing the static structure of the system?“

Graphical Notation - Dynamic

Related to the previous one, this criteria addresses the existence of a visual notation for the dynamic events. The Peer Model does this by superimposing the dynamic event visualisations onto the static structure, but other approaches are equally valid. The important aspect is whether the tool offers a visual representation of the running system and hence, the question is: “does the tool establish a graphical notation for the dynamic events happening within the system?“

Time Control

During simulation or exploration of a trace, complete control over the time dimension is crucial for the user. The ability to stop time, forward and rewind as well as jumping to other moments in the execution chain are required interaction techniques the tool has to offer in order to fulfil this criteria. The central question is: “does the tool give the user complete control over the time dimension to explore a given set of events?“

Temporal Overview

While a detailed visualisation of a system’s state at a certain moment in time is essential, sometimes users prefer a more outlined presentation of the unfolding events. In this case, an overview that compresses the time dimension and visualises a set of consecutive events is needed. Tools looking to fulfil this criteria need to answer the following question positively: “does the tool offer a representation of the events in a more compressed form that allows for a temporal overview?“

Tracing Format

The presence of a trace format enables post-mortem debugging of running systems. Traces can be passed around and shared between developers, used to describe bugs and act as documentation. Even when simulation tools for a coordination model approach exist, having a dedicated trace is beneficial. The central questions for this criteria are: “does the coordination model approach offer a standardised format for logging and storing event data?“ and “can the evaluated tool load and visualise these traces?“

Automatic layouting

All evaluated approaches feature a static visual notation that resembles graph structures or flowcharts. Generally speaking, all share the notion of *elements* that

represent some form of system component and *links* that connect them. While their semantics are entirely different for each approach, the overall presentation of models is similar. To relieve users from having to manually arrange elements or route links, automatic layouting capabilities can be employed. Thus, the central question is: “does the tool offer automatic layouting capabilities for positioning elements and routing links?“

Layout Customisation

The counterpart to automatic layouting is the manual (re-)arrangement of elements by the user. While having the option to delegate the layouting task to the tool is beneficial, sometimes the user wants to change the precalculated layout. A reason for this can be a semantic affiliation of nodes that should be grouped together, which the automatic layout algorithms can not apprehend. For a tool to fulfil this criteria, it has to answer the following question: “Does the tool let the user manually place and reposition elements?“

Information Hiding

Selective information hiding is an important technique for simplifying complex coherences. It serves to fulfil the *Information Seeking Manta* by enabling users to focus on currently important parts while hiding unneeded complexity. The tool is evaluated whether it can simplify the visualised system by selectively hiding components, either interactively or in an automated fashion. Especially when dealing with larger systems, being able to filter unrelated components is crucial for an efficient workflow. This criteria is met when the tool in question is capable of hiding complexity from the user in order to facilitate the understanding. The question is thus: “does the tool offer functionality to hide currently unimportant parts of the visualisation?“

Tools

Not all discussed coordination model approaches provide tools that serve the same role as the monitoring tool for the Peer Model approach. However, tools for simulation or modelling share important aspects with monitoring, such as providing a visual representation of the underlying model and the exploration of event chains. For some approaches, such as BPMN, a multitude of tools exist for various purposes. To keep the evaluation to a manageable scope, one tool for each approach is chosen, based on its dissemination and its similarity in functionality to the tool developed in this Master’s thesis. In the following section, the evaluated tools are described in terms of their focus and role within their respective coordination model environment. Afterwards, each tool is evaluated based on the previously established criteria.

Peer Model Monitoring Tool

The Peer Model monitoring tool fulfils all stated requirements. The visual notation for the static structure of a Peer Model has been established prior to this Master's thesis and its dynamic counterpart was introduced in chapter 5. The timeline serves both as the temporal overview and as the interface for controlling the execution of the trace. A file format for Peer Model traces was developed in chapter 7 and can be imported by the tool. The visualised model is automatically layouted upon loading and can be modified afterwards by dragging and moving wirings and peers. Lastly, the developed tool supports collapsing of sub peers to hide their inner workings and a compact representation for services to save screen space.

Reo - Extensible Coordination Tools

The *Extensible Coordination Tools* (short: *ECT*) are developed as a set of Eclipse plugins to aid developers when working with Reo [24]. The tools offer graphical editing of connectors, code generation, simulation and animation of systems, among other features. Figure 8.1 shows a screenshot of ECT when modelling a synchronous communication channel that can be opened and closed.

ECT can animate models by generating flash videos of the proceedings. These

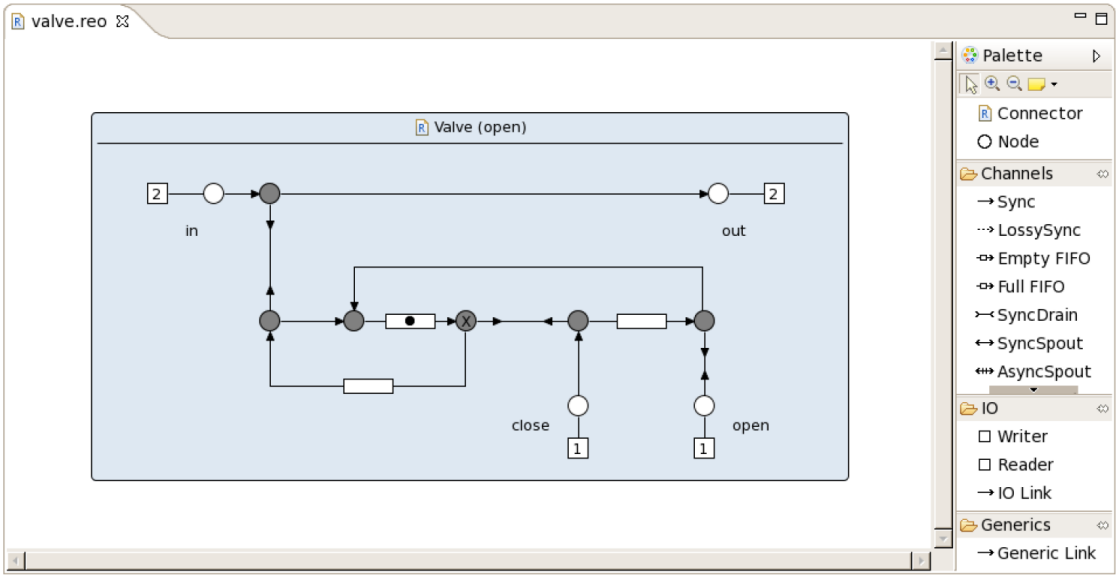


Figure 8.1: A screenshot of ECT. The center view show a model of a synchronous communication connector. The palette on the right is used for selecting components to add to the model. Image taken from [5].

