

**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

MASTER THESIS

Tomáš Filípek

Smart Infrastructure Visualization

Department of Distributed and Dependable Systems

Supervisor of the master thesis: doc. RNDr. Petr Hnětynka, Ph.D.

Study programme: Computer Science

Study branch: Discrete Models and Algorithms

Prague 2016

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In.....Date.....

signature

I would like to thank my supervisor doc. RNDr. Petr Hnětynka, Ph.D. for valuable advice he offered me during making of this work.

Název práce: Vizualizátor inteligentní infrastruktury

Autor: Tomáš Filípek

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: doc. RNDr. Petr Hnětynka, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Výpočetní síla přenosných zařízení v poslední době výrazně stoupla. Jednou z výhod, kterou to přináší, je možnost nasazení nových druhů distribuovaných systémů, jako například komponentových systémů založených na ensemblech (EBCS). Z praktických důvodů jsou zpravidla EBCS systémy před dokončením testovány pomocí simulací. Často však bývá obtížné interpretovat výstup z takových simulací, neboť je obvykle ve formátu XML, jenž je uzpůsoben strojovému čtení. Vytvořili jsme aplikaci pro grafickou vizualizaci těchto dat - v současné podobě dokáže graficky znázornit výstup z aplikací postavených na JDEECo komponentovém modelu, ale může být snadno upravena i pro jiné EBCS systémy. Aplikace dokáže znázornit komponenty a ensembly, jejichž grafickou realizaci je možné měnit pomocí připravené skriptovací konzole. Dále aplikace nabízí možnost rozšíření pomocí zásuvných modulů. Provedené měření výkonu ukazuje, že při použití typických vstupů aplikace běží rozumně rychle.

Klíčová slova: DEECo, JDEECo, smart Cyber-Physical Systems, Vizualizace

Title: Smart Infrastructure Visualization

Author: Tomáš Filípek

Department / Institute: Department of Distributed and Dependable Systems

Supervisor of the master thesis: doc. RNDr. Petr Hnětynka, Ph.D., Department of Distributed and Dependable Systems

Abstract: Computational power of mobile devices has been continuously improving in the recent years. One of the benefits which it brings, is feasibility of new kinds of distributed systems, such as Ensemble-Based Component Systems (EBCS). For practical reasons, EBCS systems are usually tested using simulations before being released. However, it can be difficult to interpret the simulation output, as it is usually contained in XML format, which is more suited to be read by machines than by people. We provide a visualizing application, which creates a graphical representation of such a simulation output. Out of the box, it is able to visualize data from applications built on top of the JDEEC_o component model, but it can be easily modified to accept output from different EBCS applications. It is able to visualize both components and ensembles and provides a scripting interface to modify the graphical output. In addition, it has an extensibility mechanism for adding new functionalities. Our benchmarking shows that the application is expected to run reasonably fast in typical scenarios.

Keywords: DEEC_o, JDEEC_o, smart Cyber-Physical Systems, Visualization

Contents

1. Introduction	8
1.1. Goals	9
1.2. Structure of the Thesis	10
2. Background	11
2.1. Smart Cyber-Physical Systems	11
2.2. Ensemble-Based Component Systems (EBCS)	12
2.3. DEECo Component Model	13
2.4. JDEECo: an Implementation of DEECo	15
2.5. Delimitation of the Visualizer Application	16
2.6. Goals Revisited	17
3. Description of the Solution	19
3.1. Resolution of the Goals	19
3.1.1. Support for Environment Infrastructure	19
3.1.2. Customizable Environment Presentation	21
3.1.3. Support for EBCS Components	25
3.1.4. Support for Ensembles	28
3.1.5. Support for Large Input Files	31
3.1.6. Reasonable Loading Times	33
3.1.7. Platform Independence	33
3.1.8. The Scripting Interface	34
3.2. Basic Architecture of the Application	35
3.2.1. Parsing the Input	35
3.2.2. Internal Representation of the Input	36
3.2.3. Transition to JavaFX Representation	37
3.3. Evaluating the Performance	39
3.3.1. Employed Techniques	39
3.3.2. Searching the Big Files	42
3.3.3. Input Processing	46
4. Using and Customizing the Application	50
4.1. Supplying the Input	50
4.2. Extending the Application with Plugins	51
4.2.1. Information Plugin	52
4.2.2. Filter Plugin	53
4.3. Scripting Interface	53
4.3.1. Customizing the Environment	54
4.3.2. Customizing the Components	56
4.3.3. Customizing the Ensembles	57

5. Related Works	59
6. Conclusion and Future Work	60
Bibliography	61
List of Abbreviations	63
Attachments	64

1. Introduction

In the recent years, the computational power of mobile devices has considerably improved. It is not uncommon that smart-phones in the middle price range now ship with up to 8-core processors and RAM memory comparable to that of low-end desktop computers. Some would argue that the main benefit of this technological evolution is an increase of comfort for mobile users. Certainly, it is now much more pleasant to work with a mobile internet browser or a video player, as it no longer takes ages to load the application. However, it also created new opportunities – with the improved hardware, it became possible to consider previously unfeasible forms of distributed software systems.

A prime example of such systems, only made feasible by the improved hardware, are the *smart cyber-physical systems (sCPS)* [4][5]. We can characterize them as distributed, decentralized, heterogeneous and open-ended systems, which can moreover interact with the surrounding environment. The interaction is supposed to be carried out both ways; the system may retrieve information from the environment as well as influence the environment in a certain way. A nice example of an sCPS is an intelligent navigation system for cars, which monitors the traffic (using communication with other members the sCPS) and chooses the optimal route based on both the static map and the current traffic.

To help with implementing an sCPS, the *ensemble-based component systems (EBCS)* [2] software concept was developed. It is a mixture of different computational models and concepts which are otherwise usually used in a separate manner. EBCS borrows notions and design principles from component-based software engineering, agent-oriented computing, ensemble-oriented systems and control-systems engineering. The architecture of such systems is based on components, whose behavior is completely determined locally. Communication of components is done exclusively through dynamically created ensembles.

Having an appropriate software concept is still a long way from building an actual application that will run the sCPS. *Dependable Ensembles of Emerging Components (DEECo)* [2][8] is a refinement of the EBCS into a software engineering model. In other words, DEECo can already be used by software architects to design the structure of an application [6], unlike sCPS, which is just a general concept. To enable the design process, DEECo provides the software architects with three pivotal concepts – the *runtime*, *component* and *ensemble*. As the components and ensembles can be easily translated into classes in an object-oriented programming language, the main challenge in implementing DEECo was creating a proper runtime. This was addressed by *JDEECo* [2][7], a pure Java implementation of DEECo. It provides a software library for building new applications on the DEECo principles, thus effectively making the final step on the journey from an abstract sCPS to a working application.

Using the DEECo software engineering model, together with its Java implementation, it is possible to build sCPS systems. However, a new problem arises. When developing desktop software or even web applications, we usually have good opportunities to test the product on the target platforms. We can simply test the application on a local machine with configuration similar to that of customers' machines. Possibly, we need to set up a *faux* web server in the case of a web application. However, DEECo applications are different, since they are highly distributed with potentially high number of participating computers. We usually can not afford to perform testing in a real-world environment, as it would most often include the challenge of finding a sufficient number of people who are willing to participate in the testing process. Even if this proved to be no problem, the testing would probably take a lot of time, which is sometimes not desirable.

Instead of testing the DEECo applications on a network of real machines, the *MATSim* [3] simulation system can be used. While it originated as a simulation system solely for agent-based systems, it nicely fits the needs of EBCS applications, as well. It offers extensive parametrization of the simulation process, thus making the testing process very flexible. The output of the simulation is produced in the form of XML documents.

1.1. Goals

While certainly XML is a reasonable means to store structured data such as simulation output, it comes with a drawback – it is arguably more suited for being read by a machine than by humans. Generally, it becomes quite challenging to interpret an XML file by sight even if it spans just about a few pages of text.

In this thesis, we aim to provide a software tool that would visualize the data obtained from EBCS applications – such as the DEECo applications mentioned above. In most cases, the input will be obtained from simulation of such systems, but it is not a requirement. Before we state the goals of the work, we need to make clear a few terms.

The graphical output of the application will consist of two logical components – the *infrastructure* and *presentation*. The infrastructure consists of graphical shapes which represent the elements of the EBCS application. These include the environment (for instance a road map), the moving components and the dynamically changing ensembles. The infrastructure thus provides the most important part of the output – the information about how the application execution went. The presentation, unlike infrastructure, does not bring any new information to the table – it is the graphical design of the infrastructure elements. For example, a part of the presentation is a definition of how the DEECo components are visualized (such as a dot/square/arbitrary picture...).

The application is required to support the visualization of EBCS applications' infrastructure, including the environment, components and especially ensembles.

Because the EBCS applications may represent systems of various characters, the output presentation must be highly customizable. This way it is possible to modify the presentation of individual input data in order to visualize it so that it is clear what the original application actually did.

Apart from the support for infrastructure and customizable presentation, the application should meet these additional criteria:

- Support for large input files
- Reasonable loading times
- Clean architecture and extensibility
- Platform independence
- Scripting interface

1.2. Structure of the Thesis

In the following chapter we expand on the topics concerning the background behind this work. We describe in detail the notions of *smart cyber-physical systems (sCPS)*, *ensemble-based component systems (EBCS)*, the platform of *dependable ensemble of emerging components (DEECo)*, and *JDEECo*, the Java implementation of DEECo. We finish with characterization of the applications that can be built on the DEECo platform.

In the third chapter we provide an analysis of the software work which forms the basis of the thesis. We start with a discussion of the goals, introduced in the previous section. For each of the goals, we describe the approach we have taken to solve it. The next part of the chapter contains an overview of the application architecture, especially of how the input data are processed. Finally, the chapter concludes with a performance evaluation, where the application performance is measured. The results are interpreted with regards to the initial goals.

In the fourth chapter we present a few tips on how to use the application in an efficient way. It begins with a presentation of the configuration file feature, which is a way to specify input for the application. Introduction of the plugin extension mechanism follows. Finally, the scripting interface is introduced, including the discussion of how it can be used to solve basic tasks, and illustrated by a number of code examples.

In the fifth chapter, we put the presented work in the context of the current works in the field of smart cyber-physical systems. We compare it with the existing visualizing tools, and we list the benefits and drawbacks of using other visualizing solutions for EBCS applications.

The work concludes with the sixth chapter, which provides an overview of the presented work with regards to the goals which were originally set.

2. Background

In this chapter we are going to discuss the concepts and notions which we work with in the following chapters. These include the *smart cyber-physical systems (sCPS)*, the *ensemble-based component systems (EBCS)* and finally, the *DEECo* software engineering concept.

2.1. Smart Cyber-Physical Systems

A *smart cyber-physical system (sCPS)* [4][5] is a collection of potentially high number of elements situated in a real-world environment. Each element of the sCPS is presumably a mobile gadget, a phone, or a computer which runs a specific piece of software. Typically, the elements are physically located at different places, and their position may change continuously with time, which is in agreement with the assumption that the elements are portable forms of computers, which either move by themselves or are carried by users in their luggage.

The elements are able to extract certain types of information from the surrounding environment, such as their position or the local temperature. To enable the extraction, the elements are equipped with specialized sensors. Additionally, they can also retrieve specific kinds of information from other elements of the system. To make the interaction with the environment complete, the elements can influence the physical environment, as well. For this reason, they are often equipped with a variety of actuating devices.

To characterize an sCPS in basic notions, we list the main properties [5] of such a system:

- *Decentralization* – there is no central element that would, in the case of malfunction, paralyze the whole system.
- *Distributed character* – the individual elements are located on different computers and communicate through network. Behavior of the system as a whole is an emergent result of the behaviors of its elements.
- *Heterogeneous nature* – the elements need not be of the same type. For example, one element might be an embedded computer in a car, while the other is a mobile phone.
- *Interaction with physical environment* – as mentioned above, the system members can interact with the physical environment (as well as with each other).
One additional property is often assumed:
- *Open-endedness* – system elements can connect and disconnect at seemingly random times.

As an example of an sCPS we may mention an intelligent navigation system. Each navigation system gadget corresponds to an element of the sCPS. Usually, the gadgets are located inside cars, but may also be worn by pedestrians or other vehicles. The navigation software in each gadget then monitors the current position

on the map, dynamically refining the optimal route to the destination. On top of that, it also monitors the current traffic, using communication with other navigation system elements. In the end, the optimal route is determined both by the current position and the traffic situation.

In this example we may see how the interaction with the environment works both ways. The behavior of an individual element is affected by the environment when it selects the optimal route based on the traffic and position – both of which are retrieved from the sensors. On the other hand, the element can influence the behavior of other elements by selecting a particular route to go, thus increasing the traffic along the way.

The remaining sCPS characteristics are easy to spot in the car navigation example, as well. Decentralization is apparent, as the cars determine their route locally, not using any central server. The system is distributed, since the software runs on each of the elements, and these elements are equal in role/importance. The individual element may not only be cars – the navigation system may also be present in bicycles, trucks, or in a pedestrian's mobile phone, which is what makes the system heterogeneous. Finally, the open-endedness is ensured by allowing the navigation gadgets to be turned on and off, effectively connecting and disconnecting them from the system.

The computational power of portable gadgets and computers has notably increased in the recent years. Together with the solid coverage by cell-phone signal and improved mobile internet bandwidth, this allowed for building a number of new sCPSs. Although the hardware still poses a few challenges to deal with – such as battery life or weight of the gadgets, the main focus of the sCPS developers is now aimed at the software part of the systems.

Designing a reasonable software for an sCPS may present many challenges. Since the combination of the basic characteristics of sCPS, listed above, does not fit in with any of the usual software-engineering concepts [4][5], a new approach needed to be developed [2].

