TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

OLLI-PETTERI HIRVONEN

DESIGNING A BACKEND SERVICE AND DATABASE STRUCTURE FOR MONITORING DISCRETE MANUFACTURING SYSTEMS

Master’s Thesis

Examiner: Prof. Jose L. Martinez Lastra
Examiner and topic approved on 29. November 2017

ABSTRACT

Olli-Petteri Hirvonen: Designing a Backend Service and Database Structure for Monitoring Discrete Manufacturing Systems
Tampere University of Technology
Master of Science Thesis, 46 pages, 5 Appendix pages
November 2018
Master's Degree Programme in Automation Technology
Major: Factory Automation and Industrial Informatics
Examiner: Professor Jose L. Martinez Lastra

Keywords: monitoring system, event-based, directed graph, Petri net

In industrial applications production monitoring is a vital part of its optimization. Our goal is to develop a system that will follow the flows of items through a manufacturing process and calculate different KPI-values for them. In this thesis the aim is to create a backend service for accepting the data, storing and preprocessing it before sending it to the frontend. The user interface or more detailed data-analysis are not part of the scope.

In this thesis different data models are inspected for storing the data and their performance is compared. Through a literature view Petri nets and directed graphs are selected as the main candidates. The theory and applications for both are studied. It is discovered that while Petri nets are more versatile they are also heavier in comparison to directed graphs.

Along this thesis an application is developed on basis of Insolution's Inspector-software on which the directed graph is applied. With this the requirements from the company are reviewed, the most important of which are lightness and scalability. Also some decisions made with the development are explained.

There is also a simulated system developed for testing the data models with. A simple generator is implemented to produce events as they would come from a real system which are then stored into the database. This data is retrieved and processed using both data models and their durations are measured.

With the tests implemented, it is concluded that the directed graph performs slightly faster in most cases. It is also concluded that a Petri net offers more for modelling and control but for pure monitoring a directed graph is more efficient.

TIIVISTELMÄ

Olli-Petteri Hirvonen: Taustapalvelun ja tietokannan rakenteen suunnittelu diskreetin tuotannon seurantaa varten
Tampereen teknillinen yliopisto
Diplomityö, 46 sivua, 5 liitesivua
Marraskuu 2018
Automaatiotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Factory Automation and Industrial Informatics
Tarkastaja: professori Jose L. Martinez Lastra

Avainsanat: seurantajärjestelmä, tapahtuma pohjainen, suunnattu graafi, Petri net

Teollisuudessa tuotannon seuranta on tärkeä osa sen optimointia. Tavoitteenamme on kehittää järjestelmä, joka seuraa kappaleiden virtausta tuotantoprosessissa sekä laskemaan niille erilaisia KPI-arvoja. Tämän työn osuus tästä on taustapalvelu, joka on vastuussa datan vastaanottamisesta, tallettamisesta ja esilaskennasta. Käyttöliittymä ja tarkempi data-analyysi eivät kuulu tämän työn piiriin.

Työssä käsitellään erilaisia malleja joilla tietoa voidaan tallettaa ja niiden suorituskykyä vertaillaan toisiinsa. Kirjallisuustutkimuksesta saatiin vaihtoehdoiksi suunnattu graafi ja Petri net. Molempien teoriaa ja käyttökohteita käydään läpi. Tästä havaitaan Petri netin olevan monipuolisempi, mutta myös raskaampi malli suunnattuun graafiin verrattuna.

Työn ohessa kehitetään myös Insolutionin Inspector-ohjelmiston pohjalta sovellusta, johon sovelletaan suunnattua graafia. Tässä yhteydessä käydään läpi yritykseltä tulleita vaatimuksia sovelluksen suhteen, joista tärkeimpiä ovat keveys ja skaalautuvuus, sekä selostetaan suunnittelussa tehtyjen päätösten taustoja.

Työssä toteutetaan myös simulaatio aidosta järjestelmästä, jonka kanssa kokeillaan molempia tutkittavia tietomalleja ja niiden suoriutumista erilaisista laskenta tehtävisä. Toteutetaan yksinkertainen generaattori joka luo tyhjästä tapahtumia joita tallennetaan tietokantaan. Tätä dataa sitten käydään läpi eri tavoin ja mitataan kuinka kauan kummallakin tietomallilla sen suorittamiseen menee.

Näistä testeistä päätellään lopulta suunnatun graafin olevan kevyempi ja suorituskykyisempi haluttuun seurantajärjestelmään. Petri netin todetaan tarjoavan enemmän mahdollisuuksia mallintamiseen tai ohjaamiseen, mutta puhtaaseen seurantaan yksinkeratinen graafi on tehokkaampi.

PREFACE

The subject for this thesis was received from Insolution Oy as an idea to extend the existing Inspector-system.

I want to thank Juha Katajisto, Atte Sipilä and Arttu Tamminen for the subject and for all the times we spent together brainstorming. I also want to thank all who supported me during this work.

Tampere, 21.11.2018

Olli-Petteri Hirvonen

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

FMS	Flexible Manufacturing System
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
OEE	Overall Equipment Efficiency
PLC	Programmable Logic Controller
SA	Scrapped Amount
SQL	Structured Query Language
TPA	Total Produced Amount
UML	Unified Modeling Language
WIP	Work In Progress

A	Finite set or arcs in a directed graph
H	A directed graph
P	Place in a Petri net
T	Transition in a Petri net
v	Vertex in a directed graph
V	Finite set of vertices in a directed graph
$\forall v \in V$	Applies for all v that are elements of set V

1. INTRODUCTION

1.1 Research Environment

The results found in this thesis are aimed to be used with Insolution's Inspector product-family. It is a data acquisition product that was originally developed by Fastems. It was originally called Fadector and used for monitoring the efficiency of cnc-machines connected to Fastems' FMS-system. In 2013 the software was sold to InSolution and its name was changed to Inspector.

The basic version of Inspector collects real-time data from connected machines and visualizes it in several refined forms for the user or other connected services. Inspector has also been used as a platform for several tailored projects where the data collection is handled in different ways but the data manipulation and visualization are basic Inspector functionality.

1.2 Motivation

Since Inspector is capable of monitoring continuous production, the next step for it would be to monitor discrete production. The application should be able to receive very simple, event-based data from the monitored system, thus making it possible to extend the usage from continuously running machining tools to other kinds of systems as well. These might include for instance material flows inside a factory, progession of an order, path walked by a person inside a facility, or how an employee moves between projects.

Originally the data is thought to include nodes that act as check-points in the system and events that occur when an item checks in at a node. The nodes would contain only a unique identification and a more user friendly name. The events would consist of an id, the id of the node where the event happened, a timestamp and the id of the item.

The implementation should be able to visualize the flow of items from this data, displaying the possible bottlenecks of a system, WIP times, amounts and histograms, and the general flow of items through the system. The initial idea for architecture of the system can be seen in Image 1.

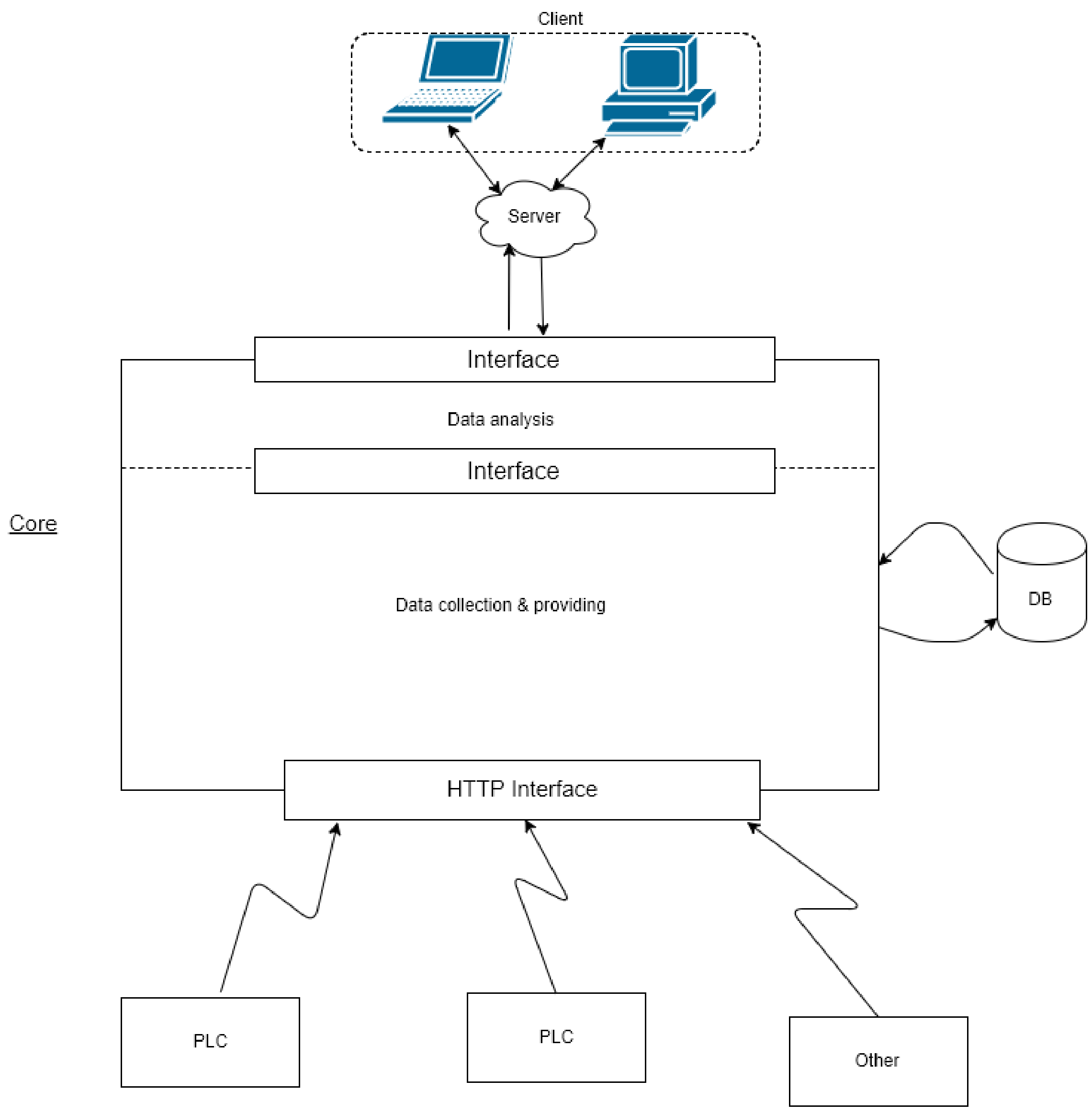


Image 1. The overall architecture of the Inspector-system.

1.3 Problem definition

The problem is to decide what kind of backend service should be developed to monitor discrete, event-based flows. There are several parts in this that are to be defined; The structure of the database could follow a ready model like a directed graph or should something new be developed. The sources of the data can be PLCs monitoring the system, a user-interface through which a user can enter the data, both or something else. The interfaces between the core service and the data collection devices as well as between the backend and frontend services can use a specific standard or be coded explicitly.

1.4 Constraints by the Reseach Environment

Insolution requires that the implementation, which should be done witht his thesis, can be attached into the existing version of Inspector. This means that the database engine will be Microsoft's SQL server, and the source code should be in line with the companys coding policies and use the .Net-framework.

1.5 Goal of the Thesis

The goal of this thesis is to find a way to implement the backend service for this application, and find the best way to do it. In this case the backend will consist of collecting, storing and processing the data. Different methodologies, tools and standards are to be studied and evaluated according to the requirements set by Insolution. Either some of them are to be implemented as they are or after some modifications, or something new would be developed.

The thesis aims to answer the following questions:

- What kind of structure should the database use?
- Where can the data come from?
- How should the application communicate with the data sources?
- How will the data be processed before giving it to the frontend?
- How should the backend communicate with the frontend?

1.6 Outline of the Thesis

This thesis is divided into five parts; The first part is the introduction where the background and motivation for this thesis is explained. The second part contains background research done on the topic from literature. The third part describes a simulated system that was built to test the models that are to be used in the actual application. In the fourth part is explained how the application that was done on the side of this thesis was developed and what kind of decicions were made. The final part contains the conclusions made and ideas on how to continue with this.

2. LITERATURE AND INDUSTRIAL PRACTICES REVIEW

2.1 Modelling Tools

An important part of creating a monitoring system, is to decide how to store the data collected that it will be as effective as possible for later processing. The structure should be simple but flexible in order to make monitoring different systems easy. The monitoring system is desired, in its simplest, to be able to make sense from simple timestamps linked to item and node ids. If the process flow of the physical system expands, more nodes would be added either manually, or automatically according to an input from the system.

2.1.1 Petri Nets

Petri nets were created by Carl Adam Petri and published in his doctoral thesis in 1962 [1]. The basic version of the data model consists of four types of components; places, transitions, arcs and tokens. The arcs are directed, meaning they only work in one direction, and connect places to transitions and vice versa. Arcs have by default a capacity of one, but it can be defined otherwise. Places have an unlimited capacity, whereas transitions have no capacity and tokens only pass through them. It is also possible to define a specific group of tokens to specific places using for instance colors.

A transition is triggered when all its input-nodes contain enough tokens to fill the capacities of the arcs between the places and the transition. When it triggers, an amount of tokens is taken from each place, equivalent to the capacity of the arc connecting the place to the transition. Then new tokens are created to equal the capacities of the out-going arcs, and they are distributed to the output-places of the transition. Image 2 and Image 3 visualize this functionality.

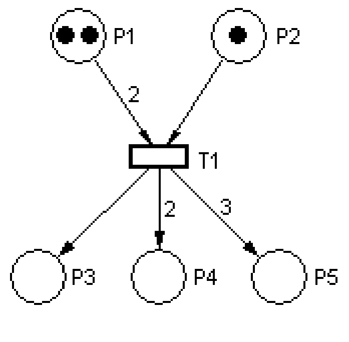


Image 2. Petri net ready to trigger.

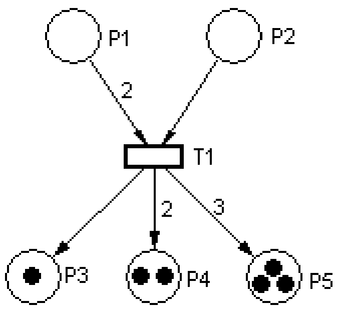


Image 3. Petri net after it was triggered

It is also possible to define an external event to trigger a transition. This could be for example a delay of some seconds. In this case the input-places of the transition only enable it when they contain the required amount of tokens. When the transition is enabled, it waits for the external event, deletes the input tokens, and creates new output tokens. This kind of Petri nets are called non-autonomous, where as the ones without any external events are autonomous.

2.1.2 Discrete, Continuous and Hybrid Petri Nets

The basic version of a Petri net is discrete. The flow of the system is based on discrete transitions and events, and only whole tokens are created and removed. David and Alla [2] suggest creating a continuous model, by splitting the tokens down into marks, and increasing the amount of these marks, while decreasing their size, to approach infinity. They would have the transitions fired successively for a defined amount of times, to move a token from a place to another in smaller pieces. These events would then be approximated to create a continuous model of the system as seen in Image 4. A continuous Petri net would be visualized using a double line as in Image 5.

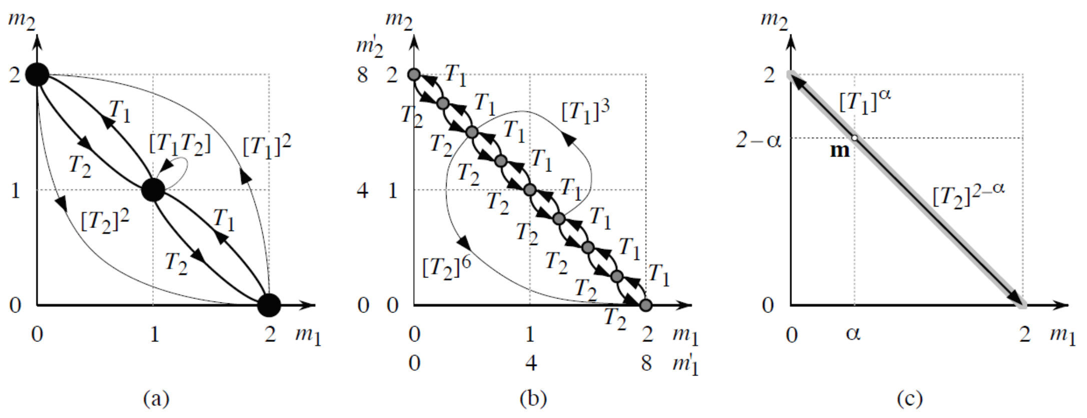


Image 4. Splitting transactions down to create a continuous model. [2] p.120

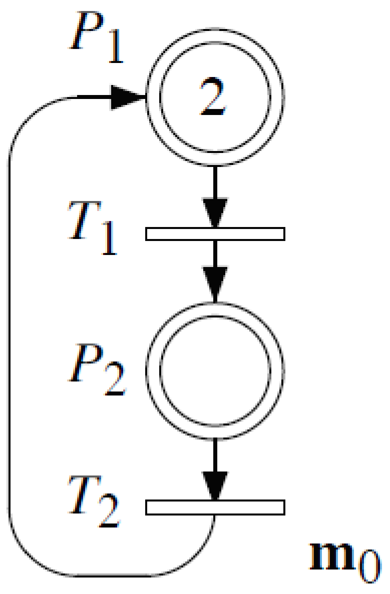


Image 5. A continuous Petri Net. [2] p.123

David and Alla [2] also introduced a hybrid version of discrete and continuous Petri nets where the model can contain both discrete and continuous places and transition. They suggest using them, for instance, in continuous systems that are controlled by discrete componenets and vice versa.

