Ανοικτό Πανεπιστήμιο Κύπρου

Σχολή Θετικών και Εφαρμοσμένων Επιστημών

Μεταπτυχιακή Διατριβή
στα Πληροφοριακά και Επικοινωνιακά Συστήματα

Designing and Implementing a Visualization Solution for a Multi-Agent System

Viktorija Leonova

Επιβλέπων Καθηγητής
Loizos Michael

January 2015

Ανοικτό Πανεπιστήμιο Κύπρου

Σχολή Θετικών και Εφαρμοσμένων Επιστημών

Designing and Implementing a Visualization Solution for a Multi-Agent System

Viktorija Leonova

Επιβλέπων Καθηγητής
Loizos Michael

Η παρούσα μεταπτυχιακή διατριβή υποβλήθηκε
προς μερική εκπλήρωση των απαιτήσεων για απόκτηση

μεταπτυχιακού τίτλου σπουδών
στα Πληροφοριακά Συστήματα

από τη Σχολή Θετικών και Εφαρμοσμένων Επιστημών
του Ανοικτού Πανεπιστημίου Κύπρου

January 2015

Abstract

In this work, we are giving an outline to a visualization application, which will provide a user with a visual representation of the output produced by a multi-agent simulation during its run and originally delivered in the form of statements in DAMMAGE language [01]. Our purpose here is to discuss the boundaries of the project, its proposed architecture and generally to define a path for its implementation.

A solution proposed in this work includes two parts. The first is a graphic interface, which takes as an input a plain text log file and demonstrates to the user a customized visual representation of its content along with providing him or her with the possibility to navigate through it, acquire the needed information in a handy manner and observe the changes between the consecutive states in dynamics. The second is a set of customization elements, used for the visualization in an individualized manner, including user-defined images.

We hope that the proposed solution will improve the user experience in working with the multi-agent simulation system and provide the observers with new opportunities that before would take too much effort to explore.

Περίληψη

Σε αυτή την εργασία δίνουμε το περίγραμμα μιας εφαρμογής η οποία παρέχει στο χρήστη οπτική αναπαράσταση του αποτελέσματος προσομοίωσης πολλαπλών πρακτόρων κατά τη διάρκεια της λειτουργίας ενός προσομοιωτή στη γλώσσα DAMMAGE [01]. Σκοπός μας εδώ είναι να εξετάσουμε τα όρια του έργου, την προτεινόμενη αρχιτεκτονική του, και γενικά να καθορίσουμε μια πορεία για την εφαρμογή του.

Η λύση που προτείνεται στην παρούσα εργασία περιλαμβάνει δύο μέρη. Το πρώτο μέρος είναι η γραφική διεπαφή, η οποία λαμβάνει ως είσοδο ένα απλό αρχείο κειμένου με το ιστορικό της προσομοίωσης και δείχνει στον χρήστη την οπτική αναπαράσταση του περιεχομένου του, μαζί με την παροχή σε αυτόν της δυνατότητας να περιηγηθεί μέσα από αυτό, να αποκτήσει τις απαραίτητες πληροφορίες με πρακτικό τρόπο, και να παρατηρήσει τις μεταβολές μεταξύ διαδοχικών καταστάσεων. Το δεύτερο μέρος είναι ο καθορισμόν επιλογών από το χρήστη, με στόχο την οπτικοποίηση με εξατομικευμένο τρόπο, συμπεριλαμβανομένων και της χρήσης εικόνων που ορίζονται από τον χρήστη.

Ελπίζουμε ότι η προτεινόμενη λύση θα βελτιώσει την εμπειρία των χρηστών σε συνεργασία με το σύστημα προσομοίωσης πολλαπλών πρακτόρων, και θα παρέχει στους παρατηρητές ευκαιρίες εξερεύνησης της προσομοίωσης για τις οποίες το υφιστάμενο σύστημα δεν ήταν επαρκές.

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Loizos Michael for his patience and thoughtfulness. His energy was always there to tap upon, and his ideas served as a constant source of inspiration. Without his precious input and critical feedback, this work would be much less of a one.

I would also like to thank Elektra Kypridemou for her genial help and priceless comments.

My gratitude is also due to Mihails Gondarevs, a programming genius, who helped me to defeat the challenges of XAML / WPF technologies, and Valdis Murzins, who introduced me to the world of user interface assessment.

Separate thanks to the patient friends of mine, who agreed to serve as model organisms for my inhumane experiments in the area of user interface evaluation and stalwartly sustained the trials.

Introduction **8**

1.1 General Overview 8

1.2 Related Works 10

The Existing System and Limitations of Its Visual Interface **13**

2.1 Overview of the System 13

2.2 Structure 14

2.3 Operation of the System 16

2.4 Output and Usability 17

Problem Analysis and Proposed Solution **19**

3.1 Problem Formulation 19

3.2 Logical Entities 21

3.2.1 Log File and Its Processor 21

3.2.2 Market State 21

3.2.3 Agent 22

3.2.4 Item 23

3.2.5 Key-Value Pair 25

3.2.6 Event 26

3.2.7 Clock 27

3.3 Basic Functionality 28

3.3.1 Demonstrating the Market State 28

3.3.2 Reflecting the State Changes as They Happen 29

3.3.3 Reproducing the Simulation Run 30

3.3.4 Reflecting the On-Going Simulation 31

3.3.5 Navigating the Market States 31

3.3.6 Customizable Look 32

3.3.7 Global Lists 32

3.3.8 Point of View 33

3.3.9 Connecting as an Administrator 33

3.3.10 Connecting as an Agent 34

Design and Development **35**

4.1 Implementation 35
4.1.1 Input of the Visualization Module 35
4.1.2 Input Format 38
4.1.3 Input Processing 41
4.1.4 Classes 44
4.1.5 Output Generation 49
4.2 Visualization Details 51
4.2.1 General Assumptions and Conventions 52
4.2.2 Customization Elements 54
4.2.3 Agent Frame and Its Contents 55
4.2.4 Market State and Related Entities 59
4.2.5 Visualization Implementation 67
4.3 Control Elements 68
4.3.1 Main Slider 68
4.3.2 Navigation Element Group 69
4.3.3 "Admin" Button 70
4.3.4 "Config" Button 71
4.3.5 "Connect as Agent" Button 72
4.3.6 Exit Button 73
4.3.7 Miscellaneous Controls 73

Interface Assessment 74
5.1 Used Methods 74
5.2 Participants 76
5.3 Evaluation Procedure 76
5.3.1 Object of the Evaluation 77
5.3.2 Participants 77
5.3.3 Think Aloud Method 77
5.3.4 Heuristic Evaluation 78
5.3.5 Cognitive Walkthrough 78
5.4 Results and Data Analysis 79
5.5 Summary 81

Future Work 82

6.1.1 Supporting the History of Agent Actions 82
6.1.2 Item Movement History 83
6.1.3 Supporting the Reproduction of Data Stream from Network Connection 83
6.1.4 Application Interoperability 84
6.1.5 Improved Visibility of the Agent Account 84
6.1.6 Supporting a Manual Run Mode 84

Conclusions 86

Glossary 88

Bibliography 90

Example of an Environment State B-1

Example of a Visualization Customization File B-1

List of Heuristics for Interface Assessment C-1

Issues Identified during Interface Assessment D-1

D.1 Issues Identified by Think Aloud Method D-1
D.2 Issues Identified by Cognitive Walkthrough Method D-4
D.3 Issues Identified by Heuristic Evaluation Methods D-4

User Manual E-1

E.1 Application Deployment E-1
E.2 Starting the Application E-1
E.2.1 Changing Configuration Settings E-2
E.2.2 Running the Visualization E-4
E.2.3 Available Functionality E-5

Flow Chart of the Application F-1

F.1 Flow Chart F-1

Chapter 1
Introduction

In this chapter, we would like to give an outline to the present state of the Multi-Agent Systems area and to introduce the reader to the goals of this work and motivation underlying the development of the solution we propose. We also review related works in the field.

1.1 General Overview

Over the past decade, our society has come to experience the explosive growth of the interconnectivity, social and otherwise, affecting virtually all spheres of life and dramatically changing all we were used to. This growth has brought along the development of all related fields, including the e-commerce. The fraction of transactions handled online has increased rapidly over the past years, and the volume of relevant information has become overwhelming.

The situation has come to the point where a sheer amount of information available to users makes it only partially useful — human perception and analysis capabilities are pretty limited compared to a number of commercial institutions available online, and a volume of information stored in those. However, working with a massive amount of information is the job typically associated with computers, and some researchers have looked to employ those in order to harness the tide of information. However, as the web is ever-changing and very dynamic environment, a simple static system is not enough. What is actively studied by the multiple researchers, though, is the possibility to create a proxy agent, acting on the behalf of a specific body and in its best interests.

There are numerous directions in this field, which the research has targeted. Some take it large-scale and build agents that make predictions about the product pricing and the market conditions [02], while some are trying to develop an agent that could act as a proxy with a certain degree of freedom and is driven by the intent of maximizing the service value for a customer [03]. In certain areas, such as stock markets, software agents, if the ones without the inherent ability to learn and less free in their actions, have become virtually omnipresent, basically defining the appearance of the nowadays stock markets and affecting the behavior of their human counterparts [04].

It is only natural that before such agents are released to the real-world environments they need to be tested and their efficiency is to be evaluated and assessed. Unfortunately, when we are talking about asynchronous agents, the traditional methods of testing fail to be adequate due to computational complexity. Thus the need for the simulation environments and the way to observe and analyze the behavior of the agents.

In our work, we are examining an example of such environment — DAMMAGE framework, consisting of a language coupled with a market environment [01] that provides the possibility for an arbitrary number of agents, both human and software, to connect to a virtual market environment and interact with each other. One of its strong sides is that this environment can serve for both completely virtual runs and for a fully-fledged real-life interaction — the difference is only how the results would be interpreted in the real world. By defining different domains, it can serve as a framework for different interaction models. One of such is the English auction domain, where agents can open and close auctions, raise bids and so forth. However, the output of the environment, including the actions of the agents, auctioned items and the like, while readily observable in the form of a log file, is not easily understood, as it is in essence a list of consequent PROLOG-compatible descriptions of market states in textual format.

This work is conceived with the intention to endow potential users of the virtual environment with a tool that would make it easy to see what happened during its run at any given moment or is happening right now, in human-friendly, understandable way — namely, with the complex of the visualization mechanism and the means to control it. For this purpose, we have designed and developed the software that we are proposing as a solution. In this thesis, we are outlining the problem and our concept of proposed solution, reviewing the challenges, and discussing the design and development choices as well as the details of the implementation.

1.2 Related Works

As the Multi-Agent Systems area has experienced a great advent for the last few years, thanks to the dramatic increase in the computational power, new techniques for their implementation and the frameworks employing these techniques, so have done the visualization methods. This is only natural, as due to that human perception is in the greatest part visual [05], visualization helps understanding the multi-agent systems at any stage of its implementation [06]. There have been quite a few works on this subject, studying different aspects of the visualization for agent-based models. Within the course of this thesis, we have explored them and analyzed several of those works, choosing the representatives for the different facets of this field of study.

As our primary concern was designing a user-friendly and understandable visualization, and assessing it afterwards, the most relevant subjects are modelling toolkits and frameworks along with their assessment techniques. As of now, there is a plenty of visualization systems on the market, providing the users with a possibility to visualize multi-agents models. These systems include toolkits like NetLogo [07], CAVE [08] and even tools supplied with a scripting language for visualization, like MODAVI [09]. Some researchers go as far as building 2D and 3D visualizations for real-world interactions with engines that have a built-in physics module and realistic real-world objects, such as chairs or tables [10]. However, all of these systems, some of which are visualizing agent-based models in highly aesthetic and comprehensive way, have one attribute in common that makes them of very limited applicability for our case: none of them depicts an inner state of an agent and all of them are in general focused not on the agents and their interactions, but on the general view of the environments [11]. This makes a perfect sense, when the research contains a huge number of the agents and is focusing on the general course of events, and not on the details of specific interactions. A typical example of such representation can be observed on Figure 1.1.

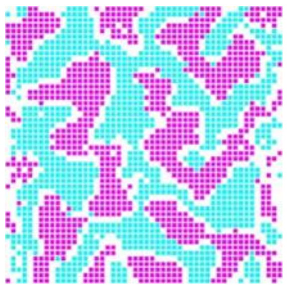


Figure 7.1. Philip Rutten's visual representation of a multi-agent system. Every agent is motivated to assume the same color with its neighbors. [12]

However, the area of the utmost interest for us is the economic environments and their visualization, especially the auctioning-related models. The formal framework, developed by A. Artikis et al [13], serves as an example of an economic environment with a notion of agents that can open auction for something or raise bids, their powers, permissions and well-defined rules of inference. It is also coupled with a visual interface but, although this interface allows to see distinctively the market states in a form of Prolog statements, as well as to see the members of the environment and the important entities, it is still of a limited observability for users not familiar with Prolog and its syntax.

In addition, there is such framework as a visualization assistant system, ViA [14], designed and developed with picturing multi-agents interactions in the economic environment with auctions, among other applications, in mind. It facilitates the production of a pretty detailed visualization of a multi-dimensional environment with high number of agents, bids and instruments. Although, alike the tools described above, it focuses on the environment, picturing the bids and the asked quantities, the bid wars, et cetera, hardly giving the agents any attention.

Another shot at visualization of the multi-agent systems used for auctioning and other formalized purposes was made by A. Bogdanovych et al, when they have proposed a methodology for developing multi-agent systems as 3D electronic institutions. Namely, they propose to separate such development in two almost independent stages: firth, the visually-assisted specification of the system, including the definition of roles, allowed actions, obligations, and so forth [15], second, the automatic generation of these institutions, and finally, annotating the generated environment with 3D Virtual Worlds [16]. As a result, a human agent can connect to the application server and interact with other market participants through visual interface, similar to the ones employed in first-person computer games. Artificial agents, in they turn, can connect to the application ignoring the visualization layer and communicate with the others using the specification language. They will be visible for other participants in the same way.

While the representations of multi-agent systems as 3D electronic institutions developed under this framework undoubtedly allows human participants a comfort of visual interaction, there is one quality that the framework lacks: it doesn't provide the convenient way for an observer to examine the agents and their inner states. The only possibility for a user, except of seeing the avatars and actions of other market participants, is to observe the agent when it is acting on his behalf. As the environment is separated into institutions, there is not much a common participant can observe.

One more important facet that we have examined within this work is the assessment of the developed visualization. Although, as there are numerous metrics specifically developed for multi-agent simulations [17], those are also designed for visualizations that cover the entire environment, picturing only a limited set of agent features, such as orientation, distance and so forth. Thus, we had to resort to using more general metrics for assessing the interface of a general software application.

Chapter 2
The Existing System and Limitations of Its Visual Interface

In this chapter, we intend to describe the functionality of the existing system, as well as its operation and motivation behind its development. We also discuss current limitations of its applicability, inferred by the minimalism of its visual interface and implying the opportunity to improve the usability of the system by endowing it with a visualization module.

2.1 Overview of the System

In our work, we have studied the system, developed by L. Michael, D. Parker and A. Pfeffer and described in "Specifying and Monitoring Economic Environments using Rights and Obligations" [01] published in 2009. This article, as its name implies, focused on specifying and monitoring economic environments, and presented a formal scripting language coupled with a run-time system that we shall refer to as the simulation system. The scripting language employed by the system is effectively an extension of Prolog and allows defining rights and obligations explicitly, as well as provides for the possibility to define complex action-based transactions. The rights and obligations are derived from the property and are considered first-class goods, which allows for the recursive definitions, such as a right to sell a right or an obligation to buy an obligation. The run-time system represents an agent-mediated environment that allows monitoring and enforcing rights and obligations of the agents, in the sense that for each particular action of an agent the system would check whether this action complies with the rights possessed by the agent and whether or not any obligations of any agent were violated as a result.

This system has defined several domains, in other words, several sets of definitions. Each domain defines its own set of item classes, possible actions and their implications, rules for generating rights and obligations in the result of an action, et cetera.

The reasons for implementing such system are logical: in the modern world, agent-mediated economic environments are omnipresent due to the advent of the electronic commerce. Virtually every device that has access to the internet can join one or the other electronic market. There is an ever-present need for reliability and trust in such environments, which means that agents should act in the interests of the represented parties. Thus, providing a framework with documented rules and producing observable and verifiable results can be seen as a step towards satisfying this need. Such system can be used to counter the real market challenges, such as an auctioning of license plates by the Road Transport Department of Cyprus [18], or for experimental testing of different sets of market settings and options and their influence on the agent behavior. Other possible areas of application include modelling social and economic processes or studying different agent models in the field.

However, with all these advantages, the system lacks one quality — a visibility. As of now, the output of the system is either stored in the form of page-long statements (see Appendix A for an example) or are demonstrated on the console in the same format. One willing to examine the state of the market at the given time can, of course, get a tool that allows text parsing, but doing so is not too convenient, and the possibility of introducing human error while looking on the cramped text is high. This is the rationale behind our proposed application: we think that the usability of the system will be significantly improved, if the data hidden in these statements will be displayed in a neat and handy visual form.

2.2 Structure

As of the moment, the original system includes a complex engine developed by Dr. Loizos Michael et al, as we mentioned in the previous section, serving as a virtual environment for the agent communication. By using different domains, the system can serve as a marketplace for both human and software participants, or as a game board. In our work we are focusing on the English auction domain of the simulation, however, it can be customized for other domains as well. In this work, we also occasionally give examples from the Coin domain. The purpose of the system is to provide the environment, mainly of economic nature, where participants can exchange money, items, rights and obligations. The system provides the possibility for agents, both human and computational, to connect to the simulation and participate in such exchange.

The system encompasses four service modules written in Prolog, ensuring the operation of the system, and supplies a few additional ones, namely, a module allowing human participants to connect to the simulation and a set of artificial agents. There are the following core modules:

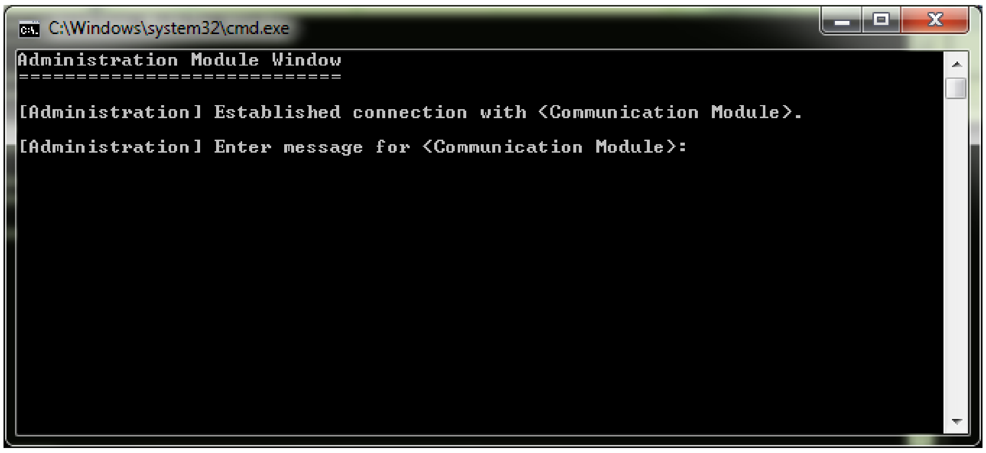
1. Communication module, whose function is to provide the interoperability between all parts of the system and the agents, both human and software. The communication is carried over through a socket connection and can be performed from remote machines.

2. Administration module. As the name implies, it is used to administrate the system and serves for sending commands to it. The module accepts user input from a console and passes it to the communication module, which processes it and acts accordingly.

3. Registration module. Its main purpose is to register the agents in the system, supply them with unique ids and ensure that they can interact with the simulation.

4. Simulation module, the last and the most important. It is essentially a heart of the system. Simulation module is the one providing the agents with the possibility to use the environment. It includes sending them the states of the environment and the updates of those, supplying them with rights and obligations in the result of environment events, etc. It also includes a built-in master agent, which controls the performance of the obligations and otherwise manages the state of the environment.

These modules are run as console applications, in the listed order. Below a console of the administration module can be seen:


C:\Windows\system32\cmd.exe

```
Administration Module Window
============================

[Administration] Established connection with <Communication Module>.

[Administration] Enter message for <Communication Module>:
```

Figure 2.1. Visual appearance of the administration console. On the console, one can see a connection message and an invitation to enter the command. No other clues of functionality or the current state of things are provided.

All communications with the system made by the agents or by its administrators are performed using DAMMAGE language and the events happening during the simulation run are described using the same notation.

Although, however intuitive and easy learnable, this language is not native for humans, thus one willing to understand what have happened during the run will have to decipher the statements in it, which may prove to be a toilsome work, especially if the statements are of a considerable length.

2.3 Operation of the System

System operates in the following manner: first of all, its basic modules, listed in the previous section, are to be run in the predefined order to provide the operational environment, or the simulation. One should supply the simulation with an indication, what domain should be used. After that, system sets up the environment and starts the simulation. At this point, the agents, human or computational, can connect to this simulation. As per English auction domain, which I am focusing on, on entrance each agent is given an item (an “apple”) and some amount of money. However, with the changes in the domain, the definition of what agents possess in terms of items and money, may not be pre-defined, but be determined in some other way.

Therefore, the agents may possess items and / or to be looking to come into possession of some, along with the budget. In addition to the actual assets, the agents may possess rights to the object, which in its turn may, be a right, an obligation or an asset, or be a subject to some kind of an obligation. Upon connecting to the system, these agents become participants of the simulation marketplace. As such, they may at their sole discretion perform actions, naturally associated with this role. For example, they may create an auction for an item in order to sell it; or they can place a bid on someone else's auction hoping to buy a desired item; or they can query the attributes of the items if they are entitled to do so. In addition, they may perform the same operations on the rights they own or their obligations of some kind. In order to do so, they ought to send a message to the simulation, using DAMMAGE as a command language.

```
> open_auction(MyAuction, apple(25), 1)
```

Figure 2.2. Example of a command sent by an agent. This line commands a simulation module to open the auction MyAuction for item "apple(25)" with opening price of 1.

Upon each change, for example, caused by an action performed by an agent, or simply upon the passing of a certain time interval, a market state is written to the log file by simulation module using the same DAMMAGE syntax. The same market state update is communicated to all agents via the communication module. Thus, the market state at any given moment can be reproduced accurately. However, with virtually any number of agents bigger than one, the description of the market state tends to acquire a considerable length, thus heavily impairing the readability of the log file.