2.2. Ensemble-Based Component Systems (EBCS)

To match the needs of various sCPS, a new software design concept has been developed – the *Ensemble-Based Component Systems (EBCS)* [2][1][12]. It can be seen as a mixture of the following computational models [2][8]:

- Component-Based Software Engineering [10][13]
- Agent-Oriented Computing [11]
- Ensemble-Oriented Systems [2][15]
- Control System Engineering [16]

EBCS concept has adopted the notion of a component from the *component-based software engineering*, which makes it possible to model the system members as well-

encapsulated and reusable entities. The behavior of a member is based solely on a piece of data, called "belief", that is held inside the member and describes the surrounding environment. Of course, the belief may not be always complete or accurate. As a result, the behavior of the system as a whole is described by the behavior of its individual members; there is no global way to change it. The reader may have noted that this is actually one of the main principles of *agent-based computing*. As for the *ensemble-oriented systems*, the main asset EBCS borrows from them is the communication model. System members can only exchange information in one way, which is done implicitly by the runtime. The communication is attribute-based – attributes are data associated with a system member. The attribute values are exchanged between system members when they are members of an "ensemble" – these are created and dismissed dynamically based on the member localization and their attribute values. Last but not least, EBCS is inspired by *control system engineering* in the way it ensures the *operational normalcy* of components, that is, how it makes sure that the components hold reasonably accurate belief about the environment. It is achieved by employing soft-real-time control feedback loops, with carefully determined periods between individual loops.

2.3. DEECo Component Model

Of course, the EBCS is just a concept. When developing the software part of an sCPS we need a more rigorous approach. To this end, the software-engineering model of *Dependable Ensembles of Emerging Components (DEECo)* [1][2] was developed. It employs the EBCS concept, described in section 2.2. On top of that, it defines actual software-engineering notions and semantics that make it possible for software architects to design a new sCPS application. To make the difference clear, look at the situation from the perspective of a software architect – EBCS only describes an approach to software design, while DEECo should lead to an actual application architecture.

The main notions DEECo works with are that of a *component* and *ensemble*. To provide an environment in which these two notions are implemented, there is also the DEECo *runtime framework*. It carries out those operations on components and ensembles that are considered to run implicitly in the DEECo model. Components are, from the developer's point of view, a self-contained piece of code that can be developed, deployed and run independently of other components. Internally, components consist of two logical parts – *knowledge* and *processes*.

Knowledge is the data that the component holds. It is organized as a collection of key-value pairs, with the option to have a tree node in place of a primitive value. This effectively makes it a tree structure with primitive values in the leaves. The data that knowledge contains define the component's *current state* and the *functionality* that the component provides. The current state of the component is, informally, a collection of values of those keys that somehow characterize the component; these keys are of course determined by the actual application that is built on DEECo model. The functionality that the component provides is specified by the

implemented interfaces. *Interface* is a collection of keys that the implementing component must contain. Interfaces behave in a similar way to those in popular programming languages – a component may feature zero, one or multiple interfaces, and key in a component may be specified by multiple interfaces at once. As will be described later, a component sometimes acts on the outside as an instance of one of the implemented interfaces. This is especially useful in some of the ensemble operations that require the input component to contain certain keys – in other words, it requires the component to implement a certain interface.

A *process* is a procedure that operates on the component knowledge. A list of knowledge fields, along with the information whether they act as input/output/input+output, is associated with each process. Each process is run by the runtime, either periodically in regular time intervals (specified for each process), or when triggered by an event, usually a change in a knowledge key (specified for each process). When it is determined that the process should run, first the input knowledge fields are atomically retrieved from the knowledge, then the actual procedure is run and finally, the results are atomically written to the output knowledge fields. No other external data may be accessed from within the procedure than the specified knowledge keys.

An *ensemble* is a model of connection among components. The motivation for having a kind of binding/connection among components is to allow them to communicate. To keep things simple, the ensembles model has been designated as the only means of communication in DEECo. Each ensemble is essentially a group of components where one of the components acts in a privileged role, named *coordinator*. The other components in the ensemble act as *members*. These roles, however, apply only for the one specific ensemble. As there may be a high number of ensembles in a DEECo system as a whole, each component is allowed to participate in multiple ensembles; in each of the ensembles, the component may act in a different role. As a result, it does not make sense to talk about a component being coordinator, unless a context of a specific ensemble has been specified.

Each ensemble is created dynamically by the runtime framework. To determine the coordinator and members of the ensemble, the *membership function* is called in sequence on every ordered pair of components, determining that they are in the ensemble with the first component acting as the coordinator. This way, after the function has been applied on every ordered component pair, we get the full information about which components are involved in the ensemble.

The *ensemble definition* contains five pieces of information. First, the interface of the coordinator, which is a restriction on which components can potentially act as coordinators in this ensemble. If a component does not implement this interface, the membership function is not even called on the component pairs where the component acts as coordinator. The second part of ensemble definition is the interface of the member, which works similarly to the coordinator interface restriction. Third, the membership function is included. It is a binary predicate and can only access those knowledge fields of the potential coordinator and member which are part of the

interfaces described above. Fourth part of the definition is *knowledge exchange*. It is a procedure which describes the interaction among ensemble components. It receives the coordinator and a collection of the members as parameters and manipulates the knowledge entries in those components. Of course, the exact instructions contained in the procedure depend on the actual application built on DEECo model. As in the case of membership function, the knowledge exchange function can only access those knowledge field specified by the interfaces described above. The last entry in the ensemble definition is *scheduling* of knowledge exchange. It is a specification of when should the knowledge exchange be carried out – it can be either done periodically (i.e. in regular time intervals with the interval specified here accordingly), or when triggered, i.e. when a specified knowledge field has changed its value. Generally, the scheduling is here defined the same way as in the component process scheduling.

2.4. JDEECo: an Implementation of DEECo

In order to successfully transform the application architecture into a working piece of software, we need an implementation of the notions and tools described by the DEECo model. *JDEECo* [7] is precisely what we want – a plain implementation of the concepts described by DEECo, written in Java. It is an interesting project from a software engineer's point of view – the main challenge it deals with is mapping the concepts of the model to the constructs of an object-oriented programming environment.

A *component* is simply represented as a Java class, which extends a specific base class and is marked with appropriate annotations. The knowledge of a component is determined to be the set of class member variables – the runtime framework then uses reflection to access them. If a knowledge field should be a tree node, there is a special wrapper class prepared for this purpose. The interfaces implemented by the component are not explicitly marked; they are determined dynamically using reflection. Finally, the processes of the component are provided as properly annotated static methods in the component class. The input and output knowledge field are given as method parameters, again with appropriate annotations.

An *ensemble* is represented as a Java class, as well. The ensemble class extends a specific base class and is marked with dedicated annotations. In contrast with the DEECo model, the interfaces of coordinators and members are not explicitly marked in the ensemble class; they are determined dynamically by looking at the knowledge fields required in the membership and knowledge exchange functions. The membership predicate is provided as a properly annotated static function. The knowledge fields of the coordinator and member, which the function needs to access, are given as method parameters. The knowledge exchange function is, similarly to the membership predicate, implemented as a properly annotated static method. The knowledge fields of the coordinator and member, which it needs to access, are given as method parameters. Finally, the DEECo ensemble definition contains information

about scheduling. JDEECo delivers special annotations which provide scheduling options, when applied on the ensemble class.

2.5 Delimitation of the Visualizer Application

With the Java implementation of DEECo, we are ready to build a variety of EBCS applications. They will most likely be the software parts of sCPS systems, since this is what the main motivation was behind creating the EBCS and DEECo abstractions. Although the EBCS applications may be of various types, they bear a few common traits, mostly connected with the notions defined by the EBCS concept – the components and ensembles. Recall that the EBCS applications most often use the components to model entities moving through some kind of an environment, while ensembles help to model the communication among the entities.

It is generally a challenging task to debug and test distributed applications, and the EBCS applications are no exception. To perform testing on real machines, it usually requires a task of finding a sufficient number of machines and their users, who are willing to participate in the testing process. Testing of distributed applications which run on mainstream tablet operating systems is still somehow feasible, as it typically only suffices to persuade users to install the application from the central repository. However, when the distributed application runs in embedded gadgets, things get more difficult. For this reason, an adequate simulation environment is usually used instead of testing on real machines.

For EBCS applications, the *Multi-Agent Transport Simulation Toolkit (MATSim)* [3] can be used to simulate a real environment. Among the many upsides of this solution is the good integrability with the EBCS applications. The degree at which the tool is configurable is also decent.

In the current form, the presented application accepts the input data in the format produced by the MATSim framework simulations. As a result, the input data comes in a standard form, the usual MATSim event log format. However, the MATSim event log files do not contain the whole information about the simulated application. The missing part is especially the ensemble log, which must be exported from the EBCS applications separately.

Our application has an architecture, which allows extending it, so that it fits the needs of different EBCS applications. However, we had to choose a type of EBCS applications to which the visualizer will be tailored out of the box. We chose two classes of applications to be supported right away – *moving cars* and *people in a building*.

The *moving cars* applications have their components contained in vehicles, for instance cars. The environment, in which they reside, may be represented as a simple map – a network of nodes, representing points of interest or crossings, and links, which represent streets and connect the nodes. The vehicles move solely through the

provided links. Ensembles may represent any kind of connection among vehicles, as the visualizing application is only interested in the participants of the ensembles, not the interpretation of ensemble membership.

The components of the *people in a building* applications represent individual people, as they move through a flat or building. Alternatively, the components may also represent some kind of robots which move inside a building, for example robotic vacuum-cleaners. The main difference from the *moving cars* applications is that the corridors, through which the components move, do not have to be of a straight line shape. For example, the corridors may have a curved shape. Ensembles may, like in the *moving cars* applications, represent virtually any kind of relationship.

2.6 Goals Revisited

We shall now review the goals of this work. Since we have discussed the essential background, we can take the goals one by one, and explain their meaning. In the subsequent chapter we will discuss how we managed to meet the goals in our application.

- *Support for Infrastructure*

The application has to be able to show the environment structure, along with the EBCS components and ensembles. Since the components are subject to movements, the visual output must be properly animated so that it shows how the individual components moved through the environment as time went by. The ensembles must be visualized in way which enables the user to tell, when a component assumes a certain role in the ensemble (a coordinator, for instance).

- *Customizable Output Presentation*

The infrastructure is completely described in the input XML files, unlike the presentation, which is partially left undefined in the input files. The application must choose a reasonable default presentation, which would make it easy to perform a visualization, as the output would then need no further major refinements. Still, the users must be allowed to change any aspect of the presentation to fit the needs of the individual input data – the range of possible DEECo applications is very large, so the desired way of presenting the output may differ from one application to another. This may be useful, for example, when we need to show just a selection of the participating components.

- *Support for Large Input Files*

When run (or simulated, as well), the DEECo applications produce output steadily throughout the duration of their run. Especially when the process lasts for a long time, the size of the output XML files may rise up to gigabytes. It is thus essential that the visualizing application will work well with the big files. Specifically, it should allow the user to specify a time interval of the EBCS application run, which will then be extracted by the visualizing application from the input files and visualized.

- *Reasonable Loading times*

As the application loads the input files, regardless whether just a part of them or the whole data, it parses the input to an in-memory representation of the data. The visualization then runs on this parsed form of the data. If the input files are large, the user should be able to specify a section of the files to be visualized. As a result, the size of the input which will be visualized is reasonably small. Our goal is to make sure, that the loading and parsing process does not take an excessive amount of time.

- *Clean Architecture and Extensibility*

The DEECo platform provides means to develop a large scale of applications. Apart from the car navigation system example, we may also mention a planning system for groups of robotic vacuum-cleaners – a set of robotic vacuum-cleaners move through the building and cooperate on the cleaning process. As it seems, the environment in which the DEECo application is situated is not always the same. It can be, for instance, a city street network, corridors and rooms inside a building, or a topographical map. The actual gadgets which run the application are also of varying character – for example, it can be cars, robotic vacuum cleaners, or mobile phones. The visualizing application must have a clean and transparent architecture, so that it is possible to extend it easily in order to fit the needs of a new DEECo application, or any EBCS application. For example, a new EBCS application may potentially be situated in a three-dimensional environment. The visualizing application should be designed so that it is clear where and how the extension should be added. In the presented form, the visualizer should be ready to accept the output of the applications which are situated in a road/street map, or a building interior (i.e. corridors and rooms).

- *Platform Independence*

It is essential that the application runs at least on the three major desktop platforms – the Windows operating system, Linux and OS X/macOS. A reasonable choice of the programming language is then Java, which runs on all of the systems. For the graphical output, the JavaFX library will be utilized, because it is a modern GUI platform which moreover became a standard part of the Java Runtime in the last major release.

- *Scripting Interface*

The application will contain means to customize the presentation of the output. The basic options will be available as GUI controls – for example, the color of the background, visibility of ensembles (as a whole, or zooming of the output. For more refined customization a scripting interface must be available, with the ECMAScript language used.

3. Description of the Solution

In this chapter, we are going to present the software work, which forms the basis of the thesis. In the first section, we describe how we have approached the solution of the goals set in the previous chapters. The follow-up section contains a basic overview of the application architecture, especially how the input data are processed. In the final section we measure the application performance and interpret the results.

3.1. Resolution of the Goals

We will now consider the individual goals, set in the previous chapters. For each goal we dedicate one subsection, where we describe, how we have decided to approach the solution of the goal.

3.1.1. Support for Environment Infrastructure

EBCS applications may be situated in various environments, such as a network of streets in a city, corridor system inside of a building or a topographical map of a mountainous terrain. The EBCS elements, which carry out the application logic, may as well be of very diverse character – for example a car, a person or a robot. However, the situation is not as complicated as it seems.

We can observe that in virtually any EBCS application, the basic structure of the environment can be described by means of *nodes* and *links*. A node is a specific point in a coordinate system, representing a point of interest. In most applications a node corresponds to a kind of an intersection, be it of roads, paths or corridors. The links, on the other hand, represent connections between the nodes. A single link connects two nodes in a specified order, that is, it can be thought of as a directed arrow between the two nodes.

Recall the two classes of DEECo applications presented in section 2.5, the *moving cars* and the *people in a building*. We show how the environments used in these applications can be modeled using the nodes and links abstraction. In *moving cars*, the environment is a map, which consists of roads in a country, or streets in a city. In either case, when we have a road network, it can be logically described as a set of points together with defined connections between them – the points are the road intersections, while the connections are the actual roads. Now we can easily represent these points as nodes, and the connections as links. When we use this *nodes and links* description of the map, the only information we lose is the details of the road trajectories – i.e. where the road goes through between its endpoints. We consider this issue to be of presentational character, and discuss its solution in the next section.