videos superimpose dynamic information onto the static model. Colour is used

to differentiate between active an inactive components. Data moving across connectors is animated by small, coloured circles. While the animated views are informative, the user does not have proper control over the animation. ECT provides no functionality for interacting with the animations other than starting and stopping them. Users can not rewind or jump to certain events in the animation. Furthermore, ECT does not provide an overview on the occurring events. The model's proceedings need to be observed through the animation.
While ECT offers modelling capabilities using intuitive drag and drop mechanics, the users are tasked with layouting the placed components themselves. No functionality for automatic layouting exists. However, ECT is capable of selectively hiding the inner workings of complex connectors and therefore meets the *Information Hiding* criteria.

Petri Nets - CPN Tools

CPN Tools is a tool for editing, simulating and analysing coloured petri nets. It allows developers to design coloured petri nets and features syntax checking and code generation. A simulator can be used to generate scenarios and see them unfold live. Similar to the approach chosen for the Peer Model monitoring tool, CPN Tools superimpose dynamic event data onto the static structure. The user may control the simulation similar to how video and music players are controlled. However, no temporal overview on the occurring events is provided and no way to jump to specific moments in time exists. If the user accidentally moved too far forward, the only way to go back is to rewind the simulation to the initial state and start over. Figure 8.2 shows a screenshot of the CPN Tools simulator in action.
Users can freely position and rearrange places, transitions and connections. Horizontal and vertical *Gridlines* may be placed to help with aligning elements. However, no functionality for automatic layouting of elements or routing of links exists.
The CPN Tools offers capabilities for showing and hiding parts of the model. The user can specify which parts to visualise and arrange them using *pages* and *binders*. The tool is capable of generating files containing the events of a simulation, called *simulation reports*. However, while generating and exporting these event traces is possible, importing them again is not. User obtaining simulation reports can only explore them manually and can not visualise them again using the CPN tools.

UPPAAL

UPPAAL includes its own tool environment for modeling, validation and verification. Command line tools as well as a graphical software interface exist. UPPAAL is split into three main parts: a description language, a simulator and a model-checker. Using the description language, developers specify the structure

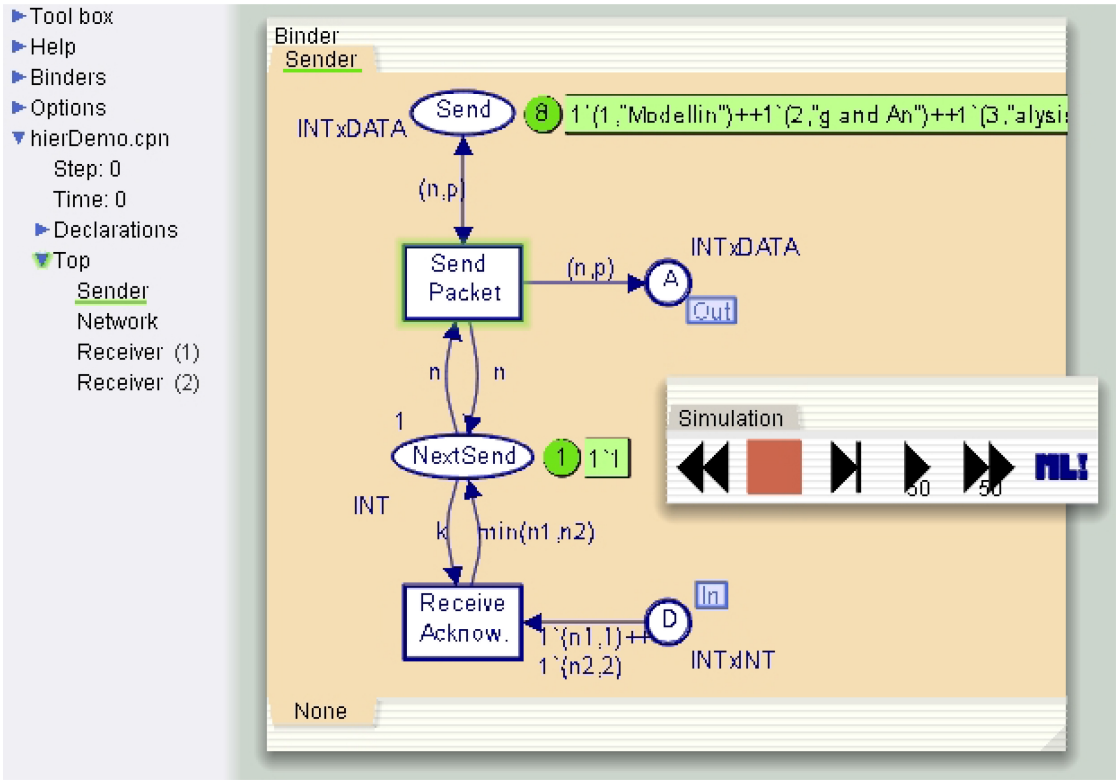


Figure 8.2: Screenshot of the CPN tools simulator. The main window shows only a part of the complete model (the *sender*). The dynamic state and events (indicated in green) are visualised on top of the static CPN structure. Note how the simulation controls resemble video or audio player controls. Image taken from [71].

of the model and describe its behaviour. As an alternative to describing the system using the textual format, UPPAAL also provides a graphical modeller.

The model-checker is capable of answering questions concerning reachability and invariance. By exploring all possible states given an initial configuration, the user may check for certain properties of the system. While the model-checker explores the complete set of possible sequences, the simulator allows the interactive and graphical visualisation of a single sequence of states. Figure 8.3 shows a screenshot of the UPPAAL simulator in action.