2.4 Output and Usability

As I have mentioned in passing in the previous sections, a running simulation engine produces state updates in two cases: if an event has happened in the environment, thus causing the state to change or the certain time period, which is typically set to twenty seconds, has passed since the previous state update. Such state is essentially a snapshot of the market environment: it is listing all objects currently found in the environment, such as items, rights, and obligations along with all their attributes. It also contains a clock, stating the time passed from the moment of a simulation start for both capturing the state and its expiration, in other words, the moment where the next state was produced. In case when the state generation was caused by an event, it also contains an event object, along with its description: what has happened and to which result. All these descriptions are given in the form of Prolog-compatible statements, declaring state(…) with a list of attributes inside. These states are communicated to the agents and are stored in the log file in plain text format.

The problem of this is as follows. Imagine a fairly modest simulation run with but a three agents. Every agent has one item and about a dozen of rights, for example, a right to the item that belongs to him, the right to open an auction for this item, et cetera. Each right has a set of attributes, and so does the item. For three agents it amount to a statement a few thousand symbols long. While logical statements are not hard to understand in itself, not many users are familiar with them, and even those who are will have difficulties understanding the statement of considerable length. In order not to give unfounded accusations, I shall give an example of one such statement. It encompasses four agents with one item each. Suffices it to say that it takes a few pages on its own! (See the Appendix A to explore the readability of such statement). It is easy to see that if a user wants to actually understand what was going on in the simulation run in some moment, it will be difficult, to say the least.

Chapter 3
Problem Analysis and Proposed Solution

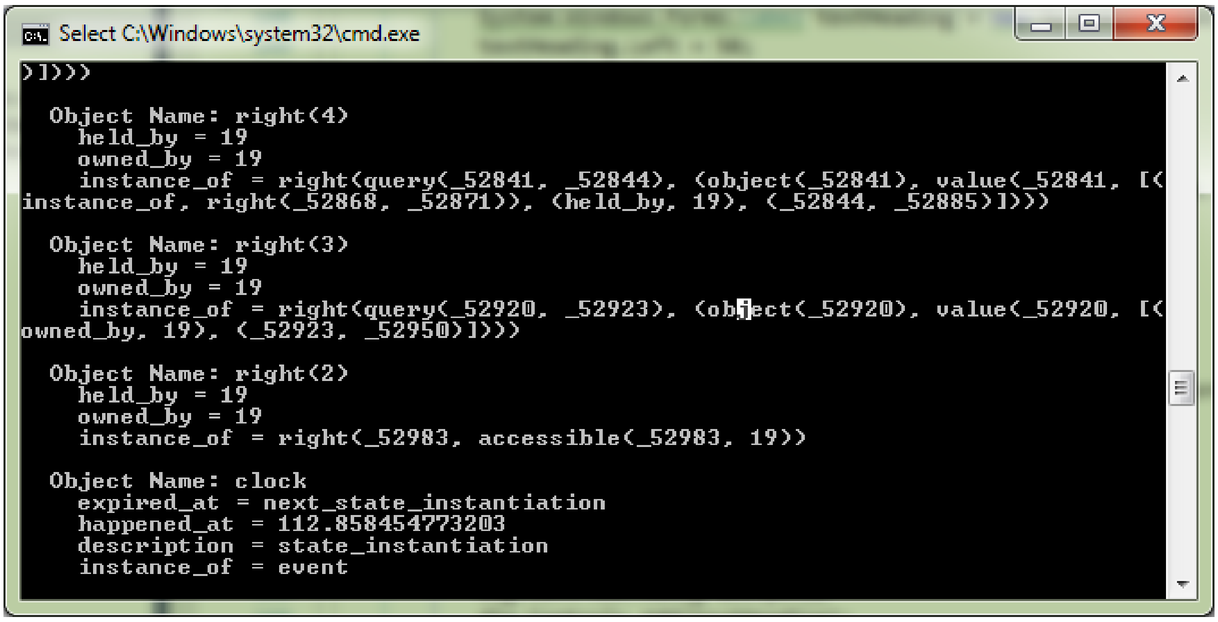
In this chapter, we examine the existing problem, give its formulation and perform its analysis, as well as give an outline to a proposed solutions, its content and scope.

3.1 Problem Formulation

As we saw in the previous chapter, the information produced by the simulation engine is given in a lengthy and unwieldy form. Plain text statements, listing all the objects and their attributes are difficult to understand and even if they contain useful information, it is hard to keep track of. Thus, we need a tool that would allow us to see the entire state at once in a user-friendly manner, as well as to see the difference between states. What we are proposing to do is to add a new external component to the system — a visualization module that processes a running simulation log file (with administrator rights if needed) and produces the visual representation of a market state change or re-plays a simulation run previously, taking the log file as an input. This would allow human watchers to see events that in the current state of things would require a considerable amount of deciphering directly. And, in case if a human agent desires to connect to the market environment, it would give him or her a valuable opportunity to see what happens without extra effort needed to process the corresponding lines.

Such module would read the log file line-by-line and picture the sequence of the market states using a set of conventions, thus providing a possibility for researchers not familiar with Prolog statements to still be able to observe the operation of the system in a convenient manner, and even those familiar with its syntax can benefit from it, as a visual representation tends to be much more compact and readable than a description in a form of a logical statement.

Also, in the original system a human agent connecting to the interface could only see the command prompt, inviting him to enter the command. If he or she wants to see what happens, the only option is to read the log file or the output of the simulation console


Select C:\Windows\system32\cmd.exe

```
>]>>

  Object Name: right(4)
    held_by = 19
    owned_by = 19
    instance_of = right(query(_52841, _52844), (object(_52841), value(_52841, [(
instance_of, right(_52868, _52871)), (held_by, 19), (_52844, _52885)])))

  Object Name: right(3)
    held_by = 19
    owned_by = 19
    instance_of = right(query(_52920, _52923), (object(_52920), value(_52920, [(
owned_by, 19), (_52923, _52950)])))

  Object Name: right(2)
    held_by = 19
    owned_by = 19
    instance_of = right(_52983, accessible(_52983, 19))

  Object Name: clock
    expired_at = next_state_instantiation
    happened_at = 112.858454773203
    description = state_instantiation
    instance_of = event
```

Figure 3.1 Simulation console outputting the market state. It shows four objects: right(4), right(3), right(2) and a clock, accompanied with their attributes.

As one can see from this picture, this is not very user-friendly. The state is written in the form of many consecutive lines, where it is hard to distinguish one object from another. The same applies to the administration console. If an administrator wishes to invoke a certain command in some specific conditions, he or she will have a hard time understanding that the conditions are indeed met without some additional coding. Thus, we intend to provide the user and / or the administrator of the system with the possibility to connect to the simulation directly from the visualization application in order to provide a seamless user experience.

3.2 Logical Entities

Having determined that the visualization tool would serve our purpose, we shall establish, what exactly the visualization module shall reflect. This shall be done by identifying the main entities of the simulation environment from an agent's or a spectator's point of view and breaking them up into smaller pieces that would be easy to observe and that would be useful to be able to look at. In this chapter, we shall give visualization examples for illustrative purposes, without going deep into the implementation details. We shall discuss the visualization details, assumptions, reasons and decisions in the Design and Development Chapter.

3.2.1 Log File and Its Processor

First of all, the greatest encompassing entity is the history file itself — the list of all consecutive market states from the beginning of simulation run to the end of the run or to the current moment. However, the representation of a single market state for the purposes of the visualization module shall be, of course, different from the logical statement, as they serve two different purposes. Thus, the visualization module should be equipped with the ability to load a market state history from file, process incoming lines and transform them into appropriate format. However, there would be no use of keeping the list of all states if we are not doing anything with them. Therefore, the visualization module should also possess such functions as retrieving a state by its number or by the time passed from the moment of a start of a system. This would endow the visualization module with the ability to browse through the log file as needed. We shall not, of course, visualize all states at once, but instead we will do so reflecting one state at a time in a consecutive manner or as requested. This brings us to the next type of logical entities, of which such log file consists, namely, a market state.

3.2.2 Market State

A market state is probably the most natural logical entity in the proposed application. One such is to be mapped to every line received from the data source. It should contain all information encompassed within it. The market state is exactly what its name implies: a state of the market environment in the given moment. The market state should contain the collection of all items on the environment, the event, if such is present, and the clock. However, this is not enough. Imagine we will only reflect the market state in such form: some visual field with a bunch of items on it. While

I believe that even this would be a significant improvement in comparison with having to read and understand the log file, the least we can do is to group items.

And the first such level of grouping would be to split the items between the agents. Notwithstanding the fact that the agents are not listed in the simulation run log explicitly, we can always determine what agents are currently present in the environment, as each agent has at least one item — its account, and is listed as its owner and holder. We shall visually represent the market state as the main visual field of the application, encompassing all the agents, clock and other entities that we will determine further.

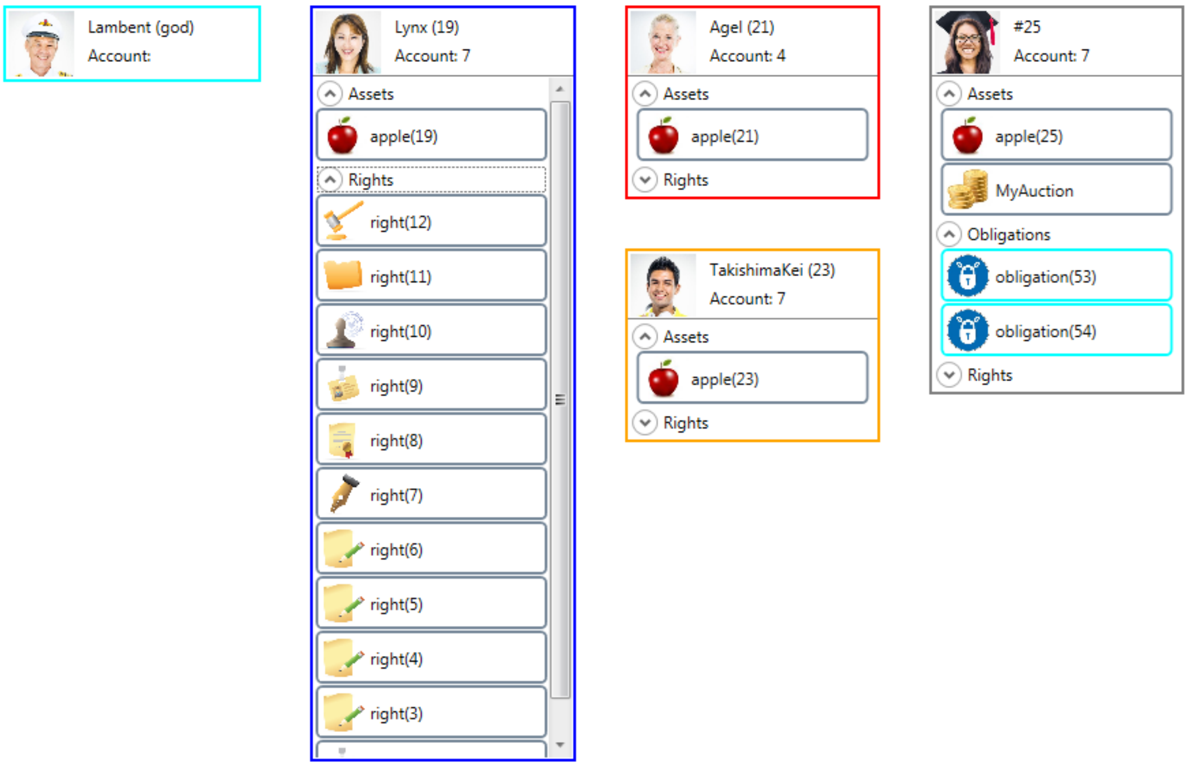


Figure 3.2. Agents of the market state, each having its items grouped further for better readability, with customized name and color. For example, the master agent with id "god" is pictured with the cyan-colored border and the icon picturing a person in captain's hat.

3.2.3 Agent

It seems only natural that our visualization will focus on the agents, and represent all the other objects in relation to them, as they are the only actors in the simulation marketplace. In real life, there are also environmental factors that may affect market conditions directly or indirectly, such as new laws or taxes, or even an earthquake that has damaged significant manufacturer's facilities, but these are not directly present here. Therefore, we will have the agent entities, picturing each of those as a separate object. The agents will be uniquely identified by their ids.

As per current implementation, an agent is effectively defined by the collection of the objects it has in its possession, most important of which in the auction-related domains being its account, representing the amount of money it has in its disposal, as it is the only item which agent has unconditionally, thus we can always be sure that if the agent is present in the market environment, we shall not miss it.

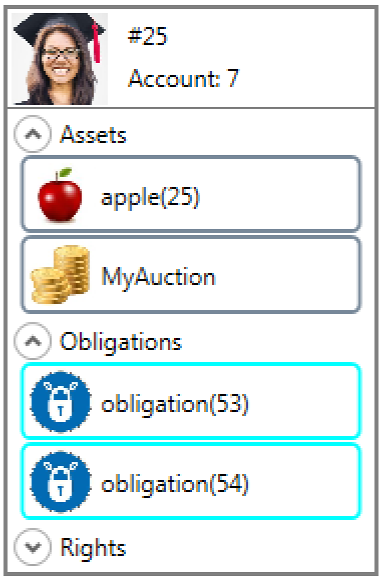


Figure 3.3. An agent in the market environment. One can see its number, account value and unique identifiers of the items it holds (apple(25), MyAuction, obligation(53) and obligation(54)). Rights section is not expanded

Having spoken that the agent is defined by its objects, we can identify the next logical entity — that is, the item itself.

3.2.4 Item

While agents will serve as a centerpiece of the visualization, about which everything revolves, the items are the building blocks from which the market state is built and are therefore the key to the visualization. The items are identified by their unique name. They contain two types of attributes: arbitrary, such as weight (for apple) and mandatory. There are two mandatory attributes for any object: held_by, describing, who holds the object at the given moment, and owned_by, which indicates the owner of the object. By examining the content of these two fields of all objects on the market, we can determine what agents are currently present in the market environment, and what items are possessed and owned by each of them.

Although the current simulation does not discriminate between different types of items, such differentiation would be very helpful in terms of human perception. It is very difficult to find a desired object in a non-structured heap, the best showcase being a needle in a haystack; however, if those objects are sorted and grouped, such task becomes much easier. There are the following logical types of items in the system:

1. Account of an agent. It is possessed by every agent and cannot be alienated from it. We use this fact to guarantee that we will not overlook any of the agents present in the market environment.

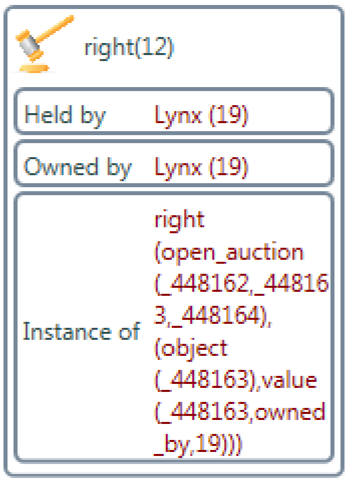
2. Assets. "Things", objects possessed by the agent that he can dispose of at its own discretion. In the auction domain, those can be sold and bought. They cannot disappear from the environment: if an agent leaves the environment, all his assets are handed over to the master agent.

3. Rights. A right is a special case of the object, which is defined by two components: the action the holder is entitled to and the conditions of its effectiveness. As per description of the system, given in "Specifying and monitoring economic environments using rights and obligations" [01] right(`#action`, `#condition`) denotes the right to execute action #action whenever condition `#condition` is true. The right is exercised by the agent that holds the right if the agent invokes the action `#action`. In the simulation, the right possessed by an agent, is essentially an instance of a corresponding domain class right. Item attribute "instance_of" contains the information regarding the actions and the conditions of the right. Each time an agent tries to invoke an action, the master agent checks, if it indeed has the right to do so.

4. Obligations. It is also a special case of the object. As per definition from the same source [01], obligation(`#satisfy`, `#violate`, `#sanction`) denotes the obligation of ensuring that condition `#satisfy` is satisfied no later than condition `#violate`, under the penalty of invoking action `#sanction`. Such sanctions are imposed by the master agent. Like the rights, the obligations held by an agent are the instances of the corresponding obligation class and their "`instance_of`" attribute denotes the details of this obligation.

5. Special objects. They are described in the same way like the other items, but in fact, they exhibit a special behavior and thus it makes sense to treat them separately. For English auction domain, such special objects would be the items of the "English auction" type.

As these entries significantly differ, it would make sense to equip the item with their types to distinguish easily between the items belonging to different types and to group them accordingly in visualization. Also, we naturally want to display the complete list of attributes for each object.


right(12)

Held by	Lynx (19)
Owned by	Lynx (19)
Instance of	right (open_auction (_448162,_448163,_448164), (object (_448163),value (_448163,owned_by,19)))

Figure 3.4. A right to open an auction with its attributes: "held_by", "owned_by" and "instance_of". Instance of contains the details of the right: the agent has the right to open the auction on the item _448163 if _448163 is an object, and if _448163 is owned by agent #19. The item frame is sporting the customizable icon, corresponding to the right to open an auction.

3.2.5 Key-Value Pair

A key-value pair is the most basic entity of the market state description structure. It is used in the definition of all environment entities and their embedded structures. In fact, the values of the higher-level objects in their turn can be key-value pairs or lists of key-value pairs. Let us examine the example of the description of such object provided on Figure 3.5.

```
(MyAuction, [(last_bid_time, 347.66588475182652), (highest_bidder, 23),
(highest_bid, 2), (closing_time, undefined), (opening_price, 1), (item, apple(25)),
(status, open), (auctioneer, 25), (held_by, 25), (owned_by, 25), (instance_of,
english_auction)])
```

Figure 3.5. Example of an auction object as it is contained in the log file. It can be seen that the structure on the higher level contains the key "MyAuction" and the value — list of attributes, each of which is also a key-value pair.

The figure above shows us the description of an object, as it is listed amongst the others, being an integral part of the environment state description. In the hierarchic tree of the enclosed key-value pair, every one of these has a unique key in relation to its siblings. We see that the described object has the key "MyAuction" and a value representing the list of attributes, each of which is a key-value pair with a unique key within this list.

Taking into the account the abovementioned, we have implemented a logical entity KVP in our application and have been using it extensively for processing and parsing the output.

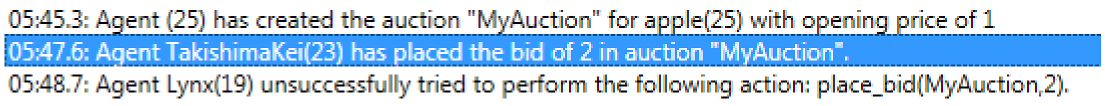
3.2.6 Event

An important part of the environment state description is the event object. Notwithstanding the fact that it is not mandatory, it plays an important role in the simulation — it is the only source of information about the actions performed by the agents during the simulation run, if we do not count re-engineering of the invoked action from the changes in the environment state. Therefore, if we want to keep an accurate track of the invoked action, implementing the event object can be seen as a reasonable thing to do. Each event description contains the following information: name of the actor, in other words, *who* performed the action, the action invoked — the "verb" of the sentence, and, the last, whether the action was invoked successfully. It also contains the time, when the action has happened and has expired. As per current implementation, these two are essentially the same. The figure 3.6 provides an example of such event.

```
(event(63), [(description, invoked(23, place_bid(MyAuction, 4), successfully)),
(expired_at, 352.49216085206717), (happened_at, 352.49216085206717), (instance_of,
event)])
```

Figure 3.6. Example of an event object. Event with the name "event(63)" describes that the agent #23 has successfully placed a bid of 4 in the auction "MyAuction" approximately 352.5 seconds after the beginning of the simulation run.

The event entity, as can be seen on the above picture, contains only the description with one invocation statement. From it we can see that the agent with id 23 have successfully placed the bid of 4 on auction "MyAuction". In our application, we reflect such event as a sentence in natural language in the action log. Figure 3.7 provides an example of such representation.



```
05:45.3: Agent (25) has created the auction "MyAuction" for apple(25) with opening price of 1
05:47.6: Agent TakishimaKei(23) has placed the bid of 2 in auction "MyAuction".
05:48.7: Agent Lynx(19) unsuccessfully tried to perform the following action: place_bid(MyAuction,2).
```

Figure 3.7. Visual representations of actions in user-friendly form. In addition to the agent number, the line contains a customizable name and the sentence is formulated in NLP form.

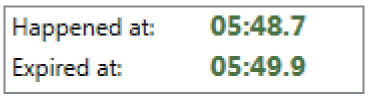
3.2.7 Clock

A market environment state contains one more small, but important entity — that is, the clock. The clock defines two time values: the time when the actions that caused the environment to assume this state happened and when this state has expired. The expiration of the state can be caused by two reasons: 1) an action was taken by some agent and thus the state has changed, and 2) the state has expired on a time-out. The time is provided in seconds that has passed from the beginning of the simulation run. We have implemented the clock entity, along with the methods for time conversion into a user-friendly format. Notwithstanding the fact that the clock is also an instance of event class, it is quite different in nature and function from the events we have discussed in the section 3.2.6 and thus we have chosen to treat and process it separately. The clock serves as a unique and the only time signature of the state and it is used for aligning the time and state numbers in the visualization application. Figure 3.8 provides an example of a clock description

```
(clock,  [(expired_at,  356.00036145187914),  (happened_at,  352.49216085206717),
(description, state_instantiation), (instance_of, event)])
```

Figure 3.8. A description of a clock object, encompassed in the environment state statement. It can be seen that the containing state has come into existence approximately 353.5 seconds after the beginning of a simulation run, and has expired at approximately 356 seconds.

In addition to the inner processing and mapping of a state to time, we also use the clock object to demonstrate to the user the time for the environment state he is observing. While the time values in the clock entity are given in seconds, we translate them into more understandable format. The visual representation of a clock can be observed on figure 3.9.