In Image 6 a hybrid Petri net is displayed providing a concrete example. In the system there is some machine providing continuous processing of materials. When the machine is working, i.e. there is a token in P2, it processes materials in its input buffer P4 and moves them into the output buffer P6. T3 monitors the amount of processed marks in P6, triggering when the amount reaches 500 stopping the machine. P5 works as a counter for taking; when it reaches 60 marks it means 60 parts have been processed and a new raw materials batch token will be taken from the warehouse P1 and transformed into 60 marks.

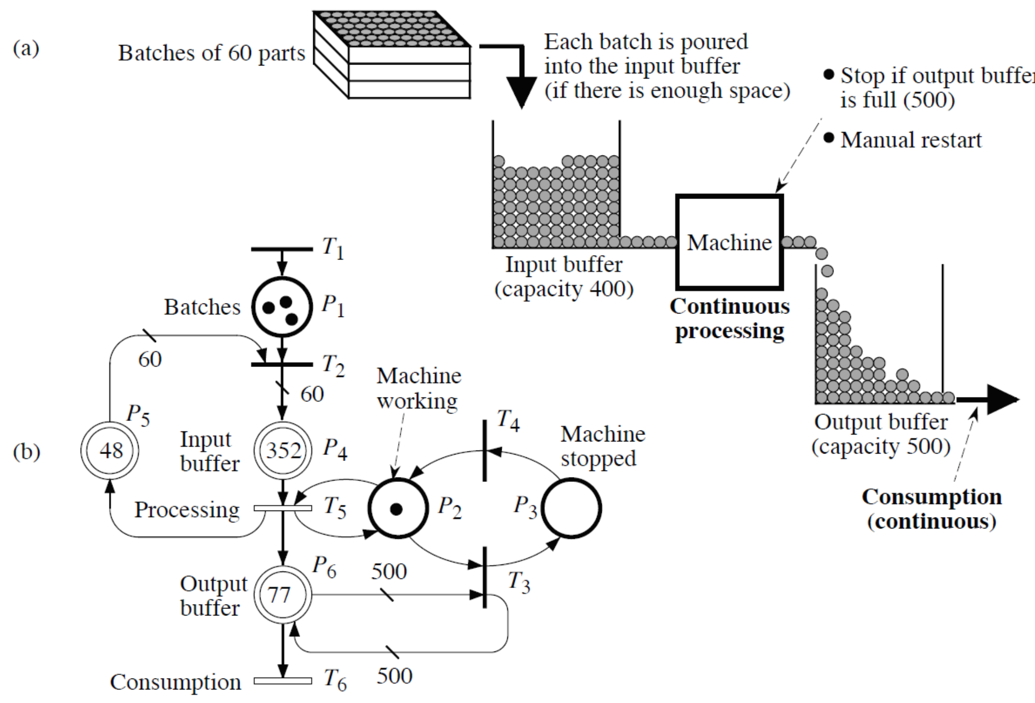


Image 6. A model of a production system represented using a hybrid Petri Net. [2]

2.1.3 Petri Nets with Data Structure

When monitoring the flow of items, as described in chapter 0, it would prove quite difficult using standard Petri nets. A requirement for the application is to be able to follow single items or batches that move along the system, and since in Petri nets the tokens that represent items aren't moved, but deleted and created in places, they cannot achieve this functionality.

Christophe Sibertin-Blanc introduced in 1985 an extension to Petri nets called Petri nets with data structure [3]. The idea is to add three concepts into the model; properties describe attributes or simple data, entity classes represent a set of properties with default values, and entities are instances of entity classes that can override the default values of the properties with actual data. Entities are used to replace tokens in Petri nets, and entity classes are assigned to both; entities and places, thus restricting entities of a certain entity class move only to places of the same class.

According to Sibertin-Blanc there should be preconditions to all transitions, which can, for instance be associated with external events or the properties of the entities in the input-places. [3] Transitions on the other hand, aren't supposed to take into account the properties of the entities used to fire them. Firing these transitions could alter the properties of affected entities while transferring them into the output-places.

The definition of Petri nets with data structure also contains a mention of registers. They are used as global entities that affect all preconditions in the model, but aren't used in any transition. [3] This leads to the properties of them being constant since they are changed only when entities are used to fire a transition.

2.1.4 Net-Condition Event System

Since the goal of this thesis is to produce a monitoring system for discrete systems, the data structure should be able to adapt to events from the system, instead of modeling and calculating the behavior of the system. An expansion has been introduced to Petri nets, called Net Condition/Event System, which would provide this functionality. In event graphs the transitions are triggered by external events and enabled by condition arcs in addition to the basic form where they are only enabled by marking, i.e. the tokens in the inpuit-places. [4] They can all be signals coming from the Petri net model itself, or outside it, for instance from a physical system or another Petri net.

An example of a Net Condition/Event System is presented in Image 7 which models the functionality of a small processing system. The system can be seen consisting of two parts; the physical system and the controller. In the example some transition trigger events, marked with the zig-zag in the arc, that are used to fire other transitions. There are also conditions for some transitions that depend on the status of specified places. These dependencies are marked with an arc that has a ball replacing the arrowhead that usually marks the direction of an arc. [4]

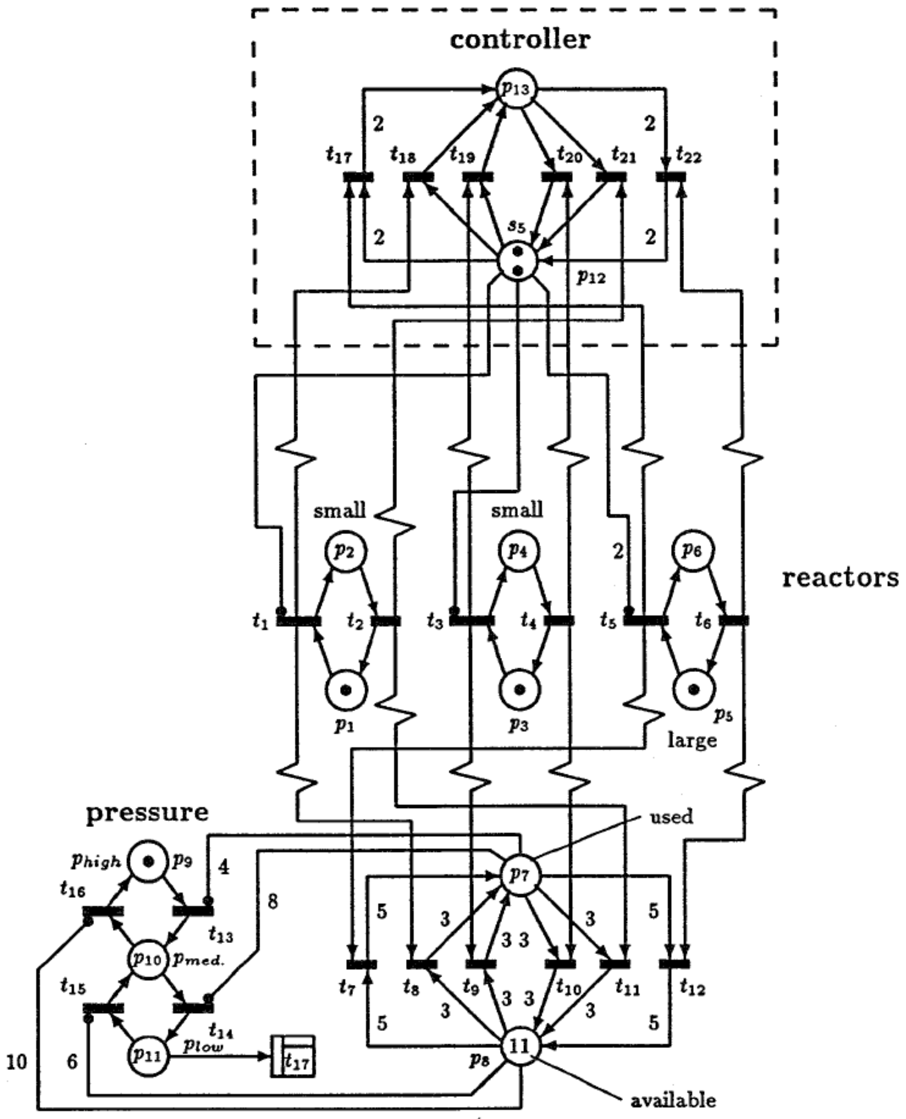


Image 7. An example Petri net Condition/Event system modeling the behavior of a small processing system

2.1.5 Petri Nets Applications to Monitoring Systems

Valette, Cardoso and Dubois presented in 1989 a model for monitoring a manufacturing system using Petri Nets with Objects and imprecise markings [5]. They decided to use objects to model the physical components of the system because it provided them with dynamic properties instead of constants that are used for instance in Predicate/Transition Petri nets. They wanted to be able to show how a token could be between places and that it could be known certainly where it was. For this they introduced imprecise markings where it is presented that an object is in different places with different propabilities and extended the Time Petri nets, where each transition has an predefined interval [6], to be Fuzzy-time Petri nets, where in addition to the maximum and minimum enabling durations another, wider pair of durations would be defined to symbolize the accepted limits.

The trio suggests that this kind of Petri Net would be suitable for both monitoring and controlling a manufacturing system. Specially for monitoring this would introduce the possibility to validate the events on the physical system against the predefined model, and its time-limits. This validation could prove useful in detecting malfunctions and possibly calling an operator when needed. [5] While this is true and useful, the usability of this kind of solution in our project, where a requirement is the possibility to modify the monitored system without reconfiguring the monitoring system, seems rather unoptimal.

2.1.6 Directed Graph

A directed graph, also known as a digraph, is a mathematical graph, that, in its basic form, consists of a finite, not-empty set of vertices (marked V) and a finite set of arcs marked (A). Arcs, that are also called edges, are ordered pairs of vertices, linking them together to create a network. The vertex, from which an arc originated from, is called the tail and the vertex the arc ends in, is known as the head. What separates a directed graph from a regular graph, is that its arcs have an assigned direction, where as in a graph they work both ways. [7]

The graph is, by definition, allowed to contain opposite arcs, for instance (u,w) and (w,u) in Image 8, but it may not contain parallel or looping arcs. Paraller arcs are multiple arcs that have the same head and tail with eachother, and a looping arc is a single arc whose head and tail are the same vertex. If these properties would be implemented, then we would have to speak of a directed pseudograph. Image 9 displays a directed graph H, which is transformed into a directed pseudograph by adding a looping arc (z,z), and two new parallel arcs (u,w). [7]

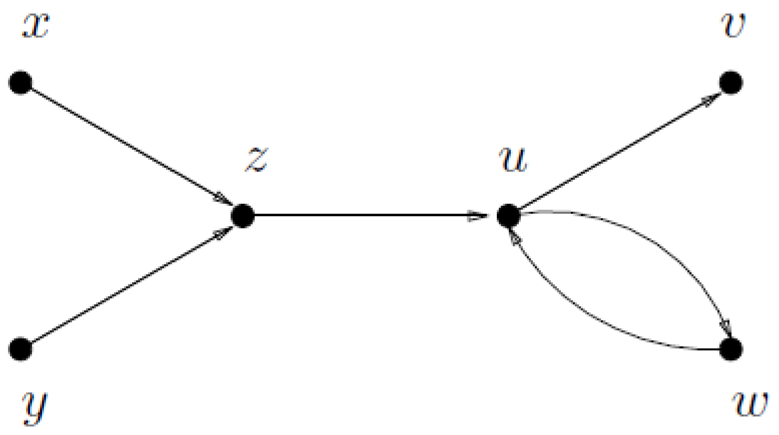


Image 8. A directed graph. [7]

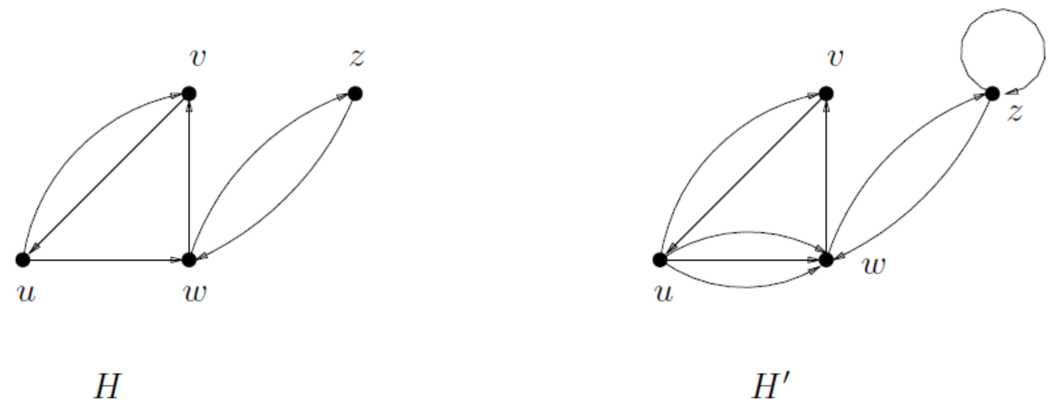


Image 9. A directed graph H and a directed pseudograph H'.[7]

A directed graph is said to be connected if every vertex is connected to the same graph. The graph is strongly connected if there exists a directed path between every two vertices. A spanning tree, on the other hand, contains a single vertex that has a directed connection to all the other vertices in the graph. This vertex is called the root vertex. A connected directed graph is shown in Image 10, where vertex E is connected to both vertices C and D, but isn't reachable from either of them. Image 11 shows a strongly connected version of the same graph, where the arc (E,C) is reversed making the graph a circle, thus enabling any vertex to be reached from any other vertex. Finally a spanning tree is presented in Image 12. The graph is also connected, but vertex D cannot be reached from anywhere else in the graph. Nontheless, there is a directed path from vertex D to all the others, making it the root vertex.

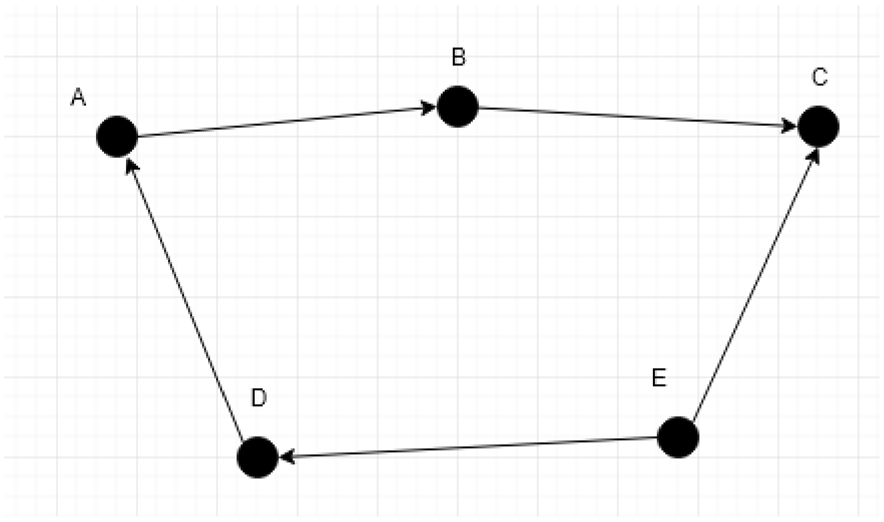


Image 10. A connected directed graph.