UPPAAL fulfils most of the specified criteria. It supports a static and a dynamic notation and offers controls over the time dimension. A temporal overview is provided in form of a separate view and an explicit tracing format called *XTR* is defined. However, it requires users to manually place nodes and does not offer automatic layouting capabilities. Furthermore, its flexibility in terms of visual encapsulation and hiding of unimportant parts of the model is limited. While it

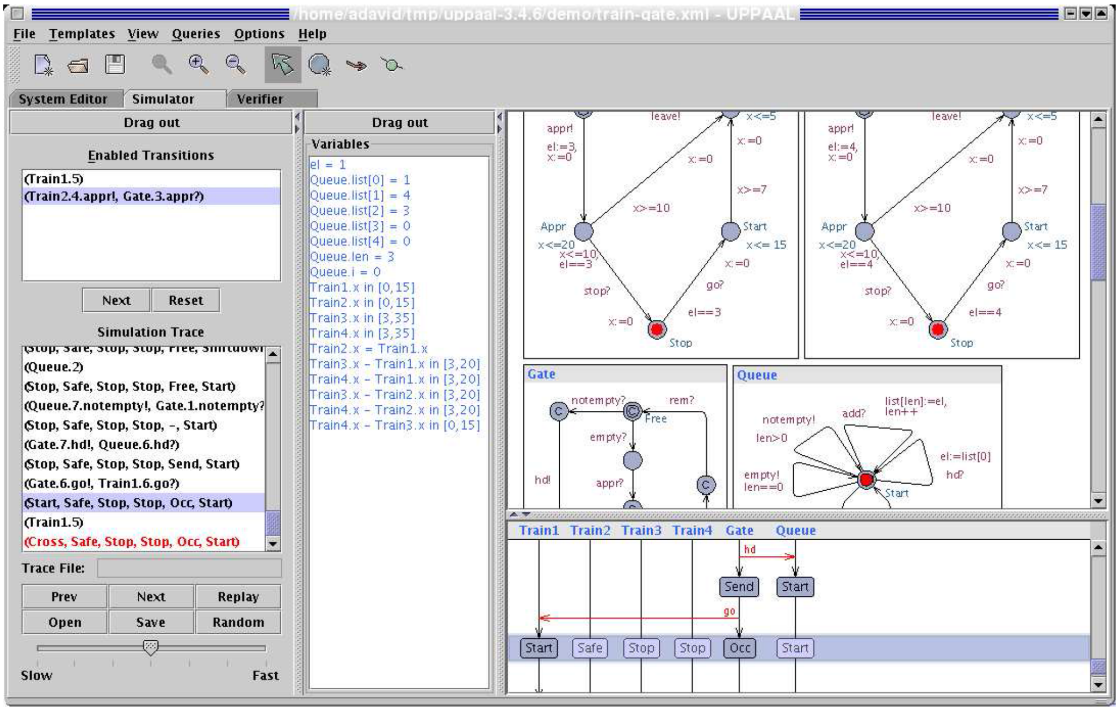


Figure 8.3: The visual simulator of UPPAAL. The main view on the right shows the model in detail, while the bottom view visualises a series of events, similar to an UML sequence diagram. The controls on the bottom left are used to navigate the trace. Image taken from [7].

supports scrolling and zooming in the main view, the general representations can not be collapsed or otherwise condensed.

WS-BPEL - EA4B

EA4B is very similar in its design and architecture to the Peer Model monitoring tool developed in this Master's thesis. Gravel, Xiang and Jianwen developed EA4B for post-mortem debugging of BPEL-based systems [32]. They define a logging format used to record the events occurring inside a BPEL-based system. These logs can be read by EA4B and visualised in conjunction with the corresponding BPEL structure definition. Figure 8.4 shows a screenshot of EA4B.

Compared to the Peer Model monitoring tool, EA4B lacks a proper temporal overview. Users are required to step through the trace step-by-step to gain insight into the proceedings. Additionally, the tool does not provide users with complete control over the time dimension. The visual trace execution may only be explored in a linear fashion. Reversing the trace to a previous moment or randomly jumping to other events is not possible. The nodes in the main view of

EA4B are automatically layouted, but can not be repositioned or modified to hide unimportant details.

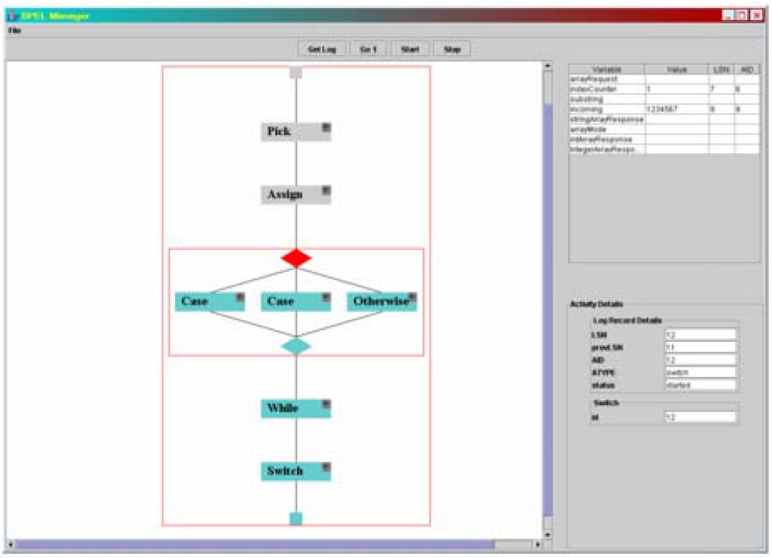


Figure 8.4: Screenshot of EA4B, used to visualise execution traces from BPEL business processes. Image taken from [32].