Happened at:	05:48.7
Expired at:	05:49.9

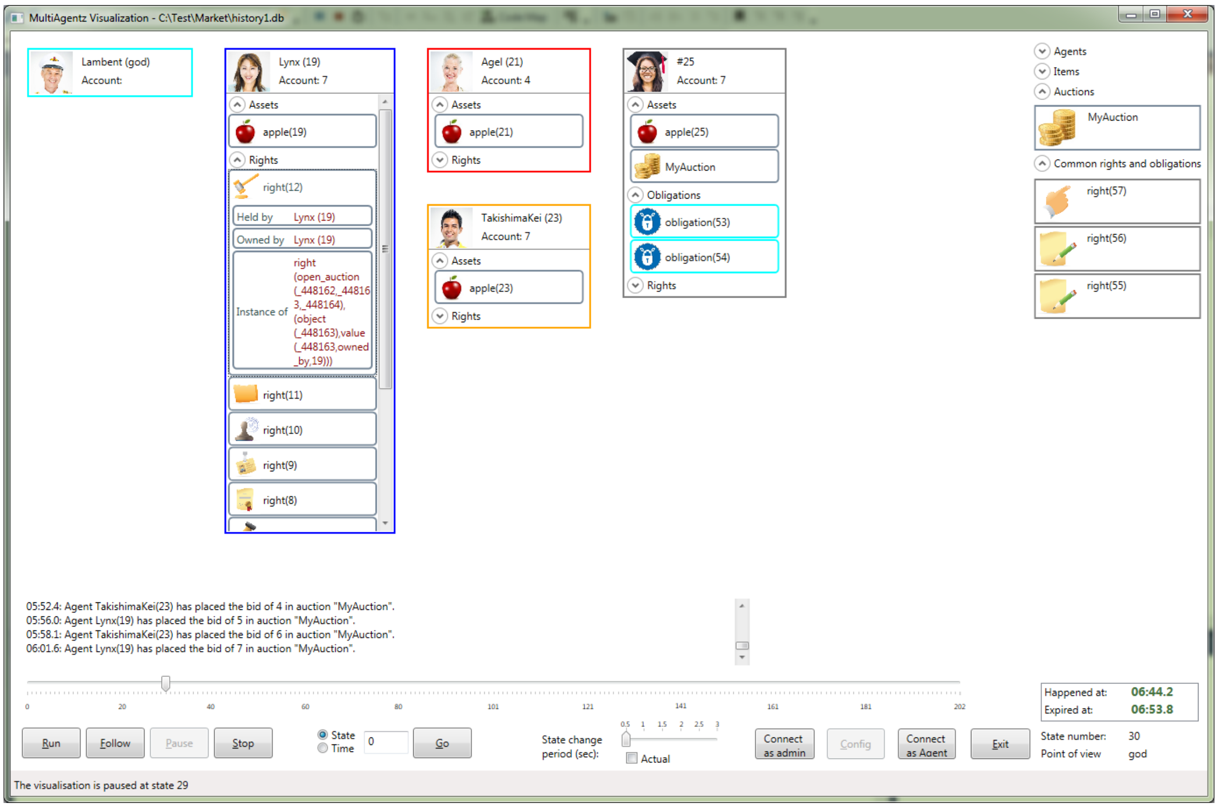
Figure 3.9. Visual representations of a clock. It shows two values, described above in a user-friendly format, in minutes and seconds.

3.3 Basic Functionality

In this section, we would like to give a description for the main functions of the visual application that we have seen as desirable and have chosen to implement. Some of them are fairy intuitive, while the others provide additional information in comparison with the original entries of the simulation output.

3.3.1 Demonstrating the Market State

First, the most important and obvious function that is targeted by our application is, of course, providing a user with the visual representation of the entries of the simulation log file — that is, of the market environment state. Everything contained in the environment state description can be observed in the application window in one or the other visual form. In the previous section, 3.2. "Logical Entities" we have discussed the content of the environment state and mentioned in passing the visual representation of its objects. All of those can be observed in the visualization. Let us examine the main window of the application, provided on the Figure 3.10.


MultiAgentz Visualization - C:\Test\Market\history1.db

Lambent (god)
Account:

Lynx (19)
Account: 7

Agel (21)
Account: 4

TakishimaKei (23)
Account: 7

#25
Account: 7

Held by Lynx (19)
Owned by Lynx (19)
Instance of right (open_auction (_448162,_448163,_448164), (object (_448163),value (_448163,owned_by,19)))

```
05:52.4: Agent TakishimaKei(23) has placed the bid of 4 in auction "MyAuction".
05:56.0: Agent Lynx(19) has placed the bid of 5 in auction "MyAuction".
05:58.1: Agent TakishimaKei(23) has placed the bid of 6 in auction "MyAuction".
06:01.6: Agent Lynx(19) has placed the bid of 7 in auction "MyAuction".
```

Happened at: 06:44.2
Expired at: 06:53.8
State number: 30
Point of view god

The visualisation is paused at state 29

Figure 3.10. Main window of the application. It contains both the visual objects, such as agents or items and navigation elements, letting the user to control the application.

The figure above demonstrates the representation of all objects that are present in the corresponding statement. All agents that are currently participating in the simulation are shown as separate entries in a form of agent frames where the header displays the agent id and account while the body lists all the items that are held by this agent, except for the special items, which are, however, shown in a special list. ("Auctions" on the figure). All attributes of an item can be observed in similar format, namely, with a header showing the name and a body listing the attributes. In order to make the visualization concise, we have designed all visual objects to be expandable / collapsible, so in a collapsed form only the object name or id is seen, and in in the expanded view the content of the object is demonstrated. The events of the market environment state are represented as the descriptive statements in the action log, below the agent area. The clock can be observed in the bottom right corner.

3.3.2 Reflecting the State Changes as They Happen

As we reproduce the run of the simulation or connect to the ongoing simulation file and reflect the states as they are produced in a visual form, we want to draw the user's attention to the changes that have happened since the previous state. In order to do so, we first compare two states and find the objects that have changed, and then change the visual representation of these changed objects to catch the user's eye.

Agents, basically, can change in one of the two ways: to join the simulation environment and to leave it. For the agent that has joined, we color its header light yellow and make its form slowly appear, coming into the existence. For the agent that has left, we dye its header dark grey and make the frame slowly disappear. Thus, their joining and leaving are represented not as the instant event, but in a continuous manner, giving the user a time to process this information. Figure 3.11 demonstrates semi-transparent agent frame upon its leaving the simulation environment.

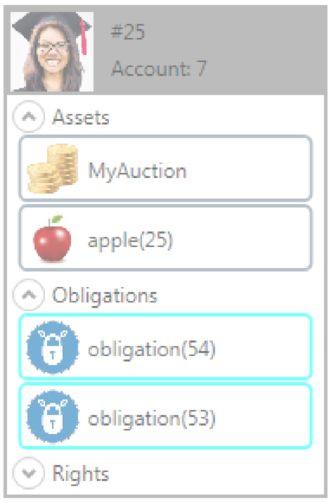


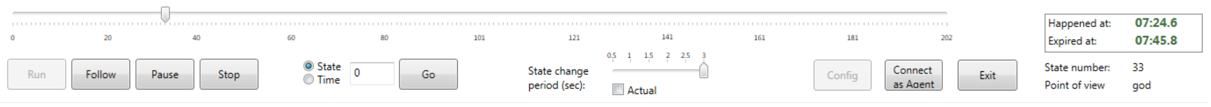
Figure 3.11. Demonstrating the state changes: a semi-transparent agent frame with grey header background.

Items, though, can change in more ways than that: in addition to appearing and disappearing, the values of their attributes can change. In a similar manner, we represent these changes by changing the background color of the item, dyeing them yellow and grey for new and deleted ones and blue for changed. The color-coding is covered in a greater detail in the section 4.2. Visualization Details, in the part 4.2.1. General Assumptions and Conventions. We also keep the marking color for the duration of one state (as of the current settings of the visualization application) to provide the continuity of the change visualization and thus greater visibility.

3.3.3 Reproducing the Simulation Run

However, there is a small meaning in representing a single state or even two states without a context. Those only interest us as a part of an entire simulation run. Thus, our program is designed for giving the user an opportunity to picture all the states in the simulation run log file one by one, reflecting the consecutive states and the changes from the previous one. In addition, we provide the information that covers the entire scope of a log file — that is, the list of all agents that have participated in simulation at some point of time, and the list of all items that were present in the market. To visualize this, we have equipped the main application window with a slider that represents the length of the log in the terms of number of states and reflects the current position of the visualization run on it. We also provide the state number for reference. In addition, we provide the time reference, showing the clock with the time that has passed from the moment of original simulation run to the point where it has reached the state that we are reproducing.

We also provide the possibility for a user to regulate the speed of state changes, allowing him to speed up or slow down the reproduction as he wishes. There is also a possibility to demonstrate the states as they have been produced, using the clock value "expired_at" to determine the duration of a current state. Figure 3.12 shows the panel with the control elements designed to support this functionality. The control elements are discussed in detail in the section 4.3. Control Elements



(● State / ○ Time) 0

State change period (sec): ☐ Actual

Happened at: 07:24.6
Expired at: 07:45.8
State number: 33
Point of view god

Figure 3.12. Control elements section in the bottom of the main window of the application. In contains the following elements (top to bottom, left to right): main slider, navigation buttons, navigation field for moving to a particular state or time with accompanying button, state change period regulation slider, application control buttons, clock, state number and point of view fields.

3.3.4 Reflecting an On-Going Simulation

We also provide the possibility for a user to observe an on-going simulation as it happens, reflecting the states as they are generated during the simulation run. This allows the user to see the changes on the fly, and manipulate the experiment conditions by invoking actions within the environment and observing the results. For online reproduction mode, we also give the possibility to stay a specific number of states behind the end of the file, which can be useful for testing some predictions or assumptions in the interactive manner. As during the offline mode, the slider shows the current number of states and the position of the current state on it. The slider marks are rewritten each time the number of states is updated, so the information is up-to-date.

3.3.5 Navigating the Market States

One of the intuitive requirements for the visualization application is for a user to be able to navigate to the specific market state without needing to go through all of the previous ones, as he may be willing to observe only a specific part of the simulation run or replay it for a few times to get a better understanding. We provide several ways for navigation through the market states.

The first of those is to move to the particular state number explicitly. It can be handy if the user knows the state number, which he wants to observe. However, if he knows the time, he can move to the state this time falls within. Another way is to drag a slider to the desired state. By clicking on the slider to the left or to the right of the pointer one can move one state back or forward, accordingly.

However, there is another navigation possibility. The Agents Pane contains the list of all agents that have participated in the given simulation run. They are equipped with the numbers of the states during which they have joined and left the simulation and buttons to navigate to these states directly. Figure 3.13 demonstrates the Agent Pane frame and interface elements that provide the mentioned functionality.

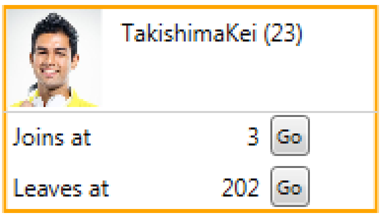


Figure 3.13. Expanded Agent Pane frame showing the navigation buttons. One can see that this agent has joined the simulation at state 3 and has left at state 202.

3.3.6 Customizable Look

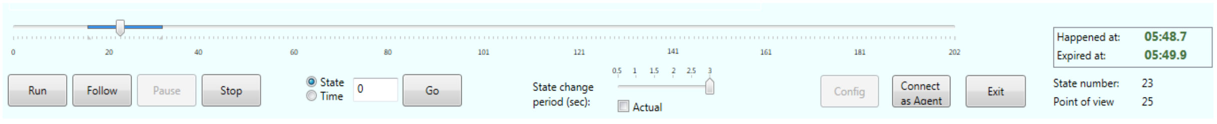
The visualization application provides the possibility to customize the icons, which are used for depicting different item classes and instances and the names used for identifying the agents. These names and icons are purely arbitrary and are not bound in any way to the class or agent id. This customization is performed by providing the visualization customization file in a specific format. This file is examined closely in the section 4.1.2 Input Format. This is done to improve the visual and textual recognition of agents and items, as it is typically easier to distinguish between names than numbers and between images than plain text. A user also has a possibility to select what classes will be considered special items for the purposes of a given visualization and provide the name for it.

3.3.7 Global Lists

We have implemented the possibility to observe at any arbitrary time a list of all items that have appeared during the simulation run, as well as a list of all the agents that participated in it. This way, when we need information about one of the items or agents, we do not have to thumb through all the states, trying to find it. Instead, we can look it up in the Reference Panel. The Agent Pane, comprising a part of the Reference Panel, also provides the possibility to navigate directly to the state where an agent has joined or left the simulation (described above).

3.3.8 Point of View

While by default the visualization application takes as an input a log file, produced by the master agent (as it is built-up in the simulation environment, essentially it means that the log file is produced by the simulation engine itself), we have implemented the possibility to pick up the log file, generated by a specific agent provided it has the name complying with our requirements, and demonstrate the market environment states from the point of view of this particular agent. All the states where at least one agent was present besides the master agent can be observed from a different point of view, if the corresponding log file is available. Upon changing the point of view, the visualization application changes the background color from white to one matching in hue the color selected for the chosen agent, and the main slider explicitly shows the period covered by the log file of this agent. The "Point of View" field explicitly indicates the id of the agent, from which point of view the simulation is demonstrated. Figure 3.14 demonstrates the panel with a colored background, marked slider and the point of view field.



Happened at: 05:48.7
Expired at: 05:49.9
State number: 23
Point of view 25

Figure 3.14. The bottom section of the main window of the application. The "Point of View" field tells that the visualization currently demonstrates the environment from the point of view of the agent #25. The slider is marked to depict the states when this agent was present in the simulation, and the lightest cyan background matches the cyan border of an agent (not shown here).

3.3.9 Connecting as an Administrator

The application provides the possibility to be run instead of the administration module of the original simulation system (covered in detail in the Section 2.3. Operation of the System). In this case, it should be run after the communication module and have the appropriately filled configuration file (described in the Section 4.1.1. Input of the Visualization Module). If these conditions are fulfilled, the user will have the visualization application providing the interface for the administration access to the system, essentially – the administration console window. This means that the administrator can send commands to the simulation itself through the communication module.

3.3.10 Connecting as an Agent

During the online run mode, there is also a possibility to connect to the simulation as an agent. This will allow a user to register in the simulation environment and participate in market relations. Through the Agent Console, he can send commands in the form of DAMMAGE language statements to the simulation and observe the results on the fly. In order for this to be possible, the simulation should be running and the user should have had run the proxy window for the agent connection.

Chapter 4
Design and Development

In this chapter, we follow the steps that we took in order to design the proposed solution and describe its development, including its stages and employed technologies. We also examine the details of the implementation, reviewing the assumptions and decisions made in the development process, and the constituent parts of the solution.

4.1 Implementation

This section is dedicated to the details of the development, examination of the processing flow, output of the existing system and its use in the proposed solution, data formats and generation of the output and intermediate entities.

4.1.1 Input of the Visualization Module

The actual input of the visualization module consists of two big parts: the configuration file and the actual files used as a source of information. In order for the visualization to be easily tunable regarding the file locations, and for the convenience of introducing the changes into the program settings, we provide these details within a small configuration file along with the possibility to change the configuration of the system from the application itself. An example of the configuration file can be observed on Figure 4.1. Let us examine its parts.

```
<?xml version="1.0" encoding="UTF-8"?>
      <!DOCTYPE MultiAgentz[
      <!ELEMENT MultiAgentz ANY>
      <!ELEMENT HistoryFileDetails ANY>
      <!ELEMENT Path ANY>
      <!ELEMENT Customization ANY>
      <!ATTLIST Path id ID #REQUIRED>
      <!ELEMENT Path ANY>
      <!ATTLIST Access id ID #REQUIRED>
]>
<MultiAgentz xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <HistoryFileDetails>
            <Path id="HistPath">D:\ DMG\dammage 1.0\history1.db</Path>
      </HistoryFileDetails>
      <Customization>
            <Path id="DmgPath">D:\DMG\dammage 1.0\domains\english.vs</Path>
            <Access id="AdminRights">False</Access>
      </Customization>
</MultiAgentz>
```

Figure 4.1. Configuration file example. It is provided in XML format and supplies the information of the paths to the simulation history file and visualization customization file. It also has a setting indicating whether the application should be run in the administrator mode.

The files used as a source of information are naturally divided into two parts: the history of simulation run and the visualization customization definition. We can see that the configuration file for the sake of readability reflects the division into these two parts.

Simulation Log File

First of all, we need the information about what we will visualize. The first part of the configuration file, "HistoryFileDetails" contains one element — the path to the log file. In the future, we would like to provide more options for the information source, such as connecting to socket and reading the market information in real time. However, this would require introducing the changes into the original system, but should such be implemented, we would add the details to this section. We describe these possibilities in the "Future Work" Chapter. We will not give here the example of such log file due to its length, but we do give the example of one statement from such log in Appendix A. Please, refer to it in order to get the impression of how the actual log looks like, bearing in mind that it contains one such line for every environment state.

Visualization Customization File

The second part of the configuration file, namely, "Customization", contains two elements: a path to customization file and an indicator of whether the application should be run replacing the administration module of the system and with the appropriate rights. Let us have a look at the form and purpose of the customization file. We provide the example of actual file in the Appendix B hereof so the reader can have an idea of how it looks like. The purpose of having a customization file is to have a set of images that will be used for the agents and items of the application, as well as names for the agents. The customization file is in essence domain-specific, as while the concept of agent is the same for all the domains, the special items differ. For example, in the "Coin" domain that allows the participants to flip a coin, the coin is a special item and should be treated in a special way, and also a right to flip a coin is unique for this domain. In "Sealed Bid" domain, such special item will be represented by a "sealed_bid" object and the rights, naturally associated with this domain will be particular for this domain.

Start-Up of the Visualization Application

Upon the start of the visualization module, the application reads from its history a path to the configuration file and tries to process its content, processing and loading information contained in the history file and visualization customization file to the memory. In case of failure to do so, application prompts the user for a path to a valid configuration file. It also reads the value of the "AdminRights" section and if it is set to "true", prompts the user for an administrator password and tries to connect to the simulation module at the pre-set host and port. In the future, these can be made available as settings of the same configuration file. We examine this possibility in detail in the "Future Work" Chapter.

As the simulation run is performed for a certain domain and the visualization file is domain-specific, user can have several configuration files. The current configuration can be set up in the configuration window, where user can observe the current settings as they are set in the configuration file. If the user selects to run the visualization, these settings will be used for its execution.

The changes, should those be needed, can be introduced into the configuration file manually, although we have provided the possibility to change the configuration file via a user interface from within the visualization application. The setting to run the program instead of the administration module cannot be changed from the user interface and has to be changed manually. It is done to prevent the accidental change of this setting.

4.1.2 Input Format

In the above paragraph, we have examined the input of the visualization module from the facet of what parts it comprises and what information is available in each part, however, we have not discussed the format of its representation. Let us do it here.

Configuration File

In order to keep in line with the current industry standards we have implemented a configuration file compatible with XML standards [19]. This approach has several advantages:

1. Intended users of a system, who in most cases will have at least some computer science background, due to the fact that they are willing to examine a multi-agent system, would likely be familiar with XML standard to at least some degree and will be able to read the configuration file without a need to study the technology first.

2. Standard libraries for processing XML files are included in all major IDEs, including Visual Studio .NET that we have used for developing this solution, thus we can use the existing functionality for its processing.

3. By using the abovementioned standard libraries, a user can easily create or modify the existing configuration file should he wish it.

Simulation Log File

However, the output of the simulation engine does not exactly comply with the paradigm described in the Section 3.2. "Logical Entities" of Chapter 3. "Problem Analysis and the Proposed Solution". This is only expected, as they serve two different purposes. The output of the engine has to capture the current market state in the form understandable by the Prolog engine and designed so that the run of the program can be reproduced from any arbitrary point, while the visualization module focuses on the concise and understandable representation of a given state from the point of view of the preceding one, in order to provide the historical aspect.

The simulation output consists of separate lines, produced one by one during the course of a simulation run, each one describing the next state of the simulation environment. A state is formally defined as a statement state(#objects), which denotes a state, where #objects is a list of objects, in the form, determined in the domain classes. For example, an apple object is defined as (#name, [(instance_of, apple), (owned_by, #owner), (held_by, #possessor), (weight,#weight)]).

The format of the actual line structurally is a multi-branching tree: the line itself is a node element with the key equal to "state" and the value equal to a list of nested nodes, representing the elements encompassed within this state. These nested elements are a clock, items currently being present on the market and a non-mandatory event element, describing the event that happened during that state, such as an action taken by an agent, which has served as a state change trigger. Each of these nodes, in its turn, contains the children elements of its own. These elements can be either nodes themselves, or key-value pairs, or lists of key-value pairs, representing the properties of the encompassing item. "held_by" may serve as an example of such property. Its value describes the id of the agent in whose possession the item is to be found at this given market state.

```
(apple(23), [(owned_by, 23), (held_by, 23), (weight, 6), (instance_of, apple)])
```

Figure 4.2. An example of a subtree depicting the item apple(23) and its attributes: owned_by and held_by, containing the ids of the agent, owning and holding this item, respectively, weight, and instance_of, indicating the class this item represents.

We have no indications of agents currently present on the market other than ids implicitly present as holders or owners of the items listed in the state. Therefore, in order to acquire the list of all agents we have to turn this tree inside out, turning the former leaves into the nodes and vice versa.

Visualization Customization File

The visualization customization file is also made in the form of logical statements for compatibility with the format of another input file. An example of such file can be observed in Appendix B. It is also formed on the basis of key-value pairs and contains two statements: visualize(#agents, #items) and special_items(#group_name, #members). The #agents here is a list of agent ids coupled with the name and path to the icon to be used in the visualization. The ids are assigned by the simulation engine and thus the name and icon serves only for discriminating between the agents within the scope of a current run, as it is much easier for a human observer to distinguish between pictures accompanied by names than between the numbers. There can also be defined a default name and icon, by using the keyword "default". The example of it provided on the Figure 4.3.

```
(default, [(image,"..\visualization\agent_images\default_agent.jpg"), (name, "An
Agent")])
```

Figure 4.3. A definition of the default agent. As a key, it uses a keyword "default" and contains two values: the icon to use as a default agent icon and a name to use as a default agent name.

The #items in the visualize() statement represent the list of the item instances, essentially, the instances of domain classes. For each of them a separate image can be defined. There names of the classes are matched to the beginning of the names in the statement, and the best match (the longest one) will be used to illustrate the corresponding object. Like with the agents, one can define the default item image, using the keyword "default". An example of the item definition is provided on the Figure 4.4. Please, note that the domain class is matched to the customization key by the start of the word, so in the example, shown on the figure, all items with the instance starting with "right(query" will get the corresponding icon, unless there is defined a longer match for them in customization file.

```
("right(query", "..\visualization\item_images\query.png")
```

Figure 4.4. An example of the item definition. Items belonging to the domain class "right", to the instance representing a right to query about something will be depicted with a provided icon.

In case of failure to acquire the specific image from the indicated file, an agent or an item will be depicted by using a defined default image, and in the case when it is not available either, with the default image provided by the application itself. Of course, due to the impossibility of guessing what kinds of items shall be in use in an arbitrary simulation, we would supply only a default icon in the general case. However, due to the nature of the simulation, we can almost safely bet that there would be instances of two entities: a right and an obligation. Thus, it makes perfect sense to have default icons for those as well. Considering this, by default, we shall provide three default icons: an item, a right, and an obligation. They shall be stored within the application folder under /Icons and can be manually replaced, should the user so desire.