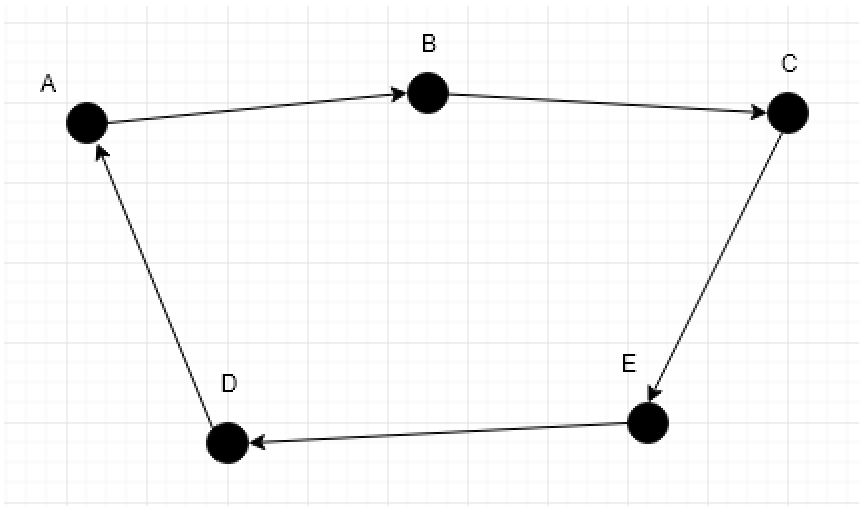


Image 11. A strongly connected directed graph.

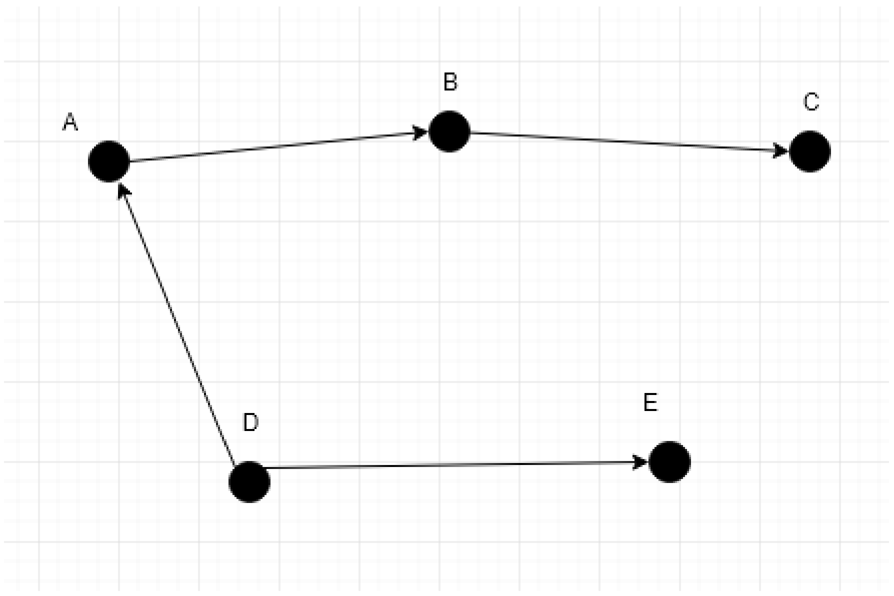


Image 12. A spanning tree where vertex D is the root.

2.1.7 Properties of Directed Graphs

A directed graph can have a weight assigned to its elements, both vertices and arcs. This is a numerical value, which can be used, for instance in calculating an optimal path through a network, to mark the cost needed to travel over an arc or through a vertex. In Image 13 graph D has weighted arcs, and graph H has weigted vertices. They could also be assigned to both of them. By default the weight of an element is one, an the total weight of an graph is the sum of the arcs in the graph. This can also be applied to sub-graphs, i.e. a section of a directed graph. [7]

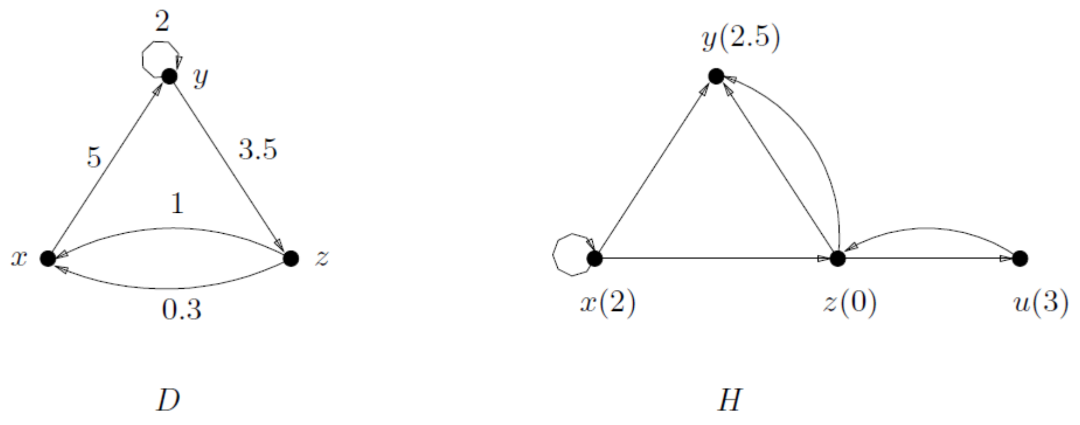


Image 13. Weighted directed graph D and vertex-weighted directed graph H. [7]

Directed graphs feature a property known as distance. This property is inherited from general graph theory. It is defined for two vertices and means the smallest amount of edges betweent them. In opposition for a regular graph, for which the distance between two vertices is the same in both direction, there path might not even exist for two directions in a directed graph.

2.1.8 Networks

A network is a special directed graph for which some extra functions are associated to the arcs: lower bound, capacity and cost. Lower bound marks the minimum limit for the flow on the arc, where as capacity marks the maximum value. The cost is a value assigned for the arcs. It can be used for comparing different paths when trying to optimize the flow through the network.

An arc can be defined as a flow which is a function transferring units or tokens from one vertex to another via an arc. Together with capacity and cost -parameters, the flow is a useful tool for controlling systems and optimizing a network. Through them can be seen the available capacity in different parts of the network, and calculate how the network could provide the most efficient output in respect to pace and cost.

A flow is considered feasible if the numerical value is between the lower bound and the capacity. The cost of the flow is the sum of costs assigned to all the arcs on the path between the endpoints. When there are no possible arcs connecting the defined vertices, the flow is zero. A network containing feasible flows is presented in Image 14.

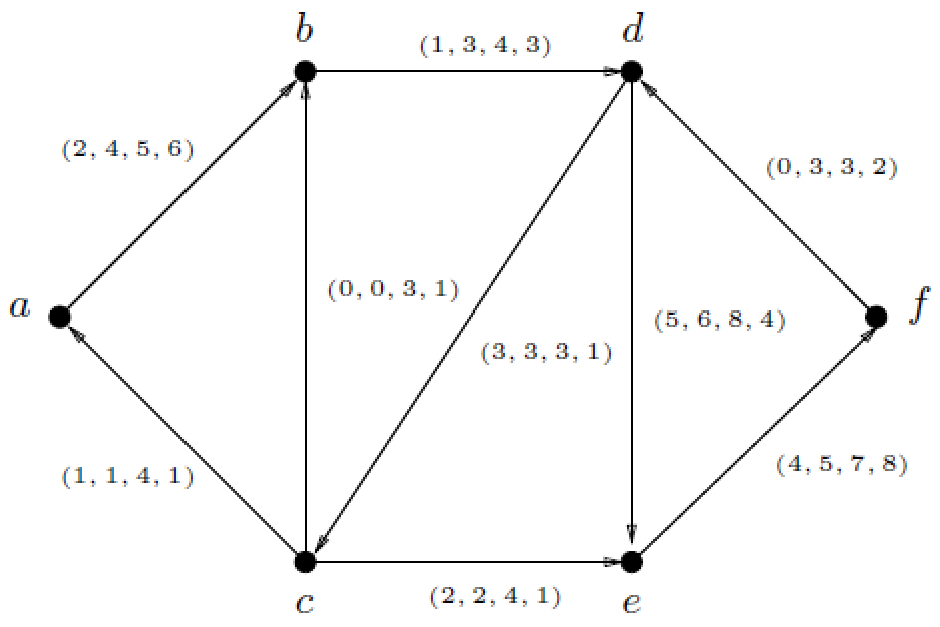


Image 14. A feasible flow. Arcs have properties lower bound, flow, capacity and cost.

A balance vector can be assigned to different sections of a directed graph, which represents the difference between incoming and outgoing flows of the section. It is calculated as in Formula 1.

$$b_x(v) = \sum_{vw \in A} x_{vw} - \sum_{uv \in A} x_{uv} \; \forall v \in V$$

Formula 1. Balance vector af a vertex.

It basically means the sum of flows that have their heads in the defined vertex is subtracted from the sum of flows that have their tails there. When the balance is positive, the vertex is called a source, and the ones with a negative balance are called sinks. Balanced vertices have a balance of 0. The sum of the balances for all vertices in a network is assumed to be zero.

2.1.9 Applications of Directed Graphs

In 2007 Natsch and others published an article where they described how they used directed graphs to model how antibiotics-prescriptions were changed on a patient. The vertices depicted the transitions between prescriptions. One stood for the transition from no antibiotics to antibiotics, another for from antibiotics to another antibiotic, the third from an antibiotic to the same, and the fourth from an antibiotic to no antibiotic at all. The researchers created a graph for each month in 2000-2004, and created an animation of them to visualize the evolution of sequential prescriptions in that time period. [8]

Kristian Movric and Frank Lewis used directed graphs in creating a cooperative optimal control for multi-agent systems. In their solution they used directed graphs that allow distributed solution, meaning those that have a simple Laplacian matrix. They had trouble with dynamic systems and therefore decided to use this solution only for fixed topologies. [9]

2.2 Database

As was mentioned in chapter 1.4 the data will be stored on the database which will be based on Microsoft's SQL-server. For both data models; Petri nets and directed graphs, the database model will be very similar. The database will contain a table for each element of the model linked with relationships together to form a complete system. This will be explained in more detail in chapter 4.1.

2.2.1 Stored Procedure

A stored procedure is a sequence of sql commands, that are executed on the database-server. It can be called like a function or method over a network, with possible parameters, and it returns a resulting table like a normal sql-query. It can also contain simple programming elements, like conditional or iterative commands. The difference between a normal query and a stored procedure is that with a stored procedure, you can execute several sql-commands over a single network-transaction, minimizing the network traffic and delay.

For instance a stored procedure can be very useful when an application wants to execute a query inside a loop. In this case the application would have to send a query over a network, and wait for an answer on every cycle of the loop. The conditional select-command on a sql server is also optimized to handle and process large sets of data much faster than an iterative method in most traditional programming environments.

As the stored procedures act like an interface to the database, the application doesn't have to be very tightly bound to the structure of the database, making it easier to maintain several applications using the same database. As long as the procedures are called with the same inputs, and they return the same outputs, the application doesn't care how the tables are arranged in the background. For instance we could allow some third-party-software to connect to our database and so the results of the data we have collected in case if a customer wanted to use some other frontend than Inspector.

Since stored procedures are stored on the server, they pose a security threat as they can be accessed over the network, and if an attacker manages to modify them, this immediately affects all client applications. In a monitoring system the biggest threat is the privacy. If an attacker can access a stored procedure, they most likely can also access sensitive data. The stored procedure only provides more processed data and results instead of the attackers having to parse it themselves. However it might be in an attackers interests to feed the user misinformation about their monitored process, so this cannot be completely ignored.

3. EVENT-BASED MONITORING SYSTEM

This chapter descripbes the actual application that we have begun to implement. It is mostly done on basis of the test-software used in chapter 4.1.

3.1 Current Status

When the development of the actual application began, there was already some data generated from a client’s system. This data consisted of events produced by the flow of processed parts in a factory. There had already been ideas about using directed graphs as a base for this application, and so the actual data has also been stored in such a fashion that it can relatively easily be transformed into a graph. The data contained vertices, called nodes in the application, which depicted the different workplaces and buffers in the flow, edges depicting the movement of pieces along these nodes, and events.

The events are used to log the actual process that is on-going and they contained information of when a specific workpiece has been noticed at a specific node in the system. From these events we aim to build the edges connecting the nodes, and by analyzing the timestamps, attach more detailed information about the flow into the edges.

3.2 Functional Requirements

The system to be monitored can be preconfigured to have a certain structure and possible limitations. The nodes in the system can be defined and possible maximum capacities set for them and their type can be selected. Then the system will receive events from the system, containing simple information: item, node and timestamp. From these events, the system will be able to expand dynamically, and create new default nodes into its database. These automatically created nodes will be modifieable later to allow refining their properties.

The user interface will be able to draw the whole network, and where the length of the edges will depict the average time spent on each transition, and the thickness of them would present the relative amount of items that have flown through it. From the graph a user can click on a single node or edge to see its properties, like the type of the node or average duration of the edge. It will be possible to select two nodes on the network, and get the properties of the flow between them.

The system will not in itself keep track of which items are in located in which node, or even how many items are in the nodes. When needed, these values are obtained by querying the database in a manner described in chapter 4.2.

3.2.1 Configuration Requirements

The system is designed to adapt to changes automatically, but some properties of it can be configured to give more info for the user. The nodes in the system can have two properties assigned by the user; the type and the maximum capacity. The type defines into which class the node belongs to; Check point is the default, and is meant to be used as a point through which items travel instantenously. Buffer nodes represent the storages along the line and are used for calculating the stand-stills of items. Value adding nodes are used for actual processing and by that for calculating the efficiency. The output nodes are the exit-points of the system, and items passing into them are no longer taken into account when calculating the current situation. Naturally they still affect the overall statistics. All these types behave the same inside the monitoring system, but by classifying them we can provide more potential for calculating different performance indicators.

There must be a possibility to configure the maximum capacity of nodes, meaning how many items they can contain at a time. This system is not meant to control the actual flow, so at this stage it cannot prevent a node from overfilling, but it can give an indicator to the user when the node is full, and when there are too many items in a node.

Nodes can be assigned a group, indentified by an integer value. The idea behind this is to make it possible to split a single workplace or such into several nodes and still show that they belong together. At the time of writing there is no plan to use this information for anything else, but there are speculations that some day we might make a possibility to make automated events or restrict the flow in the system based on these groups. The aim is not to restrict the physical flow but if we, for instance, know that every time after an item has entered a specific node in a group, or workcell, the next node it will be seen at must be some specified node in the same group. This way we can provide more accurate data, i.e. in situations where an operator has forgotten to log a movement of a piece into the system.

3.2.2 Data Collection Requirements

The events in the system are aimed to be as simple as possible, to enable gathering data from a wide variety of systems. In its simplest form the event consists only of a timestamp, an item and a node where the event has occurred. In the database events will be grouped into pairs, so a single row will contain the information about both the starting time and place of a transaction, as well as the ending time and place. The duration of a transaction will also be stored on the same row.

3.3 Data Model

The database was decided to be built on basis of a directed graph. Petri nets were a good alternative, but their basic idea seems to be focused more on control and modelling than monitoring. The triggering of transitions is something that we do not want in this system since our objective is to implement pure non-intrusive monitoring. The directed graph is a pure data model, thus allowing us to maintain the data in it the way we like. Admittedly this is possible for petri nets also, but as mentioned it is not their purpose. There are three tables to contain this data; nodes, edges and transactions.

The nodes-table is used to contain the representation of the workstation in the system. The table contains the following columns: an identification and name of type string, type to dictate what kind of events are associated with it (check point, value adding, buffer or exit), capacity to mark how many items there can be at a time, and an integer value to represent the group it belongs to.

The transactions-table contains the realised events from the system. Each row has the following columns: identification, start node and time, end node and time and duration of the transaction. The transactions are meant to represent the movement of items between nodes. When an event occurs for an item at a node, the previous transaction, for the item in question, gets its end node, end time and duration -values assigned accordingly. At the same time a new transaction is created for the item, that has its start node and time set according to the current event.

The edges-table is meant to bind transactions, that have the same start and end nodes, together. The table contains the following columns: id, start node, end node, count of transactions that have realised this edge and average duration for the transactions. It is used to keep the basic data of the graph updated with a higher availability. The corresponding rows are always updated when a transaction is finished, thus keeping the data up-to-date. The main advantage of this table is for drawing a visualization of the whole graph, since when fetching the data to the frontend the nodes and edges to be drawn can be returned through a simple select-query. This will more efficient than calculating through all the transactions when the user interface is to be updated.