In *people in a building* class of applications, the environment is a ground plan of a flat or a building floor. It comprises a system of corridors and rooms, only separated

by thin walls. An important observation is that people can only move through certain paths in the ground plan – it is, for example, impossible to move through walls or some parts of rooms, where there is furniture. Thus it is possible, similarly to the case of *moving cars*, to represent the environment as a set of nodes and links. To do so, we just form a simple network of all paths where it is possible for people to move and then proceed the same way as in the *moving cars* case. Again, the representation of the path trajectories is dealt with in the next section.

With *nodes* and *links*, we now have means to represent the environmental structure of most EBCS applications. The presented visualizing application is in the present form ready for visualizing the output of the DEECo applications which follow this *nodes and links* abstraction in their environment definition.

The main advantage of modeling an EBCS application environment with nodes and links is that it allows an elegant representation in XML. The environment definition XML then just contains a set of `node` elements, each containing the definition of the corresponding node, followed by a set of `link` elements, each containing the definition of the corresponding link. The basic structure of the environment definition XML is charted in Figure 3.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<network name="prague_map">
  <nodes>
    <node id="top_left" x="10.0" y="10.0" />
    <node id="center" x="20.0" y="20.0" />
    <node id="top_right" x="30.0" y="10.0" />
    ...
    <node id="bottom_right" x="30.0" y="30.0" />
  </nodes>
  <links>
    <link id="1" from="top_left" to="center" />
    <link id="1opp" from="center" to="top_left" />
    <link id="2" from="bottom_right" to="center" />
    ...
    <link id="2" from="bottom_right" to="center" />
  </links>
  ...
</network>
```

Figure 3.1 – environment definition XML

The environment definition may also contain additional elements following the links definition, but these only deal with the presentational aspect of the output.

3.1.2. Customizable Environment Presentation

It is very much possible that the environment definition file contains just the infrastructure definition, with the presentational aspect completely left out. This is made possible by the fact that the presentation definition is optional in the input file. Therefore, our application must readily provide a reasonable default presentation for all of the output elements.

From the definition of a node, we can get at least its identification (ID) and its position in a coordinate system. After obtaining the positions of all nodes, the application can determine the boundaries of the provided map – that is, the left-most, top-most, right-most and bottom-most nodes of the map. The actual values of the coordinates are not of much importance, as only their relative positions are an interesting piece of information from the perspective of the visualizing application. The application then shows a view of the relevant part of the coordinate system, using the boundaries obtained from the available nodes. Each of the nodes is shown as a red dot on the corresponding place with regards to the node's coordinates.

The definition of a link always contains at least its identification (ID) and information about the nodes it connects, in addition to the direction in which the link leads. The links only connect the nodes which are provided in the environment definition, otherwise the input is rejected as invalid. Because we already have a visualization of the nodes (the red dots), we can proceed to visualization of the links. An individual link is shown as a straight gray line connecting the relevant nodes. The direction of the link is not explicitly marked in the visual output, as it can be easily spotted by looking at the people moving through the link (visualization of people will be discussed in the following sections). Visualization of an example environment infrastructure is provided in Figure 3.19.

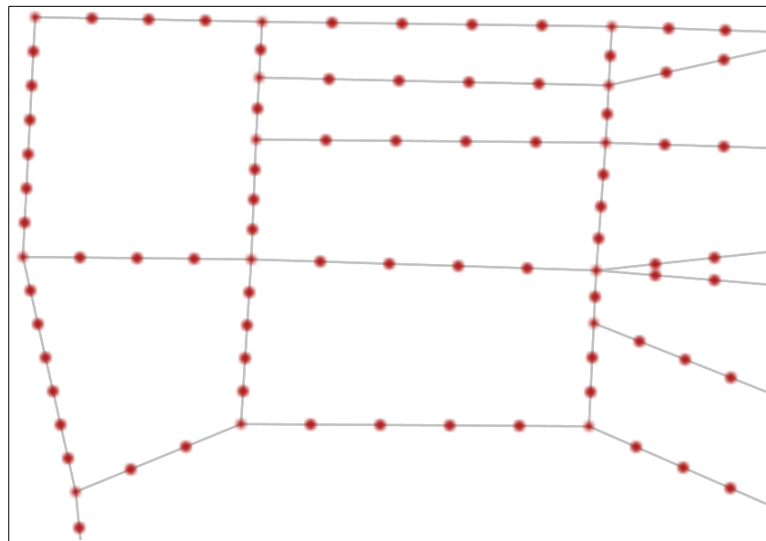


Figure 3.19 – example of the default environment visualization

The presentation of the environment infrastructure does not have to be explicitly specified in the input files. The visualization of the environment then contains just the nodes and links, as described above. However, sometimes the presentation forms an important part of the environment definition – for example, when the links represent roads in a country, it may be of great interest to know the trajectories of individual roads, as they may be curved in otherwise inconceivable ways. In that case, the presentation is usually included in the environment definition file, so that it is applied by default, without further need of fiddling with the visual output.

The environment definition may contain some parts of the presentation; the visualizing application supports the following three main presentation aspects:

- nodes presentation
- links presentation
- static background image definition

The parts of the node presentation, which can be specified in the environment definition file, consist of presentation definition for individual nodes. Not all of the nodes have to be covered – the other will just be visualized the default way, as described above. When a node is covered by the presentation definition, it means that there is a specification of which image should be used to represent the node. Alternatively, the default shape can be used, but with changed color – in that case, the color is provided.

The nodes presentation definition is provided in the input file as a set of `intersection` elements, where each of the elements specifies a certain form of node visualization. An example of such XML input is provided in Figure 3.2.

```
<intersections>
  <intersection id="simple"
               nodes="top_left,center"
               color="green" />
  <intersection id="decor"
               nodes="top_right"
               image="/home/img.png" />
  ...
</intersections>
```

Figure 3.2 – example of node presentation definition

In Figure 3.2, we see that the nodes *top_left* and *center* will be visualized as green dots, since a color has been specified. The node *top_right* will be represented by the specified image in the visual output.

The links presentation definition in the input file consists of a set of `corridor` elements, where each of the elements describes a certain form of link visualization. An example of such XML input is given in Figure 3.3.

```

<corridors>
  <corridor id="cor01" links="4,4opp">
    <link_img source="/home/road.png"
              fromx="836" fromy="1280"
              tox="1" toy="250" />
    <link_path>
      <point x="836" y="653" />
      <point x="817" y="446" />
      <point x="695" y="305" />
      <point x="483" y="251" />
    </link_path>
  </corridor>
  <corridor id="cor02" links="5">
    <link_path>
      <point x="0.2" y="0.4" />
      <point x="0.4" y="0.8" />
      <point x="0.6" y="0.9" />
      <point x="0.8" y="0.98" />
    </link_path>
  </corridor>
</corridors>

```

Figure 3.3 – example of link presentation definition

In Figure 3.3, we see that the links *4* and *4opp* are represented by a specified image in the visual output. In addition to the image specification in `link_img` element, there is also the `link_path` element, which defines a path through the previously specified image. The path will be used by all sCPS elements moving through the link. This way it is possible to provide a highly asymmetric image to represent a link – without the path definition, it would not be clear where the elements should move when going through the link.

The example link presentation also specifies how the link *5* will be visualized. In this case, no image is given. Only a path definition is provided – this allows assigning a polygonal path styled visualization for the link.

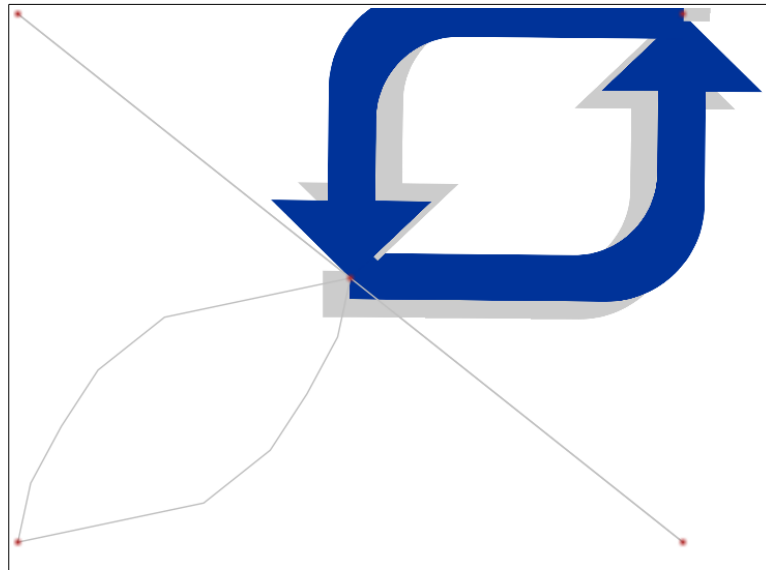


Figure 3.20 – example of a customized environment visualization

In Figure 3.20 we provide a basic demonstration of the visual output customization. The links located in the left-bottom part of the scene are shaped into a polygonal-line shape, while the links located in the top-right part of the scene are represented by a custom image, in this case a blue arrow.

The static background image definition, included in the environment definition file, contains a reference to an image file. The image will be shown in the background of the visual output of the application. This functionality can be used for two reasons:

- purely decorative purpose
- part of the environment appearance

In the former case, the background is used solely for aesthetical reasons and bears no direct connection to the nodes and links. In the latter case, the background image is somehow connected to the environment structure – it may, for instance, contain various decorations of the places where some of the nodes or links are located. In order for the connection with the environment structure to be established, we need to calibrate the position of the background image with respect to the environment structure. Thus the calibration details are included in the background image definition in the input file, as well.

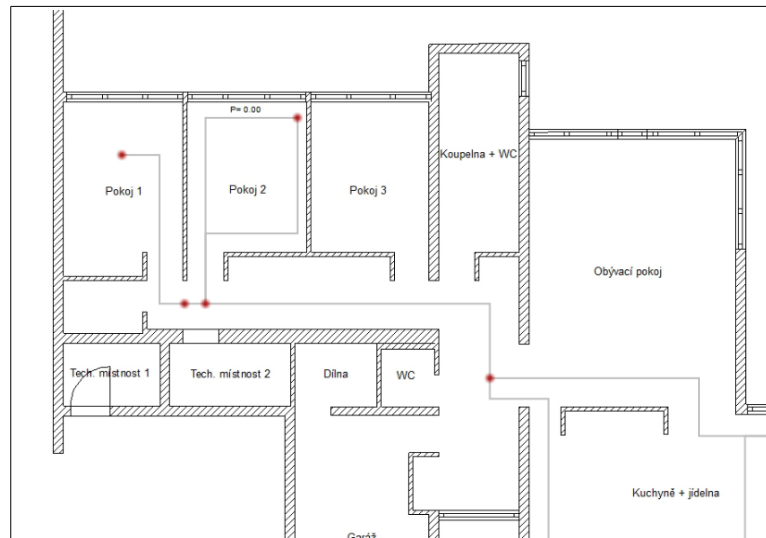


Figure 3.21 – example of customized environment visualization with background being a part of the environment

In Figure 3.21 we see an example of input where the background forms a significant part of the environment. While the links and nodes define where the components can travel, the background provides an intuitive insight into the spatial structure of the environment – we see that the links actually go through a system of corridors, so it is immediately clear why the links bear those strange polygonal shapes.

So far, we have only conceived means to modify the presentation before the actual visualization is started. We did so by including the required modifications in the input files. The application, however, allows changing various presentation aspects even during the visualization process. It can be done two ways – either by using the GUI controls, or by utilizing the scripting interface. The GUI controls responsible for manipulation of the output presentation are located in the main window of the application, right next to the visual output. They only facilitate changing a few basic properties of the presentation, such as the visibility of the nodes as a whole. For more refined modifications, the scripting interface must be used. Description of the scripting functionality is provided in the later sections.

3.1.3. Support for EBCS Components

Each EBCS application is characterized by the environment it lives in, the components and the ensembles of elements. So far we have discussed the support for the first part, the environment. Now we shift our attention to the EBCS components.

As the presented application aims to visualize the provided simulation output, we are only interested in those aspects of components which are visible from the outside – mainly the current position of the element, direction of its movement, and optionally the information whether the component is traveling inside a vehicle. Precisely this kind of information is contained in the *event log file* produced by the MATSim simulation system.

The structure of the event log file is simple – it is an XML file, consisting of a set of event elements. Each of the event elements defines an event of a specific type, which happened during the simulation. Of course, only a selected number of events is stored in the log file – as an example of an event, which is not stored, is the event that an element crossed the 38th x-coordinate in the environment. On the other hand, there is definitely a record in the log file for every event that an element entered or left a link. Since links are directed, we see that it is easy to compute the element's position from these records.

For practical reasons, the event log file is sorted in the ascending order according to the time when the individual events occurred. While it is caused mainly by the way the log file originates, as the MATSim incrementally appends data to the end, it certainly fits the needs of our visualization software. With event elements sorted, it makes it possible to search efficiently in the log file for events with specified time, without the need for loading the whole file in the memory. An example event log file is provided in Figure 3.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<events>
  <event      time="10.0"
              type="PersonEntersVehicle"
              person="V1" vehicle="V1" />
  <event      time="10.0" type="departure"
              person="V1" link="1" />
  <event      time="10.0" type="left link"
              person="V1" link="1" />
  <event      time="10.0" type="entered link"
              person="V1" link="4opp" />
  <event      time="25.0" type="left link"
              person="V1" link="4opp" />
  <event      time="25.0" type="entered link"
              person="V1" link="4" />
  <event      time="40.0" type="left link"
              person="V1" link="4" />
  <event      time="40.0" type="arrival"
              person="V1" link="4" />
  <event      time="40.0"
              type="PersonLeavesVehicle"
              person="V1" vehicle="V1" />
</events>
```

Figure 3.4 – example event log file

In Figure 3.4 we see that the event log contains events regarding the element V1. Since the presented application is currently adapted for the two classes of DEECO applications, mentioned in section Chyba: zdroj odkazu nenalezen, it is reasonable that the elements are called persons in the input log file. At first, the person V1 enters

a vehicle, then departs and travels through a bunch of links before arriving to a destination, and leaves the vehicle.