BPMN - Visual Paradigm

Visual Paradigm offers tools for modelling, simulating and analysing business models based on BPMN [89]. The tools support the BPMN standard notation for visual elements established by the *Object Management Group* (*OMG*). Figure 8.5 shows a screenshot of the Visual Paradigm BPMN simulator. A number of automatic layouting algorithms are supported that can be applied to BPMN models. Users may also rearrange elements and hide process details to suit their needs. Thus, the criteria *Automatic Layouting*, *Layout Customisation* and *Information Hiding* are fulfilled. However, the BPMN tools of visual paradigm do not support a trace format required to fulfil the *Tracing Format* criterion.

Users may use the provided BPMN simulator tool to test models. However, the simulator is aimed towards long-running business processes. Therefore it does not provide the user with a lot of control over the execution. While the simulation can be paused, rewound and the execution speed can be varied, further interaction possibilities for exploring a simulation are not available. After a simulation has finished, the user is provided with additional charts regarding resource usage and identification of bottlenecks. The static notation is enhanced with additional information for each process, which serves to fulfil the *Temporal Overview* criterion.

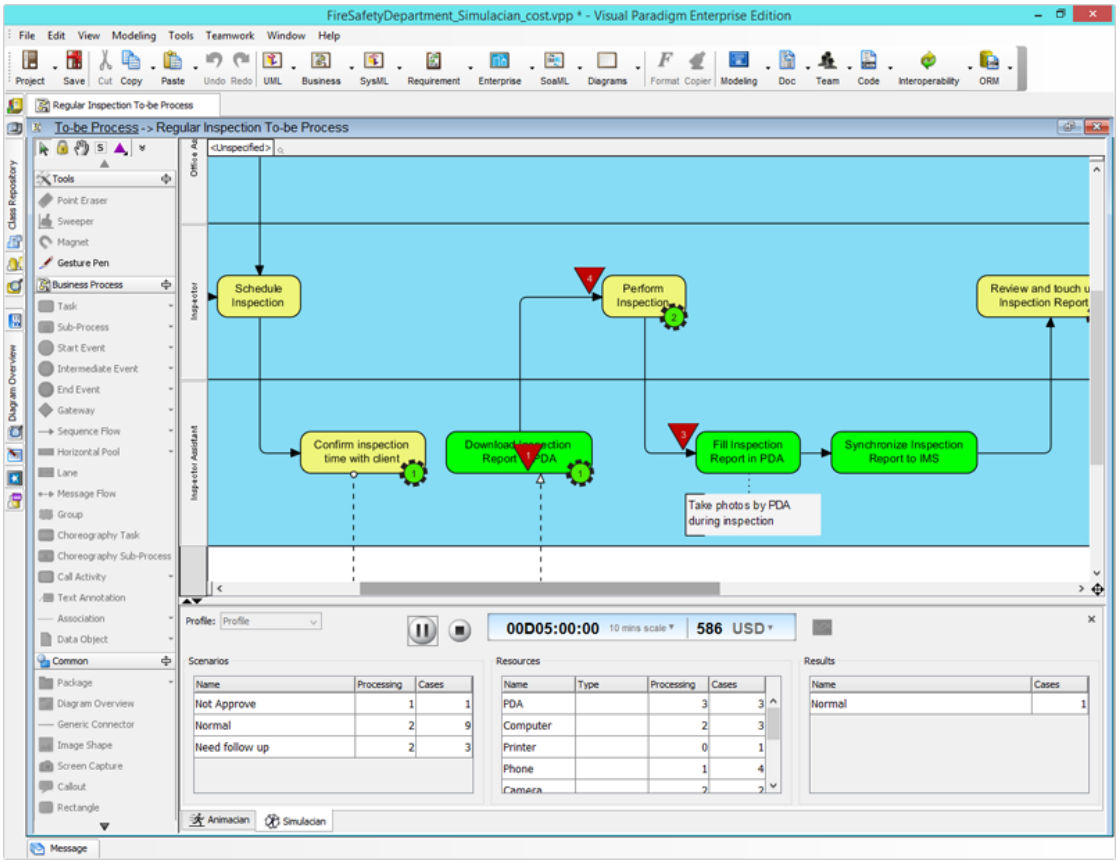


Figure 8.5: Screenshot of the Visual Paradigm BPMN simulator. Image taken from [89].

Summary

Table 8.2 presents the evaluation results and summarises the findings. The developed Peer Model monitoring tool fulfils all established criteria. Each of the other evaluated tools showed weak points in certain areas. The results reinforce the conviction that the developed tool is a feasible asset for monitoring and debugging Peer Models.

8.3 Development Evaluation

After discussing the results in the light of the posed requirements and by comparing the monitoring tool to other tools, this section evaluates the taken decisions and chosen methodologies through the course of the whole development cycle.

Interviews

	Peer Model Mon. Tool	Reo ECT	Petri Nets CPN Tools	UPPAAL UPPAAL	BPEL EA4B	BPMN VP
Static Notation	✓	✓	✓	✓	✓	✓
Dynamic Notation	✓	✓	✓	✓	✓	✓
Time Control	✓	✗	✗	✓	✗	✗
Temporal Overview	✓	✗	✗	✓	✗	✓
Tracing Format	✓	✗	✗	✓	✓	✗
Automatic Layouting	✓	✗	✗	✗	✓	✓
Layout Customisation	✓	✓	✓	✓	✗	✓
Information Hiding	✓	✓	✓	✗	✓	✓

Table 8.2: Summary of the results in matrix form. For each tool, the previously established criteria were evaluated and marked by a ✓ or a ✗, depending on whether or not the tool could fulfil the criteria.

Before starting the development of the monitoring tool, personal interviews with developers working with the Peer Model were conducted. They delivered invaluable insight into the issues the developers currently face. The interviews were mandatory for making sure that the completed software actually becomes an asset for Peer Model developers and fulfils their requirements. Without them, no proper requirement analysis would have been possible.

Technology Choices

Overall, the chosen technologies were a good fit for the development of the Peer Model monitoring tool. JavaFX offered all the necessary features needed in terms of user interface and drawing capabilities. Standard controls were easy to implement and the scene graph and its related components proved to be appropriate for the given tasks. JavaFX's binding system allowed for a non-invasive way of linking user interface controls to the data model, supporting the *Separation of Concerns* design goal.