3.4 Use Cases

The use cases for the backend service can be seen as the interface between the back- and frontend plus registering the events from the monitored system. The interface was developed together with Atte Sipilä, and it allowes the frontend to fetch the following data:

- Average inventory sizes
- Scrap percentage in a timeperiod
- Throughput time for items
- Processing time for items
- Buffer time for items
- Move time for items
- Output rate

The interface also supports configuring the system; creating, updating and deleting nodes. While the nodes are created automatically through the events, the interface will also allow for manually entering new nodes, as well as editing the configurations of existing ones. It will not be possible to delete nodes from the database since this would result in a loss of data but they can be removed from the system so that events targeting them raise a warning.

3.4.1 Configuration

For manually creating or updating the properties of a node two methods are provided for the user interface -section of the application for taking in the data for that. With the new node -method described in Table 1, the user inserts the name, type and maximum capacity of the node. The name is the only mandatory parameter, by default the type will be check point and the maximum capacity will be zero meaning that there is no limit. The return value is a Boolean indicating success.

Table 1. The properties of the new node -method.

New Node Parameters		
Parameter	**Optionality**	**Data type**
Name	no	string
Type	yes	Enum
MaxCapacity	yes	integer
Out values		
Value	**Data type**	
Return value	boolean	

For updating the node only the id is a mandatory property. For the others, the properties that are set are updated for the node with the matching id in the database. The return value works as it does with the new node -method; a Boolean value indicating success.

Table 2. The properties of the update node -method.

Update Node Parameters		
Parameter	**Optionality**	**Data type**

Id	no	Guid
Name	yes	string
Type	yes	Enum
MaxCapacity	yes	integer
Out values		
Value	**Data type**	
Return value	boolean	

3.4.2 Get Average Inventory

The user interface can fetch the average inventory sizes of the system. The backend will search through the events and calculate how many items have been in buffers at specific times. The backend takes as parameters the time window for which this data is to be fetched and the resolution in which the results are to be returned. It is also possible to define, through optional parameters, which buffer-nodes and which orders or item-types are to be taken into consideration.

The resolution is given as an enumerable, for which the alternatives will be month, week, day, hour and all. All is the default value, and in this case the backend will only return a single value. The time window is split into sections according to the resolution, and the average will be given for each section separately. The average will be calculated accordingly:

$$\frac{B1 + B2}{2}$$

Formula 2. Average inventory.

where B1 stands for the buffer size at the beginning of the time window, and B2 the size at the end.

When several orders or item-types are defined, the backend will split the results for each of them separately. The result it gives will be a dictionary-data type, where the key is the identifier of the item type, and the value is a list containing the the averages for each time window that are specified.

Table 3. Get average inventory parameters and return values..

Parameters		
Parameter	**Optionality**	**Data type**
Start time	no	DateTime
End time	no	DateTime
Item type	yes	Guid
Selected buffers	yes	List<Guid>
Resolution	yes	Enum
Out values		
Value	**Data type**	
Return value	Dictionary<Guid, List<float>>	

In Image 26 a rough workflow for the use-case is displayed. First the given time window is split into smaller time slots according to the resolution given as a parameter. These time slots are then handled individually inside a loop, where the amount of items in the selected buffers inside the time slot is queried from the transactions.

Since a transaction contains both start and end times, it is easy to find all transactions at the specified node that have started before a specified time and ended after it. The items linked to these transactions can be deducted to have been at the node at the time. This method is applied to both the start and end times, thus calculating the amounts of items at each time, and to these amounts we can apply the formula defined before in Formula 2.

3.4.3 Get Scrap Percentage

For fetching the scrap percentage, the frontend must supply the time window, and possibly the item types or orders for which the percentage will be calculated. The percentage is calculated:

$$\frac{SA}{TPA} * 100\%$$

where SA stands for scrapped amount, and TPA for total produced amount. The return value will be a dictionary with a guid as key, for defining the item or order in question (if multiple specific ones were given as parameters) and the value a float for the percentage value.

Table 4. Get scrap percentage parameters and return values.

Parameters		
Parameter	**Optionality**	**Data type**
Start time	no	DateTime
End time	no	DateTime
Item type	yes	Guid
Selected nodes	yes	List<Guid>
Out values		
Value	**Data type**	
Return value	Dictionary<Guid, List<float>>	

3.4.4 Get Throughput Time

Throughput time can be fetched for either a single item, with optional parameters to define a section of the graph by the start and end nodes. The method will return a long-value describing the amount of seconds the item has spent in the system, or between the specified nodes. The time is calculated as the difference between the timestamps for the entry-event and the exit-event.

Table 5. Get throughput time parameters and return values for a single item.

Parameters		
Parameter	**Optionality**	**Data type**
Serial number	no	String
Start node	yes	Guid
End node	yes	Guid

Out values	
Value	**Data type**
Return value	long

It is also possible to get the throughput time for multiple items on a single query. The item types in question must be given as a list, as well as the time window which is to be taken into consideration during this calculation in addition to the parameters used for the single item. The return value is a dictionary, with the item type as key, and a list containing the throughput time for each individual item in seconds as a long-value.

Table 6. Get throughput time parameters and return values for multiple items.

Parameters		
Parameter	**Optionality**	**Data type**
Item type	no	List<Guid>
Start time	no	DateTime
End time	no	DateTime
Start node	yes	Guid
End node	yes	Guid
Out values		
Value	**Data type**	
Return value	Dictionary<Guid, List<long>>	

In Image 25 a sequence diagram is used to visualize the rough workflow for this use-case. If start node is defined, the core queries for the chronologically first transaction that has started from the specified node, and is associated with the item in question. Otherwise the first of all transactions with the item is fetched. The same logic is applied for fetching the last transaction for the item, with the exception that if no end node is defined, the transaction with which the item exits the system is selected. If no such transaction is found, a null value will be returned. In a successful query, the time difference between the two transactions will be calculated and returned.

3.4.5 Get Process, Buffer and Move Time

Process time can be fetched for a single item, using the serial number, or for multiple items, in which case a list containing all the wanted item types must be provided along with the time window. The process time is calculated as the sum of the times each item has spent in value adding nodes. The return value is a dictionary with a guid as the key for each value adding node, and the value is a long describing the seconds the item has spent there. For multiple items, this dictionary is the value of another dictionary that has the item type as the key. For multiple items, the time is presented as an average.

Table 7. Get process, buffer and move time parameters and return values for a single item.

Parameters		
Parameter	**Optionality**	**Data type**
Serial number	no	string
Out values		
Value	**Data type**	
Return value	Dictionary<Guid, long>	

Table 8. Get process, buffer and move time parameters and return values for multiple items.

Parameters		
Parameter	**Optionality**	**Data type**
Item types	no	List<Guid>
Start time	no	DateTime
End time	no	DateTime
Out values		
Value	**Data type**	
Return value	Dictionary<Guid, Dictionary<Guid, long>>	

Process, buffer and move time can all be fetched in the same way. The only difference is when the process time is calculated from the value adding nodes, the buffer time is the time spent in buffer nodes, and move time is the time the item has spent between nodes. The parameters and return values are identical.

3.4.6 Get Output Rate

Output rate is meant to return the amount of processed items in a certain time period. The method requires the time window as a parameter, and takes resolution and item types as optional parameters. As a result it returns a dictionary, with the item type as the key, and a list containing the amount of processed items in each time slot as the value. The time slots are defined according to the given resolution, by dividing the time window according to the given resolution. The resolution is an enumeration with the following possibilities:

- Month
- Week
- Day
- Hour
- All

with all selected as the default value.

Table 9. Get output rate parameters and return values.

Parameters		
Parameter	**Optionality**	**Data type**
Start time	no	DateTime
End time	no	DateTime
Resolution	yes	Enum
Item type	yes	Guid
Out values		
Value	**Data type**	
Return value	Dictionary<Guid, List<int>>	

3.4.7 New Event

The new events from the monitored system will be entered through a separate interface that will be provided for the devices or services actuating the monitoring; plc-controllers, web applications, Raspberry Pi -computers etc. The interface only provides one method; new event. This method gives the piece id, node id and a timestamp from a new event in the system, it is stored in the database and the data affected is updated. The update means the current location of the piece is changed, the capacities of both nodes are updated; the node the piece arrived to and the node where it was before, and the count of the edge connecting the nodes is increased, or in case there is no edge yet it is created.

3.5 The Architecture

The 4+1 view model is used to describe the architecture of the application. The model consists of four views that represent the construction of the system; logical view in Image 15, development view in Image 17, process view in Image 18 and physical view in Image 16 plus a view to represent the use cases in Image 19 [10]. The architecture used in this system is developed on top of the existing Inspector-system. The main idea was to separate the database, user interface, event processing and data processing logic from each other.

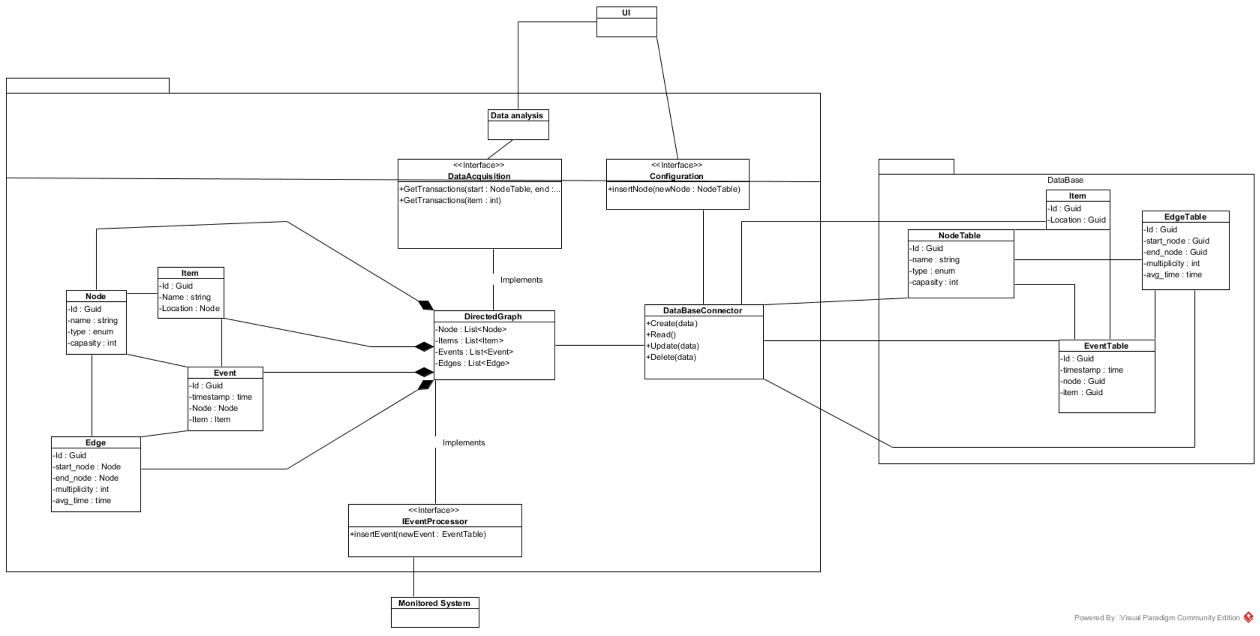


Image 15. Logical view of the system as a UML class diagram.

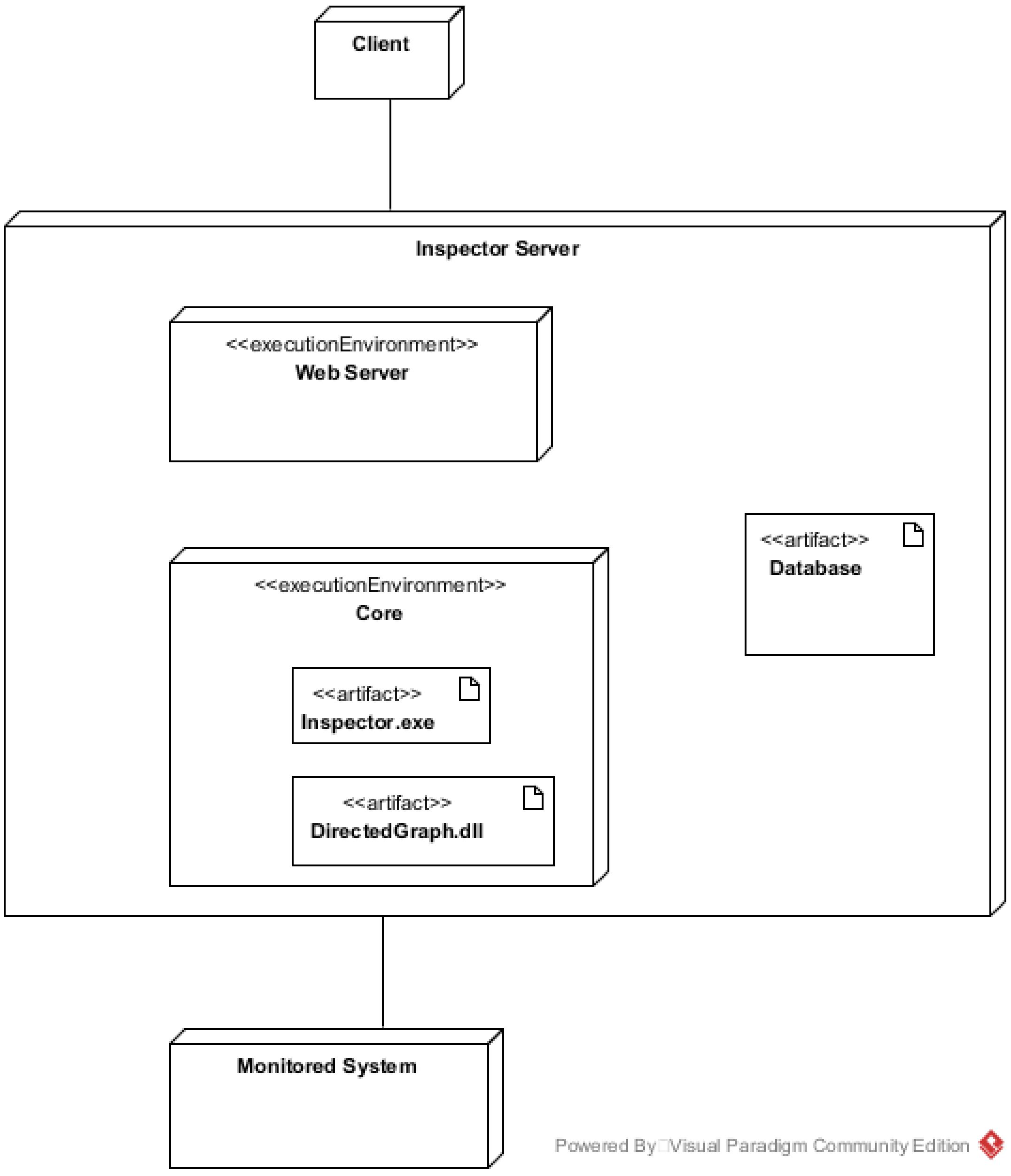


Image 16. Physical view using a UML deployment diagram.

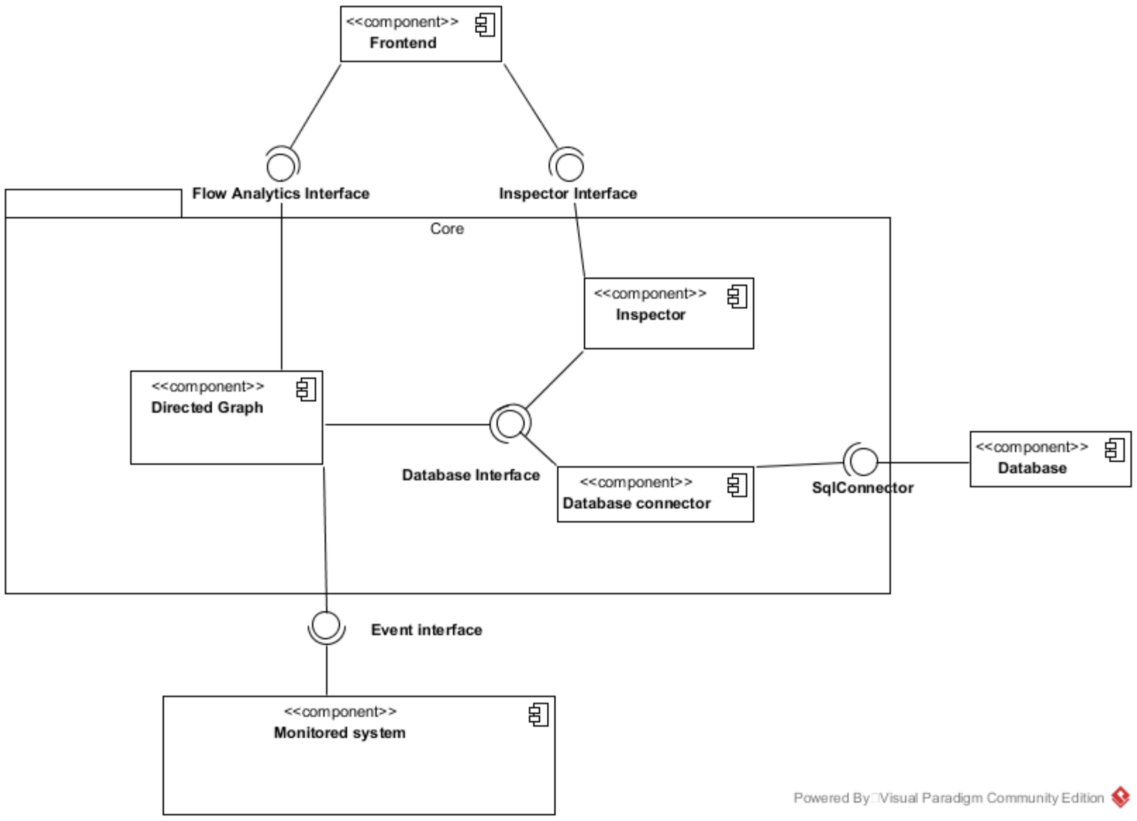


Image 17. Development view using a UML component diagram.