All of the interesting pieces of information about the elements can be put in the form of events in the log file. As a result, no other input file is needed to contain further information on elements. Unfortunately, a typical event log file will be much larger than just a few lines. How the big input files are handled is discussed in section 3.1.5.

Presentation of elements is simple, compared to the case of the environment. By default, the individual elements are shown as green dots in the visual output. If it is desirable to change this visualization, it can be changed using the scripting interface, to be described in the following sections. The visualization can be changed in two ways. First option is just to change the color of the dot, which is how we can simply differentiate various elements. The second option is to visualize selected elements using custom-provided images. For example, sometimes it may be reasonable to change the element visualization to an image of car, such as when we work with output from a car navigation system. On the other hand, once the image is provided, we can not modify it directly, like we can manipulate the dot color. The only way to manipulate the provided image is to change it to another image.

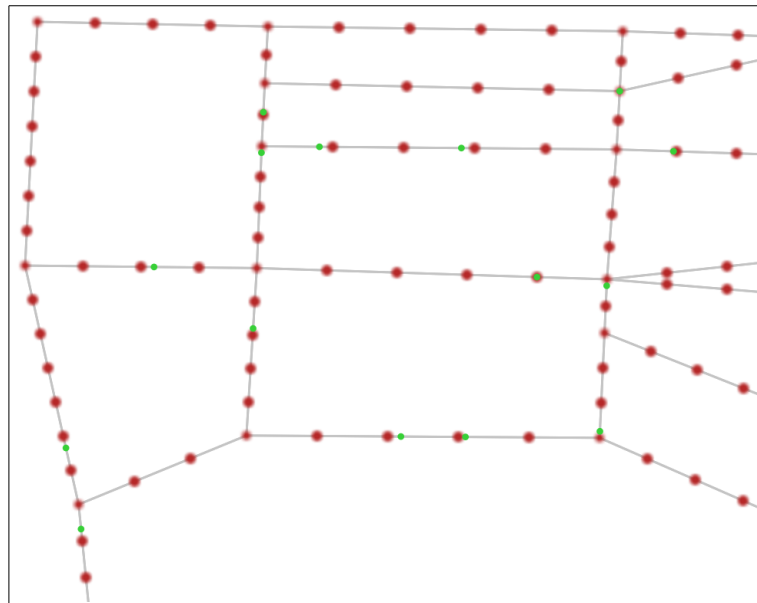


Figure 3.22 – the default component visualization

In Figure 3.22 we see the default presentation of the EBCS components, which is a green dot for each of the components. An example of a customized component presentation is given in Figure 3.23, where the components are represented as small cars.

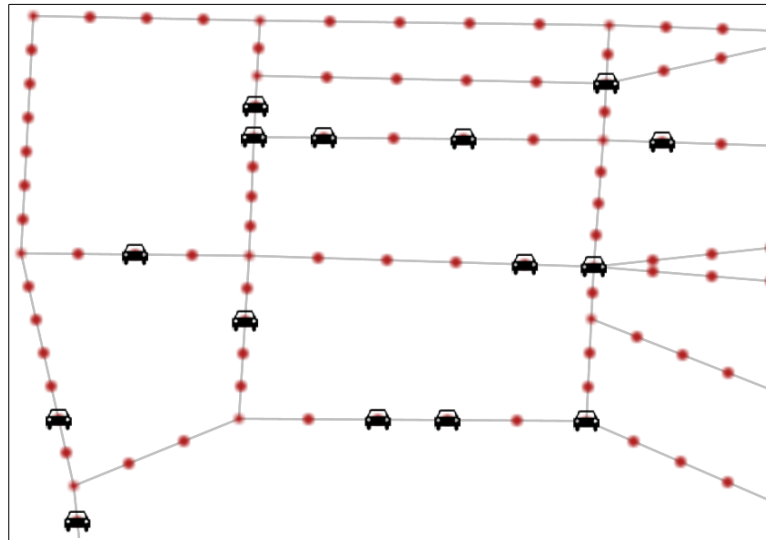


Figure 3.23 – example of a customized component presentation, where components are represented as small cars

3.1.4. Support for Ensembles

The final part of the simulation output is the information about ensembles. From the perspective of the simulated application, the essential aspect of the ensembles is that they provide means of communication among components. However, from the visualization point of view, the knowledge exchange itself is not interesting at all, since it only manipulates the internal knowledge of components and it only changes the component behavior indirectly. We can obtain the log of components' behavior in a direct way from the event log file.

The second part of ensemble definition – the membership function, is much more suitable for visualization. To obtain the complete history of the ensemble membership function values for each of the available ensembles, a separate ensemble event log file is provided as an input to the visualizing application. It is possible to derive the value of each of the ensembles' membership function at any time during the executed simulation, using the log file.

The ensemble event log file is an XML document which contains a set of `event` elements. Each of the elements represents a change of the ensemble membership function at a specified point in time – note that the log file only contains changes in the ensemble membership function values, and not a regular snapshot of their values at a given time interval, as it is sometimes used in other systems.

Similarly to the MATSim event log file, the ensemble log file is sorted in the ascending order. Again, the main reason is that it is written incrementally by the simulated application as it runs. The practical impact is that searching the log file becomes easy.

An example ensemble event log file is provided in Figure 3.5.

```

<?xml version="1.0" encoding="UTF-8"?>
<events>
  <event    type="ensemble"
           coordinator="V20"
           member="V6"
           membership="true"
           ensemble="CapacityExchange1"
           time="25871" />
  <event    type="ensemble"
           coordinator="V2"
           member="V3"
           membership="true"
           ensemble="CapacityExchange2"
           time="25871" />
  <event    type="ensemble"
           coordinator="V20"
           member="V6"
           membership="false"
           ensemble="CapacityExchange1"
           time="25873" />
  <event    type="ensemble"
           coordinator="V2"
           member="V3"
           membership="false"
           ensemble="CapacityExchange2"
           time="25873" />
</events>

```

Figure 3.5 – example ensemble log file

In Figure 3.5 we see that the elements *V20* and *V6* enter the coordinator-member relationship of the ensemble named *CapacityExchange1*, only to split up again after a few seconds. The same can be said of elements *V2* and *V3*, though these two enter and leave a different ensemble, *CapacityExchange2*.

The presentation of ensembles is somewhat tricky. For a given ensemble, we have at most one coordinator, and if there is one, then there is also at least one member element. There are a few ways how to visualize the membership relation:

- Add a "tool-tip" icon next to the corresponding nodes. The tool-tip manifests that the node acts as a coordinator/member in the given ensemble.
- Enclose the ensemble coordinator and members into a graphical shape such as a polygon or an ellipse. This way we see that a node has a role in the ensemble by checking that it lies inside the enclosing shape.
- Connect the coordinator by a line segment with each of the ensemble members.

The first approach is good when we need to check the roles of a single node in various ensembles. On the other hand, the application visual output may get too cluttered as the number of ensembles gets larger.

The second approach is useful in situations when we do not care about the positions of nodes in the coordinate system of the environment. It is then possible to simply relocate the members of the ensemble to the vicinity of the coordinator, and then enclose them all in a circle. The situation may get a little bit disorganized, however, when there is a number of ensembles. Finally, we must note that the assumption about irrelevance of node positioning is very restrictive – in most DEECo application, we believe, the position of elements forms a very important part of the output.

The third approach eliminates the drawbacks of the first two. When there is a number of ensembles, the visual output remains clear, without the excessive number of tool-tips, which would occur in the first approach. Moreover, the nodes do not have to be relocated in any way, which is not the case with the second approach.

We describe in detail how the visualization of ensembles looks like, when the third approach is chosen. For a given ensemble, a random color is assigned. For each ordered pair of nodes, if they are in the coordinator-member relationship of the ensemble, they are connected by a line segment. The line segment has the color associated with the ensemble. The different colors associated with different ensembles make it possible to easily tell the ensembles apart.

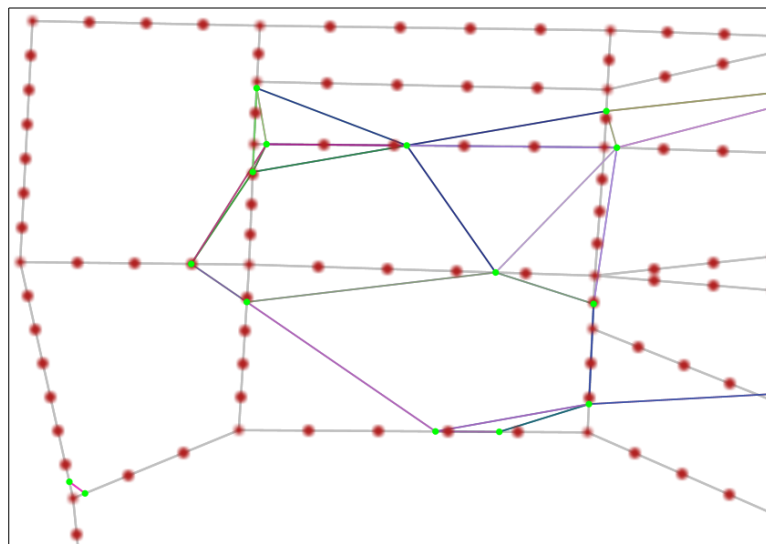


Figure 3.24 – the default presentation of ensembles

In Figure 3.24 we see the default ensembles presentation on an example input. We see that a coordinator of an ensemble is connected with the members by lines of the same color.

As it is the case with the environment and elements, the ensemble presentation is modifiable. The GUI of the application provides basic options for this end – the

option to show/hide all the ensembles, or show/hide just selected ensembles, depending on their name, coordinator, member. In addition, the application provides the scripting interface, to be discussed in the following chapters, which allows modifying the ensemble presentation in a more refined way.

3.1.5. Support for Large Input Files

The DEECo applications may contain a considerable number of components and ensembles. When simulated, it causes the output log files to grow at a steady rate. If, moreover, the simulation is run for a longer period of time, the log files' size can go up to gigabytes. This is obviously a problem when we try to visualize the data; it is no longer possible to just load the whole data into memory and work with it.

To address the challenge of large input files, the application allows specifying a time interval of the original simulation, which will be visualized. The portion of the log files which corresponds to the specified time interval will then be loaded and visualized in the usual way, loading it to the memory and then working with the in-memory copy.

There arises a problem of how to search the log files quickly so that we can find the events within the specified time interval. We are going to need a solution for the following problem P: *find the position in the log file, at which there is the first element with time value equal to the one requested, or the nearest greater value*. If we can find a reasonably efficient algorithm solving the problem, we can just call the algorithm for the enclosing time values of the specified interval and we have the position in the file of the events within the time interval.

It is apparent that the problem described above is very similar to the problem of searching for a value in an ordered array. The ordered array search problem has been studied extensively, see for example [17][18]. Methods have been developed which make it possible to search the ordered array in expected $O(\log \log n)$ time. These algorithms assume that we work with an array, where it is cheap (resource consumption-wise) to access an element at a given index. However, our situation is slightly different – for a given offset in the file, we do not care about the byte which lies there. Instead, we seek the time value of the nearest event element. Even though it is not an extremely performance-expensive procedure, it surely requires some work. As a result, we can not simply disregard the constant factors hidden in the algorithm complexity – as the maximum expected input size is in the order of gigabytes, or units of terabytes at most, the input we search in is considered rather "small" from the theoretical point of view. On such inputs, we believe that the simple binary search algorithm performs at least as well as the asymptotically faster algorithms such as interpolative searching.

We are going to describe how to employ the binary search algorithm to solve the problem P. The basic structure of the algorithm is the same as in the original algorithm. The difference lies in what value we assign to each offset in the file – as

mentioned above, we do not simply read the byte at that position, but we seek the nearest event element and read its time attribute value. We do it in a straightforward manner – at the given file offset, we read a sequence of bytes, in which we then search for the time attribute of the first event element. The process is charted in Figure 3.6.

```
/* Gets the value of the time attribute of the nearest
event element at the specified position in the given log
file. */
function getTime(file logfile, integer offset) {

    /* Size of the excerpt to be loaded */
    integer bufferSize = 4096;

    /* Load an excerpt of the log file into memory,
starting at the position specified by the method
parameter */
    String excerpt =
        system.readChars(logfile, offset, bufferSize);

    /* Get the position of the first occurrence of an
event element in the excerpt. */
    integer eventPosition = excerpt.findFirst("<event");

    /* Delete everything in the excerpt before the first
event element. */
    excerpt.deleteUntil(eventPosition);

    /* Preamble of the time attribute value. */
    String timeAttr = 'time=\'';

    /* Get the position of the time attribute of the
first event element in the excerpt. */
    integer timePosition = excerpt.findFirst(timeAttr);

    /* Delete everything before the first time attribute
value in the first event element. */
    excerpt.deleteUntil(timePosition + timeAttr.length);

    /* Interpret the leading digits as an integer. */
    integer timeValue =
        excerpt.leadingDigits().toInteger();

    return timeValue;
}
```

Figure 3.6 – pseudo-code of the algorithm which gets the time-stamp of the nearest event element at a given position in the input log file

Note the second step of the algorithm from Figure 3.6 - it loads the excerpt of the file into memory. We could avoid it by just reading the bytes from the input file one-by-one until we reach the event element and its time attribute. However, if we read a chunk of the input file at once, we avoid the potential performance overhead of the unnecessary number of read operations.

3.1.6. Reasonable Loading Times

When faced with big input files, the application provides a remedy, which can be dissected into two phases. First, it is possible to define a section of the input which will be its only visualized part. The details of this functionality were presented in the previous section. After a section of the input is delimited, we are left with a trimmed input which can not be further reduced. Therefore, it is essential that the application does not take a long time processing the input, before the visualization is shown. According to Nielsen Norman Group [20], most users are discouraged (i.e. they close the application) when they have to wait for at least 10 seconds during any phase of the application run. Therefore, we aimed to keep the loading times below this limit. We have measured the performance of the presented form of the application; the methodology, along with results and interpretation, is provided in section 3.3.

3.1.7. Platform Independence

The days when Unix was restricted on server machines and virtually every desktop ran Windows are over. Desktop forms of Unix and especially Linux are now more widespread than ever, thanks to the emergence of a number of free distributions. Moreover, as the Mac OS shifted from a custom OS to the Unix standard, we now have quite a handful of active Unix-compliant computers.