The special_items() statement defines what items play a special role in the visualization. For those two properties are defined: group_name, used for a caption in the corresponding visualization area, and #members, indicating the list of the domain classes that are to be treated in a special manner. For the "English auction" and "Sealed bid" domains we define the group name as "Auctions", while for the "Coin" domain we consider "Coin" to be a suitable name.

```
special_items((group_name, "Auctions"), (members, ["english_auction"])).
```

Figure 4.5. An example of the special_items() statement. It contains two elements: the group name, which will indicate the name of the header of the special items, and the members — the names of the classes that will be considered special items.

4.1.3 Input Processing

The very first thing that the module has to do is to decipher the configuration file and load up the files. The configuration file is processed using standard XML libraries, and the input files are loaded from the indicated paths. The actual input processing starts with the log file, as to be able to visualize anything at all, the first we need is a source of information. The engine can run without a customization file, using the application-level default images, however, the visibility will be significantly impaired.

Simulation Log File

Thus, the first step is processing the simulation engine output. For each statement, we are creating the state object that will contain all the information essential for the proper visualization of the state. We start with the first statement. It will serve as a starting point for the visualization. In order to do the initial mapping, we perform the depth-first parsing of a tree, collecting all references to agent ids and corresponding items. For each state, we will keep the list of participant agents and their items, the clock values, and the state number. We also keep a separate list of special items and common rights and obligations — the ones not belonging to any particular agent, but applicable to any agent of the market environment.

Each time we encounter a previously unseen agent id, we add it to the list. If the agent id was discovered as the value of held_by attribute, we would also add the item to the agent's list of possessed items. If this was not the case and the agent id was found as a value of owned_by attribute, it would merely serve as a reference. Upon the completion of the parsing we shall have the complete list of agents currently present on the market and for each of those — the list of items in their possession.

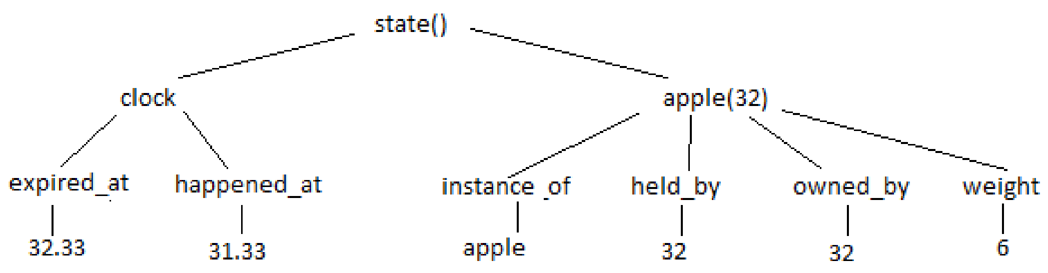


Figure 4.6. An excerpt from the state tree, presenting a state with two objects, clock and apple(32), their attributes and values of those.

Also, for each agent id we encounter, we check, if there is an agent log present. As per convention, the name of the agent log file is a combination of master log and an agent id. For example, if master log name is "history_db", the log of agent #25 will have the name "history_25.db". These logs are used to provide the point of view of an agent upon request, if such log is available. The information regarding agent log availability is stored in the memory.

At the same time, parallel to collecting of agent ids, we shall parse the item records themselves. The item information is essentially a subtree of a state tree, and is examined in the same travel over it. For each item, the corresponding object will be created in the application. All nested elements of the item will be examined and mapped to the appropriate fields. We intend to treat all nested elements of a simple structure, namely constituted of a key and a value, as descriptive attributes, providing alternative, more user-friendly description of an attribute. For example, the key of the owned_by attribute is replaced by "Owned by", which comply with English better. However, due to the special nature of possession and ownership attributes, we will treat them in manner specific to their nature.

```
Item
{
    Dictionary<String, String> StringAttributeList =
        {{“Weight”, “6”}};
    String Owned_by = “23”;
    String Held_by = “23”;
    String Key = “apple(25)”;
    String InstanceOf = “apple”;
    ...
    ItemType Type = ItemType.Asset;
    ElementStatus Status = ElementStatus.Unchanged;
}
```

Figure 4.7. An example of a mapped object. It can be seen that mandatory attributes “held_by” and “owned_by”, as well as item key and instance are mapped to the predefined fields, while the rest of the attributes are stored as a list containing name-value pairs. Additional fields like item type and status are also present.

One can see that we have introduced a few other attributes not present explicitly in the original description. One of such is the item type, which we intend to use to better visualize the items. The available variants of the item type is asset, right, obligation and attribute, of which the latter is not the real item in the sense of the application but the element of the description. Another useful field is the status, which we use in determining whether the element has changed since the previous step, and how exactly.

For each item, we also check, whether it is a special type of an item. In that case, we shall not only add in to the collection of agent items, but also, keeping the references of the owner, store them in separate list of special items. For the English auction domain, we would store in such list all objects of english_auction class.

As we examine the states one-by-one, we also do such thing as population of the simulation run scope-wide lists. Those are the lists of all agents that participated in the simulation during the period covered in log and all items that were at any time present in the market environment. For each agent we store such information as the state number, when it has entered the market environment and the state number where it has left. It is done in order to implement the possibility of easy navigation to such state.

Visualization Customization File

Essentially, the format of the visualization customization file is the same tree-like hierarchic structure, bases on key-value pairs. During its processing, first we parse the visualize() statement and acquire listed agent ids along with the location of their image path and name, and list of item classes with the path to their image files. We keep a special data dictionary in the application, where we add the information regarding the image path for each agent and item we encounter. We also store the values of the attributes "group_name" and "members" of special_item() statement and use those for visualization and object type determination.

4.1.4 Classes

The main building blocks of a module, written in object-oriented language, are classes. Let us define the main classes that we need to represent the states of the environment. Those are logically divided in separate groups: visual elements that will be represented in graphical form, core classes that are serving for processing the input into proper form and support the provided functionality, ancillary elements that support the operation of the module and enumerations, basic types for encoding certain number of specific values for a given property. The class diagram can be observed on Figure 4.8. Let us examine them one-by-one.

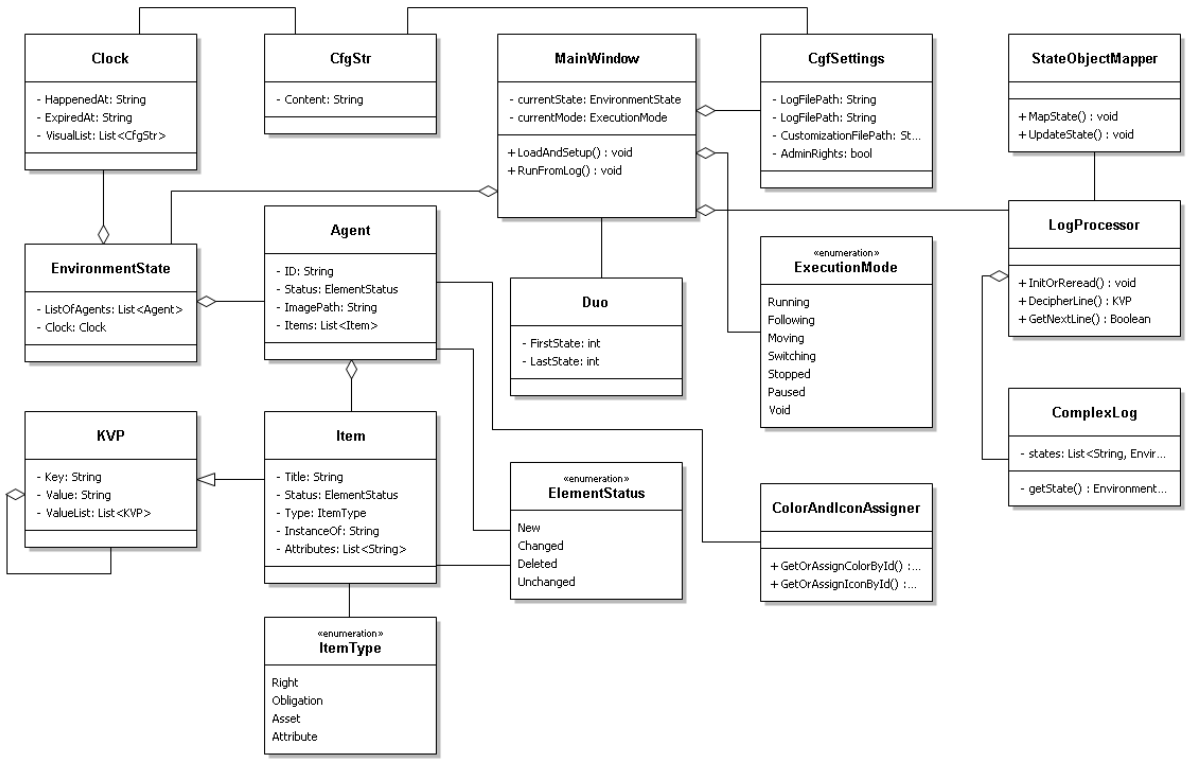


Figure 4.8. Class diagram of Visualization Module. It contains all the classes of the application and relations between them. We examine all classes one-by-one below.

Visual Elements

Those are the first classes that come to mind when talking about visualization application. The most obvious candidates for implementing them are logical entities, the first of which being environment state itself, as it is what we are essentially trying to visualize. The next two obvious candidates that we will have to represent in any case are agents and items, as the state—in our paradigm—contains the agents, which, in their turn, contain the items. We have also chosen to implement a clock and event as separate classes, and those are contained in the market state along with the list of agents.

Environment State class is represented as a number of visual components on the main application window, as described in Section 4.2.4 Market State and Related Entities. Thus in order to be able to reflect all the constituent elements pertaining to the particular market state, we shall keep them all together. Thus, Environment State is basically an aggregation of the list of agents, special items, event and clock.

As the important characteristics of an agent are its id and account, *Agent* class contains the corresponding fields. Naturally, it also contains the list of items belonging to this agent. Except for those fields naturally associated with the agent, it also contains such properties as First Step and Last Step, which represent the number of the state from log file, when an agent has joined the simulation environment and when it has left. These properties are populated by processing entire log file and comparing the lists of agents in each state. For visualization purposes Agent class also has such properties as title, border brush, which is set to a distinct color for better recognition with the help of Color And Icon Assigner class, that we will examine below, in paragraph "Core Classes". An important field of the class is Status field, which is set to New, Unchanged or Deleted during the environment state update in the course of the visualization run. The Source property serves for storing the agent icon.

The items of the simulation are also stored in a separate *Item* class. It has the key, serving as its unique identifier and properties "held by" and "owned by", which we discussed in Section 3.2.4. "Item" and Subsection 4.1.2.1. Simulation Log File of Section 4.1.2. Input Format. Another field that is inherently present in the original log file record is Instance Of, which tells us what class is the specific item an instance of. As any item can have an arbitrary set of descriptive attributes, we store them in String Attribute List field, which stores these attributes in (name, value) format. The InstanceOf property is used for determining, what image will be used for the representation of this particular item, as in our application the images are supplied per item class, or, in case of rights, per specific right type. Item class also has the field Type, which can be asset, right and obligation and is used for grouping Items per type within the agent frame, as described in Section 4.2.3 Agent Frame and Its Contents, in paragraph 4.2.3.1 Item grouping. The Item type is determined during the log file processing. Item class exposes the method `KVPToItem()` for converting an arbitrary hierarchic key-value pair from initial log statement (see subsection 3.2.5 Key Value Pair from Section 3.2 Logical entities) into Item class. This method is extensively used during the initial processing of log file statements.

The special item entity is implemented as an instance of the same Item class, as the required details do not differ.

Clock class is relatively simple and is used for storing values of time, when the state became effective and have expired. However, in addition to the values in seconds that are present in the original statement, we also store the value in conventional time format for representation in user-friendly way. We also keep the number of the state in the clock, something that is not present in the original statements and is populated during the input processing. *Clock* class is also exposing the `ToTime()` method for converting floating point seconds to user-friendly time in hours, minutes and seconds.

Event class, dubbed *System Event* due to the naming restrictions in .NET, is represented as the message in the action log on the main application window. It has only a timestamp and message properties, but we chose to implement it as a separate class, as the original information about action comes in the form of logical statement and to reflect it in the form of natural language statement it required the parsing logic, which we have encapsulated in the class.

Another visual class is *Configuration Window*. It is used to provide the user with a friendly way to load, change or save the current application configuration and exposes the corresponding functionality. To store the configuration settings and pass them to the main application, it uses the instance of *ConfigurationSettings* class as a container.

And, finally, the framing class, which manages the coordination of User Interface flow and processing flow, is *Main Window* class. It both serves as a whiteboard for graphic elements and environment states and links the controls to the underlying logic. Main Window stores the objects that are forming the current state of the environment, such as list of agents, as well as their visual representation, and such control fields as current execution mode, current configuration and so forth.

Core Elements

However, in order implement the required functionality, and as part of it, to read the actual entry from file, parse it and map to KVP object, and in general to guide and manage the execution, the visual element objects are not enough. For processing, parsing, and so forth we shall need another classes — the ones holding the corresponding functions.

The first one of such is *LogProcessor* class. It has two main functions: processing the simulation log file and processing the visualization customization file. Upon the start of the application, *LogProcessor* should be initialized according to the current configuration. It has the method `InitOrReread()` which handles the initialization. It reads the log file, processes it and with the help of *StateObjectMapper* class creates the dictionary of all states of the current log file. It has the property Index that denotes the number of state the application is currently reflecting and Count, storing the number of states. It has also methods for browsing this dictionary by time, number of the state or consecutively. For example, `GetNextLine()` method returns the next state and increments the Index. What relates to the visualization customization file, the LogProcessor parses the file, indicated in the current configuration, and creates the dictionary in the form of *AgentAndItemDataDictionary* class, which stores and indexes all the visualization information provided in the customization file in the form of indexed list and exposes the methods for retrieving icon source by agent id or item key and for getting agent name by its id., namely, `GetItemSourceByID()` and the like.

Another important functional class is *StateObjectMapper*, whose primary responsibility is mapping the initial state tree in form of KVP entry discussed earlier to the entities of our application — namely, market state, agents and so on.. *StateObjectMapper* has a range of methods for initial parsing and updating objects of different type. As a rule, these methods receive as an input two consecutive environment state descriptions and produce the updated state. The example of these methods is `UpdateAgentItems()`, which receives as an input two instances of Agent class, examines the items of both, and updates the list of items, governing by the difference of two lists. The status on each of the items is updated accordingly.

For visualization purposes, not as of the moment related to customization, as we will describe in Subsection 4.2.1 General Assumptions and Conventions of Section 4.2 Visualization Details, we randomly assign each agent its own distinctive color. Also, when switching the point of view of the application to other agent than master agent, we shade the application window in a light version of agent color in order to facilitate recognition and intuitive understanding of that fact. The class *ColorAssigner* deals with distribution of these distinctive colors and their lighter version used for shading. It has a built-it set of distinctive colors and their lighter version, and assigns those through the exposed method `GetOrAssignColorById`(). On the first call for the particular agent id this method assigns the color and adds agent id and its color to the special dictionary, as well as marks the color as used. On consecutive calls, it returns the color from the dictionary.

Ancillary Elements

However, to support the seamless operation of the core and visual object classes we also need some "helper", ancillary classes, acting as intermediaries between relatively low-level processing and visualization itself. The first such is the "workhorse" of the application — key-value pair.

KVP class does not have much inside it, but most of the parsing is done by isolating key-value pairs and then processing them further in the manner specific to the type of the object, like, clock or event. It has three fields: key, value and list of key-value pairs, at that only one of the two can be populated at any given type. Either the element has the value or it contains the list of embedded key-value pairs. Visual class constructors, for example, `Item()`, take *KVP* as an input and construct the class with the values elicited from it.

ComplexLog class is used for storing the environment states indexed by agent id. The purpose of it is synchronizing the points of view of several agents that participate in the simulation. For each state number it has a collection of records in a format (agent id, environment state) for each agent that has the simulation log file of its own.

Class *Duo* is the littlest class of the application. It is used to store the numbers of agent first and last state and has only two according fields. The EnvironmentState keeps a list of all agents steps in format (agent id, duo) for the purposes of managing Point of View functionality.

ConfigSettings class is used for storing current configuration details and passing it as an input to the *LogProcessor* or *AgentAndItemDataDictionary* instances, which, in their turn, can load up the log or simulation file details from it. It has the following fields: Path To Log File, Path To Customization File, and Administrator Rights. Its exposed methods are `SaveToFile()` and `LoadFromFile()` with obvious functionality. Ancillary class *CfgStr* is implemented for the latter, with a purpose of raising a notification (and, accordingly, updating the information displayed by the application) each time the value of one or the other fields changes. Along with the other visual elements, configuration settings are defined in its own template within the main window.

Enumerations

In our visualization module to keep the values of such fields as element status in line with one another, we have implemented several enumerations.

The first of them is abovementioned *ElementStatus*, which is used for choosing a background color of an agent or an item. It has the following values: New, Changed, Deleted and Unchanged, referring to the corresponding states of the objects.

Second is *ElementType*, used for logical grouping of the items within Agent Frame. It has the values Asset, Right, Obligation and Attribute.

In implementing the navigation elements and basic functionality we encountered the need to keep a track of the current execution mode or state, for example, to differentiate between running and following modes. Thus we have the enumeration *ExecutionState* with values Running, Following, Switching, Moving, Paused, Stopped and Void, used to describe the associated states of the application.

4.1.5 Output Generation

The general flow of output generation consists of the several stages, and begins with processing a configuration file. Upon the start of an application, the current configuration file is read and parsed with the help of ConfigSettings class. Then the visualization log is parsed by LogProcessor class. Each line of the file is parsed by LogProcessor and mapped to a state by StateObjectMapper class. Every such state contains a list of all agents with items held by them, current action, clock and special items. After all log file have been processed, the lists of all agents and of all items are filled up and added to the states. The states are stored in a state dictionary of ComplexLog class under the agent id of master agent. If during this mapping other agents are encountered, the LogProcessor checks for log file, belonging to these agents. If there are such logs, they are stored in the same state dictionary under a relevant id. The visualization customization file is also read and parsed, and the information contained in it is loaded into AgentAndItemDataDictionary to be used as a reference. This process is repeated each time a new configuration file is loaded into application.

After all the information has been parsed, sorted and mapped, the application waits for user to start the simulation by pressing one of the two buttons: Run or Follow. These only affect the flow in the conditions of when the next state will be displayed. In case of Run we start consequently displaying all states in the log file, paying no attention whether the states have been added to it or not. With Follow we keeping a certain distance from the end of the file and shall not start processing the next state if this distance has not changed, in other words, if the states has not been added to the file.

In any case, we start with the first state in the file. With the Follow we also set the distance to the current number of states between the first line and the end of the file.

We map the first state and bind its components, such as list of agents, to the visual elements of the main window. By setting this binding and providing a set of templates in WPF / XAML, which indicate how the particular element should be displayed, we achieve that all the bound elements will be pictured in pre-defined way and the change of the element will be reflected in a given manner. The agents, reference lists, special items list, clock in the corner of an application, the state number and point of view are all bound to keep in-line with the current state. The customization elements are referenced in the objects themselves, by exposing a Property that calls an appropriate method. For example, Agent class exposes a property *Source* denoting the bitmap image of agent icon. This property is implemented as a call to AgentAndItemDataDictionary class asking to get agent icon by its id. The dictionary, in its turn, checks, whether the customization file contained the valid file path for this particular agent id, and if yes, reads it and returns in a form of a bitmap. If not, it checks, if customization file has provided a default agent icon and returns it, if such exists. If not, it returns the default image provided by the application itself.

If during the state display time user has pressed a Pause button, the execution is paused and the next state will not be displayed before user will not continue the run. If the user has switched the point of view of an agent, the appropriate state of its log is mapped in the manner described above and execution continues from there.

After the state has been displayed, we wait a certain amount of time before displaying a next one. In case of Follow mode we wait until the next state will become available, in case of Run mode, we are governed by the State Change Period, discussed in Subsection 4.3.2. Navigation Control Group of Section 4.3. Control Elements.

When we are due to display the next state, we acquire it from the LogProcessor and start updating the state mapped to the visual elements in order to reflect all changes that have taken place since the last state. We go over the lists of agents, items and special items, comparing their elements. If the agent was present in the previous step and is not in the current one, it is marked as deleted; if the situation is opposite, it is marked as new. For each agent we compare the lists of its items in two steps in the same manner. In addition, however, for each item, we compare all of their attributes, and should any be different, the item is marked as changed. These statuses will be reflected visually with background colors. A for the lists of all agents and of all items, we go over those and compare them with the lists of agents and items currently present in the simulation state. Those not present are marked in grey.

With each state change, we also move the main slider of an application in to reflect the current position in the log file, and, if the number of states in the file has been changed, update the values of the slider itself.

This process is repeated for each next message. In case of log file, upon reaching its end, the application does not stop, but instead, keeps re-reading the file to see whether the new lines have been added. This provides the possibility to run the application from the log file of the running application, and get the visualization virtually as soon as the lines are added to the log, with a latency of a few milliseconds.

The general application flow chart can be observed in Appendix F.

4.2 Visualization Details

The idea of separating the actual visual representation from the basic classes seems feasible, as we may choose to reflect only an arbitrary part of the actual environment and to manage what to represent and how independently of class content management. Thus, we shall need a representation class for each of the aforementioned classes. We chose to implement the market state as separate window, while the visual agent of an agent is essentially a template. Similarly, the visual representation of the clock is a template and is located within the market state alongside the list of agents. Item has both its own window and a template, which are used depending on whether the item is observed as a part of agent item list or separately. We have also implemented the Agent Pane, listing all the agents that we present at one or the other point during the simulation run, and

Item Pane, listing all such items. There are also Special Items pane, and Common Rights and Obligations Pane, listing the appropriate items present in the current state. The visualization of those happens with the help of control templates. We shall discuss the details in the rest of the section.

4.2.1 General Assumptions and Conventions

As we mentioned in the Input Processing section, we keep the agent and item data dictionary, populated with the information obtained from visualization customization file. Thus, for every agent we check, if such dictionary contains an entry for its id, and if yes, assign to it the appropriate image and indicated name. Those images and names are not specific to particular human or software agents, but instead, they are mapped to the simulation-issued agent id and serve only as a visual clue for user to differentiate between those. The same process takes place for each item that appears in the simulation. As of the moment, the image can only be assigned to the class, and not to the specific object, but such functionality can be added in the future. We examine it in the Future Work Chapter.