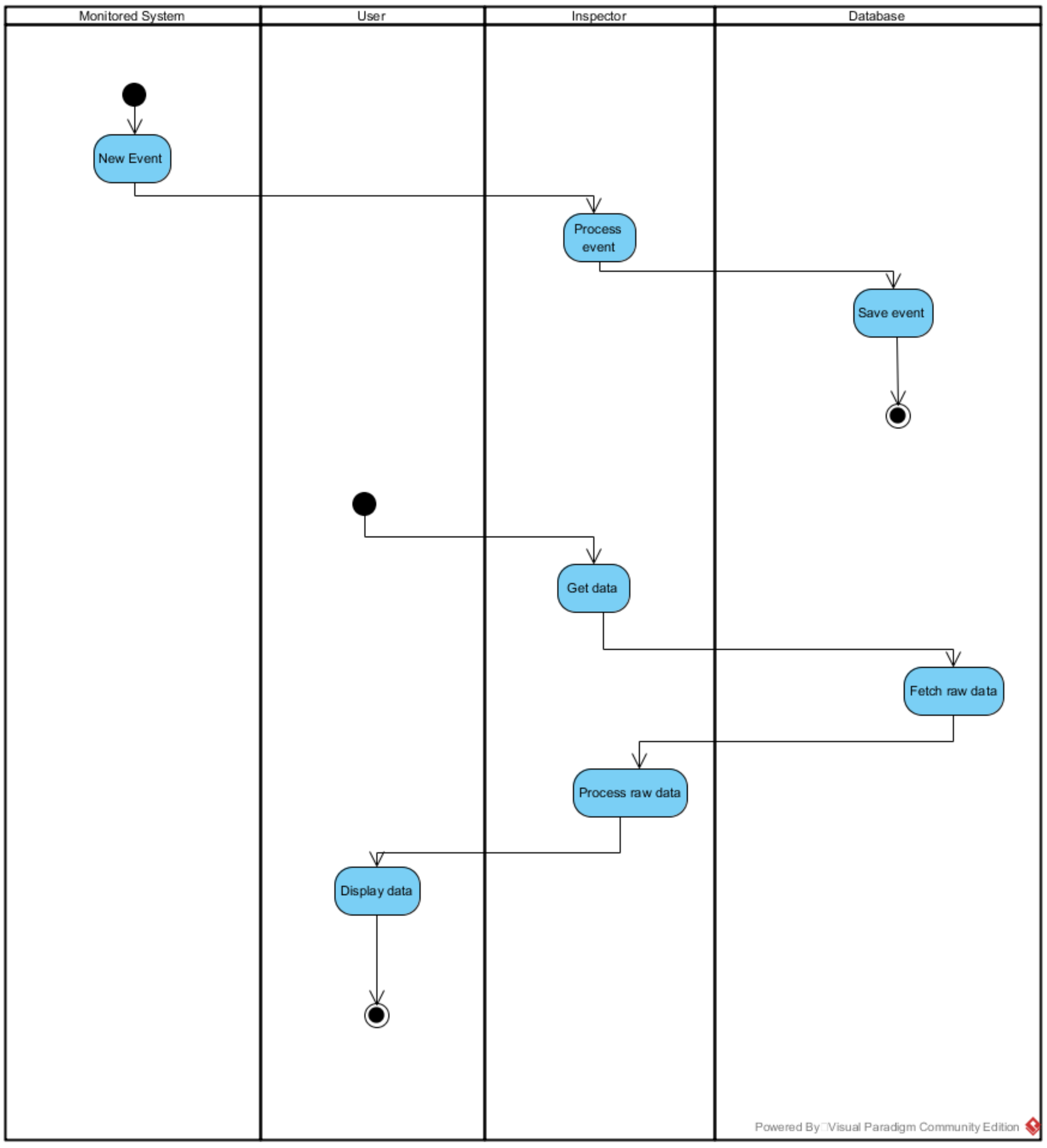


Image 18. Process view of inserting an event into the database and for general data acquisition to the user as a UML activity diagram.

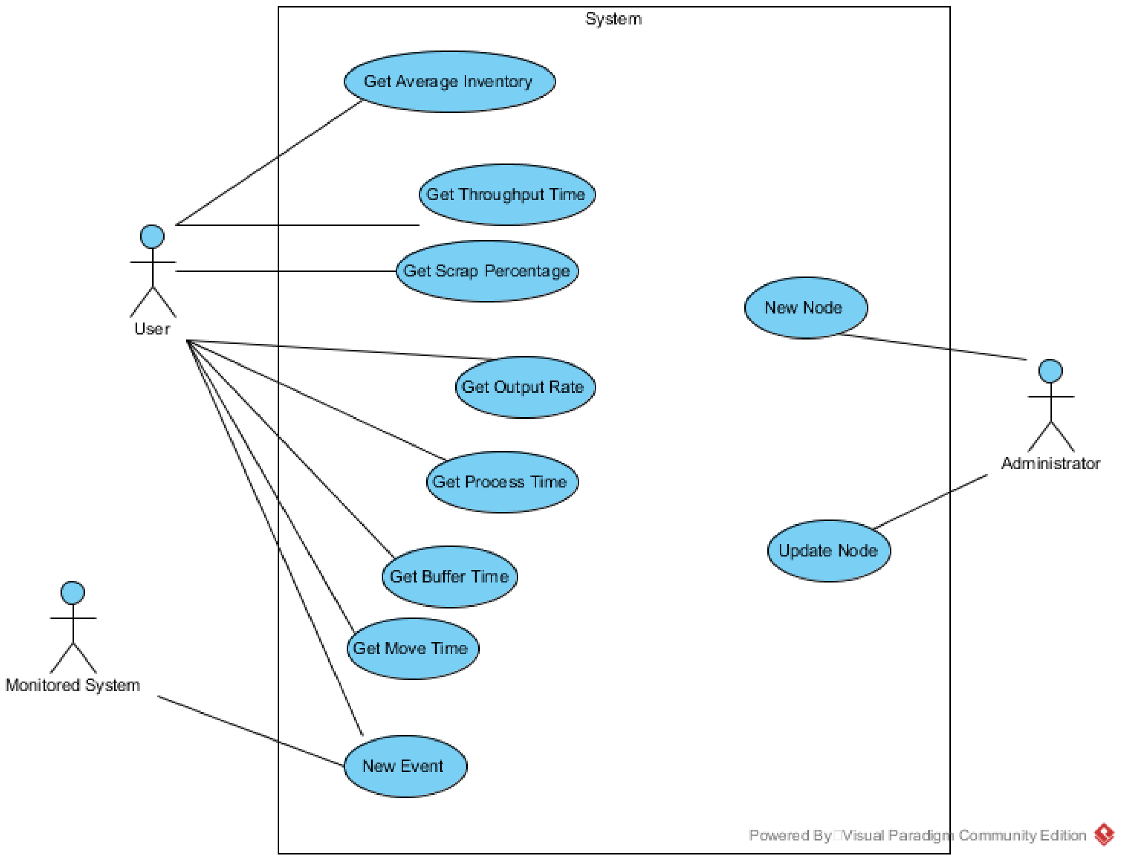


Image 19. UML use case -diagram of the system.

4. SIMULATION SOFTWARE PROTOTYPE

4.1 Performance Comparison Tests

I created a small test setup on a database to compare how the performances on databases using the two data models, directed graph and Petri net, react to different scenarios. In the setup I made a single database, on Windows SQL -server version 14.0.1000.169, where I created the needed tables for both test-cases. The system was run on .NET 4.5.2.

4.1.1 Structure with the Directed Graph

An initial version of the database was designed at Insolution office with the directed graph which is now used here. There are three tables: nodes, edges and transactions. Nodes represent the workstations and transactions realized movements of items between the nodes. Edges are used to store the cumulative data of transactions to make drawing the graph faster. A row in the edges-table represents a realized connection between nodes, the start- and end nodes, the count of how many times it has been triggered and the average duration of realized corresponding transactions. The database is explained in detail in Table 10 and Image 20 where as Image 21 describes the simulated system as a directed graph.

Table 10. Database tables for directed graph

Property	**Type**	**Optional**
	Nodes	
Id	Unique identifier	no
Name	String	no
Type	Enum (value adding, buffer, check point, exit)	no
	Transactions	
Id	Unique identifier	no
Start node	Unique identifier	no
End node	Unique identifier	yes

Start time	Date time	no
End time	Date time	yes
Edges		
Id	Unique identifier	no
Start node	Unique identifier	no
End node	Unique identifier	no
Average duration	Time	no
Count	Long	no

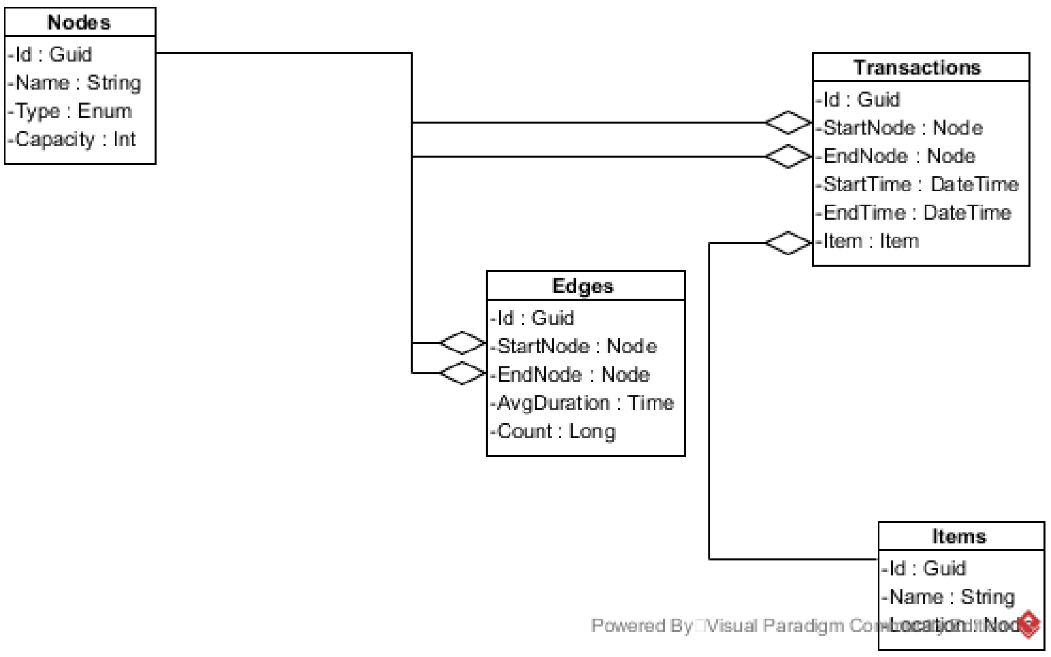


Image 20. The database structure for a directed graph.

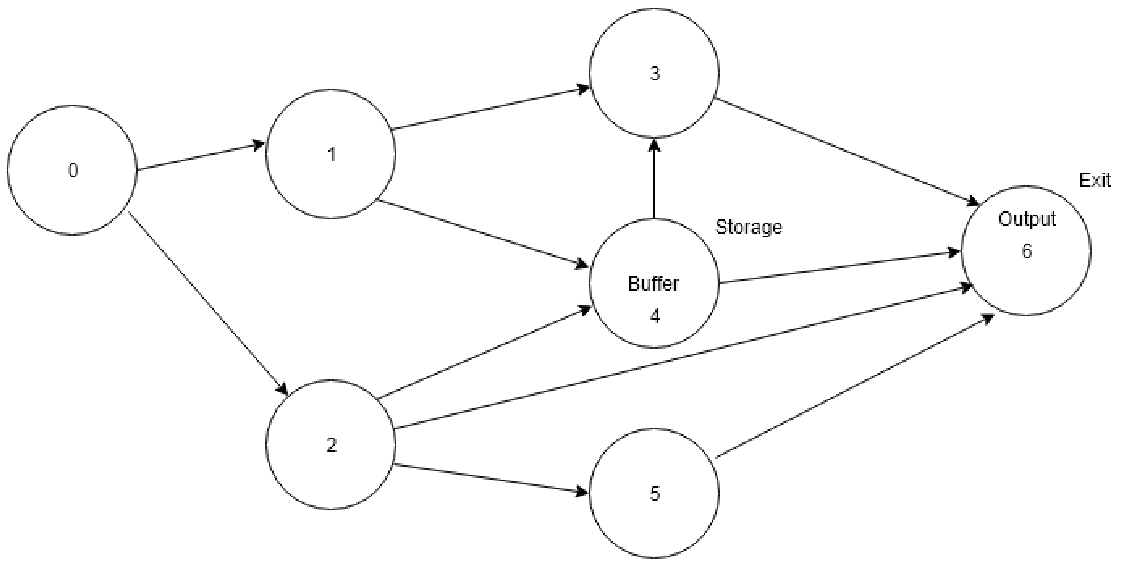


Image 21. The architecture of the demo system with a directed graph.

4.1.2 Structure with Petri Nets

For Petri nets the tables had to be from scratch. The database uses the same idea as with the directed graphs; there is a table for each of the components of petri nets; places, transitions, tokens and connectors plus a fifth table for events as displayed in Image 22 and Table 11. The events-table was added to enable following flows of tokens through the network, since neither the transitions nor the connectors themselves contain the information of when they have been fired.

Opposed to normal behavior of Petri nets, the tokens in the simulation are not deleted and new ones are not created at the transitions, but they are transported through them from places to others. This decision was made in order to follow individual items through the network meaning that the network must be designed so that each transition has an equal amount of incoming connections as it has outgoing ones. In the test case the transitions will have one of each, since the aim is to be able to move single items independent of each other. The the test system is presented in Image 23 as a Petri net.

As the transitions in Petri nets are considered to be immediate, without any delay, it was decided to include two connectors in every event; one beginning at a place and ending at the transition and the other beginning at the transition and ending at another place. Since in the test-system every transition has only one incoming and one outgoing connector this desing decision is unnecessary but this leaves room for later to implement the basic model of Petri nets where there can be several incoming and outgoing connectors.

A connector simply contains the identification of the place and transition it connects. In addition it also contains an enumeration marking the direction in which it leads. This property is used to tell apart the incoming and outgoing connectors in a system, since it is not meant to be possible for items to travel backwards. Despite this the system does support moving the item anywhere, but separate transitions need to be defined for this.

Another difference to normal Petri nets, is that the transitions are not meant to function autonomously. The events are supposed to act as triggers notifying the data model of changes in the physical system, and transitions are used to mark the time a item moves between two workplaces, or places in the data model.

Table 11. Database tables for Petri nets.