The Java platform provides good programming environment for platform independent development. What makes it especially suitable for our purpose is that it offers a library for GUI development, so we do not have to care about different GUI elements implementations on different platforms, as this is taken care of by the Java platform. There are actually two mature GUI libraries included in the Java Development Kit (JDK) – the Swing and JavaFX.

Swing library has been a part of the JDK for years. It offers all the basic building blocks needed to create standard desktop applications. However, not much work has been done on extending the Swing functionality in the recent years. The Java updates which concerned Swing were mainly focused on bug fixing and performance enhancements.

In 2008, the first version of JavaFX platform has been released. It was originally a product separate from Java. A custom scripting language JavaFX Script was used to build JavaFX applications. In the following versions, the JavaFX platform changed drastically – now it is fully integrated into the JDK, with programmatic access to all

features of JavaFX. Many new features were introduced, including support for GPU acceleration. Since version 8 of the JDK, JavaFX became a part of the standard Java library, thus it became a reasonable choice for new GUI applications.

According to the Oracle website [19], Swing will remain a part of the JDK for now, but it will only be updated for security-related issues. No feature updates are expected to happen anymore. On the other hand, JavaFX now takes Swing's place as the preferred means to develop GUI applications, with a good chance that it will be well maintained for the following years.

Since JavaFX seems more perspective for the future than Swing, we have chosen it as the library to be used by the GUI part of our application. The main drawback of the choice is that not all Java installations have the JavaFX library included by default, which is especially valid for older Java versions.

3.1.8. The Scripting Interface

In order to alter the presentation of the visual output, the user may follow one of the following two approaches. The first option is to use the GUI controls. However, this way it is only possible to do simple modifications of the visual output presentation. The second approach is to use the bundled scripting console, which allows much more refined modifications than the dedicated GUI controls. Using scripts, it is possible to alter the basic presentational aspects of virtually every element of the visual output.

As the standard Java library provides a handy scripting API (JSR-223, the Java Scripting API), we have decided to leverage it. What we are going to need is an implementation of a scripting language. There is a large number of scripting languages with JSR-223 compliant implementation on JVM, so we have picked a few of the most well-known and assessed their suitability for our application:

- *Groovy*

First released in 2007, the Groovy language is a scripting language developed primarily for the JVM. Its syntax is fairly similar to that of Java, which makes it easier to learn Groovy for Java developers. Most typical usage is in web development.

- *ECMAScript*

ECMAScript, or its dialect JavaScript, is probably the most notorious of all scripting languages. It is the client-side scripting language of choice for most websites, and as a result, most programmers have at least basic knowledge of the language. Importantly, this is the only scripting language with an implementation bundled with the JDK.

- *Jaskell*

A Java implementation of Haskell, Jaskell is one of the most purely functional languages available on the JVM. As Haskell is often used in functional programming classes at universities, it is quite popular in the academic sphere.

- *Jython*

A Java implementation of Python. First released already in the 1990s, its development partially stalled in the mid 2000s for various reasons. Nowadays it is mainly used for mathematical computations, where it leverages the power of the underlying JVM.

We have chosen ECMAScript as the scripting language for our application. The main reason was the high popularity of the language, meaning that most users of our application are likely to know at least the basics of ECMAScript. Another advantage of choosing ECMAScript is that it is bundled with the JDK, so that the language implementation does not have to be shipped with our application.

The instructions on how to manipulate the individual parts of the visual output using scripts are provided in section 4.3 as a part of the user instructions.

3.2. Basic Architecture of the Application

When the application is run, a usual work-flow is typically followed – the user specifies the input and starts the visualization. While this is a simple task from the user's perspective, it consists of many sub-tasks and operations under the covers. In this section, we try to explain how the input processing is carried out by the application. In order to achieve that, we present the basic architecture of the application.

3.2.1. Parsing the Input

We start at the point when the application has been provided with the input specification, inside the GUI controls. The primal operations are done by the `cz.filipekt.jdcv.SceneImporterHandler` class, which does two things, in sequence:

- Validate the contents of the input fields
- Parse the input files into in-memory representation

The former action is trivial, as it only requires to check, whether the fields contain the correct data type input (i.e. a number when required, or a path). The latter action is more complicated and its execution is delegated to various utility classes.

The parsing process uses the `cz.filipekt.jdcv.xml.XMLExtractor` utility class, which makes it easy to parse XML files. Essentially, it is a wrapper for the Java implementation of SAX (JAXP). It accepts a location of an XML file, together with

an instance of `org.xml.sax.ContentHandler`, which is a standardized accessory to the SAX parsing process, which contains instructions on what to do on various occasions during the file parsing. The `ContentHandler` implementation usually provides a means to obtain the parsed form of the XML file, after SAX parsing is done. Of course, the handler does not have to save all the information from the XML file; it can choose what to put in the parsing output. The output of the parsing process can be in any format. The parameters which the `XMLextractor` utility takes, are summarized in Figure 3.17.

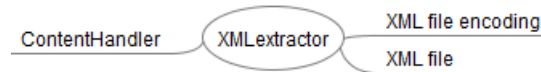


Figure 3.17 – XMLextractor and its input parameters

In our application we have chosen to parse the complete information from the input XML files; if some the data is not required for visualization, it is simply not used. We believe that it prepares ground for potential future enhancements of the application. As for the format of the parsed output, our application follows a principle, that upon parsing, no other processing should be done (there is time for that later). As a result, the output format we have chosen is very simple – most of the elements of the XML files are transformed in a one-to-one way to a single class instance. The parsing process we have just described is charted in Figure 3.16.

```

String path = ensemblePathField.getText()
String encoding = ensembleEncodingField.getText()
EnsembleHandler handler = getHandler()
XMLextractor.run(path, encoding, handler)
List<EnsembleEvent> parsed = handler.getResult()
  
```

Figure 3.16 – how ensemble event log file is parsed

3.2.2. Internal Representation of the Input

We aim to provide a universal software tool, which is able to visualize the output of most EBCS. While in the current state, it is mostly suited to the needs of the DEECO platform applications, it bears an internal structure which makes it relatively easily modifiable to fit any other EBCS output. One of the architectural traits which forms the basis for the universality, is the use of an internal, *generalized representation* of the input data. This way, the application transforms the parsed input data into the generalized representation, and no further operations are performed on the original parsed input. This approach has a significant advantage – it prepares the application for future modifications. For example, not every EBCS produces the output in XML format. Using the generalized representation, it is now sufficient to just add the parsing front-end for the different file format, and no other modifications of the program structure is needed. The situation is charted in Figure 3.18.

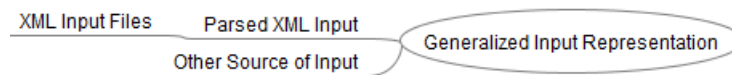


Figure 3.18 – generalized input representation as an entrance point to input processing

We shall now look at the generalized input representation. The environment is represented using the *nodes and links* abstraction, which is the same as in the case of input from DEECo applications. The ensemble events are represented the same way as in the DEECo log file, as well, which is an ordered sequence of individual events. The representation of the individual XML elements is the same as what is obtained from the parsing process.

The MATSim events are, however, represented in a slightly different way than in the parsed DEECo input. The events which are not directly related to the movements of EBCS components are omitted, and the same goes for the information contained in the various XML attributes. The only types of events, which are contained in the generalized representation of input, are those determining that a component entered/left a map link, and that a component entered/left a vehicle. The information on movements through the links is used to determine the component's trajectory. Meanwhile the vehicle information helps the visualizing application to tell, whether the component is on a journey, and therefore eligible for visualization.

Implementation of the generalized input representation is provided by the `CheckPointDatabase` class, which is a database of a number of *checkpoints*. A checkpoint is a definition of where a given output element is at a given time, as was touched upon in the characterization of the generalized input representation above. The database of the checkpoints contains, on top the checkpoint collection, various utility procedures such as searching for checkpoints related to a specified output element.

The conversion process, which transforms the parsed XML input into the generalized representation, is carried out by the `SceneImporterHandler` class, on top the operations described in the previous sections. It is started as soon as the input file parsing is finished; as a result, the application can then dispose the parsed input, which might free up some memory space.

3.2.3. Transition to JavaFX Representation

The generalized representation of the input contains precisely the information we need in order to visualize the input. However, the actual visual output is managed by the JavaFX platform, which uses different data representation. Hence we need to transform our generalized representation of input data into a format which is accepted by JavaFX.

First, we need to visualize the environment defined in the means of nodes and links. We need to create a visual representation for each of these objects. For simplicity, here we suppose that the default presentation is used – if that is not the case, the differences are worked out by parametrization of the default process. The visual representation will be a shape, realized as an instance of a JavaFX class `javafx.scene.Node`. Of course, each of these JavaFX shapes will be located appropriately in the *output pane*, which is a place in the application window, where the visualization is shown. As for the environment, there is no motion in place, so we do not have to care about potential animation.

Next, we need to create a visual representation for each of the EBCS components. Again, for simplicity we will suppose that the default presentation is used. For each of the components, a single shape implementing the `Node` interface is added to the output pane as a visualization of the component. Unlike the environment, we must take care of proper output animation in this case. We use the JavaFX *timeline* functionality for this end. It is an abstraction of the animation of vector graphics. We provide the timeline with a collection of *key-frames*, where each of the key-frames defines the position of selected visual output elements at a specified time. In other words, we tell the timeline where and when should each element of the output be located at given times. The actual graphical animation is then taken care of by the JavaFX runtime. The information needed for the key-frames to represent the input correctly, is almost exactly the information contained in the *generalized input representation*, described in the previous section. Therefore the conversion is pretty straightforward, and we only have to deal with proper transformation of time units and coordinates. The unit conversion is necessary, since the user probably wants the visualization to run for a reasonable time, which can be defined upon visualization start-up. Also the coordinates in the simulated EBCS application will most likely not map well on the computer monitor, so a conversion is necessary.

Finally, the ensembles are taken care of in a similar way to the EBCS components. Using the list of ensemble events, appropriate shapes are created and added to the *output pane*. Each of the shapes represents a value of the ensemble membership predicate. In order to animate to visual output, the *timeline* feature of the JavaFX platform is employed the same way as in the case of EBCS components.

Implementation-wise, the whole conversion from the generalized input representation to the format accepted by JavaFX timeline is contained in the `cz.filipekt.jdcv.MapScene` class. An instance of this class represents a scene, which was operated by an EBCS application – i.e. it consists of the environment, components, ensembles and their movements. We consider the transformation of the data obtained from the simulated application to be scene-dependent, so we have included the above-mentioned conversion in the class.

3.3. Evaluating the Performance

One of the measures by which we can usually assess the success of an application is its performance – if it runs slow, it will discourage users [20] from using it. We have analyzed our application and identified two possible bottlenecks, whose performance measurement makes sense with respect to the total application performance. In the following sections, we first discuss the techniques we use for performance measuring and then in the following two sections we move on to the description of the two parts of our application which we measure. Each of the sections concludes with an interpretation of the measured results.

3.3.1. Employed Techniques

Because our application is implemented in Java, we will have to deal with a few peculiarities regarding the virtual machine environment. First, recall how the just-in-time compilation (JIT) works in Hotspot, the reference implementation of Java which is contained both in OpenJDK and Oracle JDK. Since version 8 of the JDK, the *tiered compilation* mode is the default behavior of the JIT compilation. The application first runs for a short while in an interpreted mode, then gets compiled by the *C1* JIT compiler. The process does not end here; the Hotspot collects various kinds of runtime information about the application, such as which methods are used more frequently than the others, or which branches are frequently chosen in the *if* statements. After a reasonable amount of runtime information is obtained, the *C2* JIT compiler kicks in and creates a new compiled code for the application, this time highly optimized according to the previous experience.

As a consequence, we will have to take into account the phases of compilation, in order to get meaningful testing results. In order to eliminate the effect of compilation on the performance, the usual tactics is to first warm up the virtual machine. During the warm-up process, we repeatedly call the procedure whose performance we are later going to measure. It is important to call precisely the same method which we intend to measure later, because the goal is to achieve the performance behavior of JVM as after running our application for a while. If the number of repetitions of method calls during warm-up is big enough, the methods will get compiled, preferably to the final form (with *C2* compiler). The actual performance measuring will then use the compiled forms of methods, and therefore:

- Measurement results are not skewed by an occasional compilation process
- Measurement results reflect how the application behaves after being run for a while

It is essential that during measuring, the whole application is started up. The reason is that the JVM uses some of the *optimistic type optimizations* for compilation. Specifically, one of such optimizations is the *devirtualization* of method calls - in case there is only one subtype of a class available, the JVM just replaces all virtual method calls on the baseclass with a simple *inlined* implementation from the only

subclass, which results in better speed performance of the call, compared to the usual virtual method call. However, if this happens during measuring and not later when the whole application is run, we will get misleading measurement results. Thus it is important to have the whole application started when doing performance measurements of any part of the application – that way we try to prevent the situation when only a small subset of classes is loaded for measurements and the JVM performs some extra optimizations which would not be performed under normal circumstances when the application is run.

Another peculiarity stemming from the use of Java platform is the memory management. Since the memory is managed automatically, we can not tell with certainty when exactly the garbage collection process will start. Unlike the compilation-related issues, we are unfortunately unable to eliminate the possibility of its occurrence – the garbage collection can not be turned off in Hotspot. On the other hand, we might decrease the probability of such an issue by advising the virtual machine to perform a garbage collection right before the measuring starts. This is done by the standard library method `System.gc()`, which makes sure that by the time it returns, best effort was made to collect garbage. There is, however, no guarantee that the garbage collection actually runs upon calling the method. Then we may expect that during the measuring phase, the garbage collection is performed only when really necessary.

In summary, there are two actions we need to perform on the same virtual machine before we begin with the performance measuring. First, we need to warm the machine up by repeated calls to the methods we are going to measure. Next, we need to perform garbage collection in order to decrease the probability that it runs during the measuring process. Obviously, we can not carry out these actions on a different instance of the JVM. Therefore it is not possible to use the usual performance measuring tools such as `time` utility on Unix, as they have to be executed with the same OS process as the measurements.