The color-coding is the important tool, heavily employed by the visualization application. We color-code two main things: ownership of the item and status of an object.

For color-coding of the item ownership we developed a special class, described in details in “Classes” Section, that assigns to each agent a distinct color, selected from the list of easily differentiated colors, and this color is used for a border of the agent frame to serve as a reference, as it is used as a visual key for marking items owned by this entity but possessed by the other agents. In agent frame, we list the items, possessed by this agent. If such item is owned by the same agent, the item frame has light grey border, in order to avoid visual noise, and if not — it is pictured with the border in the appropriate agent color. Figure 4.9 shows the example of such color-coding.

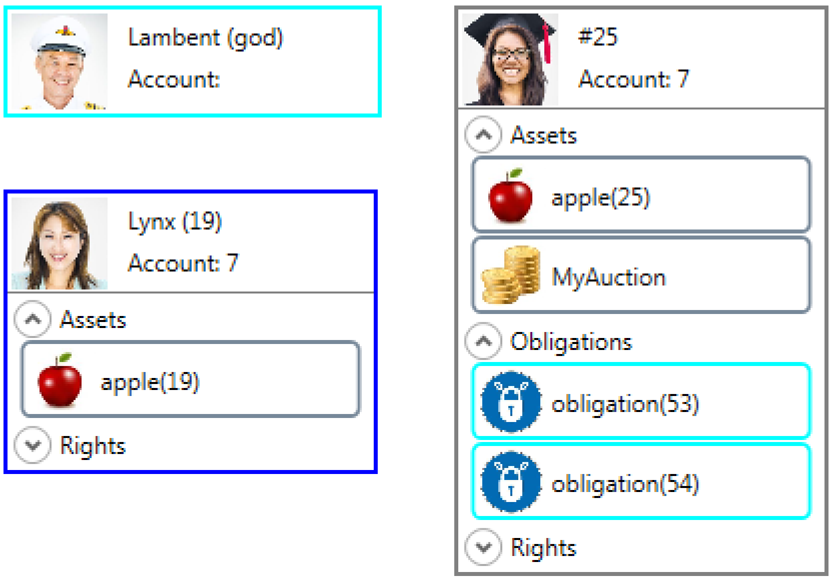


Figure 4.9. Color-coding for item ownership. It can be seen the Master agent (god) has cyan color assigned to it, and, accordingly, that agent #25 holds two obligations, obligation(54) and obligation(53) with cyan border, which means they are owned by a master agent.

On figure 4.9. the observer can see three agents, each of which has its own color used as a border of their frame: master agent (god) has cyan border, agent #19 has orchid border, and agent #25 has red one. Agent #19 holds the `apple(19)`, which is pictured with light grey border, which means that this object is owned by the same agent #19. The same is also true for object `apple(25)`, held by agent #25. However, this is not the case for items `obligation(53)` and `obligation(54)`. Without looking into details of those objects, one can see that they are portrayed with cyan border, which is the color of master agent, which means these obligations are owned by `god` agent.

Another important application for color-coding is reflecting the agent or item status. For those types of changes, we make use of commonly used conventions for marking up disabled and new elements. The disabled are typically pictured in low contrast monochrome colors, while the new item notion is commonly represented by using other color than one used for the elements of the same type within the application, and often the lighter one (or warmer in hue). Thus, we have selected the grey color for deleted elements and yellow for the new ones. We have arbitrarily chose the light blue for changed elements. In case of changes in the market state the element of interest will be agent, and in the case of agent — the possessed item. We do not, however, mark the agent as changed as there is small meaning in doing so. We intend to keep the removed items in visual field for the duration of one step, explicitly marking them as removed. We have implemented the colors and background properties in such a way so that setting the attribute "Status" on the element would trigger its redrawing in colors selected for picturing the removed elements. Low contrast grays seem to be the fitting choice for the task.

We treat the new elements n similar manner. For the duration of one state, they possess lively yellow background color. In the current implementation, it look as follows:

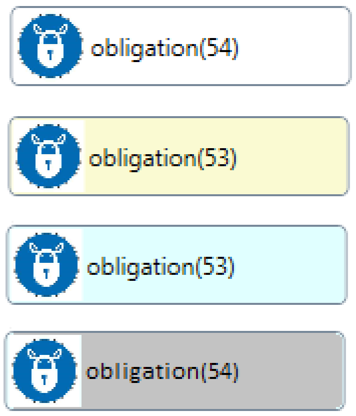
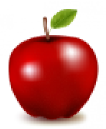


Figure 4.10. Color-coding for object status. Representations of regular (white), added (yellow), altered (blue) and removed (grey) item.

4.2.2 Customization Elements

In the course of development of this visualization application we were focusing on customizing the English auction domain and have supplied the icons for all rights and items, normally observed during the simulation run. Here we list all the images, used in our customization with the instances depicted by those.

Image	**Object**
	Domain class `apple`
	Domain class `coin`
	Domain class `enlish_auction`

	Instance of the `right(create_var())`. Depicts the right to create a variable.
	Instance of the `right(issue_o())`. Depicts the right to issue an obligation.
	Instance of the `right(issue_p())`. Depicts the right to issue possession of the right.
	Instance of the `right(open_auction())`. Depicts the right to open an auction
	Instance of the `right (sell())`. Depicts the right to sell an object.
	Instance of the `right(place_bid())`. Depicts the right to place a bid in the auction
	Instance of the `right(query())`. Depicts the right to query about the item
	Instance of the `right(take_on())`. Depicts the right to take on an obligation.
	Instance of the `right(_)`. Depicts the right to an object that the members of the party the holder belongs to are entitled to.
	Instance of the `right(flip_coin())`. Depicts the right to flip a coin
	Default icon for an unknown right.
	Default icon for any kind of obligation.

Table 4.1. Images, used in customization

4.2.3 Agent Frame and Its Contents

It seems only natural that our visualization will focus on the agents, and represent all the other objects in relation to them, as they are the only actors on the simulation marketplace. In real life there are also environmental factors that may directly or indirectly affect market conditions, such as new laws or taxes, or even the earthquake that has damaged significant manufacturer's facilities, but these are not directly present here. Therefore we have developed the agent entities, picturing each of those as a separate object. The agents are uniquely identified by their id. As we want to accommodate the application for the user-friendly depicting of many agents, we decided to use the collapsible representations, so the user can choose, which agents he or she wants to view in detail, and which not. Collapsed agent representation can be observed on figure 4.11.

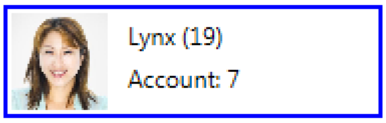


Figure 4.11. Collapsed visual representation of an agent. Only agent icon, name, id and account are visible, while the list of items agent holds is hidden.

This representation supplies only the most essential of the agent details: its id, name, icon and the current account. In addition, the background of the agent frame header, both collapsed and expanded represents the status of the agent, according to the color-coding notation, described above. Light yellow marks the agent that has entered the market environment, while dark gray is used for the one that has left. The collapsed agent frame, same as the expanded one, is bordered in a distinct color used for color-coding. Upon a click on the collapsed frame, it expands, letting the user observe the items, hold by an agent.

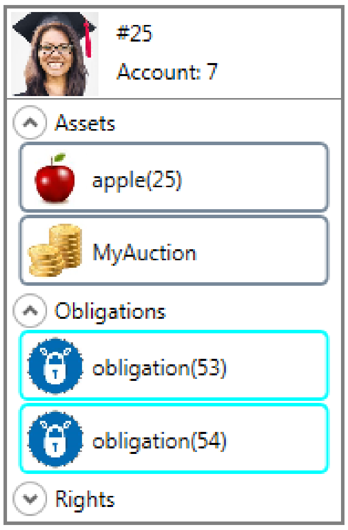


Figure 4.12. Expanded agent frame. In addition to agent header, containing agent icon, its name, id and account, one can observe the objects held by the agent, divided in three groups.

This representation provides the observer with the following important agent information: its name (id), name, icon, list of items and current account. The name and the icon are populated using the customization file and can be replaced as user wishes. There is a possibility to provide each individual agent with the color of user's choice in the future and we have listed it in Possible Further Improvements section.

We have also implemented additional visual cues for agents joining and leaving the market environment. The initial opacity of an agent frame is zero, and upon joining it changes to one during the time interval of one second, so the agent frame kind of surfaces. Upon the leaving the market state, agent frame in the same manner slowly vanishes.

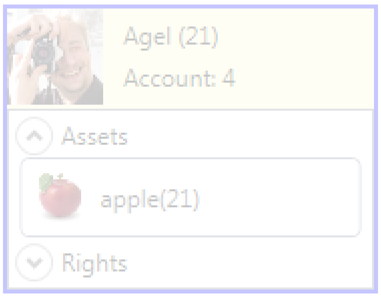


Figure 4.12. Almost transparent agent frame with yellow background during the joining.

Model release notice: We have obtained a verbal permission from the people depicted on the agents icons to use these in our course project. Written release can be obtained if necessary.

Item Grouping

As one can see on the agent frame, represented on Figure 4.12, the items belonging to an agent, are separated into three distinct groups, representing the three types of objects in the environment: assets, rights and obligations. In order not to clog up the visual space, we implemented these three groups as expandable lists, which can be opened by clicking on the group header. As the rights of an agent in the English auction domain are typically numerous, by default the "Rights" group is demonstrated in collapsed way, and assets and obligations, as those are few — in expanded. In the image indicated above, one can see expanded “Assets” and “Obligations” sections and collapsed “Rights” section. The example of expanded “Rights” group can be observed on Figure 4.13.

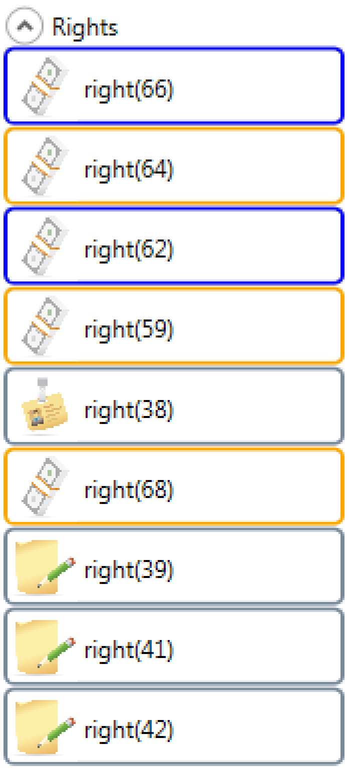


Figure 4.13 Expanded “Rights” section. One can see that the different icons are used for depicting different kinds of rights, as well as different border colors are used to mark the rights owned by different agents.

Item

While agents serve as centerpieces of the picture, we would also want to see the objects of their actions — items. Items form the second most important entity of the application. For every agent we will reflect the items being at the given time moment in his possession, along with his account. We visually mark the items, which while being possessed by one agent, belong to another by use the color-coding, described above. It also seems useful to provide user with the possibility to examine the item in detail upon his / her request. In that way, for each item we shall depicture its attributes, its owner and holder. As we have mentioned above, in Input Processing section, in order to improve the recognition of the entities, we are supplying items with relevant icons, according to the type of an item. If the class icons are provided, we use those, if not — we shall use default generic ones.

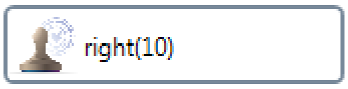
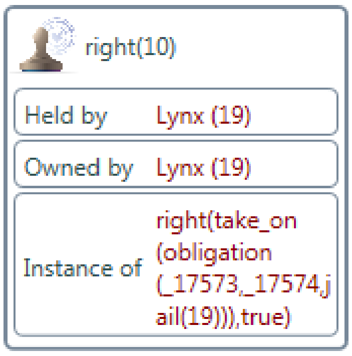

right(10)

Figure 4.14. Collapsed item frame. It contains only item name, item icon (representing its class) and its owner, represented by the border color. Its holder is implicitly present as a encompassing frame. Other item attributes are hidden.

In order to provide the exhaustive details for every item and at the same time to keep the general view concise and easy manageable, we implemented the expandable template for an item frame. The item frames are first demonstrated in the collapsed form, as shown on Figure 4.14, but upon request (most naturally implemented as a click on the item) the item frame expands, and there the item details might be observed.


right(10)

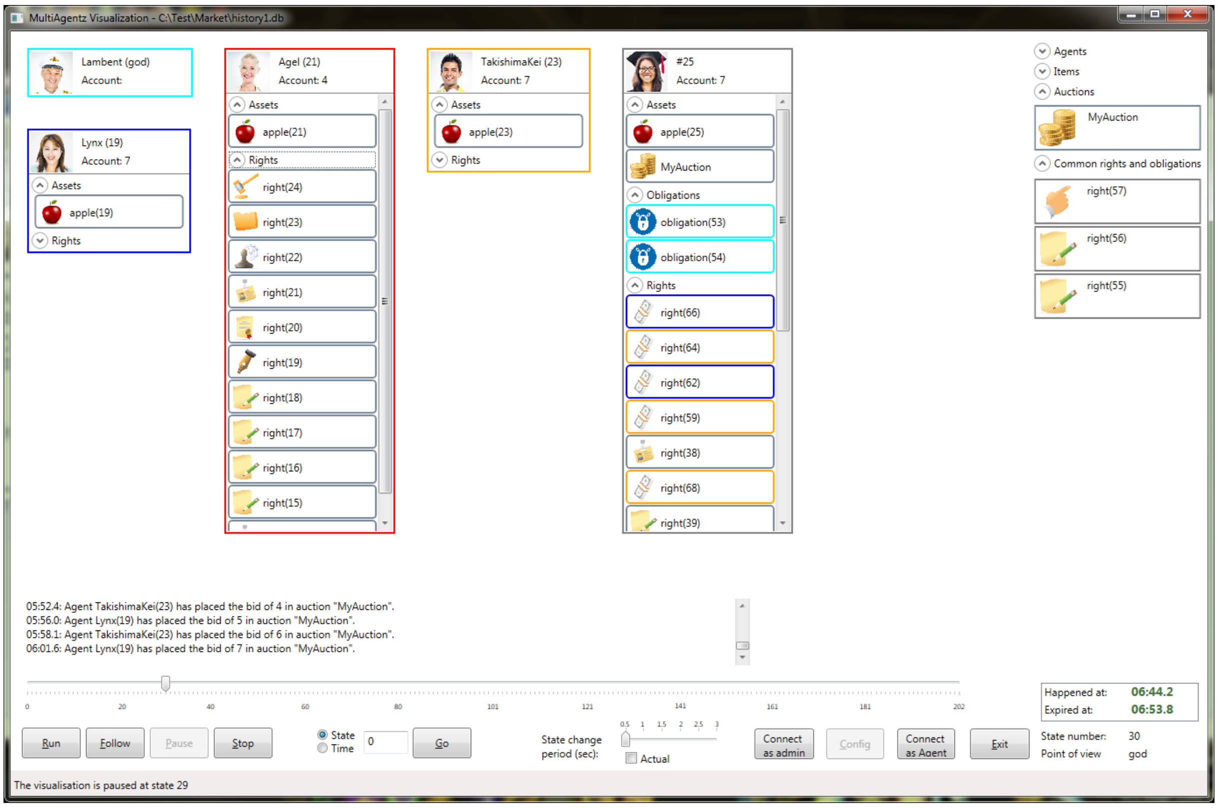
Held by	Lynx (19)
Owned by	Lynx (19)
Instance of	right(take_on (obligation (_17573,_17574,jail(19))),true)

Figure 4.15. Expanded item frame. In addition to the information, present on the collapsed frame, the expanded frame lists explicitly all the attributes of the object.

The color of the border is not the only means to identify the owner of the item. As can be seen on figure 4.15, on expanded frame one can see the owner, the holder, the element the item in question is an instance of, and any number of arbitrary descriptive attributes, such as weight, color and the like. As one can see, we provide the names of the agents in addition to their ids in the according fields for better identification. Also, collapsed and expanded frames alike, color-code the item status in background color, as indicated above.

4.2.4 Market State and Related Entities

Market state is probably the most natural logical entity in this application. One such is mapped to every line received from data source and contains all information encompassed within it. Market state is exactly what its name implies: the state of the simulation market in the given moment. As of the moment, market state contains the collection of agents, which in their turn contain the collection of items, the event, and the clock, but with further development of the application we can supply the market state with additional information, for example, historical, such as the last transaction, or analytical, like the total budget currently on the market. These possibilities is covered in Future Work Chapter. Visually the market state is represented as the main frame of the application. However, in order to supply all the necessary functionality, this frame contains some additional elements. Let us have a look on the main window of the application, identify its elements and examine them in details.



05:52.4: Agent TakishimaKei(23) has placed the bid of 4 in auction "MyAuction".
05:56.0: Agent Lynx(19) has placed the bid of 5 in auction "MyAuction".
05:58.1: Agent TakishimaKei(23) has placed the bid of 6 in auction "MyAuction".
06:01.6: Agent Lynx(19) has placed the bid of 7 in auction "MyAuction".

Happened at: 06:44.2
Expired at: 06:53.8

State number: 30
Point of view god

The visualisation is paused at state 29

Figure 4.16. Main window of the application. It contains the following main parts (left to right, top to bottom): agents field, reference panel, action log, navigation control group, application control buttons, state information block.

The top left corner of the main window is taken by control elements, which we will examine in the next section. For now, let us focus on the rest. The top left corner is occupied by the collection of all agents, which are present in the market environment in the given step. We have examined the Agents in detail in the previous paragraph. It can be seen that each agent has its own distinctive color and its own image. However, for the agent #25 the customization information was not provided and thus he has the default icon and no name of its own.

Below the agents frame there is the log of all events that has happened during the previous states of the simulation run. Those are given in a text format and have a timestamp.

The right side of the main window contains Reference Panel, representing the items and agents, not indicated explicitly in the current market state, containing four panes for four lists — panes, listing all agents of the simulation, all items of the simulation, special items of the state and common rights and obligations of the state.

All four of the panes are made expandable and scroll automatically when their length exceeds the allocated space.

The bottom right corner is assigned to the state information. There is the clock, showing the start and expiration time for the state, state number and the agent, from whose point of view we observe the simulation run.

Actions

In the aforementioned way we shall represent the subject and the objects of an action, but we would like to reflect the actions themselves. One way to do it is to show explicitly the changes that took place during the last change of state. For this purpose, agent's list of possessed items is visually separated in four parts: the ones that were in his possession during the previous state that has undergone no changes, the ones that were altered since, newly acquired and deleted. We use the background of an item shown in the agent item list to reflect the information regarding item status, color-coding the changes as described in Section 4.2.1 General Assumptions and Conventions. However, we want not only to see the aftermaths of some action, we want to have some idea about the performed action. At the same time, not all the actions have the consequences that can be immediately observed visually. Thus, we also implemented the action log, in which the actions taken by the agents can be observed. The example of such log can be observed on Figure 4.17.

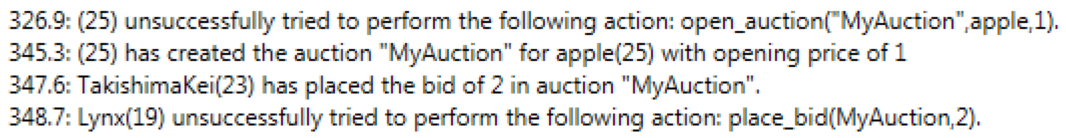

```
326.9: (25) unsuccessfully tried to perform the following action: open_auction("MyAuction",apple,1).
345.3: (25) has created the auction "MyAuction" for apple(25) with opening price of 1
347.6: TakishimaKei(23) has placed the bid of 2 in auction "MyAuction".
348.7: Lynx(19) unsuccessfully tried to perform the following action: place_bid(MyAuction,2).
```

Figure 4.17. Action log. It lists in chronological order the actions performed by the agent. The entries contain the time when the action took place, the actor and the action itself, where possible, translated into a natural language form

As can be seen from the illustration, the log provides the following information: timestamp, the actor and the action. In addition to actor id log incorporates such customization element as agent name, if such is provided. However, depicting the actions is customized in one more way. For successful actions, we perform the parsing of the original statement and reassemble it in the form of natural language statement, to make it more readable for a user.

Reference Panel

Reference panel occupies the top right side of the main window. It contains the reference lists of different elements of the simulation run in the form of expandable panes. Figure 4.18 demonstrates the Reference Panel with expanded Special Items Pane.

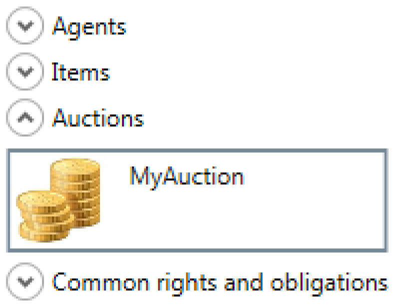


Figure 4.18. Reference Panel with expanded Special Items section. Its header name is defined in a visualization customization file.

These four panes are:

1. Agents Pane. It contains the list of all agents, participating in the simulation during the current run; however, it differs functionally from the agents frame occupying the middle area. This pane contains the frames for all agents that were at any moment present in the environment, not only for the ones present in the current state. This pane can be used for easy navigation to the state when it has joined the visualization and when it has left.

2. Items Pane, which contains the list of all items of the asset type, referenced in the simulation during the current run. It in similar manner lists all the asset-type items that were present during the entire simulation run. This allows the user interested in specific item not to search for an agent that has it, but to be able to examine it straightaway

3. Special Items Pane, containing the domain-specific special items present in the environment during the current state. For English auction and Sealed bid domains the items listed will be auctions, for coin domain — coin item. In this way user can always see the auctions or a coin in separate place and does not have to ruffle through the items of the agents to find one he wants. The name in the header of Special Items Pane is domain specific and is set up in customization file, as described in Input Format Section. On Figure 4.18 it can be seen that the Special Items Pane bears the heading "Auctions".

4. Common Rights and Obligations Pane, containing the rights and obligations common for all agents present in the environment in the current state. For example, when an agent opens an auction, every agent present in the market environment has the right to place a bid. Such rights or obligations are common for all participants in nature and are listed separately

Agents Pane

As we mentioned in Input Processing Section, during the log file processing we collect a dictionary, stating when every agent has joined and left the simulation. In case of the real-time run, this dictionary is populated during the run, and updated as soon as new information shows up. The background of the agent frame on the Agents Pane reflects, whether this agent is present in the environment in the current state. Figure 4.19 shows Agents Pane example.