Jackson also turned out to be a capable library to fulfil the given requirements. Although the roundabout way of having to develop an intermediate data structure and performing the reference replacements for peers, wirings and containers manually was cumbersome, other libraries would have arguably required the same approach. Otherwise, Jackson behaved as expected and was pleasant to use.

The general architectural approach of leveraging the MVC pattern proved to be a good decision. The parts of the monitoring tool are adequately decoupled. Especially the JavaFX dependency was kept within the view-related parts of the software. Components of the software can be replaced with relative ease, should the need arise. Furthermore, components such as the layouting framework can be

extracted to be used within other software projects.
As no tested graph layouting library was able to meet the posed requirements, developing the Peer Model layouting framework from scratch was the only viable solution. And while it was a good deal of work, it was the only way to ensure the algorithms could be modified to fit the Peer Model layouting use case.

Peer Model Representation

Leaving the already established visual notation for Peer Models intact as much as possible was a beneficial decision. With only slight modifications, it was possible to superimpose the dynamic events resulting from the trace file onto the static structure. Developers were immediately familiar with the visualisation as the static structure provides the necessary context. And while the dynamic notations were novel, their purpose and intent was reasonably intuitive.
Using *Overview+Detail* as the central visualisation technique proved to be an excellent fit for the complexity of the data and the requirements of developers. The main view allows focussing on a single moment, while the timeline provides a temporal context that embeds the focus and offers exploration capabilities. Developers can quickly make sense of a Peer Model and its corresponding traces.
Providing context and time control functionality with the timeline worked out well. A timeline is a reasonably well known concept used in many different applications, such as animation software, audio editors and video cutting software. Developers generally quickly grasp the concept of encoding the time dimension onto an axis and have little problems using the timeline.

File Formats and Workflow

The monitoring tool relies on being supplied with VIL and TIL (trace) files. Both VIL and TIL were developed using the JSON data format. Only with the availability of both, the tool can demonstrate its strengths. The proposed workflow specifies that the VIL file is generated from the DSL file prior to its use. However, this additional preprocessing step presents an impediment to an efficient workflow, even though the conversion process itself is automated and only has to be performed once for a specific model. Ideally, the developer should be able to load the DSL directly into the monitoring tool. However, implementing a full PM-DSL parser would have gone beyond the scope of the thesis and is therefore left for future work.
The development of the format for Peer Model traces (and the resulting TIL file format) was a cornerstone for the monitoring tool. The tool can only visualise the data that is provided by the trace. Thus, great care was exercised when deciding on what data should be captured and how the data should be structured. Close collaboration with the Peer Model developers was vital for reaching conclusions and developing a stable and suitable trace format. JSON is a human-readable text

format, so in cases when the visual monitoring tool can not be used, developers can still explore the trace and gain insights. The TIL structure is straight forward and easy to learn for developers previously unfamiliar with the format. Similar to the VIL workflow, taking the circuit through a dedicated TIL file can be cumbersome in certain tasks. While the file based approach is vital for debugging and monitoring Peer Models that are used in the field by examining historical traces, it has disadvantages during the development phases. Having to explicitly load new traces (and altered VIL files) while developing a Peer Model is a suboptimal solution. Ideally, the monitoring tool should support the direct coupling to a running Peer Model system without the need for dedicated, intermediate files. This potential extension is discussed in chapter 9.

Layouting

The developed layouting framework proved to be an excellent fit. While there was no specific research conducted on the layouting results in order to objectify the framework's suitability, developers were generally content with the produced layouts and did not feel the need for a lot of re-arrangement of elements. Many of the encountered Peer Models did not require any manual layout adjustments at all. In chapter 6, criteria for efficient graph layouts were introduced. Peer model visualisations are similar to graph structures in nature and are therefore assessed with the same criteria. The developed layouting framework aims to specifically reduce certain aspects, such as minimising edge bends through penalising turns in the link routing step, or the employment of a packing algorithm to reduce the overall size of the resulting layout, thereby reducing edge lengths as well. Extensive formal evaluation is postponed to a future work, but intuitively, the layouting framework successfully meets the proposed criteria.

Interaction and Exploration Features

One of the central goals of the monitoring tool was to allow the efficient exploration of traces gathered from running Peer Model systems. Being able to focus on data that is of interest while hiding other data is crucial, as the amount generated by Peer Models quickly goes beyond the constraints of what a developer can grasp at a time. The monitoring offers multiple ways of selectively hiding or showing information. One such feature is the ability to collapse peers to hide their inner workings. Another is the compression of services into their compact form to reclaim valuable screen space. Furthermore, developers can selectively gather information about individual entries simply by clicking on them. And last but not least the ability to focus on and skip between moments in time with the timeline. These and other features enable developers to quickly skim through the trace, find the aspects that are of interest to them and work out what they want to know.

CHAPTER 9

Future Work

In the following chapter, potential areas for future works are discussed. This includes the development of complementary visual tools covering other aspects of the Peer Model development process as well as additional features and functionalities for the monitoring software itself.

9.1 Real-Time Monitoring

One of the main potential areas of future research is the extension of the current tool beyond *Post-Mortem Debugging*. The general idea is that the visualisation software hooks directly into a running Peer Model system, without the indirection of using a TIL file. The tracing format was built with this potential requirement in mind and allows for sending events as soon as they occur without buffering or delay. The monitoring tool must receive the events in real-time, process them and visualise them immediately. Facilities for controlling this behaviour must be tied in with the already established interaction features. This approach would allow true real-time monitoring and debugging of Peer Models. Developers could attach directly to running models and explore the processes while they are happening, allowing for a tremendous increase in productivity.

9.2 Modelling

While visual modelling software for other distributed systems exist, the Peer Model approach still lacks such a tool. Peer models are currently designed and modelled with the DSL or by directly writing code in one of the programming languages supported by the Peer Model. While this is a feasible approach, developing a visual tool for designing Peer Models would unify the whole process and bridge the gap between the textual

notation given by the DSL and the visual notation. Furthermore, developers new to the Peer Model could start designing without having to learn the DSL syntax.