Property	**Type**	**Optional**
Places		
Id	Unique identifier	no
Name	String	no
Capacity	Int	no
Transitions		
Id	Unique identifier	no
Name	String	no
Connectors		
Id	Unique identifier	no
Transition	Unique identifier	no
Place	Unique identifier	no
Direction	Enum	no
Tokens		
Id	Unique identifier	no
Name	String	no
Location	Unique identifier	yes
Events		
Id	Unique identifier	no

Timestamp	Date time	no
First connector	Unique identifier	no
Second connector	Unique identifier	no
Token	Unique identifier	no

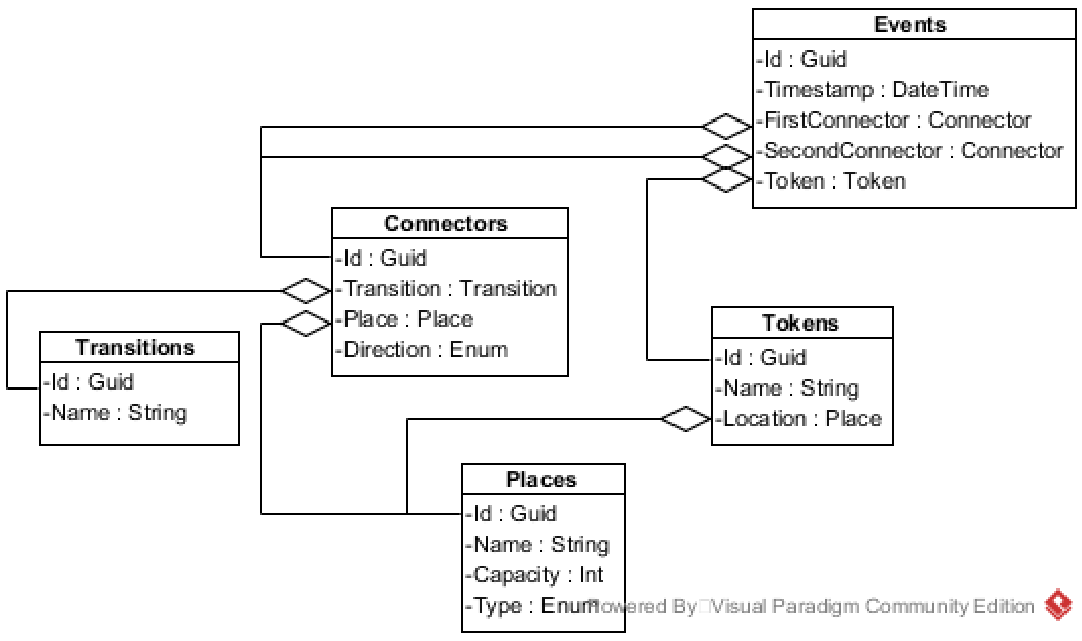


Image 22. The database structure for a Petri net.

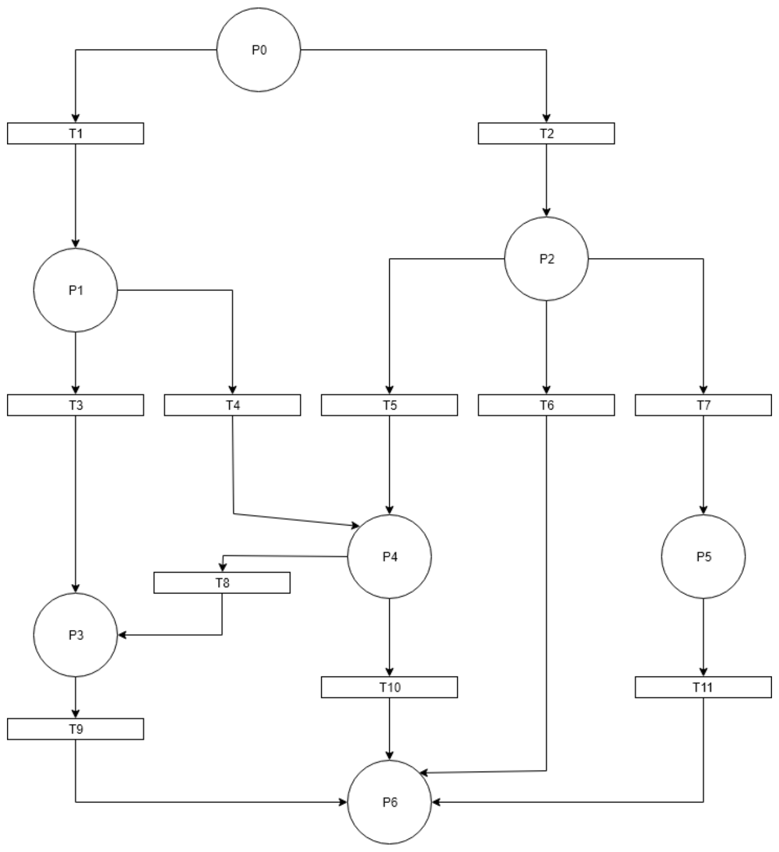


Image 23. The simulated system as a Petri net.

4.1.3 Test Scenarios

For comparing the data models the following test scenarios were created:

1. Inserting a new item into the system.
2. Inserting a new event into the system.
3. Fetching the main data to draw the system.
    a. The data is drawn from a cache.
    b. The data is drawn from the database.
4. Calculating all formed paths between two workplaces.
5. Calculating the throughput for a single item.
6. Fetching the items in a selected workstation on a given moment
    a. The moment is right now.
    b. The moment is some specific time in history.
7. Calculating the average amounts of items on each workstation.

The objective for all these cases is to see how long they have to be processed. They are all running on the same computer with the ms sql server running on localhost, so that the network connection won't affect the results.

4.1.4 Test Process

For running the tests a .NET solution was created to simulate the environment with the previously defined workplaces and model it as a Petri net and a directed graph with the classes described in chapters 4.1.1 and 4.1.2. In the solution I have the workplaces defined in the database beforehand; For the Petri net the whole system, including places, transitions and connectors are already in the database. For the directed graph only the nodes are in the initial situation since the edges are generated into it as events occur between nodes.

The workflow begins by fetching both systems into the cache and creating a single item into the entry point -workplace. The workplaces were hardcoded to match the ones specified in chapters 4.1.1 and 4.1.2 alongside with the movements workpieces are allowed to make between them. The system itself doesn't restrict the movement of workpieces but it made the simulation easier and this way it should match real life where the pieces also mostly have predefined routes. After the workpiece is generated it is run through the system using a randon number -generator to make the decisions of where to go next or whether to go at all. Most of the pieces should make it to the exit but some may be left on some other workstation.

All the events in the simulation happen the moment they are triggered but their duration is calculated with a random number -generator to be something between one and fifteen seconds. The simulation is run for 300 items and the calculations are done with 100 repetitions. The duration of each function is measured separately for the Petri net and the directed graph.

Every insertion in the simulation is executed as it would be in a real system. This means that the average durations for edges and transitions as well as the capacities for nodes and places are updated according to workpiece movements. There is an object for handling directed graphs and it contains a public interface for entering new items events and requesting data or calculations. There is another similar object for handling Petri nets which provides the same interface. The main-program of this simulation calculates the events by itself, but uses these two objects for storing and handling the data, just as in the real application it would be possible

4.1.5 Calculating the Formed Paths

The calculation of the formed paths is executed by finding all the transactions that started from the given start node. If a time interspan is given, then the transactions are filtered accordingly. From these transactions the item is selected and the latest transaction regarding that item that end in the given end node is selected. For each item, all transactions between these two are selected, ordered by time and inserted into a list so that they form the path this specific piece has travelled. All the paths of items that have a complete path are inserted into a list of lists which is then given as a return value. This is described as a sequence diagram in Image 24.

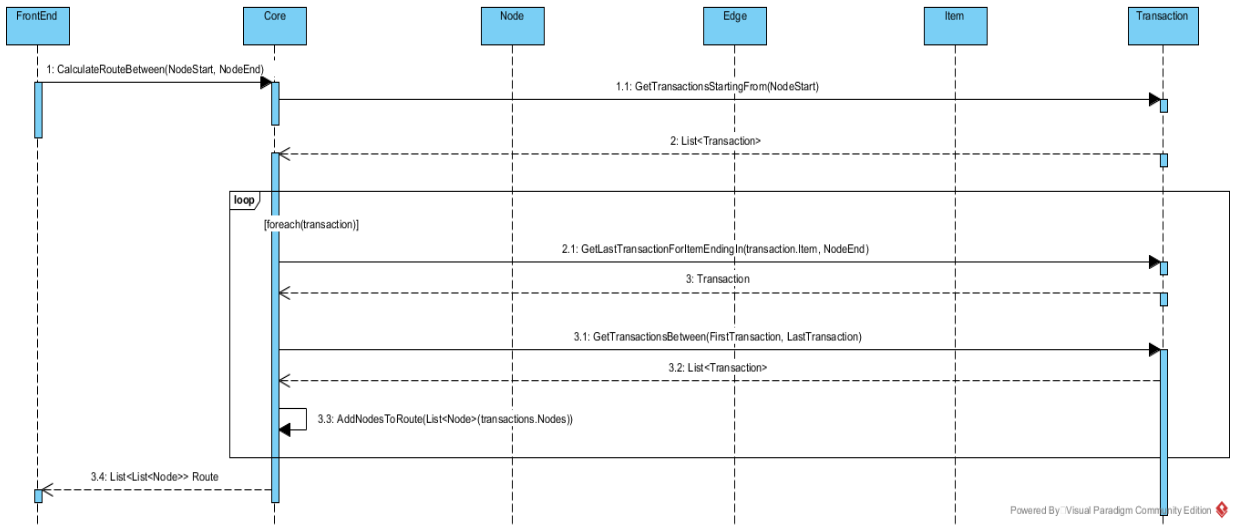

FrontEnd
Core
Node
Edge
Item
Transaction
1: CalculateRouteBetween(NodeStart, NodeEnd)
1.1: GetTransactionsStartingFrom(NodeStart)
2: List<Transaction>
loop
[foreach(transaction)]
2.1: GetLastTransactionForItemEndingIn(transaction.Item, NodeEnd)
3: Transaction
3.1: GetTransactionsBetween(FirstTransaction, LastTransaction)
3.2: List<Transaction>
3.3: AddNodesToRoute(List<Node>(transactions.Nodes))
3.4: List<List<Node>> Route
Powered By Visual Paradigm Community Edition

Image 24. Sequence diagram for calculating the formed routes between two nodes in a directed graph.

There was an idea to execute this using a recursive function to probe through the network, but this new way notices if an item has made loops during its course, possibly pointing out that there has something gone wrong in the manufacturing process.

4.1.6 Calculating the Throughput Time

A sequence diagram for calculating the throughput time for a single item in a defined time-scope is described in Image 25. It begins by finding the first and last events for the item. If starting place, ending place or both are defined the events that happen outside them in the system are ignored. The difference is finally calculated for these two events. In this simulation I used the entry and exit points of the system as these parameters so the throughputtime is calculated for the whole system. Hoda Khalil and Yvan Labiche have also suggested a method for finding all spanning trees in their work, but it poses the same problem as the recursive probing; it misses loops. [11]

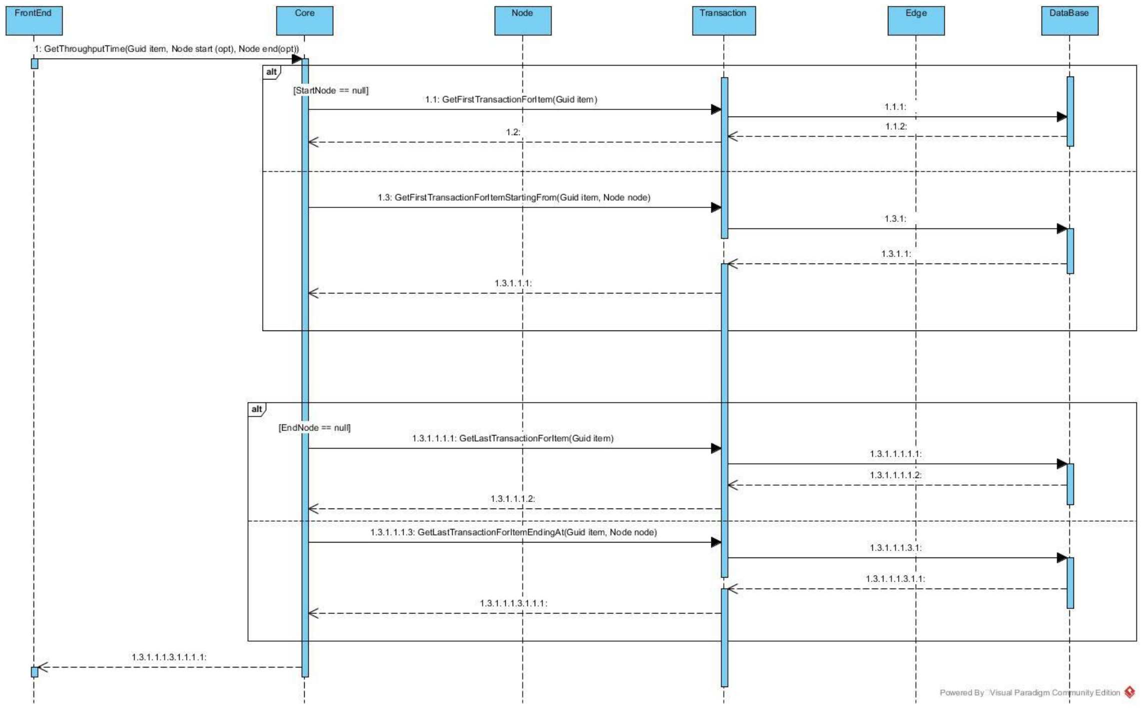

FrontEnd
Core
Node
Transaction
Edge
DataBase
1: GetThroughputTime(Guid item, Node start (opt), Node end(opt))
alt
[StartNode == null]
1.1: GetFirstTransactionForItem(Guid item)
1.1.1:
1.1.2:
1.2:
1.3: GetFirstTransactionForItemStartingFrom(Guid item, Node node)
1.3.1:
1.3.1.1:
1.3.1.1.1:
alt
[EndNode == null]
1.3.1.1.1.1: GetLastTransactionForItem(Guid item)
1.3.1.1.1.1.1:
1.3.1.1.1.1.2:
1.3.1.1.1.2:
1.3.1.1.1.3: GetLastTransactionForItemEndingAt(Guid item, Node node)
1.3.1.1.1.3.1:
1.3.1.1.1.3.1.1:
1.3.1.1.1.3.1.1.1:
1.3.1.1.1.3.1.1.1.1:
Powered By Visual Paradigm Community Edition

Image 25. Sequence diagram for fetching the throughput time with a directed graph.

4.1.7 Calculating the Average Inventory

The methods for calculating the average amount of items in workplaces in the demo software were created so that they would be easy to deploy in the real application and so they support restricting the calculation by part type. The method requires a time window as two parameters. The result can be divided by an enumerated resolution into smaller time windows. For this test, the methods were run for all parts and the results are returned as a dicionary with the key representing the workplace in question and the value being an integer representing the average amount of items.

The idea in the function is simply to calculate the amount of items at the start and end of the time window, and calculate their average value. If a time resolution is given the time window is split into smaller windows accordingly and the calculation is done for each of them separately.

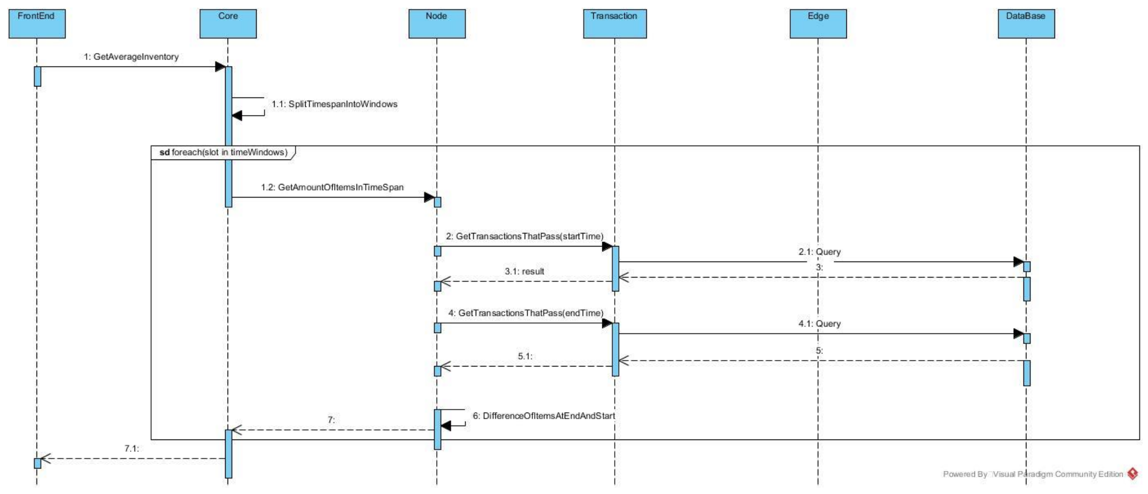

FrontEnd
Core
Node
Transaction
Edge
DataBase
1: GetAverageInventory
1.1: SplitTimespanIntoWindows
sd foreach(slot in timeWindows)
1.2: GetAmountOfItemsInTimeSpan
2: GetTransactionsThatPass(startTime)
2.1: Query
3:
3.1: result
4: GetTransactionsThatPass(endTime)
4.1: Query
5:
5.1:
6: DifferenceOfItemsAtEndAndStart
7:
7.1:
Powered By Visual Paradigm Community Edition

Image 26. Sequence diagram for fetching the average inventory in a directed graph.