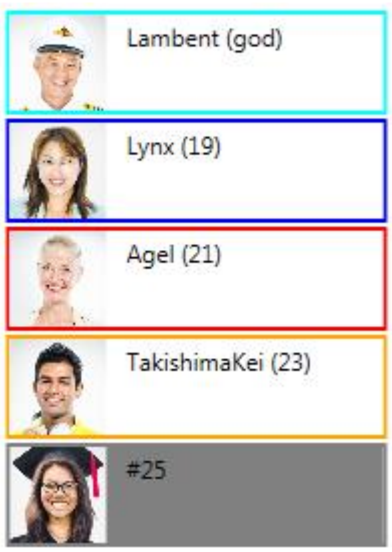


Figure 4.19 Expanded Agents Pane. There can be seen all the agents that were present in the given simulation run at some point, having a border or their own color. Grey background means that the agent is not present in the state being currently displayed.

On this pane it can be seen that agent #25 is not present in the environment in the current state, as its background is dark grey, which we find to be a suitable color-coding for an absent agent. The border colors of the agent frames are the same with used in main frame.

Agent frames in the pane are also expandable. Upon clicking on the frame, it opens, revealing the start and end state for this agent. Each of those are accompanied by buttons, clicking of which navigates the application to the corresponding state. Example of the expanded frame can be observed on figure 4.20.

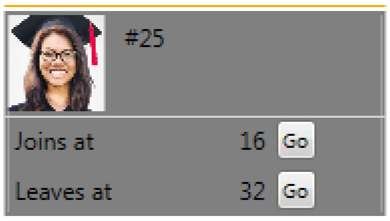


Figure 4.20. Expanded agent frame in the Agents Pane. It contains the numbers of states where the agent has joined and left accompanied with the buttons for navigation to these states.

Items Pane and Special Items Pane

Similarly to the Agents Pane, Items Pane depicts all the asset-type items that were present in the market environment at some point of time. As this information is not needed often, the Item Pane is by default shown as collapsed. Figure 4.21 shows expanded Items Pane.

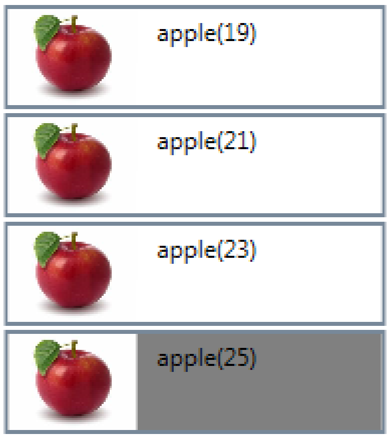
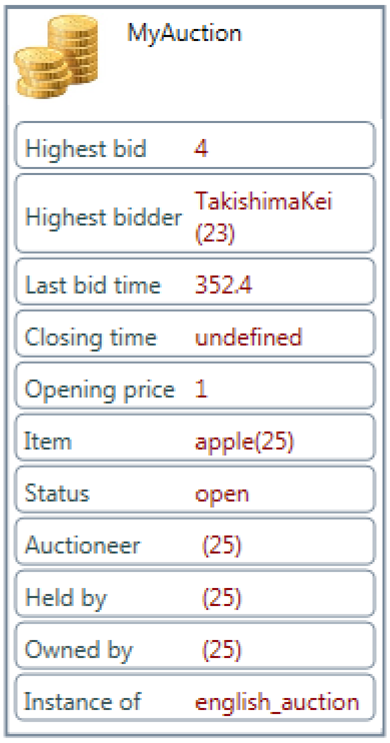


Figure 4.21. Expanded Items Pane. There can be seen all the items that were present in the given simulation run at some point. Grey background means that the item is not present in the state being currently displayed.

Unlike Items Pane, the Special Items Pane is by default expanded, as the Special Items comprise the element of special interest for a user. While Items Pane merely serve for a reference, the information in Special Items Pane is unique and useful. In the English auctions domain, the auctions are only demonstrated in Special Items Pane (and mentioned in the Action log, described above).

In order to avoid the unnecessary visual noise, the Items Pane and Special Items Pane elements are shown up in the collapsed view. Click on the element expands the frame, so all the attributes of the item can be observed. Figure 4.22 demonstrates the expanded auction frame.


MyAuction

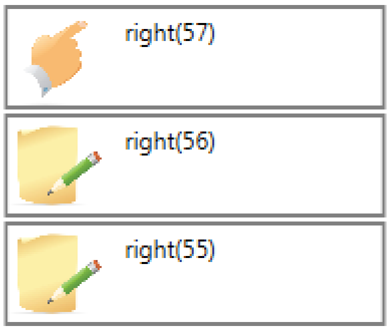
Highest bid	4
Highest bidder	TakishimaKei (23)
Last bid time	352.4
Closing time	undefined
Opening price	1
Item	apple(25)
Status	open
Auctioneer	(25)
Held by	(25)
Owned by	(25)
Instance of	english_auction

Figure 4.22. Auction details window. Alike the expanded item frame, it lists explicitly all the attributes of an auction.

This panel demonstrates all the attributes of an auction. Auction is characterized by a number of important features, which are shown on the above figure, such as the owner and the auctioneer of an auction, the auctioned item and so forth. For the sake of easy recognition, all the referenced ids, such as owner's or holder's are accompanied by associated names, if such are provided for in customization file.

Common Rights and Obligations Pane

This pane is designed in order to keep in specific place the rights and obligations, which every market environment participant is subject to. Those rights are automatically created in response for some type of event. The creation of an auction is an example of such event. It triggers the issue of three rights: right to get the information about the auctioned item while the auction is open, right to get the information about the auction while it is open and right to place bid in this auction while it is open. Figure 4.23 demonstrates the example of Common Rights and Obligations Pane after the creation of an auction, performed by one of the agents.


right(57)

right(56)

right(55)

Figure 4.23. Common Rights and Obligations Pane. Lists in collapsed form the right and obligations effective for all market participants.

The items of the pane are shown in collapsed way, and the only information available in this form is the name of the right or obligation, and the type of the right as indicated by the icon. The descriptions of the icons used by us in the current customization are given in section 4.2.1 "General Assumptions and Conventions". Alike the other frames, Common Rights frame expands on click, revealing the element details.

Clock

Another small, but important separate entity is the inner simulation clock, that shows for each state, in which moment in terms of simulation time the specific state took place. The clock shows two time values: happened at and expired at, marking the beginning and the end of the period when the market had the indicated state, measured in seconds from the beginning of simulation run. The visual representation of the clock looks like follows:

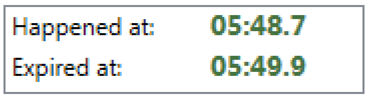

Happened at: 05:48.7
Expired at: 05:49.9

Figure 4.24. Visual clock representation. It contains two values: happened at, time when the state has come into existence, and expired at, when the next state has emerged. The values are given in a user-friendly format.

State Number and Point of View

In the bottommost right corner, we keep such useful information as market state number and point of view. The market state number is counted from the beginning of the simulation, and the point of view lists the id of the participant, whose log file serves as a source of input at this particular moment. By default, the simulation is played from the master agent point of view, but the user has the possibility to switch views, in which case the field will show the appropriate id, and all the details are demonstrated as per chosen agent log file.

4.2.5 Visualization Implementation

For implementing visual components of the application, we have selected such relatively recently available technology as WPF / xaml [20]. The very nature of visualization task is to reflect the underlying structures and as the structures in question change continuously and dynamically during the runtime, the visualization application should provide the means to represent these changes. This particular task is exactly what WFP has addressed with its data binding possibilities: through mapping data structures to visual templates the changes in the latter are automatically reflected in visual elements. For example, binding an application variable to a text property of TextBox has the effect that every time the value of this variable is changed by code, the content of corresponding TextBox will change accordingly.

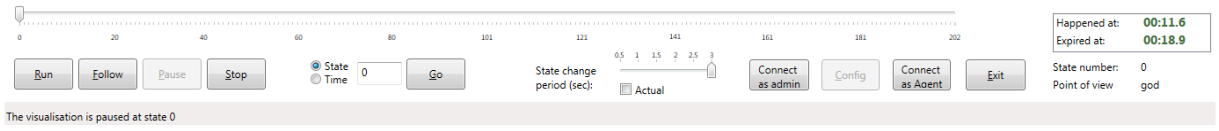
The representation of the elements themselves is to be implemented using XAML — XML-compatible markup language. XAML stands for Extensible Application Markup Language and provides a up-to-date means for defining application interfaces in a flexible manner. The appearance of elements in XAML is defined through declaring properties and attributes within corresponding element tag and is completely independent from application code. This definition is contained within separate file, thus providing for effective decoupling interface and code and easy maintainability.

In our application, we have defined four templates for representing four types of structures: the three of them are encompassing State template, the Agent template and Item template, each of which will be instantiated with the data provided by the application during the runtime. We have also defined an ancillary template for configuration settings, in order to give the user a possibility to change the configuration settings in convenient manner using the GUI instead of having to edit xml file. We use the data binding mentioned before to link the visual representation to the actual structures and thus providing for automatic update of the visual elements. We have also implemented a number of specific fields within the elements for a purpose of implicit representation of certain element properties. What we mean is that while it is always possible, for example, to explicitly mark the button as disabled, big number of such indications will clog the field of vision, hindering the perception of the whole picture and overwhelming the viewer's attention. With the development of visual interfaces since first graphic applications, some of the implicit ways of representing properties have made their way into widely accepted conventions, and in our application, we shall try to make use of some such.

However, this particular way of representation is, of course, implementing only one of the multiple possibilities, and can be changed if necessary.

4.3 Control Elements

In Section 3.2.1. Desired Functionality we have described in detail the functions that we want the visualization application to perform. For this purpose, we have developed a control panel, which can be observed in the bottom section of application's main window. Its appearance can be observed on the Figure 4.25. Let us identify its main elements and examine their function in detail. However, not all of the functions are present in the form of explicit control elements, but instead, are implemented in other forms. We will examine them after we will have covered the controls.



Happened at: 00:11.6
Expired at: 00:18.9

[✓ State] Time 0

State change period (sec): ☐ Actual

State number: 0
Point of view god

The visualisation is paused at state 0

Figure 4.25. Control panel of the application. It contains main slider, navigation control group, application control buttons, state information block and status bar.

4.3.1 Main Slider

The top part of the control panel is occupied by the main slider of the application. It has the following separate functions:

1. Providing the visual perspective of the length of a log file measured in the number of states and the current position of the visualization run in it.

2. Serving as a means of navigation. By dragging it to the desired state, the user can move to it. By clicking the slider to the left and to the right of the current position, he can navigate one step back and forward, accordingly.

3. Serving as a reference area for the navigation. When the user types a state number in the input field for navigation to state, the marker on the slider shows the corresponding position. The example can be seen on Figure 4.26.

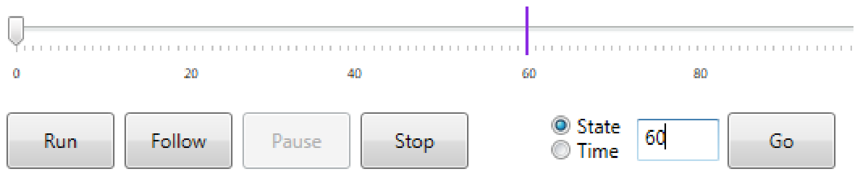


Figure 4.26. Marker on the slider, showing the position typed into navigation input field as a deep violet mart.

4. Providing the same functionality in relation to time typed into the input field (in seconds). To avoid the confusion of these two markers, they are colored differently. The example is available on Figure 4.27.

5. Gives the visual perspective for the states available from a given agent point of view. The states during which the agent was present, are visually separated. The example can be seen on Figure 4.27. In addition, this figure contains the time reference marker, described in previous paragraph.

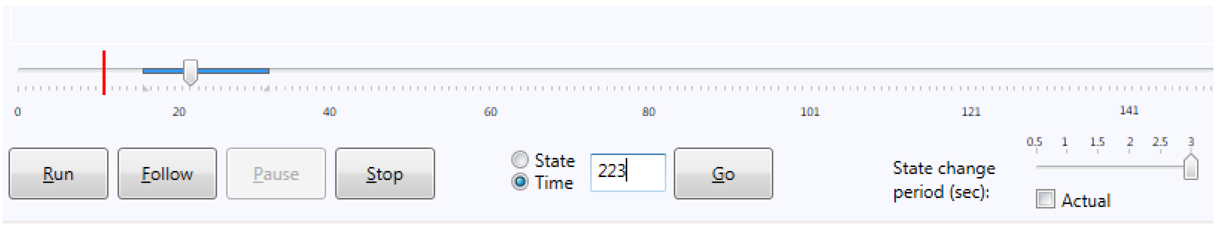


Figure 4.27. Agent presence during the simulation run marked on the slider. The time, entered into navigation input field is shown by a red mark on a slider.

4.3.2 Navigation Element Group

Figure 4.27 shows the group of control elements that are used for navigation. Those are:

1. “Run” button. Starts the consecutive visualization of environment states of a selected log file upon reaching the end of the file, waits for updates and reflects them when those are produced.

2. “Follow” button. Starts the consecutive visualization of environment states of a selected log file, however, keeping the distance of a particular numbers of states from the end of the log file. The latency is set to the current distance upon pressing the button. It is only useful for an online run, as for the offline it will keep waiting for the log file to update and stay at the same place.

3. “Pause” button. Pauses the implementation. If the after pausing the execution is resumed in by the “Follow” button, the latency is set anew.

4. “Stop” button. Stops the current visualization run.

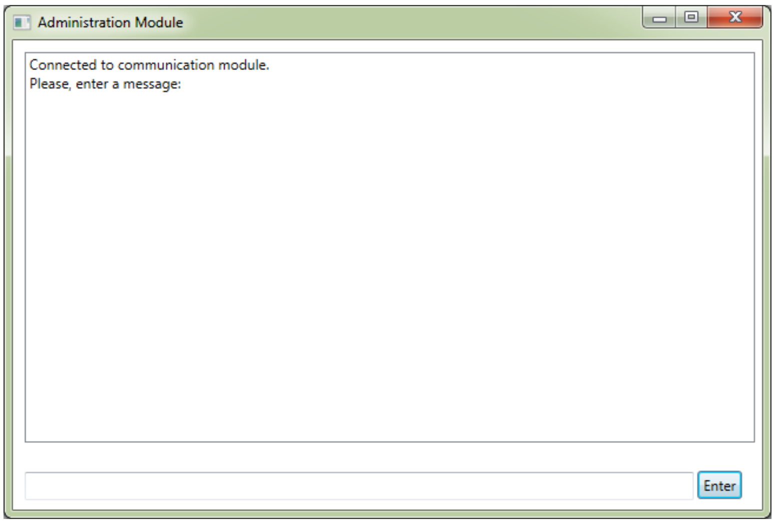
5. Radio selector, input field and “Go” button. Radio button tells the implementation how to interpret the content of an input field. Upon pressing “Go” button, the input field is interpreted as either number of state or the time from the beginning of simulation, depending on selector position, and navigates to that state or time accordingly.

6. State change period slider and checkbox. Determines the period of state changes during the visualization run in “Run” mode. If the “Actual” checkbox is ticked, the state changes are reproduced according to the time they happened during the original simulation. This means, that if state 1 has happened at 10 sec from the beginning of the simulation, in replaying this run with “Actual” tick set it will be demonstrated 10 seconds after state 0. If the tick is not set, the states change according to the values on the slider.

7. Status Bar. It serves for reflecting the current mode of visualization run, for example “The application is executing in Run mode” and for displaying error messages, for example, when user tries to go to the illegal state number, the status bar shows appropriate message in red.

4.3.3 “Admin” Button

If the value for “AdminRights” has been set to true in configuration file, the system provides a possibility to run the application instead of the administration module of the original system. Then during the start, it prompts user for a password and tries to connect to communication module. If this attempt was successful, the user has a possibility to open the Administration console by clicking this button.

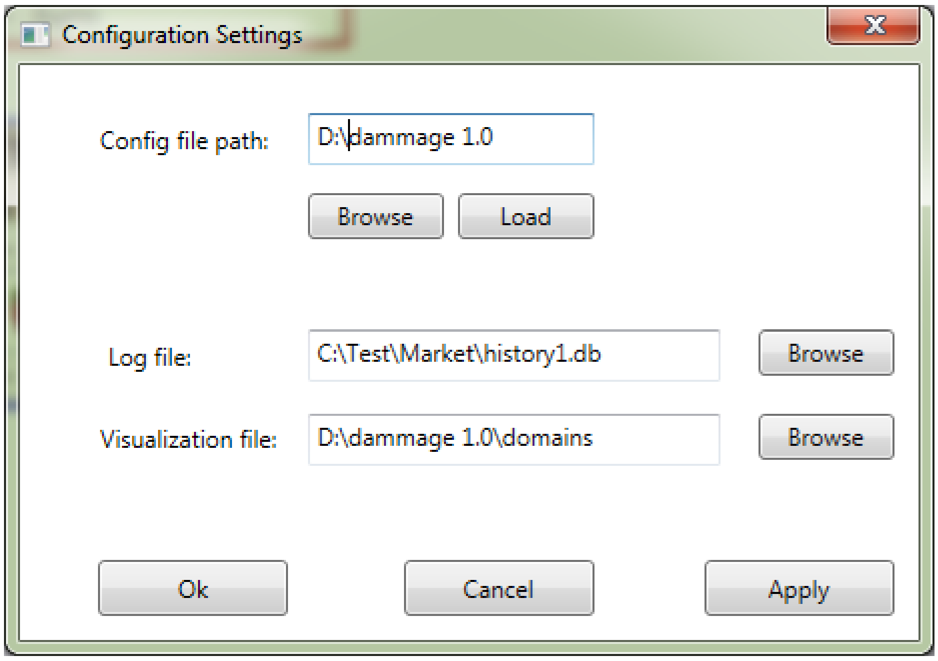


Connected to communication module.
Please, enter a message:

Figure 4.28. Administration console. It contains the message of a connection to the system, invitation to enter a message, last entered messages and the input field.

4.3.4 "Config" Button

We have covered in detail the contents of the configuration file in section 4.1.2 Input format. By clicking this button the most settings present in the configuration file become available for a change through user interface. Figure 4.30 shows the configuration settings window.



Config file path: D:\dammage 1.0

Log file: C:\Test\Market\history1.db

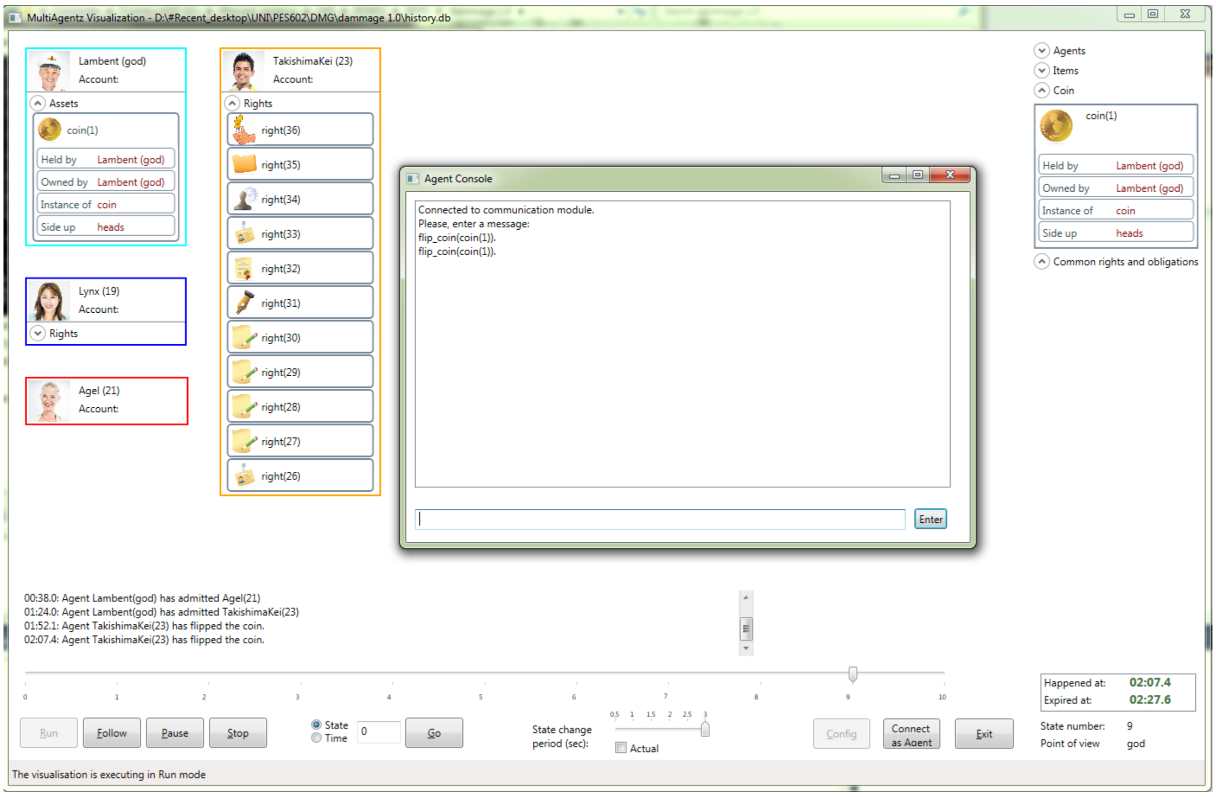
Visualization file: D:\dammage 1.0\domains

Figure 4.30. Configuration settings window. It shows the fields for paths to log file and visualization file, as well as a path to the configuration file itself, accompanied with the buttons for picking up the path and saving the configuration.

There can be observed the main settings: path to configuration file, path to log file and path to customization file. The configuration settings cannot be changed during the ongoing run. After the new settings have been saved, the main window of the applications is reloaded.

4.3.5 "Connect as Agent" Button

This button provides the possibility to connect to the simulation during the online run. The simulation and the proxy must be running for this to be successful. Upon clicking on the button, simulation tries to connect to proxy, and if succeeds, opens the Agent console window. It can be used to send the agents commands to the running simulation. The example of the main window and Agent console on top of it is provided on Figure 4.31.


MultiAgentz Visualization - D:\#Recent_desktop\UNI\PES602\DMG\dammage 1.0\history.db

Held by	Lambent (god)
Owned by	Lambent (god)
Instance of	coin
Side up	heads

```
Connected to communication module.
Please, enter a message:
flip_coin(coin(1)).
flip_coin(coin(1)).
```

```
00:38.0: Agent Lambent(god) has admitted Agel(21)
01:24.0: Agent Lambent(god) has admitted TakishimaKei(23)
01:52.1: Agent TakishimaKei(23) has flipped the coin.
02:07.4: Agent TakishimaKei(23) has flipped the coin.
```

Happened at:	02:07.4
Expired at:	02:27.6
State number:	9
Point of view	god

The visualisation is executing in Run mode

Figure 4.31. Main application window, showing the simulation run on the coin domain with the agent console window on top. It can be seen that actions invoked via console (flip coin(coin(1))) are depicted in the action log of the application.