The JVM, fortunately, provides various means of time measurement. We decided for the programmatic approach, and we use the `System.nanoTime()` standard library method to measure the elapsed time. We should note that despite its name, the method does not have to actually provide precision of nanosecond, as nanoseconds are only the used units and the elapsed time might actually change less often than every nanosecond. The actual precision is implementation dependent.

From what we have discussed so far, we can chart the algorithm of the performance measurement process, in Figure 3.9.


```

function measure() {
    warmUp()
    System.gc()
    long before = System.nanoTime()
    procedure()
    long after = System.nanoTime()
    return after-before
}

```

Figure 3.9 – pseudocode for the performance measuring

In Figure 3.9 we perform a simple measurement of performance of the (placeholder) method `procedure()`. It does the virtual machine warmup, so unexpected compilations should not happen. However, the defense against garbage collection could be better – now we only try to collect the garbage just before the measurement, and then only hope that it does not happen again. If it does happen during the `procedure()` run, our results can be skewed. To prevent it, we might opt to measure the procedure run multiple times in succession and then determine the mean value of the measured times. If the garbage collection kicks in at some run of the measured procedure, the result is less likely to be skewed. Therefore, our testing procedure then looks more like that of Figure 3.10.

```

function measure(integer repeats) {
    warmUp()
    System.gc()
    long before = System.nanoTime()
    for (i=1 to repeats) {
        procedure()
    }
    long after = System.nanoTime()
    return (after-before)/repeats
}

```

Figure 3.10 – pseudocode for the improved testing procedure

The testing algorithm of Figure 3.10 provides quite a good picture about the performance of the measured method. However, it can be improved – as of now, we have only conceived performance testing on a single JVM and single physical machine. If we can perform the same testing process on different JVMs and physical machines, we may get more telling results, as some of the unanticipated influences can be filtered out. The benefits of using multiple physical machines for testing are apparent – as the machines may have different memory hierarchy structures, the program might behave in completely different ways on the machines; other components of the computers do, of course, influence the application behavior as well. Employment of different JVMs serves a similar purpose. We can not say with certainty which JVM the user will use with our application, so we should make sure that it runs reasonably fast on all the major ones. Vast majority of computers run

either OpenJDK or Oracle JDK. Both of these implementations use Hotspot virtual machine, with only minor differences in marginal parts of the machine. On the other hand, the Java library implementations differ a little bit more – especially in the area of audio/video related classes. For example, the JavaFX library gets different implementations in the JDKs, so it makes sense to carry out tests on both of them, as in one of the tests we will be using some of the JavaFX classes.

In Figure 3.11 we list the physical machines, along with the employed JDK implementations, which we had the opportunity to use for the performance measuring purposes.

Intel® Core i5-5257U @ 2.7GHz, shared L3 cache 3MB	Intel® Core(TM) i5-3210M @ 2.50GHz
RAM 8GB LPDDR3 (1867MHz)	RAM 8GB DDR3 (1333MHz)
SSD SM0128G	Samsung SSD 840 Pro
Os X El Capitan 10.11.5	Ubuntu Linux 15.04
Oracle JDK8 update 92, 64-bit	OpenJDK 1.8.0_91, 64-bit
Configuration A	Configuration B

Figure 3.11 – configurations used for performance measuring

3.3.2. Searching the Big Files

Before we start with the performance measurements, we first discuss the algorithms whose implementation we measure, as they might shed some light on what we actually measure.

In section 3.1.5, we discussed the support for large input files. We reviewed why it is reasonable to expect large input, and how we can deal with it. One of the tools we used when faced with a large log file is the option to select a time interval, where only the events which happened during the interval are visualized. When this selection is applied, only a portion of the input file is left, which we can then easily visualize. The essential task we need to perform, can thus be formulated as follows:

Task T: Find a section of the input file, which contains all the event elements within the specified time interval.

Of course, the input file as a whole fits the description of the section, as above. As a result, we will aim to provide an algorithm which delivers such a section, but without the unreasonable amount of redundant elements. It is, however, unnecessary to require that no additional elements are included – a reasonable amount of them is fine, as we can just filter them out when parsing the XML before producing the visualization.

The approach we have taken is to approximate the result of task P, introduced in section 3.1.5, and use it to construct an approximation of the smallest solution of the task T. We have already charted an algorithm solving the problem P. It utilizes a slight modification of the standard binary search, although it does not work over an array, but over an XML file. Now we are going to describe it more thoroughly in Figure 3.7.

```
function getPrecedingLocation(integer time){
    integer lower = 0
    integer upper = filesize(logfile)
    while ((upper - lower) >= step){
        integer midPoint = (lower + upper) / 2
        integer timeAtMid = getTime(logfile, midPoint)
        if (timeAtMid < time){
            lower = midPoint
        } else {
            upper = midPoint
        }
    }
    return lowerBound
}
```

Figure 3.7 – algorithm which provides an approximation (from the left) of the position of the first event element with the time value as given

In Figure 3.7 we see that the algorithm uses the `getTime()` method, introduced in section 3.1.5. Recall, that it computes the *value of the time attribute of the nearest event element at the specified position in the given log file*. Note that as a result, the moving right bound of the binary search procedure might cross, although only once, the position of the element we are looking for. This would be a problem in the standard binary search, but in our case it is not a problem at all – the method only computes an approximation of the position, and the returned value is really a valid approximation from the left.

To obtain a reasonably small section which solves the task T, we use the algorithm in Figure 3.8.

```
function getSection(integer timeA, integer timeB){
    integer from = getPrecedingLocation(timeA)
    integer to = getPrecedingLocation(timeB+1)
    to = to + step + maxlength
    return read(from, to)
}
```

Figure 3.8 – algorithm which obtains a section which qualifies as a solution for task T

In Figure 3.8 we see that the algorithm just approximates the left and right bounds of the desired interval. To see it, we explain the meaning of the two used variables.

`step` is the same as in `getPrecedingLocation()`, and means the maximal deviation of the approximation of the precise position. `maxlength` is the maximal possible length (in characters) of an event element.

The `getPrecedingLocation()` procedure provides a left-approximation of the desired element position, with the maximum possible deviation being `step`. We can use its result as an approximation of the left bound of the interval. The right bound of the interval must, however, be approximated from the right side. Thus we must find a way to transform the outcome of `getPrecedingLocation()` into a right-approximation. Note that if we shift by `step` bytes to the right, we get at least as far as the right bound from inside the `getPrecedingLocation()` method, when it terminated. In the corner case that this internal right bound is located inside the element preceding the precise position, we must additionally move by the maximal element size to get to the following element.

In the actual implementation of the visualizer, the algorithms must deal with additional practical issues. One of them is the character encoding of the log file – unless a fixed-width encoding is used, each time we access the file at a new position, we must first determine the correct alignment of characters in the byte stream. For example, when the binary search procedure accesses the file at a position, the position may be inside a 4-byte UTF-8 representation of a single Hangul alphabet character.

It is apparent that from the performance measuring standpoint, the most interesting part of the `getSection()` algorithm are the two calls of the `getPrecedingLocation()` algorithm. The third line only consists of simple integer arithmetic, and the last line runs for a time proportional to the length of the selection. Therefore, in the following we will be measuring the performance of the `getPrecedingLocation()` implementation.

Now that we have given some insight on the algorithms which deal with big input files, we can move on to the performance measurement. The part of our application, which manages access to the big input files, is encapsulated in a single class (`cz.filipekt.jdcv.util.BigFilesSearch`). Therefore we can simply access the class's interface and test it, without any need for further integration of our testing procedure within the application structure.

We have prepared a performance-measuring utility class, which can be used to measure the performance of the big-files search algorithm. One of the advantages of this utility is that it is well-encapsulated – there is no need for any other external sources of parameters, input files or anything else. When the utility is called, it generates a MATSim event log file and performs various measurements on it. After the measuring is over, the file gets deleted immediately. The utility then proceeds to create another MATSim event log file, repeating the procedure for a few more input files. The generated input files are not all the same; they follow certain patterns which we present in Figure 3.12.

File Size	6 MB	20 MB	60 MB	200 MB	600 MB	2 GB
# of elements	100 k	100 k	1 M	1 M	10 M	10 M
Element Size	30	200	30	200	30	200
Text Encoding	UTF-16	UTF-8	UTF-16	UTF-8	UTF-16	UTF-8
	File A	File B	File C	File D	File E	File F

Figure 3.12 – files used for big-files search performance testing

In Figure 3.12 we see that the generated input files vary in several aspects. First, they differ in the number of contained event elements. That is natural, as the real input files will usually have a previously undetermined number of the elements. The real input files will also most often contain elements which have highly variable length, in characters. While some events contain just a plain identification of its cause and a timestamp, others may contain a variety of attributes with detailed information about the event circumstances. Thus our generated files have different length of the contained elements. Finally, the files also differ in the used text encoding. While not obvious at first sight, the encoding may affect the search times. The reason is because the application needs to perform character alignment whenever it accessed a random position in the input files. When simple US-ASCII encoding is used, the character aligning does not have to take place, while in multibyte, especially variable-length encodings it is definitely a necessary process.

As we described in the previous section, the performance measurements were executed for each of the input files on different machine configurations. In the following Figure 3.13, we sum up the results we got.

Config. A	48 μ s	26 μ s	74 μ s	42 μ s	111 μ s	58 μ s
Config. B	45 μ s	20 μ s	59 μ s	29 μ s	97 μ s	51 μ s
	File A	File B	File C	File D	File E	File F

Figure 3.13 – results of the big-files search performance measurements

In Figure 3.13 we see the results of the measurements of the big-files search functionality. Each row contains the results obtained by running the tests on the given configuration, introduced in section 3.3.1. The value assigned to a file and a configuration, is a mean value of a larger number of test results. The file has been searched for a fixed number of event times, and for each of these value, the search has been executed and measured repeatedly for a number of times. Of all these search results, the mean value is included in Figure 3.13.

We see that in general, the files encoded in UTF-16 often take a little bit longer to search than those encoded using UTF-8. We believe that it is caused by the fact that in UTF-16 encoded files the application does the character alignment procedure, while in UTF-8 it does not, since the generated files only contain ASCII characters. It is well-known that UTF-8 and US-ASCII are compatible in the first 127 characters, thus in the generated files, the UTF-8 representation is exclusively single-byte.

Another trait can be seen if we restrict ourselves to the input files which differ only by the file size (that is, A-C-E and B-D-F). In these situations, the search times approximately increase by an additive factor when the file size grows multiplicatively. This is the expected behavior for a form of binary search, so there is no surprise.

In conclusion, the process of searching big input files will not most likely be a bottleneck in the application performance, as the measured times are in the range of hundreds of microseconds, and the elapsed times only increase logarithmically with the size of the input.

3.3.3. Input Processing

The following is the typical work-flow of the application:

- (1) Application starts
- (2) Input files are specified
- (3) Visualization parameters are specified
- (4) Application processes the input
- (5) Visualization starts

During the application start-up, no complex tasks are performed by our application. The only work which has to be done is the loading of the Java platform and JavaFX libraries, which in most practical scenarios takes about 1-2 seconds. Chiefly because the loading time is dependent only on the Java implementation and computer hardware, and it is independent of our application qualities, we decided not to perform performance measurements of this process.

Obviously, the steps (2) and (3) last only until the user provides the required input, which is highly dependent on how well the user handles the computer. Once the step (4) is finished and the visualization is prepared to start, there are no additional situations in step (5) when the user would have to wait for anything. The only performance issue which could arise during the step (5) is a poor fluency of the visual output animation. The implementation of the visual output animation is provided by the JavaFX platform, so the hardware requirements for the animation to run smoothly are dependent on the quality of the JavaFX implementation. Although we could not achieve a jerky animation on any of the computer configurations, which we used for testing, we must advise the users of such a possibility, especially on the oldest versions of JavaFX. However, in version 8 the JavaFX implementation greatly

improved its performance, and it now adapts the computational complexity of the video output to the performance of the computer.

The step (4) seems to be the work-flow phase which could potentially be the bottleneck. For clarity, we divide it into two parts:

- (4a) Searching for the selected section of the input file
- (4b) Loading and processing the selection

In section 3.3.2, we measured the process in (4a). Now we are going to measure the performance of the process in part (4b).

When we measured the performance of step (4a), the whole file-searching functionality was embedded in a single class `BigFilesSearch`. Therefore the ground was prepared for the testing, as we just had to call a method on the class and measure how long it takes to run. In (4b), the situation is slightly different. The input processing is a task carried out by a variety of classes, where each part of the process is delegated to specialized classes and subroutines.

We have done two slight modifications of the visualizing application in order to support the performance testing process. First, we have added time-logging at the points where the input processing is just being started and where it just ends. The logged times are then made available to our performance testing utility class. Second, we allowed the application to accept the input specification in a programmatic way. It is done using the JavaFX constructs which simulate the actions of a user – as a result, from our application's view, it can not be determined, whether the input was provided by hand or programmatically.

The performance testing logic is encapsulated in a single class `cz.filipekt.jdcv.measuring.MeasureInputProcessing`, containing a `main` method. When the utility class is started, it generates a few sets of input files. After the input is prepared, the performance measuring is started. Finally, when the testing is over, the previously generated input files are deleted again. The generated input sets are not all the same; they differ mainly in the total number of MATSim event elements and ensemble event elements. An overview of the properties of the generated input is given in Figure 3.14.

In addition to the generated input, we provide one additional input set, which is packaged with the application and is not dynamically generated upon performance testing. It contains the output of a simulated system of cars, as they move through the map of a South Dakotan city, Sioux Falls.

# of DEECo elements	4 k	3 k	30 k	100	100
# of MATSim events	377 k	39 k	390 k	1300	1300
# of ensemble events	1.5 k	0	0	30 k	300 k
MATSim event log size	32.7 MB	2.51 MB	25.9 MB	78.2 kB	78.2 kB
Ensemble event log size	198 kB	0 B	0 B	3.27 MB	33.0 MB
	Input #1	Input #2	Input #3	Input #4	Input #5

Figure 3.14 – the input used for performance testing of input processing

Input #1 is the described cars simulation output from Dakota. Inputs #2 and #3 are dynamically generated upon start of the performance measuring. They bear the same structure – the map is a square, with nodes being in the corners and links forming a circle (in the graph theory sense). The DEECo elements emerge one by one from the top-left corner and circle the map once. After returning to the initial node, they stop. No ensembles are formed. The inputs #2 and #3 differ only in the number of participating DEECo elements.