4.2 Results of the Tests

After the tests were run the chart in Image 27 was drawn to visualize the average durations for each test scenario. The same data is represented in Table 12. In the table the difference-column shows how many milliseconds shorter was the calculation with the directed graph, and the ones with a negative number are colored red. The percentual-column describes how many percents the directed graph duration is of the Petri net duration. More detailed results are available in appendix A.

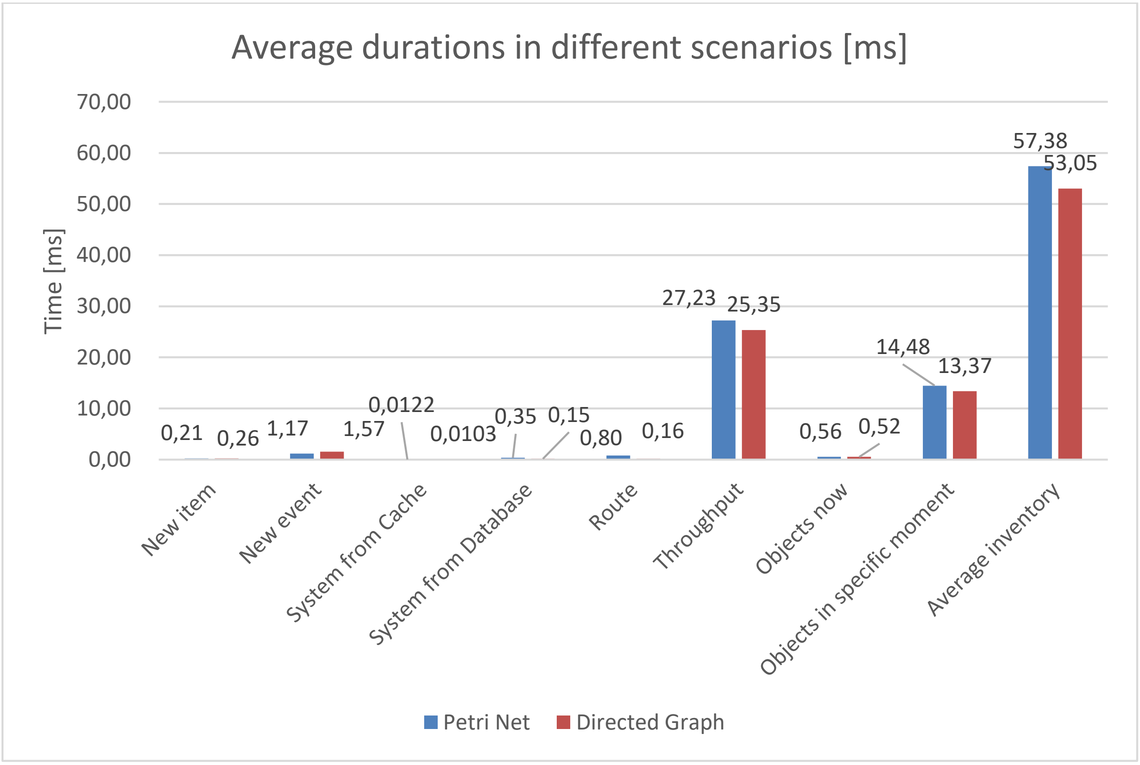


Image 27. Bar chart describing the average durations of different test scenarios.

Table 12. Average durations of different test scenarios

Scenario	Petri Net Time [ms]	Directed Graph Time [ms]	Difference	Precentual
System initiali-zation	40,47	2,14	38,33	5,30
New Item	0,21	0,26	-0,05	122,60
New Event	1,17	1,57	-0,40	134,21
System From Cache	0,0122	0,0103	0,0020	84,02
System From Database	0,35	0,15	0,20	42,00
Route Calcula-tion	0,80	0,16	0,65	19,42
Throughput Cal-culation	27,23	25,35	1,88	93,11
Objects in Workplaces Now	0,56	0,52	0,04	93,21
Objects in Workplaces at Specific Mo-ment	14,48	13,37	1,11	92,33
Average Inven-tory Calculation	57,38	53,05	4,33	92,45

From Table 12 and Image 27 can be seen that on most occasions the directed graph takes approximately 80 to 90 percent of the time the Petri net needs. The only exceptions in this test were the insertions of new items and events. It must be noted that they are also the shortest procedures resulting in a smaller absolute difference.

The system initialization mentioned in Table 12 can’t really be taken into consideration since it is supposed to only happen once on start-up. In this test the directed graph was built to be dynamic and create the edges according to events and in the initialization only the nodes were retrieved from the database. The Petri net wasn’t built to be so dynamic since its definition states that the model should be predefined. This was the reason that during the initialization in addition to the places, that represent the workplaces in my test just like nodes, also transitions and connectors were retrieved from the database. This resulted in 7 nodes to be retrieved in a single query from the directed graph in opposition to 7 places, 10 transitions and 22 connectors in three queries. This explains some of the difference in the initialization time, but still it took 20 times as much time to initialize the Petri net as it took to initialize the directed graph.

5. CONCLUSIONS AND NEXT STEPS

5.1 Concluding Remarks

From the tests described in chapter 4.1 can be seen that for a system that is supposed to focus on monitoring different flows through discrete events in a system a directed graph gives better performance in most use-cases and overall. This is mostly due to the simpler structure of the data-model. Also by definition places or transitions cannot be added dynamically into a Petri net whereas for a directed graph nodes can be added easily during execution and edges can be generated alongside the events. Of course we could have implemented an applied version of the Petri net that could have accomplished this same flexibility but the transitions and connectors would still have been a heavier model compared to the simple edges of the directed graph.

The Petri net does have its advantages if we want to simulate the functionality of a system or control it. The triggering mechanism of the transitions enhanced with external conditions makes it a powerful tool in these cases. For our application these are functionalities that we do not need or want. We decided that we want to use as nonintrusive monitoring as possible and for that reason a simple directed graph seems to be the best solution.

The data to be collected by the system could come from anywhere where a timestamp, item id and a workplace id can be identified. Simple event-based notifications from the source would be enough through, for example, a simple REST-interface. As mentioned in chapter 3.4.7, the source could be a graphical user interface or a simple barcode scanner connected to a Raspberry Pi.

The data should be preprocessed before sending it to the frontend. There are proposals in chapters 4.1.5, 4.1.6 and 4.1.7 for the most complex use-cases. In the current version all the calculations are executed on the software sometimes requiring multiple database queries to get all the necessary data. For larger quantities of data the time required for the calculations seems to grow. There is also some growth in the query-times but the looping mechanisms in the calculations seem to be the largest bottleneck in these cases.

As was described in chapter 3.5, the frontend service will be running on the same computer as the backend. It will provide the visualizations about the data to a client in a way that is not part of this thesis. The backend service will provide an interface for the frontend service consisting of methods described in chapters 3.4.2, 3.4.3, 3.4.4, 3.4.5 and 3.4.6.

5.2 Next Steps

There are some minor improvements to implement into this simple monitoring system after it has been succesfully integrated into the existing Inspector-system. There have been ideas to define possible nodes a piece can move to from a specified node, creating a kind of forced flow in the system. The idea for this is not to force the flow in the actual system, but rather to fix mistakes made by operators in logging the movements. For example if a piece must be registered when entering a workplace and when exiting it, but after the enter-event our application receives an event from another workplace our application can create the missing event automatically to create correct data.

To prevent the system from slowing down in the future due to the calculations being done on the software, as noted in the chapter 5.1, the calculations could be executed as stored procedure to reduce the amount of queries to one. The database server also should provide a more efficient way to handle and calculate the data before sending the result to the application-server.

Although controlling the flow would be a tempting idea, that would be in contradiction of the basic Inspector system and we do not wish to implement it. If it nontheless did some day be a requirement, we would need to think about the data model again if the directed graph is indeed the best solution. In that case the dynamic properties of the system would also come into question; wheter it is reasonable to control a changing system and how to implement it.

REFERENCES

[1] C.A. Petri, Kommunikation mit Automaten, 1962, Available: http://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160.

[2] R. David, H. Alla, Discrete, Continuous, and Hybrid Petri Nets, 1. Aufl. ed. Springer-Verlag, 2010, .

[3] C. Sibertin-Blanc, HIGH LEVEL PETRI NETS WITH DATA STRUCTURE, Jul, Espoo, Finland, pp. 141-168.

[4] M. Rausch, H. M. Hanisch, Net condition/event systems with multiple condition outputs, Emerging Technologies and Factory Automation, 1995. ETFA '95, Proceedings., 1995 INRIA/IEEE Symposium on, pp. 600 vol.1.

[5] R. Valette, J. Cardoso, D. Dubois, Monitoring manufacturing systems by means of Petri nets with imprecise markings, Proceedings. IEEE International Symposium on Intelligent Control 1989, pp. 233-238.

[6] P. Merlin, D. Farber, Recoverability of Communication Protocols - Implications of a Theoretical Study, IEEE Transactions on Communications, Vol. 24, Iss. 9, 1976, pp. 1036-1043. Available (accessed ID: 1): .

[7] J. Bang-Jensen, G. Gutin, Digraphs : Theory, Algorithms and Applications, 2nd ed. Springer, London, 2010, .

[8] P1099 Using directed graphs for examining transitions in antibiotic usage, in: International Journal of Antimicrobial Agents, 2007, pp. S296.

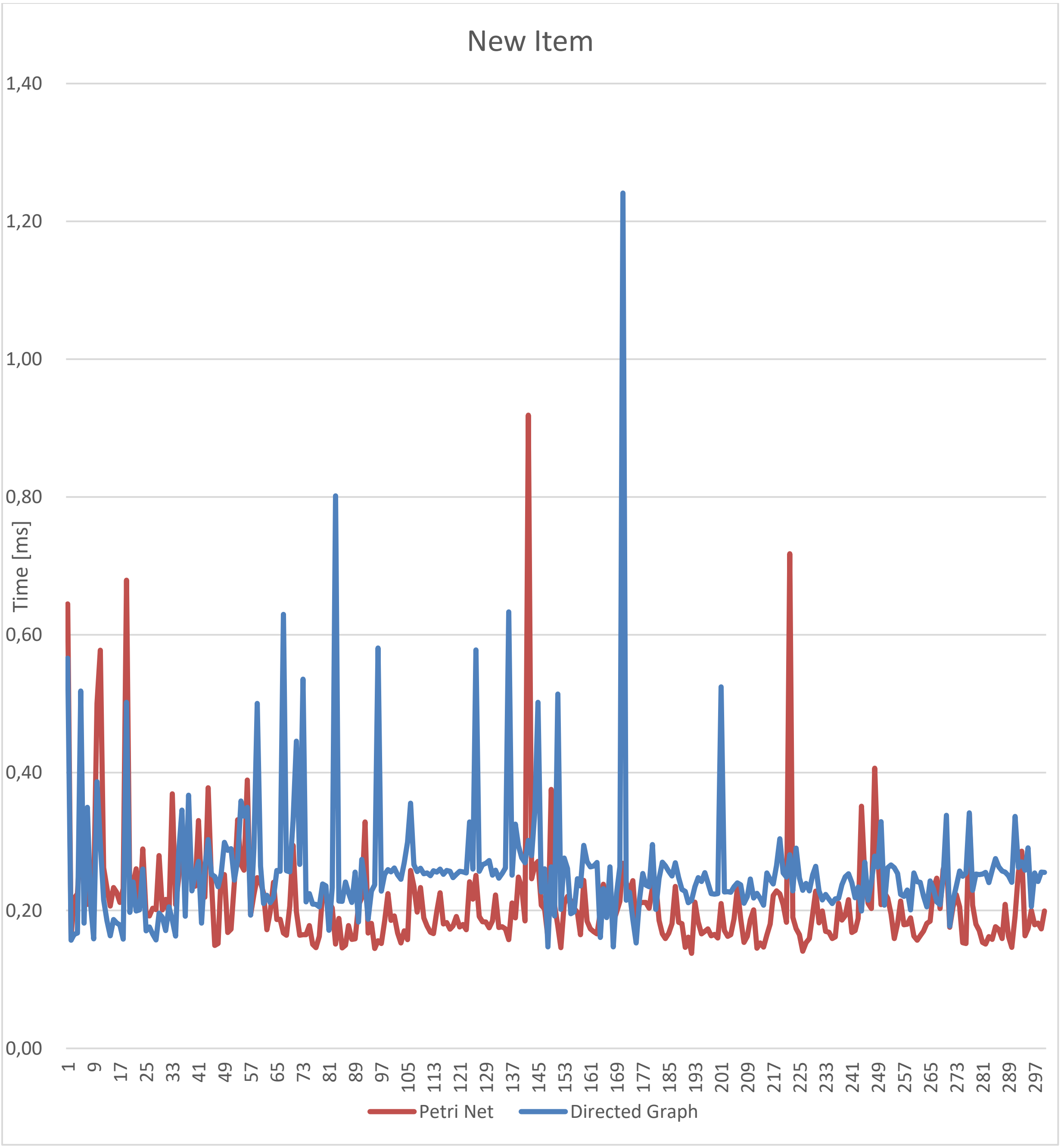
[9] K.H. Movric, F.L. Lewis, Cooperative Optimal Control for Multi-Agent Systems on Directed Graph Topologies, IEEE Transactions on Automatic Control, Vol. 59, Iss. 3, 2014, pp. 769-774.

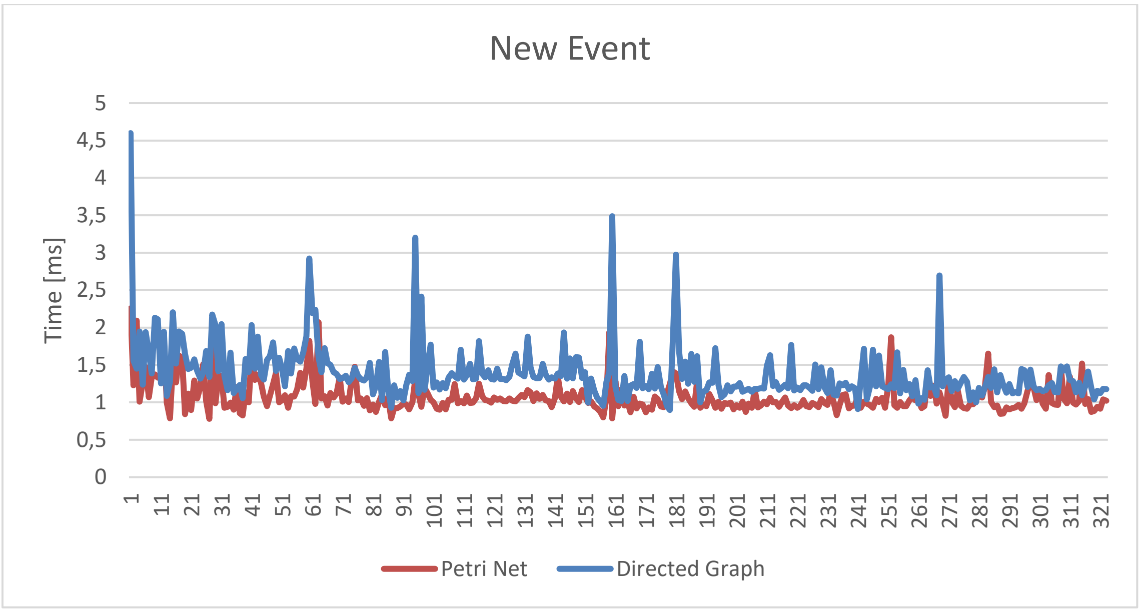
[10] P.B. Kruchten, The 4+1 View Model of architecture, IEEE Software, Vol. 12, Iss. 6, 1995, pp. 42-50.