4.3.6 Exit Button

Stops the current run and closes the application.

4.3.7 Miscellaneous Controls

One way of controlling the appearance of the frames is clicking on its header. Upon left click, item and agent frames, as well as auction frames, toggle their state between expanded or collapsed, revealing their attributes or showing only header, respectively. By clicking the Agent Pane frame, the user reveals the navigation frame, showing the states when the agent has joined the simulation and left it, along with the buttons for navigating to those states.

Control + left click on agent frame switches the point of view of the application to the clicked agent.

Chapter 5
Interface Assessment

In this chapter, we cover the motivation for assessing the interface of the proposed solution, review the employed techniques and their features and characteristics, review the results of their application and give the interpretation of acquired results.

5.1 Used Methods

One the most important quality of any software product is usability. During the development of the software applications, developers typically concentrate on the functionality, trying to cover all functional and non-functional requirement, though the fact that it is the people, who will work with the product, is often overlooked. However, in practice, users often prefer the less functional, but easier-to-use applications, as it is more comfortable and efficient to work with.

However, to be able assess the usability of the interface, first we need to agree about the meaning of the work. As per ISO 9241-11, usability is the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. [21] In its turn, effectiveness can be defined as the degree to which something is successful in producing a desired result, while the efficiency means the extent to which time, effort, or cost is well used for the intended task or purpose. User satisfaction is commonly defined as an attribute extracted from a product or service after its consumption.

In other words, the application usability measure defines, how easy and comfortable is to use the given software in order to carry over the necessary task.

In order to assess the usability of a user interface, one needs to employ some set of evaluation methods. As this issue is commonly addressed, there are many available techniques, which can be separated in two partitions: expert-based and user-based.

As the number of established usability evaluation methods makes it non-time-efficient to employ all of them, in our work we are going to implement three most popular and acknowledged of those that are selected both partitions mentioned above [22] — Think Aloud method (user-based) and Cognitive Walkthrough and Heuristic Evaluation methods (expert-based).

Think Aloud technique does not have any formal structure, and might be implemented differently by different researchers, but the main idea stays the same and includes a group of users thinking aloud while performing a task and a researcher recording their output. Typical procedure includes three steps. First, a number of users is selected for the research, introduced to the problem and is given a warm-up task to familiarize them with the notion. Then, one-by-one, they are assigned a task and proceed to carry it over, voicing their actions and issues they encounter. When they are done, they are asked to share their opinion about the evaluated system in general.

Heuristic Evaluation method is also informal assessment technique, based more on the "rules of thumb" rather than on formal research, which can be employed on a real system or on the mock-up screens. It is relatively easy to perform, but practical results will often depend of the knowledge and competence of the person performing the assessments [23]. The assessment by this technique is performed in the following manner: the expert examines every screen of the system and marks the answer to the list of heuristics. The list we used in our work is provided in Appendix C: List of Heuristics for Interface Assessment

Cognitive Walkthrough is a usability inspection method performed by the experts and based on the evaluation of typical usage scenarios. It consists of two phases: preparation and evaluation. In the first phase, the evaluators examine the tasks users are performing on a daily basis and divide them into actions, listing all the steps. In the second phase, experts walk through the scenarios developed in the preparation phase, answering a set of questions after each step. By answering the questions for each subtask usability problems will be noticed [24]. We answer the following set of questions:

1) Will the user try to achieve the effect that the subtask has?
2) Will the user notice that the correct action is available?
3) Will the user understand that the wanted subtask can be achieved by the action?
4) Does the user get appropriate feedback?

5.2 Participants

The participants of this research were divided in two groups, for performing user-based and expert-based evaluation, accordingly. For user-based methods, we have chosen a group of ten people, all of which have had previous experience with auction-related software, as we were evaluating the application using the English auction domain. For the second group we selected users who also had experience with auction-related software, but who in addition, possessed the experience in software testing, including usability testing. The subjects were chosen using the convenience sampling method.

5.3 Evaluation Procedure

To minimize the effect of statistical fluctuation, to increase the coverage of the evaluation and improve the confidence of the results, in our usability evaluation we tried to implement the following guidelines on usability evaluations:

1) Have more than one evaluator to maximize the number of obtained results
2) Provide a common set of tasks to carry out.
3) Provide a common format for problem finding reports
4) Select criteria for usability evaluation

In pursuance of the above, we have provided the evaluation teams with the set of tasks to carry on and have ensured that the findings were documented in the same format. We have also defined a severity of the found issues as follows:

1) Low — visual or aesthetic issue. The ability to carry out the task is not impaired

2) Medium — the found issue causes inconvenience to the user, disorienting him. He can still carry out the task, but feels unconfident or uncomfortable doing so.

3) High — issue significantly impacts user's ability to perform the task or prevents its completion

5.3.1 Object of the Evaluation

As the part of this work, we are evaluating the user interface of a visualization application that is designed for visualizing the output of the simulation system [01] and providing the user with a range of functions, such as navigation through the market environment states, viewing the agents and items in the customized form, observing the attributes of the environment objects and examining the actions taken by the agents.

5.3.2 Participants

As we have chosen for implementation both user-based and expert-based methods, we have selected two teams of participants. The firsts consists of ten separately interviewed users, each of which have had previous experience with auction-related software and have not seen this application before. For the second, we selected an evaluator with the Master of Science degree and previous knowledge and professional experience with interface evaluation.

5.3.3 Think Aloud Method

For evaluating the visualization application by this technique, a first group of participants was employed. In order not to compromise the results, the users were examined one at a time under researcher's supervision. To conduct the usability testing in controlled environment, all experiments were performed on researcher's computer. TeamViewer software was used to provide the users with the access to remote machine. For establishing voice communication with the users, VoIP software Skype was employed.

In this experiment users were asked to perform simple operations – open history file, find the opening time of one specific auction, count bids on this auction, and check asset and auction properties. The exact list of operations provided to participants looks as provided in Figure 5.1:

1. Open simulation log file “C:\Test\Market\ history1.db”
2. Find the time when auction “MyAuction” was opened
3. Count number of bids on “MyAuction” until state No. 32 (including)
4. Check weight of item being sold in auction “MyAuction”
5. Get status, highest bidder and highest bid of “MyAuction” in state number 22
6. Play all states from agent 25 point of view at speed 2 states per second

Figure 5.1. List of tasks, provided to the participants for Think Aloud Method

While the participants were carrying over the listed tasks, they were asked to voice their actions and issues they encountered. The researchers observed the actions of the users, listened to their feedback and created a list of issues they experienced and complained about. The complete table of encountered problems is available in Appendix D, part 1.

5.3.4 Heuristic Evaluation

For this methods, we employed the list of heuristics by J. Nielsens [24], provided in Appendix C. This method was implemented by the second group of participants. Before conducting the testing, the group has examined the list of heuristics and got familiar with the concept of the method. The complete table of found problems is available in Appendix D, part 2.

5.3.5 Cognitive Walkthrough

For this method, we have developed a scenario that represents possible actions of a regular user and covers the greatest part of system functionality. The step-by-step description of the scenario can be found on Figure 5.3.5.1.

1. Go to configuration
2. Browse for history1.db log file
3. Press “Save and Exit”
4. Press “Run”
5. Press “Pause”
6. Enter 22 in box ner “Go”
7. Press “Go”
8. Double click on “MyAuction”

9. Ctrl+Click on agent 25
10. Check checkbox "Actual"
11. Press "Run"
12. Wait till end of agent 25
13. Ctrl+Click on agent "god"
14. Press "Pause"
15. Select radiobox "Time"
16. Enter "123" in box near
17. Press "Go"
18. Press on item "Apple" on actor 21
19. Press "Stop"
20. Press "Exit"

Figure 5.2. Steps for Cognitive Walkthrough Method developed and executed by an expert.

This method was implemented by the second group of participants. Before conducting the testing, the team has familiarized itself with the method and was given all the necessary information. The complete table of found problems is available in Appendix D, part 3.

5.4 Results and Data Analysis

After performing the evaluation, we have created a set of result tables. While Heuristic Evaluation and Cognitive Evaluation methods showed only four problems each, Think Aloud method demonstrated sixty-five usability issues.

If we break down all of the identified issues by severity, the result will look as shown of Figure 5.3.

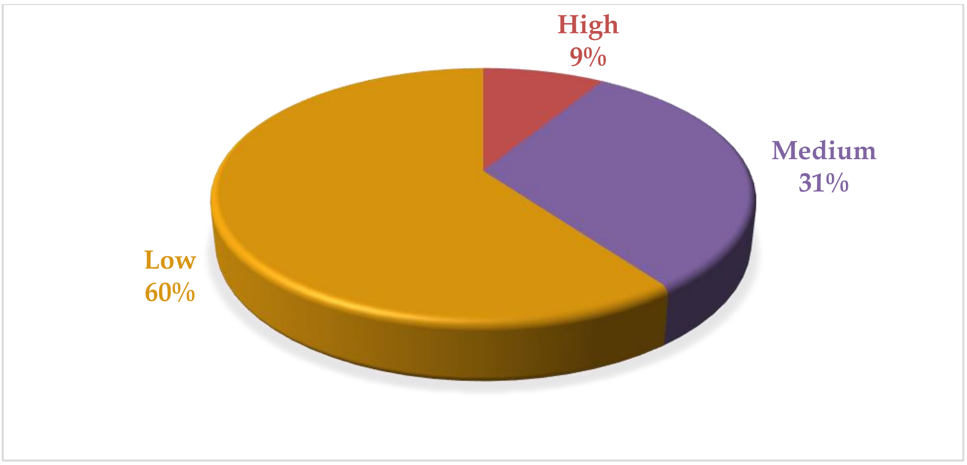


Figure 5.3. Identified issues by severity

It can be seen that roughly 10% of the problems were of high severity, about 30% of the medium and the rest 60% were low. This means that the greatest part of the problems encountered by users were of no or minor significance to usability.

Also, as prescribed by "Think Aloud" method, every participant was asked to rate the application in general on the scale of 1 to 5. The results are summarized on the Figure 5.4.

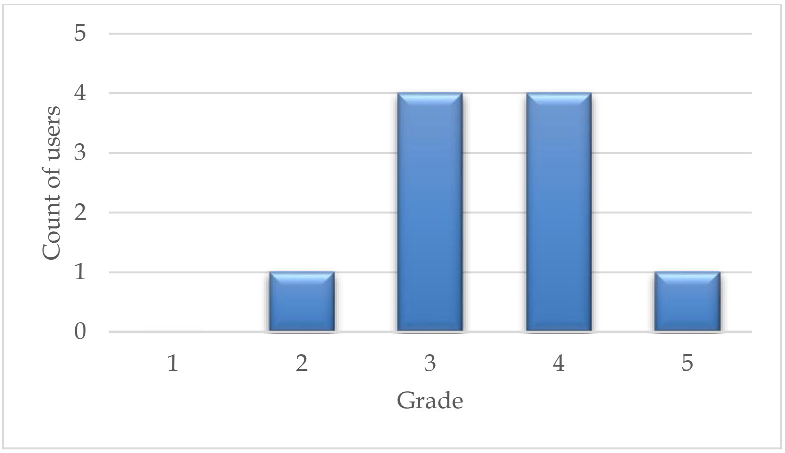


Figure 5.4. User rating

The application has received an average of 3.5 user rating. This is most probably due to the fact that however some design and coding practices were observed, we have not targeted the regular users during the development. Some of the problems, encountered while performing Think Aloud method were because participants did not know how the system works and not many of those studied the manual carefully.

Only fifteen of the encountered issues were mentioned more than once. The complete breakthrough based on the repeat count can be observed on Figure 5.5.

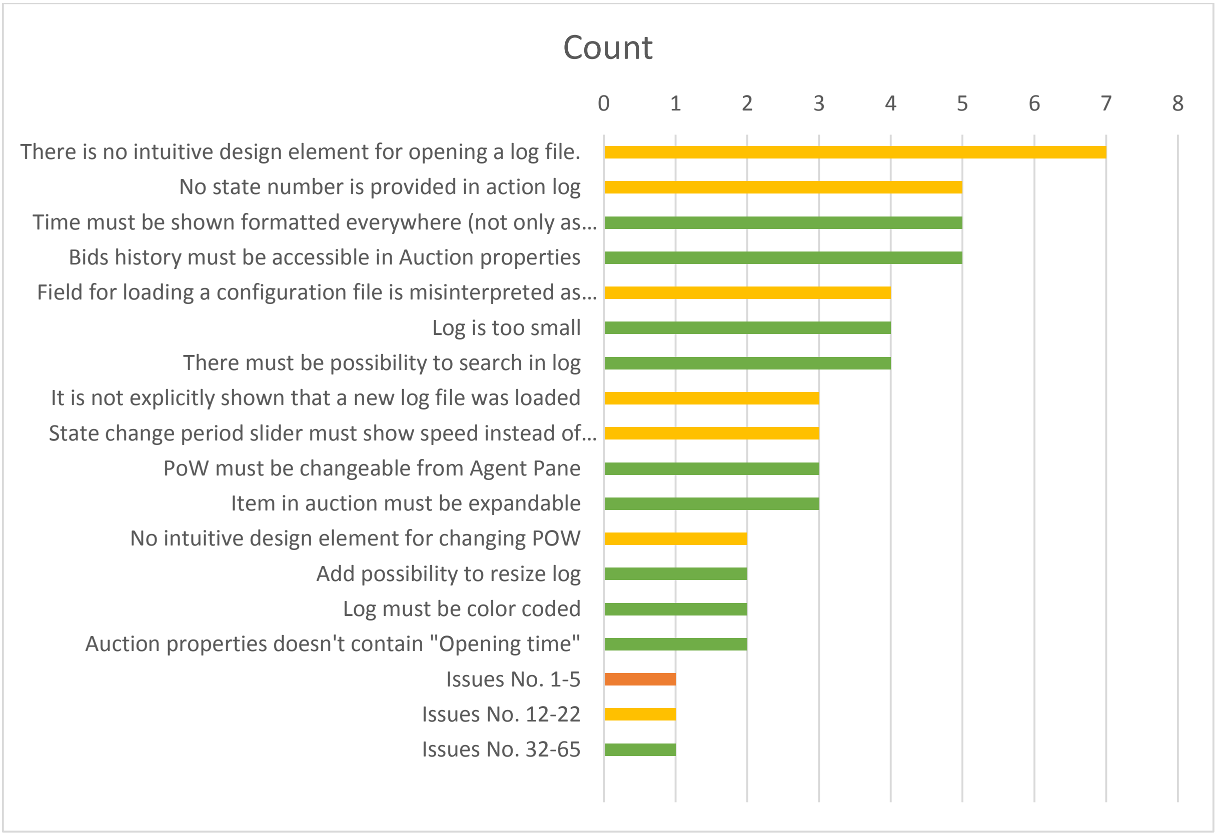


Figure 5.5. Issues found by the repeat count. The left part lists the issue, while the right shows, have many users have reported this issue. The high severity is marked in red, medium in yellow and the low in green.

5.5 Summary

We would dare to say that, however the number of issues identified during the described evaluation may seem overwhelming on the first look, the general results are fairly good. First, none of the issues that were mentioned more than once was of high severity — which means that general functionality of the system is preserved. Slightly more than half of those are of minor severity, which means they have a visual or aesthetic character, not affecting the ability to perform the tasks. And, in the end, the average user rating was 3.5, which means the participants were in general comfortable and happy with the application.

Chapter 6
Future Work

In this chapter, we review the possible improvements and extensions of the developed application, which could improve the user experience and contribute to the usability and usefulness of the visualization module.

6.1.1 Supporting the History of Agent Actions

Probably the most important thing that is lacking from the current implementation of the application is the possibility to observe all of the actions of a particular agent that have taken place on the market during the run of the simulation. In order for being able to reflect those, we shall need to be able to retrieve and process those. Thus we would propose the following:

1. Introduce the changes to the log processor. The main idea underlying these changes is to extract the happening actions and map them to the agent entities, possibly with the possibility to differentiate whether the agent has acted as an active party in the particular transaction.

2. Develop the appropriate classes supporting transaction processing and design the appropriate visual representation for it within the application paradigm.

It can be also handy to give a historic view of actions for an indicated involving the indicated item. The most convenient way of implementing this would be having agent action history class, as well as item ownership history class. The transaction history class could contain the list of the items included in the transaction, list of agents involved in transaction, the difference in the amount they had before and after transaction and the time when the transaction occurred.

The main idea underlying the reasoning in favor of having the last two as separate classes and not as simple attributes is the fact that by separating them we deepen our perspective, as we add the time dimension. If the ownership of an item is stored as item attribute, all we can say about the item in this aspect is that as of the moment is belongs to agent N. However, if we shall have a separate record, we shall be able to tell, for instance, when the item changed hands, and how often, to categorize items in terms of changing ownership, and do all other kinds of analysis, as sophisticated as we want. The questions one might ask can be as complex as "During which time of day the items that are instances of an apple class are more likely to change hands?"

6.1.2 Item Movement History

Due to the fact that we are willing to reflect the rights to the items different in nature from possession of the object, we might also want to additionally have a special class for management of entitlement. Governed by what we said above regarding the historic view, we might also want to have a class for possession of the items.

One possibility to implement that would be to have the command class shall define who has the right to an item. It would contain agent name, item id, effective and expiration time, and, finally, the type of command: possession or entitlement.

6.1.3 Supporting the Reproduction of Data Stream from Network Connection

An addition that might have come in handy would be to save the messages that we have received over network connection to the log file and give the user a possibility to access that via the control elements. This would require continuing listening to socket while visualizing the entries acquired from the log file, switching to the saved messages synchronizing those when we have reached the end of the file, and synchronizing to the socket transmission when possible. This will give a way to re-play the recorded simulation at any arbitrary time from any chosen step and to reproduce it from this log in a future. For such, it would also be convenient to add the additional setting for the saved log file name to the configuration file and configuration object.

6.1.4 Application Interoperability

There is also a possibility to enhance the main engine so it will additionally save the output in the path set in the configuration file and transmit it via the indicated socket in addition to the log saving / transmission it is performing for its own purposes. That way, the visualization module would read from this file or from this socket, respectively, not interfering with the operation of the main engine.

6.1.5 Improved Visibility of the Agent Account

As of per current implementation, the agent account and relevant number are stored along with its other attributes, essentially, with the items held by the agent. As we are certain that the account is inalienable and therefore will not change hands and its holder will remain intact during the whole run of the application, there is possibility to treat is as inseparable property of an agent and move it to the agent frame header. If we are to implement the historical view in the application, the account can be included in these changes. In the manner similar to the one proposed for the agent actions and item ownership, we could introduce account history class (something like the bank statement). Then it can be used for all kinds of analysis and for visualization, for example, for representing the move of the funds on the account during the previous step alongside its current state. Figure 6.1 demonstrates a possible visual implementation of such improvement.

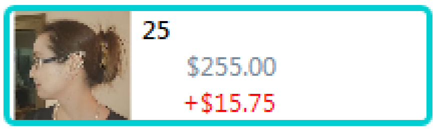


Figure 6.1. Agent frame header containing the current state of its account and the last movement

6.1.6 Supporting a Manual Run Mode

As of the moment, the application runs automatically, in "slide-show" mode in case of running from log file and changing the states upon the arrival of the update in case of socket connection. In order to give user more control over the operation of the application, we can introduce an additional run mode, so the user will select when to switch to another state, and to which one. This change would require the following alterations:

1. Adding the control element for selecting the operation mode (manual / automatic)

2. Introducing the buttons “Next” and “Previous” for switching to corresponding market states. The “Last” button can also be added.

3. Implementing the relevant logic to handle the described changes.

“AdminRights” section and if those are set to “true”, tries to connect to the simulation module at the pre-set host and port. In the future, these can be made available as settings of the same configuration file.

As of the moment, the image can only be assigned to the class, and not to the specific object, but such functionality can be added in the future

Supply the market state with additional information, for example, historical, such as the last transaction, or analytical, like the total budget currently on the market.

Chapter 7
Conclusions

Multi Agent Systems is a still-developing area, and while there are definitely undergoing researches and works in the fields, this area has not seen its zenith yet. However, as the modern development of the electronic commerce presents a growing need for the systems that can operate in dynamic, unpredictable and immense environments, the multi-agent system are being developed, as they can counter these challenges. The ever-present need for reliability and trust in using agents as mediators in the electronic commerce propels the development of frameworks where the agents and their behavior can be observed, tested and analyzed.

However, the visual representation of such systems is also in the early stage of development. Although there are plenty examples of visualization systems, for example, finding their application in a field of molecular biology [25], only several of those focus on the economic environments. And the ones that do tend to focus on the state of environment in general, as opposite to the representation of the agent inner states or explicit representation of their actions. They typically picture the system as a whole from bird's eye perspective, by, for example, picturing the agents of the first group as dots in one color and the other in another [26].

However, it is possible that for at least some of such systems the visual representation could give a new perspective for the problem. Being able to observe the inner state of the agent at any given moment and the preceding actions would allow for transparency of the processes happening inside the system in a concise and efficient manner, thus providing the opportunity to assess their behavior in a controlled environment, set as researcher needs.

In our work, we have tried to give the impression of what such system might look like. Thus, we have implemented an application that receives the input in a form of a sequence of logical statements, describing market states, and represents the agents and their inner state in form of possessed items and properties, during all the steps of the simulation run, with a possibility to observe the steps as they take place, in an online manner. This application is not an integral part of any application, and thus is not dependent on any particular implementation of the simulation marketplace or the agent design. The only thing it requires is that its input would follow the conventions of Prolog-compatible logical statements. This would give the possibility to use it for the visualization of virtually any system, the output of which can be translated into a corresponding format.

Of course, we have provided only the first iteration of such visualization application, with many limitations. We still hope that this application might serve as a basis for future applications capable of reflecting much greater amount of system phenomena and with improved user experience. We have listed the steps that we shall consider taking in this direction under the "Future Work" Section and we are hoping to continue our work on the developed application and improve it by implementing the changes that would seem beneficial after further analysis.

Glossary

Agent. Human or software entity that can perform actions in the market environment, connected to the simulation system, and identified by its ID, issued by simulation system.

Asset. A type of the item in the simulation system that represents the actual “thing” as opposed to right or obligation.

Configuration file. File, describing the current configuration of the visualization application. Contains names of the log and customization files, as well as indication of the administrator rights. Described in section 4.1.2 Input Format.