On the other hand, inputs #4 and #5 stress the ensemble visualization functionality. Like the #2 and #3, they are both dynamically generated upon start of the performance measuring. They bear the same structure – the map is a square, with nodes being in the corners and links forming a circle (in the graph theory sense). The DEECo components emerge from the top-left corner in close succession, and circle the map in a swarm. Upon returning to the initial node, they stop. Because these input files stress the ensemble functionality, the number of DEECo elements is kept rather low and the ensemble definition is enhanced in return. The ensembles definition contains a number of different ensembles, each of which is formed only once and dismissed a shortly after. All of the ensembles have one coordinator and three members.

Now that we have described the input used for performance testing, we can proceed to the results of the measurements.

Config. A	935 ms	137 ms	2233 ms	180 ms	3424 ms
Config. B	964 ms	137 ms	2678 ms	210 ms	4473 ms
	Input #1	Input #2	Input #3	Input #4	Input #5

Figure 3.15 – results of the performance measurement of input processing

In Figure 3.15 we see that we succeeded in keeping the total processing time of the input below 10 seconds, which is crucial in order to keep the user's attention, and to

possibly prevent him/her from shutting down the application for unresponsiveness [20]. On the more modern machines, we actually managed to keep the times under 4 seconds, making the user experience more pleasant.

If we look at the results obtained on inputs #2 and #3, we can deduce how the input processing time is dependent on the size of the MATSim event log. Input #3 is exactly 10 times bigger than #2, and bears the same structure. We see that the time complexity of the processing is slightly super-linear, but still remains reasonable.

Looking at the results from inputs #4 and #5, we can deduce how the input processing time is dependent on the size of the ensemble event log. Input #5 is exactly 10 times bigger than #4, and bears the same structure. The situation is similar to that of the two previous inputs – we get a slightly super-linear dependence, within reasonable bounds.

4. Using and Customizing the Application

In this chapter we are going to advise on how to use the application in an efficient way. We begin with a discussion of how the application can be provided with input data, using the configuration file feature. Next, we describe how the application can be extended by writing custom plugins. Finally, we present the scripting interface and how it can be used to solve basic tasks, illustrated by a number of practical examples.

4.1. Supplying the Input

There are basically two ways how to provide the application with an input. First, the user can fill in all of the GUI controls which deal with input specification. This includes filling in up to 7 individual text fields, which can sometimes prove tiresome. The application window with the input specification page is depicted in Figure 4.1.

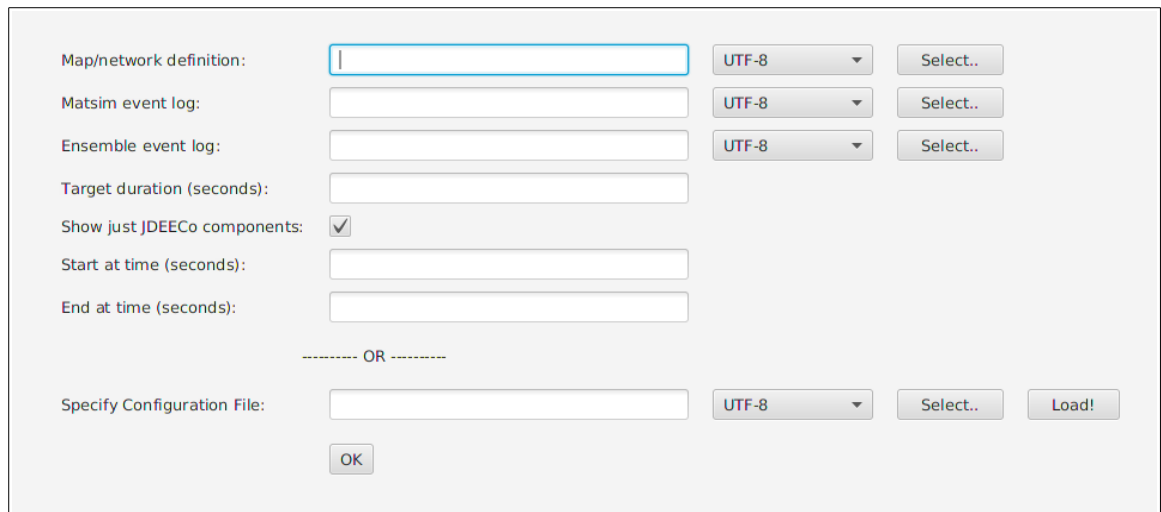


Figure 4.1 – the input specification GUI controls

In Figure 4.1 we see that the last of the GUI controls facilitates specifying a *configuration file*, which forms the second way of providing input. A configuration file is a text file in a specific format, which contains the complete specification of an input. That is, the paths to the input files, duration of the visualization, section of the input files to be visualized and others.

The main advantage of the configuration files is that they make it possible to automatize the visualization process. Once we have put the input specification into the text file form, we can run the visualization much easier than before – it is only necessary to provide the configuration file and the application does not need anything more.

In order to create a configuration file, we need to know the structure of its file format. An example configuration file is given in Figure 4.2.

```
network;example_data/network.xml;UTF-8
events;example_data/events.xml;US-ASCII
ensembles;example_data/ensembles.xml;US-ASCII
target_duration;56
just_agents>true
start_at;25000
```

Figure 4.2 – example configuration file

Inspecting Figure 4.2, we see that the configuration file consists of a set of lines, each of which contains specification of a part of the input. For every GUI control, which appears in the application window for input specification (see Figure 4.1), there is a single line in the configuration file. Of course, not all of the lines must always be present – the application can deduce reasonable default values for most parts of the input.

Notice the format of the individual lines in the configuration file. Each of them can be logically dissected into a few parts, using semicolon as a delimiter. The first of these parts is used to identify the part of the input which is specified on the given line. The second part contains the actual input specification, while the third part is sometimes used to define the text encoding of the text file referenced in the previous part.

4.2. Extending the Application with Plugins

One of the goals of the thesis was to create an application with a reasonable architecture, so that it is possible to extend it easily. The main part of the goal – the appropriate architecture, was already discussed in section 3.2. However, we have approached the extensibility goal in one additional way.

The reasons why users decide to extend our application can be most likely divided into two categories:

- The application does not accept input in a specific format
- The application does not show a certain kind of information in the visual output

We believe that the second class of problems will occur much more often than the first. The reason is that the changes in input data format are usually connected with a change in an EBCS log format, while the lack of a specific kind of information in our application output is connected with demands of individual people. As there may potentially be a number of people who use our application, the probability that some of them will find the visual output lacking, is non-negligible.

Therefore, we have equipped the application with a plugin API. Using this API, it is possible to write a standalone plugin, which can then be added to the application. What makes the plugin system powerful, is that from inside the plugin, it is possible

to access and modify any part of the visual output, such as the environment, components and ensembles, along with their visual representations. With this capability available to the plugin developer, it facilitates creating a wide selection of plugins, including various filters, highlighters or graphical mods, which completely change the visual appearance of the graphical output.

The programming details of how to create a plugin can be found in the source code documentation, included on the attached optical medium. In summary, it is necessary to extend a specific class, the `PluginWithPreferences`. Inside the plugin application logic, the presentation of the visual output elements can be modified by retrieving their preferences objects by a dedicated method, and then by calling the various methods on these preferences objects.

In order to make the plugin visible for our application, it just needs to be packaged in a JAR archive, which is marked as a service-provider in the sense of the Java service loader/provider functionality, and placed on the Java classpath. The provided service is the `cz.filipekt.jdcv.plugins.Plugin` interface.

The application comes equipped with two plugins by default. We are going to present their intended usage in the following sections.

4.2.1. Information Plugin

The *information plugin* shows detailed information about various visual output elements (links, components, etc.). Because the visual output usually contains more than just a few components and ensembles, and the environment may also get quite complicated, it is not practical for the plugin to show the details of every output element at once. Instead, the plugin only shows the details of a single output element. The user just has to click on the desired element, and the related information shows up in the plugin panel. In Figure 4.3, we see an example of how the contents of the information plugin may look, when the user clicks on a component and a link.

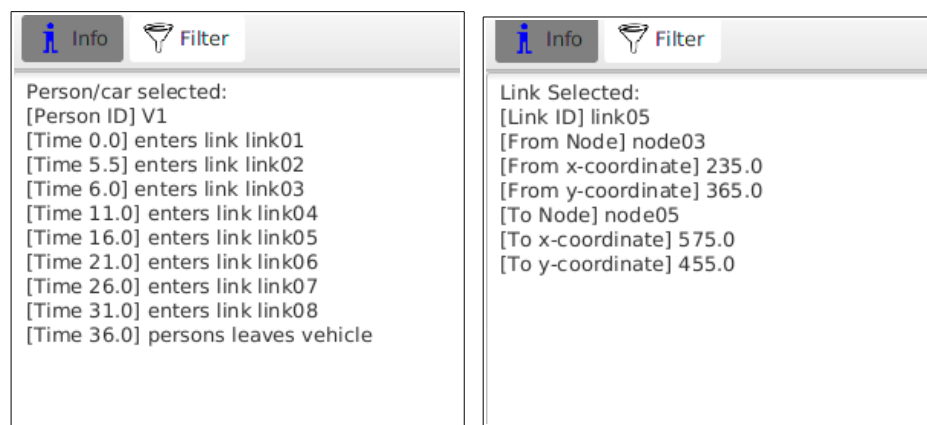


Figure 4.3 – example content of the information plugin

4.2.2. Filter Plugin

The *filter plugin* adds basic filtering functionality to the application. Whenever the use of the scripting environment is considered as a too sophisticated approach for the task, the filter plugin provides a simpler alternative.

The plugin supports filtering by an attribute value of an output element. If one such attribute value is provided, then out of the output elements of the given type, only those with the matching attribute value will be shown. For example, we can filter by an ensemble name. After the filter is applied, the ensemble membership predicate will only be visualized for the given ensemble.

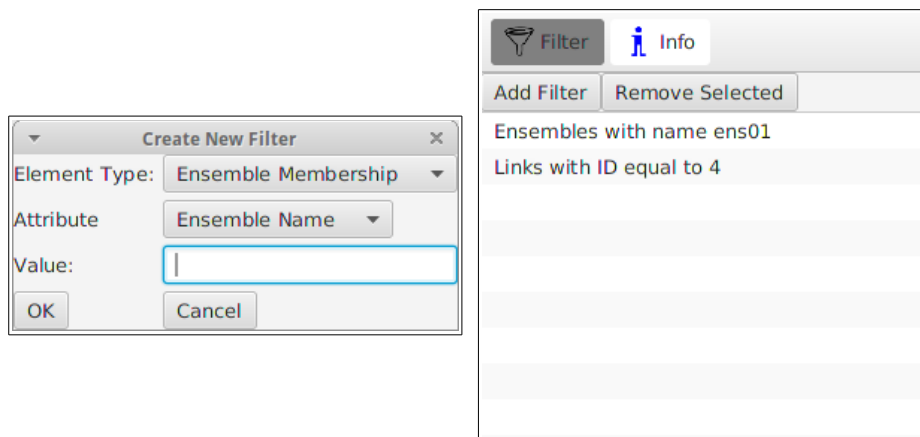


Figure 4.4 – the filter plugin in action

In the left part of Figure 4.4 we see the plugin window where a new filter is being created. It offers the possible output element types, along with the corresponding attributes by which we can filter. In the right part of Figure 4.4 the main window of the filter plugin is shown. It contains a list of filters which are currently active.

The plugin implementation uses the preferences objects, provided to the plugin code by the application. For every output element, the corresponding preferences object contains an option to make the element visualization visible/invisible, which is how the plugin manages to “filter out” some parts of the visual output.

4.3. Scripting Interface

Sometimes the options which the application GUI provides are not enough. For example, when we need a sophisticated filtering of components in the output, the *filter plugin* functionality may not fit our needs. Another case when the GUI controls can not do what we need is if we want to change the visual representation of selected components. For these and other similar reasons, the application is equipped with a scripting interface which offers a much more refined form of control over the visual output.

Recall section 3.1.8, where we reasoned about our choice of ECMAScript as the language of the scripting environment. One of the advantages of selecting ECMAScript over other languages was that it is bundled with the JDK by default. Unfortunately, the ECMAScript implementation bundled with OpenJDK is different in versions 7 and 8. Version 7 contains the Rhino engine, while in version 8 there is the newer Nashorn engine. Both engines implement the standardized part of ECMAScript. However, the non-standard language extensions they offer are slightly different. In this section we are going to present a few ECMAScript code examples, which will be written for the Nashorn engine. In order to run them on a JVM with only Rhino available, they must be modified so that they do not contain Nashorn-specific constructs.

Before we move on to the examples of how the visual output can be modified using the scripting interface, we explain the general approach the scripts take in order to have any influence at all on the visual output. The application exports a few special objects to the environment where the scripts are executed. These objects are essentially collections of *preferences objects* for the individual visual output elements. A preferences object for an output element is an object with properties, where each of the properties corresponds to a single graphical characteristic of the visual output element. Since Java does not support properties natively, we follow the JavaBeans properties naming convention, and access them using the usual getters and setters. An example of the objects which are exported to the scripting environment is `links`, which contains preferences object for each of the links – `links.get("123")` provides a preferences object for the link with id equal to "123".

In the following sub-sections we are going to present a few examples of how various output elements can be modified using the scripting interface.

4.3.1. Customizing the Environment

In order to manipulate the environment *nodes*, the application exports the `nodes` object to the scripting environment. It implements the `java.util.Map` interface, mapping the individual node IDs to their respective preferences objects. Each node preferences object contains the following properties: `id`, `x`, `y`, `circleColor`, `visible`. In order, they represent the node ID, the node's x-coordinate and y-coordinate, the color of the its default visualization, and finally the visibility of the node.

We shall now look at a few examples. We start with a basic demonstration of node customization functionality. We take the node with id equal to "123", and we look at the color of its visualization. If it is yellow, we change it to brown, as the yellow color may not be well visible at some monitors. If it has another color, we leave it unchanged. The algorithm transferred to the corresponding ECMAScript code is in Figure 4.5.

```

var prefs = nodes.get("123")
var color = prefs.getCircleColor()
if (color.equals(Color.YELLOW)) {
    prefs.setCircleColor(Color.BROWN)
}

```

Figure 4.5 – setting the node color

We see that there is available a utility class `Color`, which makes it possible to quickly select a color of one's choice. For more information about the `Color` class, consult the JavaFX documentation [9].