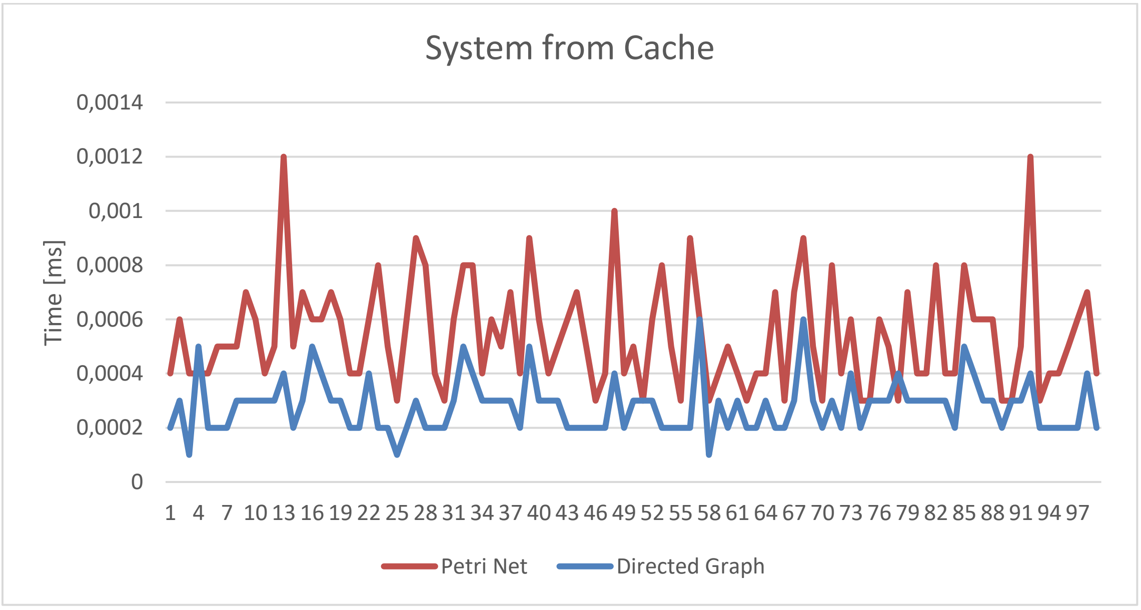
[11] H. Khalil, Y. Labiche, Finding All Breadth First Full Spanning Trees in a Directed Graph, 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), pp. 372-377.

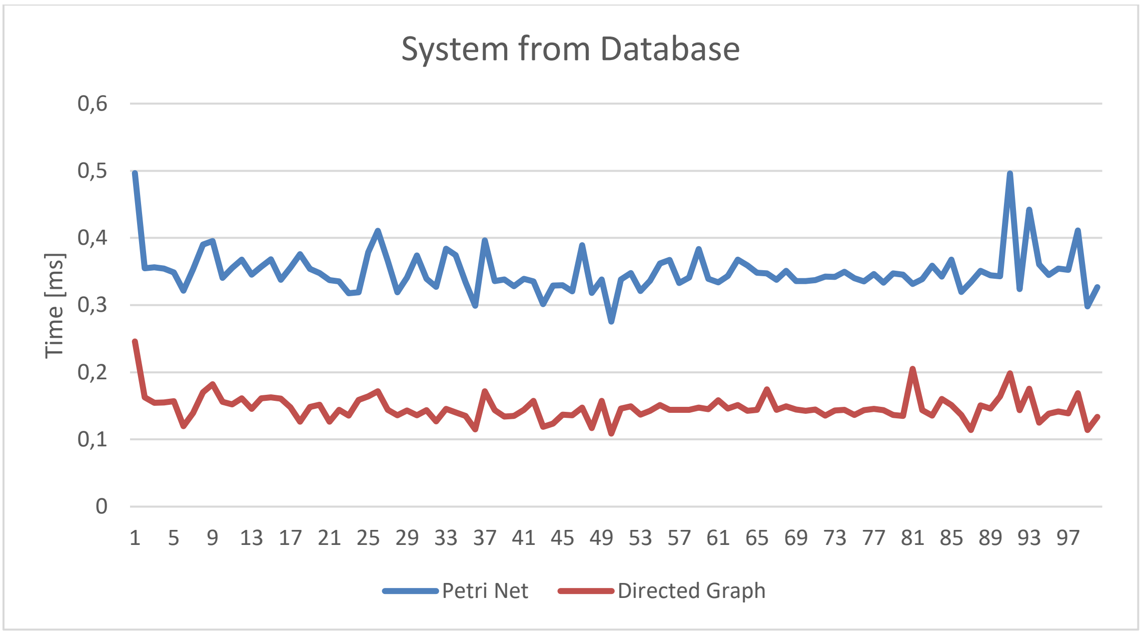
APPENDIX A: RESULTS OF SEPARATE TEST SCENARIOS

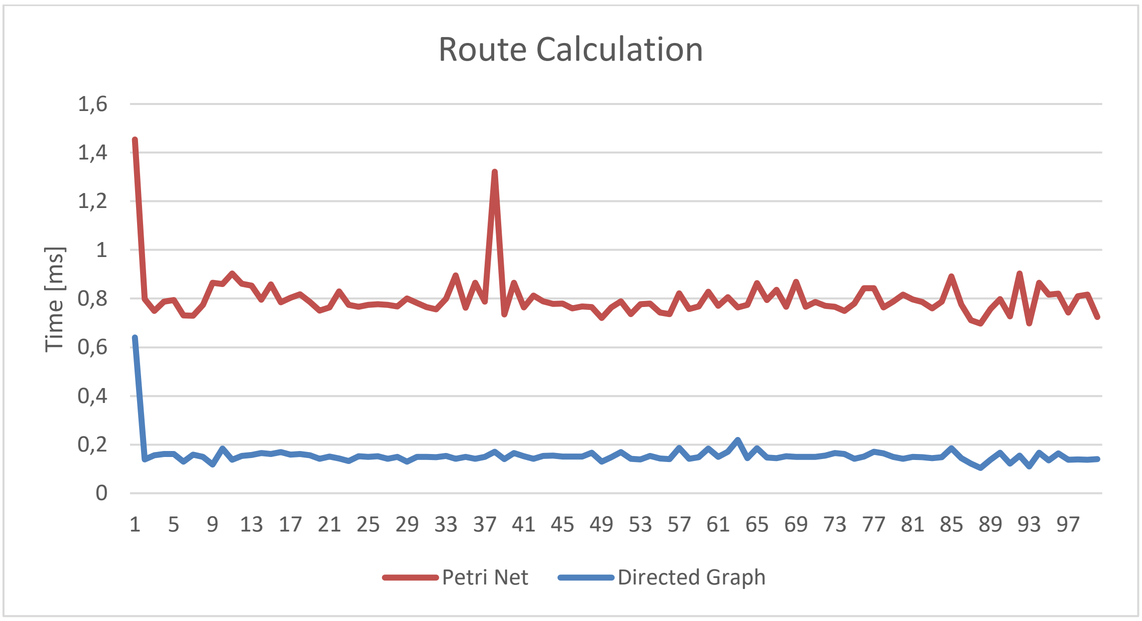
In this appendix are represented the more detailed results of the performance comparisons implemented in the thesis. Each graph represents a single test and each point marks the duration of a single execution of that test. The red values are the times used by Petri nets and the blue ones by directed graphs. Even though depicted as a line-graph the sequential values have no connection beyond being executed sequentially.

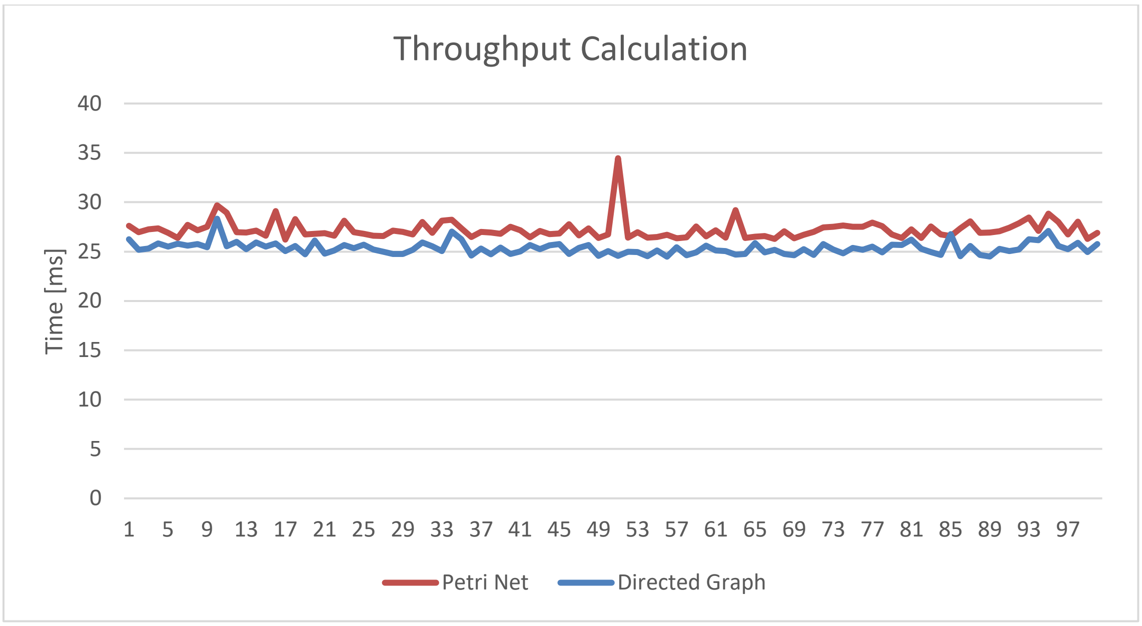


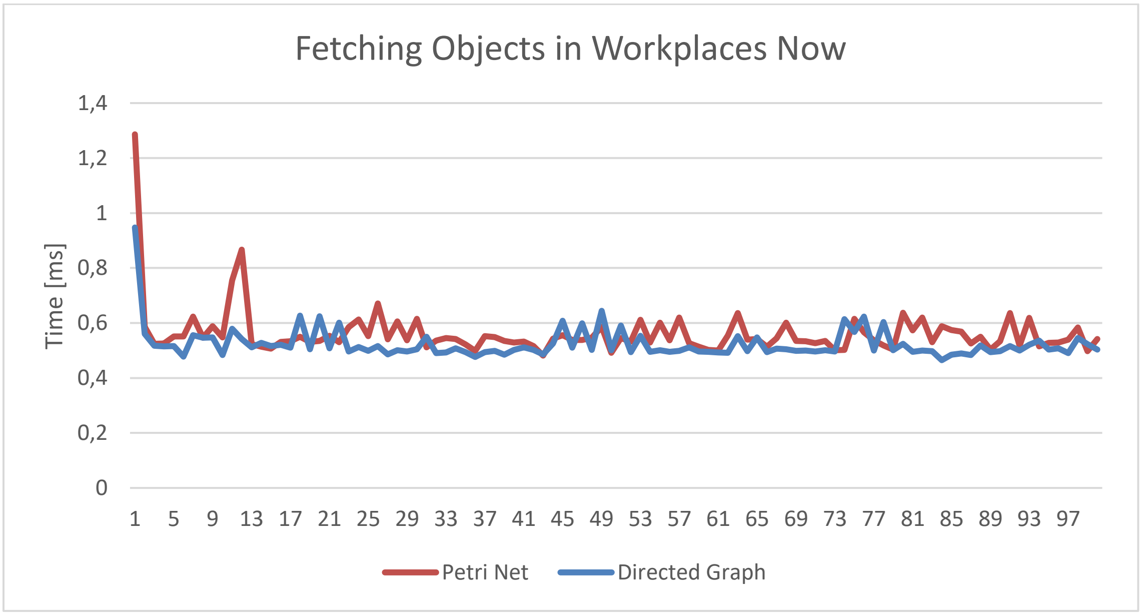

New Event
Time [ms]
5
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
1 11 21 31 41 51 61 71 81 91 101 111 121 131 141 151 161 171 181 191 201 211 221 231 241 251 261 271 281 291 301 311 321
Petri Net
Directed Graph

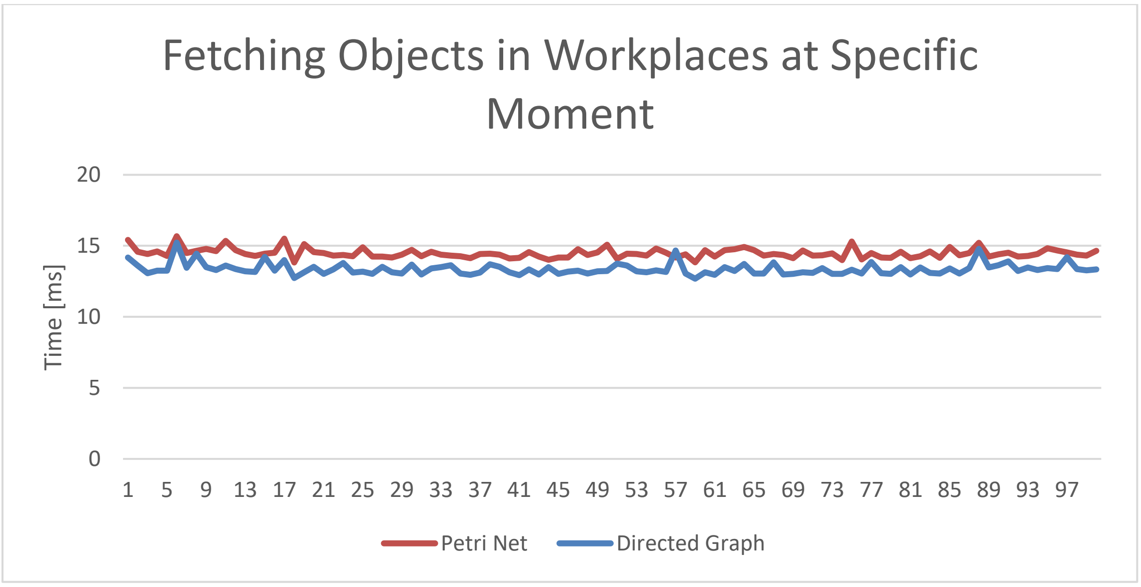

System from Cache
Time [ms]
0,0014
0,0012
0,001
0,0008
0,0006
0,0004
0,0002
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97
Petri Net
Directed Graph

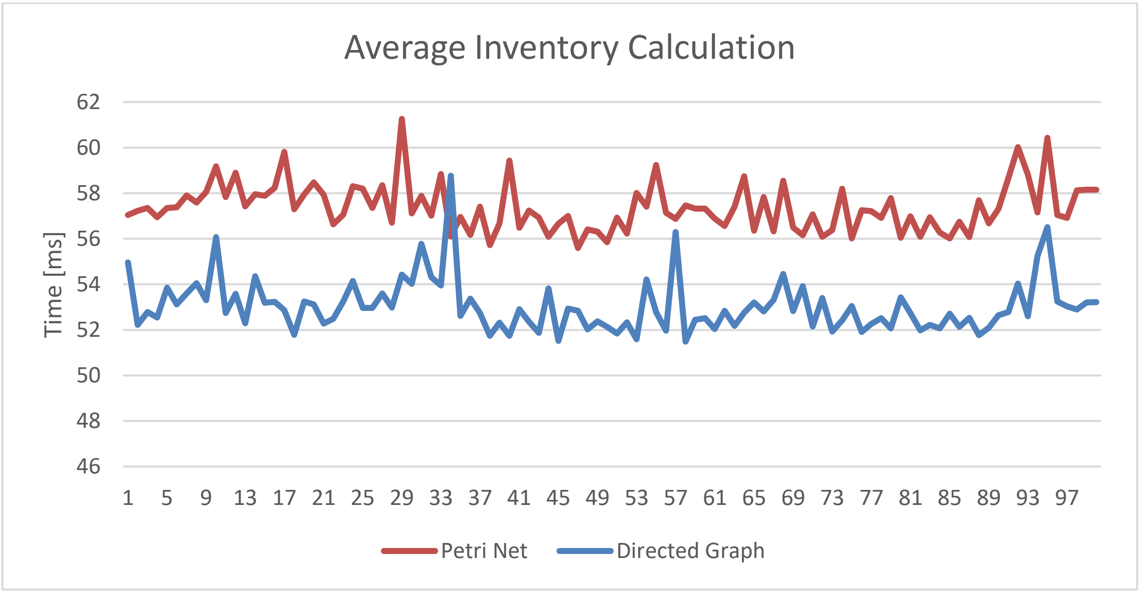

System from Database
0,6
0,5
0,4
0,3
0,2
0,1
0
Time [ms]
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
Petri Net
Directed Graph


Route Calculation
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Time [ms]
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
Petri Net
Directed Graph


Throughput Calculation
Time [ms]
40
35
30
25
20
15
10
5
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
Petri Net
Directed Graph


Fetching Objects in Workplaces Now
Time [ms]
1,4
1,2
1
0,8
0,6
0,4
0,2
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
Petri Net
Directed Graph


Fetching Objects in Workplaces at Specific Moment
Time [ms]
20
15
10
5
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
Petri Net
Directed Graph


Average Inventory Calculation
Time [ms]
62
60
58
56
54
52
50
48
46
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
Petri Net
Directed Graph