9.3 Simulation

Strongly tied to the notion of a visual modelling tool is the idea of developing a simulation framework. After having designed a model, the developer could make use of simulation facilities for testing and debugging, without the need for deployment for the majority of development. A simulator would close the cycle and, together with the modelling and the monitoring tool, provide a full development environment that is based on the visual Peer Model notation.
A case could be made for incorporating all three aspects into one coherent software package.

9.4 Peer Model Changes

An important aspect of the Peer Model is its still ongoing transformation. As new use cases and application possibilities emerge, the Peer Model specification itself evolves. At the time of writing, new features are still introduced regularly and the accompanying tools need to incorporate these features as well to stay relevant. As with all software, the monitoring tool will have to prove its worth in real-life use cases. New requirements might become manifest that have not yet been considered. Moreover, implemented features might not turn out to be as useful as anticipated and could need adjustment.

9.5 Large Traces

Future Peer Model systems might grow larger in all their dimensions than currently existing ones. Especially in terms of trace sizes, this could pose a problem. The present monitoring tool might not be capable of processing excessively large TIL files. It might also cause issues concerning the trace exploration. A large trace with a great number of events becomes cumbersome to navigate for the user. A potential future work could add features for preprocessing trace files in order to selectively choose parts of a full trace for a more detailed analysis. Instead of loading the full trace with all its events, the user could be given a quantitative overview first. Based on that, he/she could specify a timeframe to isolate areas of interest, which would then be visualised in the usual way. The feature represents another application of the information seeking mantra of providing users with an overview first and offer details on-demand.

9.6 Workflow and Features

The inclusion of a PM-DSL parser also presents a worthwhile goal. It would eliminate the need for VIL files completely, as the user could directly load DSL files into the monitoring tool. This would also improve the workflow while designing and testing Peer Models, where the static structure is in a constant state of flux.
As discussed in the previous section, the current tool does not provide any features for dealing with race conditions. Depending on how often Peer Model developers encounter such issues, the monitoring tool could be extended to offer advanced assistance in tracking down and eliminating race conditions. A similar case could be made regarding features for detecting left-behind entries and recognising obsolete structures. In general, future works could analyse the status quo in monitoring tool utilisation and explore areas of potential improvement.

9.7 Peer Model Layouting

Developing the layout framework for the Peer Model visualisation was a a major contribution. A potential future work could involve the generalisation of the developed framework for the use in other applications, for example for documentation purposes. This could also involve various performance improvements, such as exploring better suited pathfinding algorithms.

CHAPTER 10

Conclusion

The goal of this Master's thesis was the development of an interactive, visual monitoring tool for the Peer Model approach in order to help developers with their debugging and monitoring tasks. In order to identify issues with the current processes and how a visual tool can assist developers, personal interviews were conducted. The interviews served as a basis for the subsequent requirement analysis which in turn determined the focus of development.
As the Peer Model approach is relatively novel, the facilities supporting developers with their monitoring and debugging tasks were in their infancy. Among other things, no agreed upon specification for a trace format existed. Thus, a JSON based file format (called *TIL*) for recording and persisting occurring events inside running Peer Model systems was developed. A visual notation for the static structure of Peer Models had already been developed prior to this Master's thesis and was utilised for visualisations beforehand. In order to maintain its conceptual expertise, the monitoring tool visualises TIL files superimposed onto the established notation. In general, the information visualisation concept of *Overview+Detail* played a key role in the design of the monitoring tool. A specific moment in time is illustrated in full detail alongside a temporal overview that provides the necessary context. Using interaction techniques, the user can jump to different moments in time and actively explore the underlying data.
Automatic layouting of Peer Model visualisations was an important part of this work. To relieve users from manually positioning elements and routing links between them, a layouting framework capable of this task was developed. As the visual Peer Model notation is similar to graph structures, algorithms stemming from graph theory were employed and served as building blocks for the framework.
Finally, the developed tool was critically evaluated and compared to the established requirements. Potential future works were discussed and an outlook on further developments was provided.

Bibliography

[1] Nadine Abboud, Martin Grötschel, and Thorsten Koch. Mathematical Methods for Physical Layout of Printed Circuit Boards: an Overview. *OR Spectrum*, 30(3):453–468, 2008.

[2] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 74–89, New York, NY, USA, 2003. ACM.

[3] Alexandre Alves, Assaf Arkin, et al. Web Service Business Process Execution Language (WSBPEL) 2.0. *Organization for the Advancement of Structured Information Standards (OASIS)*, 2007.

[4] FARHAD ARBAB. Reo: a Channel-based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14:329–366, 6 2004.

[5] Farhad Arbab, Christian Koehler, Ziyan Maraikar, Young-Joo Moon, and José Proença. Modeling, Testing and Executing Reo Connectors with the Eclipse Coordination Tools. *Tool demo session at FACS*, 8, 2008.

[6] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[7] Gerd Behrmann, Alexandre David, and KimG. Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004.

[8] BPEL Designer Project. `https://eclipse.org/bpel/`, September 2014.

[9] BPMN Standard. `http://www.omg.org/spec/BPMN/2.0/`, September 2014.

[10] Mihal Brumbulli and Joachim Fischer. Simulation Visualization of Distributed Communication Systems. In *Proceedings of the Winter Simulation Conference*, WSC '12, pages 248:1–248:12. Winter Simulation Conference, 2012.

[11] Rodger Burmeister. ReActor: A Notation for the Specification of Actor Systems and Its Semantics. In *Software Engineering'13*, pages 127–142, 2013.

[12] Paola Campadelli, Roberto Posenato, and Raimondo Schettini. An Algorithm for the Selection of High-Contrast Color Sets. *Color Research and Application*, 24(2):132–138, 1999.

[13] S.K. Card, J.D. Mackinlay, and B. Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Interactive Technologies Series. Morgan Kaufmann Publishers, 1999.

[14] Robert C. Carter and Ellen C. Carter. Color Coding for Rapid Location of Small Symbols. *Color Research and Application*, 13(4):226–234, 1988.

[15] Andy Cockburn, Amy Karlson, and Benjamin B. Bederson. A Review of Overview+Detail, Zooming, and Focus+Context Interfaces. *ACM Comput. Surv.*, 41(1):2:1–2:31, January 2009.