Now we are going to print some details about the node to the scripting console output. We just employ the Nashorn-specific `print()` function, and call the appropriate getters. The algorithm is charted in Figure 4.6.

```

var prefs = nodes.get("123")
print(prefs.getId())
print(prefs.getX())
print(prefs.getY())

```

Figure 4.6 – printing info about a node

In the next example, we hide all the nodes with the x-coordinate greater than 100. To traverse all the preferences objects, we use the `values()` method offered by the Java Map interface. In addition, we also employ the Nashorn-specific for-each cycle for traversing Java collections. The full script is charted in Figure 4.7.

```

var allPrefs = nodes.values()
for each (var pref in allPrefs) {
    if (pref.getX() > 100) {
        pref.setVisible(false)
    }
}

```

Figure 4.7 – hiding selected nodes

In a similar way we treated the environment nodes, we can customize the environment links. In order to manipulate the links, the application exports the `links` object to the scripting environment. It implements the `java.util.Map` interface, mapping the individual link IDs to their respective preferences objects. Each link preferences object contains the following properties: `id`, `from`, `to`, `color`, `width`, `visible`. In order, they represent the link ID, the nodes where the link starts and ends, color (of the default visualization), width (of the default visualization) and finally its visibility.

We are ready to move on to the examples. First, we show how to hide a link visualization in case it is red. The code provided in Figure 4.8.

```
var pref = links.get("123")
var color = pref.getColor()
if (color.equals(Color.RED)) {
    pref.setVisible(false)
}
```

Figure 4.8 – hiding a red link

Next we show how to double the width of every link visualization. The corresponding code is in Figure 4.9.

```
var allPrefs = links.values()
for each (var pref in allPrefs) {
    var width = pref.getWidth()
    pref.setWidth(2 * width)
}
```

Figure 4.9 – widening the links

4.3.2. Customizing the Components

As for the EBCS components, the main information they bear is the way how they move. Apart from that, their presentation is quite simple – the only aspect we can manipulate is the image which represents them. By default, each component is represented by a simple green circle.

In order to change the visualization of a component from the default one to a custom image, we once again use a specific object imported to the scripting environment. It is the `components` object, which unlike the `nodes` and `links` objects is not a Java map. Instead, it offers a single `varargs` method, which is in charge of the customizing the component visualization.

We will now have a look on how we can change the visualization of a component with id equal to “comp” to a custom image. An example code is given in Figure 4.10.

```
components.setComponentImage(
    "/home/img.png", "comp")
```

Figure 4.10 – setting a custom visualization for a component

To revert the visualization to the default one, just provide a `null` value instead of an image path. If we want to apply the change of visualization to multiple components, we may add them as additional parameters – the signature of the method uses variadic arguments, as seen in Figure 4.11.


```

void setComponentImage(
    String imagePath,
    String... selectedComponents
)

```

Figure 4.11 – signature of the component visualization changing method

In order to apply the visualization change to all components, we can either provide all the individual component IDs as parameters to the `setComponentImage()` method, or equivalently, we may omit the component IDs at all, providing just the image path specification. An example of such action is given in Figure 4.12.

```

components.setComponentImage("/home/img.png")
components.setComponentImage(null, "comp")

```

Figure 4.12 – sets a custom visualization of all components, with the exception of “comp”, which is left with the default one

4.3.3. Customizing the Ensembles

The ensembles are by default visualized by a set of straight lines, connecting the components acting as a *coordinator* with the components acting in the *member* role. In order to allow modifications of the default visualization, the application exports the `ensembles` object to the scripting environment. It is a collection of preferences objects, each of which is associated with one of the straight lines providing the ensemble visualization.

The `ensembles` object contains the following properties – `color`, `visible`, `coordinator`, `member`, `ensembleName`. In order, they represent the color of the straight line, visibility of the line, the identification of components which the line connects and finally the ensemble name. Of course, the last three of the properties are read-only.

In the first example usage, we paint with red color all the ensemble lines with the coordinator “comp” and ensemble name “ens”. The corresponding code is given in Figure 4.13.

```

for each (var pref in ensembles){
    if ((pref.getCoordinator().equals("comp")) &&
        (pref.getEnsembleName().equals("ens"))) {
        pref.setColor(Color.RED)
    }
}

```

Figure 4.13 – painting red a specific ensemble with given coordinator

Next, we can hide all other ensemble lines than those painted red in the previous example. The corresponding code is given in Figure 4.14.

```

for each (var pref in ensembles){
    if ((pref.getCoordinator().equals("comp")) &&
        (pref.getEnsembleName().equals("ens"))) {
        pref.setVisible(true)
    } else {
        pref.setVisible(false)
    }
}

```

Figure 4.14 – hiding all ensemble lines not meeting specified criteria

In the final example, we want to print some information about each of the lines which provide ensemble visualization. The corresponding code is given in Figure 4.15.

```

var count = 0
for each (var pref in ensembles){
    print("Item " + count + ": ")
    print("Ensemble=" + pref.getEnsembleName() + ", ")
    print("Coord.=" + pref.getCoordinator() + ", ")
    print("Member=" + pref.getMember())
    print()
    count += 1
}

```

Figure 4.15 – printing info about the ensemble visualization lines

5. Related Works

The EBCS concept is a combination of multiple standard software-engineering models and concepts, with one of them being *component-based software engineering (CBSE)*. As CBSE is not a new notion and has been studied for years, there are a few visualizing solutions available for output from such systems. However, they all share a common drawback – none of them has any kind of support for ensembles. Therefore, they can only be used to visualize the environment and components, with the ensembles left out. We are not aware of any other visualizer, other than our application, which would be able to visualize the ensemble-related data.

One of the CBSE visualizers is *OTFVis* [21], which is visualizing tool for output obtained from MATSim-based simulations of CBSE systems. Its main advantage is probably the support of hardware graphics acceleration, which enables the application to handle big input data comfortably. Among its drawbacks we can mention a poor user interface and a stalled state of development in the last years – the latest release dates back to 2012.

Probably the most advanced visualization solution for CBSE systems is *Senozon Via* [22], a commercial product for visualization of MATSim output, among others. It offers numerous tools and customization options, including support for visualization layering. However, as is the case with *OTFVis*, there is no support for ensembles. Another drawback is its licensing – the free version has only limited functionality, so in order to get a proper visualization tool, it is necessary to buy a license.

Finally, a piece of software with high relevance to EBCS systems is the *MATSim simulation system* [3]. It can be nicely integrated with EBCS and especially DEECO applications in order to see how the applications behave in a simulated environment. The MATSim-produced output contains information about the environment and components. However, in EBCS applications the ensemble-related output is not handled by the MATSim system, and has to be managed separately.

6. Conclusion and Future Work

We have provided a visualization solution for EBCS systems. In the current form, it accepts input data obtained from DEECo applications. However, its architecture has been designed so that it is easy to modify the application in order to make it accept data from different EBCS systems.

The application meets the criteria set in the *introduction*. First, it is written in Java and uses the standard JavaFX library for handling of the graphical output. As a result, it is implemented in a platform-independent manner.

Next, it facilitates customizing of the visual output – it is possible to modify the visualization of environment, components and ensembles. The basic customizations can be made using the GUI controls, while for the more refined ones there is a scripting environment available. In addition to the scripting interface, we have also included a plugin interface, which makes it possible to extend the application's functionality by providing standalone plugins.

The application also provides support for big files – it is possible to select a section of the input files to visualize. We have measured the performance of the section finding process, and we conclude that the process runs at under 1 ms on inputs sized up to a few gigabytes.

Finally, we have measured how long it takes the application to load various inputs. We conclude that for inputs of size up to 30 MB the loading time is kept well-below the 10s limit, which helps the users keep their attention.

There is space for future development mainly in three key areas. First, it is currently being considered to modify the concept of an ensemble in the DEECo component model, so that it can model much more general relations than just the current 1-to-many coordinator-members relation. When such a change is introduced, an opportunity arises to implement the change into our application. The application architecture has been designed in such a way, that it is ready to be modified accordingly, in order to support input coming from applications built upon the updated DEECo component model.

Second, it is possible that more of new applications following the EBCS model will be developed in the short future. If that is the case, we see it as a good opportunity to modify our work for the needs of such applications.

Last, there is currently effort being made for the JDEECo applications to produce an index file, apart from the usual logging data. The file contains mapping of the simulation time to offsets in the logging file, which makes it possible to search the input data quicker than at the present time. We consider it quite trivial to integrate this new functionality into our application, provided that the format definition of the index file is available.

Bibliography

- [1] Keznikl J., Bureš T., Plášil F., Kit M.: Towards Dependable Emergent Ensembles of Components: The DEECo Component Model, Proceedings of WICSA/ECSA 2012, Helsinki, Finland, pp. 249-252, IEEE CS, ISBN 978-0-7695-4827-2, DOI 10.1109/WICSA-ECSA.212.39, August 2012
- [2] Bureš T., Gerostathopoulos I., Hnětynka P., Keznikl J., Kit M., Plášil F.: DEECo - an Ensemble-Based Component System, In Proceedings of CBSE 2013, Vancouver, Canada, June 2013
- [3] Matsim.org, 2016, MATSim: Agent-Based Transport Simulations. [online]. 2016. [Accessed 23 July 2016]. Available from: <http://www.matsim.org/>
- [4] Beetz, K., Böhm, W.: Challenges in Engineering for Software-Intensive Embedded Systems. Model-Based Engineering of Embedded Systems. pp. 3–14. Springer (2012)
- [5] Lee E.A.: Cyber Physical Systems: Design Challenges. Proc. of ISORC'08. pp.363–369. Orlando, FL, USA (2008).
- [6] Keznikl J., Bureš T., Plášil F., Gerostathopoulos I., Hnětynka P., Nicklas Hoch: Design of Ensemble-Based Component Systems by Invariant Refinement, In Proceedings of CBSE 2013, Vancouver, Canada, June 2013
- [7] Department of Distributed and Dependable Systems at Charles University in Prague, 2013, D3S Component Group Repository. [online]. 2013. [Accessed 23 July 2016]. Available from: <https://github.com/d3scomp/JDEECo>
- [8] Bureš T., Gerostathopoulos I., Hnětynka P., Keznikl J., Kit M., Plášil F.: Gossiping Components for Cyber-Physical Systems, Proceedings of ECSA 2014, Vienna, Austria
- [9] Oracle Corporation, 2011, JavaFX Documentation Home. [online]. 2011. [Accessed 23 July 2016]. Available from: <https://docs.oracle.com/javafx/2/>
- [10] Heineman G.T. , Councill W.T.: Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley Professional, Reading 2001 ISBN 0-201-70485-4
- [11] Shoham Y.: Agent-Oriented Programming (Technical Report STAN-CS-90-1335). Stanford University: Computer Science Department, 1990.
- [12] Abeywickrama D.B., Bicocchi N., Zambonelli F.: SOTA – Towards a General Model for Self-Adaptive Systems. In Proc. of WETICE '12, 48 –53, 2012.
- [13] Baresi L., Guinea S., Tamburrelli G.: Towards decentralized self-adaptive component-based systems. In Proc. of SEAMS'08, 57–64, 2008.

- [14] Bresciani P., Giorgini P., Giunchiglia F., Mylopoulos J., Perini A.: Tropos – An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*. 8, 3, 203–236, 2004.
- [15] Holz M., Wirsing M.: Towards a System Model for Ensembles. *Formal modeling*. 241–261, 2012.
- [16] Tan K.C., Li Y.: Performance-based control system design automation via evolutionary computing. *Engineering Applications of Artificial Intelligence*, 14 (4). pp. 473-486. ISSN 0952-1976. 2001.
- [17] Mehlhorn K.: *Data Structures and Algorithms 1 – Sorting and Searching* (Monographs in Theoretical Computer Science, an EATCS Series). 1987, Berlin: Springer-Verlag, 1st Edition.
- [18] Andersson A., Mattsson C.: Dynamic interpolation search in $o(\log \log n)$ time. In *Automata, Languages and Programming*, 700:15–27, 1993.
- [19] Oracle Corporation, 2011, JavaFX Frequently Asked Questions. [online]. 2014. [Accessed 23 July 2016]. Available from: <http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html>
- [20] Nielsen Norman Group, 2010, Website Response Times. [online]. 2010. [Accessed 23 July 2016]. Available from: <http://www.nngroup.com/articles/website-response-times/>
- [21] Matsim.org, 2012, OTFVis: A visualizer for MATSim. [online]. 2012. [Accessed 23 July 2016]. Available from: <http://matsim.org/docs/extensions/otfvis>
- [22] Senozon AG, 2015, Via: Visualizing MATSim results. [online]. 2015. [Accessed 23 July 2016]. Available from: <http://via.senozon.com/>

List of Abbreviations

<i>DEECo</i>	Dependable Ensembles of Emerging Components, a component model
<i>EBCS</i>	Ensemble-Based Component Systems, a computational model
<i>JavaFX</i>	(Java framework for creating rich GUI applications)
<i>JDEECo</i>	Java implementation of DEECo component model
<i>JDK</i>	Java Development Kit
<i>JRE</i>	Java Runtime Environment
<i>JVM</i>	Java Virtual Machine
<i>MATSim</i>	Multi-Agent Transport Simulation Toolkit, simulation framework
<i>sCPS</i>	Smart Cyber-Physical System

Attachments

An important part of the thesis is the presented software work. It is provided on the attached optical disc, as well as in the following online repository:

`github.com/filipekt/JDEECovisualizer`

The data contained in both sources contains the source code of the application, as well as the compilation and start-up scripts needed to run the software. The basic directory structure of the provided data is as follows:

```
--bin          [Compile & run scripts for the
                application]

--example_data [Example input data for the application]

--plugin_projects [Source code for provided plugins]

--plugins      [Provided plugins, already compiled]

--resources    [Mostly graphics used by the application]

--src          [Source code of the application]

--thesis       [Text of the thesis]

README.txt     [Basic instructions on how to start the
                application]

[and other miscellaneous files]
```