[16] MICHAEL K. COLEMAN and D. STOTT PARKER. Aesthetics-Based Graph Layout for Human Consumption. *Software: Practice and Experience*, 26(12):1415–1438, 1996.

[17] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[18] B. Craft and P. Cairns. Beyond Guidelines: What Can We Learn from the Visual Information Seeking Mantra? In *Information Visualisation, 2005. Proceedings. Ninth International Conference on*, pages 110–118, July 2005.

[19] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web Services Web: an Introduction to SOAP, WSDL, and UDDI. *IEEE Internet computing*, 6(2):86–93, 2002.

[20] Josep Díaz, Jordi Petit, and Maria Serna. A Survey of Graph Layout Problems. *ACM Comput. Surv.*, 34(3):313–356, September 2002.

[21] Paul S. Dodd and Chinya V. Ravishankar. Monitoring and Debugging Distributed Realtime Programs. *Softw. Pract. Exper.*, 22(10):863–877, October 1992.

[22] Peter Eades, Xuemin Lin, and W. F. Smyth. A Fast Effective Heuristic for the Feedback Arc Set Problem. *Information Processing Letters*, 47:319–323, 1993.

[23] Eclipse Foundation. https://www.eclipse.org/, September 2014.

[24] Extensible Coordination Tools. http://reo.project.cwi.nl/reo/wiki/Tools, September 2014.

[25] Rainer Fehling. A Concept of Hierarchical Petri Nets with Building Blocks. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1993*, volume 674 of *Lecture Notes in Computer Science*, pages 148–168. Springer Berlin Heidelberg, 1993.

[26] C.J. Fisk, D.L. Caskey, and L.E. West. ACCEL: Automated Circuit Card Etching Layout. *Proceedings of the IEEE*, 55(11):1971–1982, Nov 1967.

[27] HERBERT J Fleischner, DENNIS P Geller, and F Harary. Outerplanar Graphs and Weak Duals. *J. Indian Math. Soc*, 38:215–219, 1974.

[28] Thomas M. J. Fruchterman and Edward M. Reingold. Graph Drawing by Force-Directed Placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991.

[29] Chris Glasbey, Gerie van der Heijden, Vivian FK Toh, and Alision Gray. Colour Displays for Categorical Images. *Color Research & Application*, 32(4):304–309, 2007.

[30] Graphviz - Graph Visualization Software. http://www.graphviz.org/, September 2014.

[31] JGraphX. https://github.com/jgraph/jgraphx, September 2014.

[32] A. Gravel, Xiang Fu, and Jianwen Su. An analysis tool for execution of bpel services. In *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on*, pages 429–432, July 2007.

[33] Paul Green-Armytage. A Colour Alphabet and the Limits of Colour Coding. *JAIC-Journal of the International Colour Association*, 5, 2010.

[34] Eric Haines. Point in Polygon Strategies. *Graphics gems IV*, 994:24–26, 1994.

[35] Philipp Haller and Martin Odersky. Scala Actors: Unifying Thread-based and Event-based Programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

[36] Carl Hewitt. Actor Model of Computation: Scalable Robust Information Systems. *arXiv preprint arXiv:1008.1459*, 2010.

[37] Yifan Hu. Efficient, High-Quality Force-Directed Graph Drawing. *Mathematica Journal*, 10(1):37–71, 2005.

[38] Walter L. Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical report, College of Computer Science, Northeastern University, 1995.

[39] J. Huselius, Daniel Sundmark, and H. Thane. Starting Conditions for Post-Mortem Debugging Using Deterministic Replay of Real-Time Systems. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 177–184, July 2003.

[40] Jackson JSON Processor. `http://wiki.fasterxml.com/JacksonHome`, September 2014.

[41] Sakait Jain and Hae Chang Gea. PCB Layout Design Using a Genetic Algorithm. *Journal of Electronic Packaging*, 118(1):11–15, 1996.

[42] JavaFX. `http://docs.oracle.com/javase/8/javase-clienttechnologies.htm`, September 2014.

[43] Kurt Jensen. Coloured Petri Nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer Berlin Heidelberg, 1987.

[44] David S Johnson and M Garey. Computers and Intractability: a Guide to the Theory of NP-Completeness. *Freeman&Co, San Francisco*, 1979.

[45] JUNG - Java Universal Network/Graph Framework. `http://jung.sourceforge.net/`, September 2014.

[46] Rajesh K Karmani, Amin Shali, and Gul Agha. Actor Frameworks for the JVM Platform: a Comparative Analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20. ACM, 2009.

[47] Richard M. Karp. Reducibility among Combinatorial Problems. In RaymondE. Miller, JamesW. Thatcher, and JeanD. Bohlinger, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer US, 1972.

[48] Kenneth L. Kelly. Twenty-Two Colors of Maximum Contrast. *Color Engineering*, 3(26):26–27, 1965.

[49] eva Kühn, Stefan Craß, and Thomas Hamböck. Approaching Coordination in Distributed Embedded Applications with the Peer Model DSL. In *Proceedings of the 40th Euromicro Conference series on Software Engineering and Advanced Applications (SEAA)*, 2014.

[50] eva Kühn, Stefan Craß, Gerson Joskowicz, Alexander Marek, and Thomas Scheller. Peer-Based Programming Model for Coordination Patterns. In Rocco De Nicola and Christine Julien, editors, *15th International Conference on Coordination Models and Languages (COORDINATION), held as part of the 8th International Federated Conference on Distributed Computing Techniques (DisCoTec)*, volume 7890 of *Lecture Notes in Computer Science*, pages 121–135, Florence, Italy, June 3-5 2013. Springer.

[51] eva Kühn, Stefan Craß, Gerson Joskowicz, and Martin Novak. Flexible Modeling of Policy-Driven Upstream Notification Strategies. In *29th Symposium On Applied Computing (SAC)*, Gyeongju, Korea, March 24-28 2014. ACM.

[52] 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)*, Turku, Finland, 2015. IEEE. to appear.

[53] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[54] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3/e*. Pearson Education India, 2012.

[55] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[56] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[57] Christian Lipski, Kai Berger, and Marcus Magnor. vIsage - A Visualization and Debugging Framework for Distributed System Applications. In Vaclav Skala, editor, *Proc. WSCG 2009*, volume 2009, pages 1–7, Plzen, Czech Republic, February 2009. UNION Agency – Science Press.

[58] Shakuntala Miriyala, Gul Agha, and Yamina Sami. Visualizing Actor Programs Using Predicate Transition Nets. *J. Vis. Lang. Comput*, 3(2):195–220, 1992.

[59] J. Moc and D.A. Carr. Understanding Distributed Systems via Execution Trace Data. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 60–67, 2001.

[60] Mono Project. `http://www.mono-project.com/`, September 2014.

[61] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[62] mxGraph - Interactive JavaScript HTML 5 Diagramming Library. `https://www.jgraph.com/mxgraph.html`, September 2014.

[63] Robert H. B. Netzer and Barton P. Miller. What are Race Conditions?: Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[64] L. Nowell, R. Schulman, and D. Hix. Graphical Encoding for Information Visualization: an Empirical Study. In *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on*, pages 43–50, 2002.

[65] Helen C. Purchase. Metrics for Graph Drawing Aesthetics. *Journal of Visual Languages & Computing*, 13(5):501 – 516, 2002.

[66] Helen C. Purchase, David Carrington, and Jo-Anne Allder. Empirical Evaluation of Aesthetics-Based Graph Layouts. *Empirical Software Engineering*, 7(3):233–255, 2002.

[67] Helen C. Purchase, Robert F. Cohen, and Murray James. Validating Graph Drawing Aesthetics. In FranzJ. Brandenburg, editor, *Graph Drawing*, volume 1027 of *Lecture Notes in Computer Science*, pages 435–446. Springer Berlin Heidelberg, 1996.

[68] Qt Project. `http://qt-project.org/`, September 2014.

[69] Qt Quick. `http://qt-project.org/doc/qt-5/qtquick-index.html`, September 2014.

[70] N. Quinn and M.A Breuer. A Forced Directed Component Placement Procedure for Printed Circuit Boards. *Circuits and Systems, IEEE Transactions on*, 26(6):377–388, Jun 1979.

[71] 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 WilM.P. Aalst and Eike Best, editors, *Applications and*

Theory of Petri Nets 2003, volume 2679 of *Lecture Notes in Computer Science*, pages 450–462. Springer Berlin Heidelberg, 2003.

[72] Dominik Rauch. Implementing and Evaluating the Peer Model with Focus on API Usability. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2014.

[73] Trygve Reenskaug. The Model-View-Controller (MVC) - Its Past and Present. `http://home.ifi.uio.no/trygver/2003/javazone-jaoo/MVC_pattern.pdf`, 2003.

[74] Reo Coordination Language. `http://reo.project.cwi.nl/reo/`, September 2014.

[75] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a Modern Approach*. Pearson Education, 2 edition, 2003.

[76] Raja R. Sambasivan, Ilari Shafer, Michelle L. Mazurek, and Gregory R. Ganger. Visualizing request-flow comparison to aid performance diagnosis in distributed systems. *Visualization and Computer Graphics, IEEE Transactions on*, 19(12):2466–2475, Dec 2013.

[77] Susanne Schacht. Formal Reasoning About Actor Programs Using Temporal Logic. In *Concurrent object-oriented programming and petri nets*, pages 445–460. Springer, 2001.

[78] Gerald Schermann. Extending the Peer Model with Compositional Design Patterns. Master's thesis, Vienna University of Technology, 2014.

[79] B. Shneiderman. The Eyes Have It: a Task by Data Type Taxonomy for Information Visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343, Sep 1996.

[80] Marjan Sirjani. Rebeca: Theory, Applications, and Tools. In *Formal Methods for Components and Objects*, pages 102–126. Springer, 2007.

[81] Scott F. Smith and Carolyn L. Talcott. Specification Diagrams for Actor Systems. In *PROCEEDINGS OF THE SECOND WORKSHOP ON HIGHER-ORDER TECHNIQUES IN SEMANTICS, ELECTRONIC NOTES IN THEORETICAL COMPUTER SCIENCE*. Elsevier, 1998.

[82] Spotfire.net. `http://www.spotfire.com`, September 2014.

[83] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for Visual Understanding of Hierarchical System Structures. *Systems, Man and Cybernetics, IEEE Transactions on*, 11(2):109–125, Feb 1981.

[84] K Thulasiraman and MNS Swamy. Acyclic Directed Graphs. *Graphs: Theory and Algorithms*, 1992.

[85] Jakob Tonn and Silvan Kaiser. ASGARD – A Graphical Monitoring Tool for Distributed Agent Infrastructures. In Yves Demazeau, Frank Dignum, JuanM. Corchado, and JavierBajo Pérez, editors, *Advances in Practical Applications of Agents and Multiagent Systems*, volume 70 of *Advances in Intelligent and Soft Computing*, pages 163–173. Springer Berlin Heidelberg, 2010.

[86] Edward Tufte. *Envisioning Information*. Graphics Press, Cheshire, CT, 1990.

[87] Using the Meta-Object Compiler (moc). `http://qt-project.org/doc/qt-4.8/moc.html`, September 2014.

[88] F. van Ham and B. Rogowitz. Perceptual Organization in User-Generated Graph Layouts. *Visualization and Computer Graphics, IEEE Transactions on*, 14(6):1333–1339, Nov 2008.

[89] Visual Paradigm. `http://www.visual-paradigm.com/`, September 2014.

[90] Colin Ware. *Information Visualization: Perception for Design*. Information Visualization: Perception for Design. Morgan Kaufmann, 2013.

[91] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. Cognitive Measurements of Graph Aesthetics. *Information Visualization*, 1(2):103–110, June 2002.

[92] Douglas Brent West et al. *Introduction to Graph Theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[93] Stephen A White. Introduction to BPMN. *IBM Cooperation*, 2(0):0, 2004.

[94] W.M. Zuberek. Timed Petri Nets: Definitions, Properties, and Applications. *Microelectronics Reliability*, 31(4):627 – 644, 1991.

List of Figures