Customization file. File, containing the visualization customization information, namely, paths to images for item classes and instances, paths to images and names to agents, as well as the classes treated as special items and their group name. Customization file structure is described in section 4.1.2 Input Format.

Environment state. See *Market state*.

Log file. File, containing the entire collection of statements, describing market states of the specific simulation run and used as a source of information. Log file structure is described in section 4.1.2 Input Format.

Market state. Complete description of the market state during the specific time interval, exhaustively listing all items and their attributes.

Master agent. An entity of the simulation engine, embedded in the simulation module. Has the powers to enforce the sanctions upon other agents and check their rights. If an agent leaves the simulation, its items and account are transferred to master agent. Master agent is identified by id “god”.

Reference Panel. Element of the interface, representing the items and agents, not indicated explicitly in the current market state, containing four panes, listing all agents of the simulation, all items of the simulation, special items of the state and common rights and obligations of the state

Right. Type of the item in the simulation system that is an instance of domain class “right(#action, #condition)” and represents the rights of an agent to do #action while #condition holds. Actions of an agent are checked against its rights by master agent and the actions that agent does not have the rights to, are not allowed.

Obligation. Type of the item in the simulation system that is an instance of domain class “obligation(#condition1, #condition2, #sanctions)”. Represents the obligation of an agent to satisfy #condition1 no earlier than #condition2 is satisfied under the threat of #sanctions.

Simulation system. Market environment simulation multi-agent system, written in Prolog and described in Chapter 2 “Existing System and Limitations of Its Operation”.

Simulation system run. A collection of all consecutive states of market environment, stored in the log file.

Special item. Item, belonging to a defined domain-specific class that should be treated in special manner. For English auction domain special items are instances of “english_auction”, for coin domain — instances of “coin” class.

Visualization application. The application developed within a course of this work and intended for visualizing the market states of the simulation run and their changes and providing the user with the set of functions for navigating through it and observing its parts in a handy way.

Bibliography

[01] L. Michael, D. C. Parkes, A. Pfeffer. "Specifying and monitoring economic environments using rights and obligations". Autonomous Agents and Multi-Agent Systems 20(2):158–197, 2010.

[02] W. Ketter, J. Collins, M. Gini, A. Gupta, P. Schrahter. "Tactical and Strategic Sales Management for Intelligent Agents Guided by Economic Regimes". Information Systems Research 23(4): 1263–1283, 2012

[03] C. A. Knoblock. "Building Software Agents for Planning, Monitoring, and Optimizing Travel". In Proceedings of the Eleventh International Conference on Information Technology and Travel & Tourism (ENTER 2004), pages 1–15, Cairo, 2004.

[04] J. Grossklags, C. Schmidt. "Artificial software agents on thin double auction markets: a human trader experiment". In Proceedings of the IEEE/WIC International Conference on the Intelligent Agent Technology, pages 400-407, 2003

[05] C. Koch. In "The Cognitive Neurosciences III" edited by M. S. Gazzaniga, Chapter X "Conciousness". MIT Press, 2004

[06] A. Dorin, N. Geard. "The Practice of Agent-Based Model Visualization", Artificial Life, 20(2): 1–19, 2014.

[07] D. Kornhauser, W. Rand, U. Wilensky. "Visualization Tools for Agent-Based Modeling in NetLogo". In Proceedings of the Agent 2007 Conference on Complex Interaction and Social Emergence, Chicago, 2007.

[08] A. Louloudi, F. Klügl. "Visualizing Agent-Based Simulation Dynamics in a CAVE — Issues and Architectures". In Proceedings of the Federated Conference on Computer Science and Information Systems, pages 651–658, ISBN 978-83-60810-22-4, 2011

[09] A. Grignard, A. Drogoul, J.-D. Zucker. "Online Analysis and Visualization of Agent Based Models". In Proceedings of the 13th International Conference on Computational Science and Its Applications, pages 662—672, Ho Chi Minh City, 2013

[10] M. Kögler. “Simulation and Visualization of Agents in 3D Environments”. Diploma thesis, University of Koblenz and Landau, Faculty of Informatics, 2003

[11] C. Nikolai and G. Madey. “Tools of the Trade: A Survey of Various Agent Based Modeling Platforms”. Journal of Artificial Societies and Social Simulation 12(22):2, 2009

[12] Philip Rutten. “Multi-Agent System Based on Economist Thomas Schelling’s Model for Segregation”. Figure is posted on Supracodex site in “Computational Design” section. Retrieved 01.12.2014. All rights belong to the authors.

[13] A. Artikis, J. Pitt, and M. Sergot. “Animated specifications of computational societies”. In Proceedings of First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS’02), pages 1053–1061, 2002

[14] C.G. Healey, R.St. Amant, J. Chang. “Assisted Visualization of E-Commerce Auction Agents”, Graphics Interface 2001 (GI 2001), pages 201–208 , 2001

[15] M. Esteva, D. de la Cruz, C. Sierra: “ISLANDER: an Electronic Institutions Editor”. In Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '02), pages 1045–1052, 2002

[16] A. Bogdanovych, M. Esteva, S. Simoff , C. Sierra, and H.Berger “A Methodology for Developing Multiagent Systems As 3d Electronic Institutions”. In Proceedings of the 8th International Conference on Agent-Oriented Software Engineering (AOSE’07), pages 103–117, 2007

[17] D. Kornhauser, U. Wilensky, W. Rand. “Design Guidelines for Agent Based Model Visualization”. Journal of Artificial Societies and Social Simulation 12(21):1, 2009

[18] L. Michael, D. Stavrou. “Auctioning License Plate Numbers”. In Proceedings of the 7th Multidisciplinary Workshop on Advances in Preference Handling (M-PREF’13), Beijing, China, 2013

[19] World Wide Web Consortium. “XML 1.0 Specification (Fifth Edition)”. Retrieved 01.12.2014.

[20] A. Nathan. “Windows Presentation Foundation Unleashed”, 2006. ISBN-13: 978-0672328916

[21] Official Website of the International Organization for Standardization (ISO). Retrieved 01.12.2014.

[22] M. W. M. Jaspers, “A comparison of usability methods for testing interactive health technologies: Methodological aspects and empirical evidence”, International Journal of Medical Informatics, 78(5):340–353, 2009.

[23] C. Wharton et al. "The Cognitive Walkthrough Method: a Practitioner's Guide" in J. Nielsen & R. Mack "Usability Inspection Methods"

[24] J. Nielsen “Usability Engineering”, ISBN-13: 978-0125184069, 1993.

[25] P.J. Polack, M.M. Carvalho, T.C. Eskridge. “Visualizing Multi-Agent Systems”. In Proceedings for 2013 IEEE/WIC/ACM International Conference on Web Intelligence, 2013

[26] G. M. Faustino, M. A. de Cerqueira Gatti, C. J. Pereira de Lucena, M. Gattass. “A multi-agent-based 3d visualization of stem cell behavior”. Monografias em Ciência da Computaçâo MCC44/08, Department of Informatics – Pontifical Catholic University of Rio de Janeiro, 2008.

Appendix A

Example of an Environment State

Below we have provided the example of a statement describing market environment state with four agents currently present on the market. As one can see, such statements are lengthy to say the least and hardly readable. As an exercise for a curious reader, I may suggest trying to establish, how many objects are in the possession of agent #23 during the state described by the statement.

```
state([(clock, [(expired_at, 247.8211739519611), (happened_at, 227.51401245221496),
(description, state_instantiation), (instance_of, event)]), (account(23), [(amount, 7),
(owned_by, 23), (held_by, 23), (instance_of, account)]), (apple(23), [(owned_by, 23),
(held_by, 23), (weight, 4), (instance_of, apple)]), (right(36), [(held_by, 23), (owned_by,
23), (instance_of, right(open_auction(_453668, _453669, _453670), (object(_453669),
value(_453669, owned_by, 23))))]), (right(35), [(held_by, 23), (owned_by, 23), (instance_of,
right(create_var(_453707), true))]), (right(34), [(held_by, 23), (owned_by, 23),
(instance_of, right(take_on(obligation(_453739, _453740, jail(23))), true))]), (right(33),
[(held_by, 23), (owned_by, 23), (instance_of, right(_453768, (object(_453774),
value(_453774, [(held_by, 23), (uses, _453791)]), member(_453768, _453791))))]), (right(32),
[(held_by, 23), (owned_by, 23), (instance_of, right(issue_p(right(_453826, (_453830,
_453831)), _453832), (object(_453838), value(_453838, [(owned_by, 23), (instance_of,
right(_453826, _453830))]))))]), (right(31), [(held_by, 23), (owned_by, 23), (instance_of,
right(issue_o(right(_453886, (_453890, _453891)), _453892), (object(_453898), value(_453898,
[(owned_by, 23), (instance_of, right(_453886, _453890))])))]), (right(30), [(held_by, 23),
(owned_by, 23), (instance_of, right(query(_453943, _453944), (object(_453943),
value(_453943, [(instance_of, event), (_453944, _453963)])))]), (right(29), [(held_by, 23),
(owned_by, 23), (instance_of, right(query(_453992, _453993), (object(_453992),
value(_453992, [(instance_of, obligation(_454011, _454012, _454013)), (held_by, 23),
(_453993, _454024)])))]), (right(28), [(held_by, 23), (owned_by, 23), (instance_of,
right(query(_454053, _454054), (object(_454053), value(_454053, [(instance_of,
right(_454071, _454072)), (held_by, 23), (_454054, _454083)])))]), (right(27), [(held_by,
23), (owned_by, 23), (instance_of, right(query(_454112, _454113), (object(_454112),
value(_454112, [(owned_by, 23), (_454113, _454132)])))]), (right(26), [(held_by, 23),
(owned_by, 23), (instance_of, right(_454158, accessible(_454158, 23)))]), (account(21),
[(amount, 4), (owned_by, 21), (held_by, 21), (instance_of, account)]), (apple(21),
[(owned_by, 21), (held_by, 21), (weight, 4), (instance_of, apple)]), (right(24), [(held_by,
21), (owned_by, 21), (instance_of, right(open_auction(_454245, _454246, _454247),
(object(_454246), value(_454246, owned_by, 21))))]), (right(23), [(held_by, 21), (owned_by,
21), (instance_of, right(create_var(_454284), true))]), (right(22), [(held_by, 21),
(owned_by, 21), (instance_of, right(take_on(obligation(_454316, _454317, jail(21))),
true))]), (right(21), [(held_by, 21), (owned_by, 21), (instance_of, right(_454345,
(object(_454351), value(_454351, [(held_by, 21), (uses, _454368)]), member(_454345,
_454368))))]), (right(20), [(held_by, 21), (owned_by, 21), (instance_of,
right(issue_p(right(_454403, (_454407, _454408)), _454409), (object(_454415), value(_454415,
[(owned_by, 21), (instance_of, right(_454403, _454407))])))]), (right(19), [(held_by, 21),
(owned_by, 21), (instance_of, right(issue_o(right(_454463, (_454467, _454468)), _454469),
(object(_454475), value(_454475, [(owned_by, 21), (instance_of, right(_454463,
_454467))])))]), (right(18), [(held_by, 21), (owned_by, 21), (instance_of,
right(query(_454520, _454521), (object(_454520), value(_454520, [(instance_of, event),
(_454521, _454540)])))]), (right(17), [(held_by, 21), (owned_by, 21), (instance_of,
right(query(_454569, _454570), (object(_454569), value(_454569, [(instance_of,
obligation(_454588, _454589, _454590)), (held_by, 21), (_454570, _454601)])))]),
(right(16), [(held_by, 21), (owned_by, 21), (instance_of, right(query(_454630, _454631),
(object(_454630), value(_454630, [(instance_of, right(_454648, _454649)), (held_by, 21),
(_454631, _454660)])))]), (right(15), [(held_by, 21), (owned_by, 21), (instance_of,
right(query(_454689, _454690), (object(_454689), value(_454689, [(owned_by, 21), (_454690,
_454709)])))]), (right(14), [(held_by, 21), (owned_by, 21), (instance_of, right(_454735,
accessible(_454735, 21)))]), (account(19), [(amount, 7), (owned_by, 19), (held_by, 19),
(instance_of, account)]), (apple(19), [(owned_by, 19), (held_by, 19), (weight, 6),
(instance_of, apple)]), (right(12), [(held_by, 19), (owned_by, 19), (instance_of,
right(open_auction(_454822, _454823, _454824), (object(_454823), value(_454823, owned_by,
19))))]), (right(11), [(held_by, 19), (owned_by, 19), (instance_of,
right(create_var(_454861), true))]), (right(10), [(held_by, 19), (owned_by, 19),
(instance_of, right(take_on(obligation(_454893, _454894, jail(19))), true))]), (right(9), -
[(held_by, 19), (owned_by, 19), (instance_of, right(_454922, (object(_454928),
value(_454928, [(held_by, 19), (uses, _454945)]), member(_454922, _454945))))]), (right(8),
[(held_by, 19), (owned_by, 19), (instance_of, right(issue_p(right(_454980, (_454984,
_454985)), _454986), (object(_454992), value(_454992, [(owned_by, 19), (instance_of,
right(_454980, _454984))]))))]), (right(7), [(held_by, 19), (owned_by, 19), (instance_of,
right(issue_o(right(_455040, (_455044, _455045)), _455046), (object(_455052), value(_455052,
[(owned_by, 19), (instance_of, right(_455040, _455044))]))))]), (right(6), [(held_by, 19),
(owned_by, 19), (instance_of, right(query(_455097, _455098), (object(_455097),
value(_455097, [(instance_of, event), (_455098, _455117)]))))]), (right(5), [(held_by, 19),
(owned_by, 19), (instance_of, right(query(_455146, _455147), (object(_455146),
value(_455146, [(instance_of, obligation(_455165, _455166, _455167)), (held_by, 19),
(_455147, _455178)]))))]), (right(4), [(held_by, 19), (owned_by, 19), (instance_of,
right(query(_455207, _455208), (object(_455207), value(_455207, [(instance_of,
right(_455225, _455226)), (held_by, 19), (_455208, _455237)]))))]), (right(3), [(held_by,
19), (owned_by, 19), (instance_of, right(query(_455266, _455267), (object(_455266),
value(_455266, [(owned_by, 19), (_455267, _455286)]))))]), (right(2), [(held_by, 19),
(owned_by, 19), (instance_of, right(_455312, accessible(_455312, 19)))])]).
```

Appendix B

Example of a Visualization Customization File

Below we have provided the example of a customization file, used to determine what images are to be used for the different classes of the simulation, and what names and images are to be used for agents joining the simulation.

```
% ====================================================================
% Visualization Specification
% ====================================================================

visualize([(god, [(image,"..\visualization\agent_images\agent00.jpg"), (name,"Lambent")]),
(19, [(image,"..\visualization\agent_images\03.jpg"), (name,"Lynx")]),
(21, [(name, "Agel"),(image,"..\visualization\agent_images\agent01.jpg")]),
(23, [(name, "Takishima Kei"), (image,"..\visualization\agent_images\agent02.jpg")]),
(default, [(image,"..\visualization\agent_images\default_agent.jpg")])],
[(apple, "..\visualization\item_images\apple.jpg"),
("right(query", "..\visualization\item_images\query.png"),
("right(open_auction", "..\visualization\item_images\open_auction.png"),
("right(issue_o", "..\visualization\item_images\issue_o.png"),
("right(issue_p", "..\visualization\item_images\issue_p.png"),
("right(take_on", "..\visualization\item_images\take_on.png"),
("right(create_var", "..\visualization\item_images\create_var.png"),
("right(_", "..\visualization\item_images\underscore.png"),
("right(place_bid", "..\visualization\item_images\place_bid.png"),
("right(open_auction", "..\visualization\item_images\open_auction.png"),
("right(", "..\visualization\item_images\right.jpg"),
(english_auction, "..\visualization\item_images\english_auction.jpg"),
(obligation, "..\visualization\item_images\obligation.jpg"),
(default, "..\visualization\item_images\item.jpg")]).
special_items((group_name, "Auctions"), (members, ["english_auction", "sealed_bid"])).
```

Appendix C

List of Heuristics for Interface Assessment

Below is provided the list of heuristics that we used in our work, as presented in “Usability Engineering” by J. Nielsens [24]

*Visibility of system status***.** The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

Match between system and the real world. The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

User control and freedom. Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

Consistency and standards. Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

Error prevention. Even better than good error messages is a careful design, which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

Recognition rather than recall. Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

Flexibility and efficiency of use. Accelerators -- unseen by the novice user -- may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

Aesthetic and minimalist design. Dialogues should not contain information, which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

Help users recognize, diagnose, and recover from errors. Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

Help and documentation. Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

Appendix D

Issues Identified during Interface Assessment

D.1 Issues Identified by Think Aloud Method

No.	*Problem Description*	*Count**	*Severity*
1	Active window is not redrawn after the new log file is selected (contains old log file data)	1	High
2	Action log width is too small, so the long lines do not fit	1	High
3	Agent Pane frames do not provide the same functionality as the normal agent frames	1	High
4	Movement of slider before visualization was started should start the visualization (or should not be allowed)	1	High
5	When "Actual" speed is set it will not be changed to other, while one step will not finish	1	High
6	There is no intuitive design element for opening a log file.	7	Medium
7	No state number is provided in action log	5	Medium
8	Field for loading a configuration file is misinterpreted as a field for loading a log file	4	Medium

No.	*Problem Description*	*Count**	*Severity*
9	It is not explicitly shown that a new log file was loaded	3	Medium
10	State change period slider must show speed instead of period	3	Medium
11	No intuitive design element for changing POW	2	Medium
12	Color dictionary is now rewritten every time	1	Medium
13	Agent resize moves them in main window. Hard to follow were agent will be.	1	Medium
14	If agent item is expanded while agent is "appearing", it forces agent frame to minimise	1	Medium
15	The groups state (colapsed / expanded) is not saved when minimising/expanding aent	1	Medium
16	Time, state number and PoW are visually separated	1	Medium
17	No warning that we got out of states available for persons point of view	1	Medium
18	Error message is hard to find	1	Medium
19	Impossible to scroll log while running	1	Medium
20	Should not mark all agents as new on PoW change	1	Medium
21	“Held by” attribute is shown in common rights and obligations group	1	Medium
22	Lag while changing states by slider	1	Medium
23	Time must be shown formatted everywhere (not only as seconds)	5	Low
24	Bids history must be accessible in Auction properties	5	Low
25	Log is too small	4	Low
26	There must be possibility to search in log	4	Low
27	PoW must be changeable from Agent Pane	3	Low
28	Item in auction must be expandable	3	Low
29	Add possibility to resize log	2	Low
30	Log must be color coded	2	Low
31	Auction properties do not contain "Opening time"	2	Low
32	Agent frames colors are too bright	1	Low
33	Empty agent has empty box under him	1	Low
34	If account is empty, display "-" instead of nothing	1	Low

No.	*Problem Description*	*Count**	*Severity*
35	Agent frame size is not changed dynamically to reflect the number of items	1	Low
36	No indication what can be opened, and what can not.	1	Low
37	Minimising/expanding agent slightly changes its size	1	Low
38	Configuration window contains irrelevant data (path to configuration file)	1	Low
39	In browse all file types can be selected (not only the expected ones)	1	Low
40	Visualization does not start after loading the log file	1	Low
41	Hidden path name must be replaced by "…"	1	Low
42	Not intuitive that “Run” starts simulation	1	Low
43	After log visualization is started, “Run” button must be renamed to “Resume”	1	Low
44	Add “next” and “previous” buttons	1	Low
45	"Go to state" must accept enter as well as "Go" button	1	Low
46	Scroll bars in log are not handy	1	Low
47	All history log must be created and shown based on step, you are in	1	Low
48	Log must be separated by border from agents	1	Low
49	Identifiers should be separated from text	1	Low
50	25th agent has different name format and is differently shown in log (not consistent)	1	Low
51	Log has white border (looks bad in PoW)	1	Low
52	PoW must be displayed ad name as well as agent id	1	Low
53	It should be possible to change PoW from PoW in right bottom corner	1	Low
54	Auction must have button for going to the last state of an auction	1	Low
55	Show border on Reference Panel	1	Low
56	Show border for each menu in Reference Panel	1	Low
57	Hidden agents and items must be shadowed instead of coloring gray (including pictures)	1	Low
58	If there is nothing to expand, disable expanding button	1	Low

No.	*Problem Description*	*Count**	*Severity*
59	Slider must be brighter	1	Low
60	On hovering slider and on slider movement state number should be shown	1	Low
61	Slider has small font size	1	Low
62	Slider can be dragged to places where there are no information from the current agent point of view	1	Low
63	Slider does not go to selected point when clicked on that point	1	Low
64	State numbers in slider are not right below state indicator	1	Low
65	Different distance between slider step numbers	1	Low

*Count means the number of participants that mentioned the issue

D.2 Issues Identified by Cognitive Walkthrough Method

No.	*Problem Description*	*Count*	*Severity*
1.	“Save and Exit” can be confused as exit from application	1	Medium
2.	It is hard to see that new log file is loaded	1	Medium
3.	When pressing “Run” again it can be misunderstood as “Run from beginning” instead of “Resume”	1	Medium
4.	Not clearly defined how time must be entered in “Go to” box	1	Medium

D.3 Issues Identified by Heuristic Evaluation Methods

No.	*Problem Description*	*Count*	*Severity*
1.	Not displayed which log file is currently opened	1	Medium
2.	Not displayed that changes in configuration has taken place	1	Medium
3.	It is impossible to move some states back without losing log till current state	1	Medium

No.	*Problem Description*	*Count*	*Severity*
4.	Error messages not explaining what must be done to prevent error	1	Medium

Appendix D

User Manual

E.1 Application Deployment

The solution proposed by us consists of three parts: the code of the solution and executable files, developed in C# with the usage of XAML/WPF, the configuration file in XML format, the customization elements in form of graphic files and customization file.

In order to deploy the application, we would recommend to:

1. put the customization file under /domains folder of DAMMAGE folder
2. put the visualization folder under DAMMAGE folder
3. change the values of the log file and customization file tin the configuration file to the full path to history file and customization file from p. 1, accordingly. Put it somewhere.
4. change the value in the solution-provided file "path_to_conFiguretxt" to the full path of a configuration file from p.4

E.2 Starting the Application

To run the application, one should run the executable module AgentzRebuilt.exe. Upon the start of the application, user can see the main application window, as shown on Figure C.1.1. The full name and path of the log file can be observed in the title. This file is the one, which will be used during the run of the application, unless it is changed. By clicking Config button, one can observe the current path to visualization customization file.

In order to perform the visualization run, user should press the Run or the Follow button in the navigation group section.

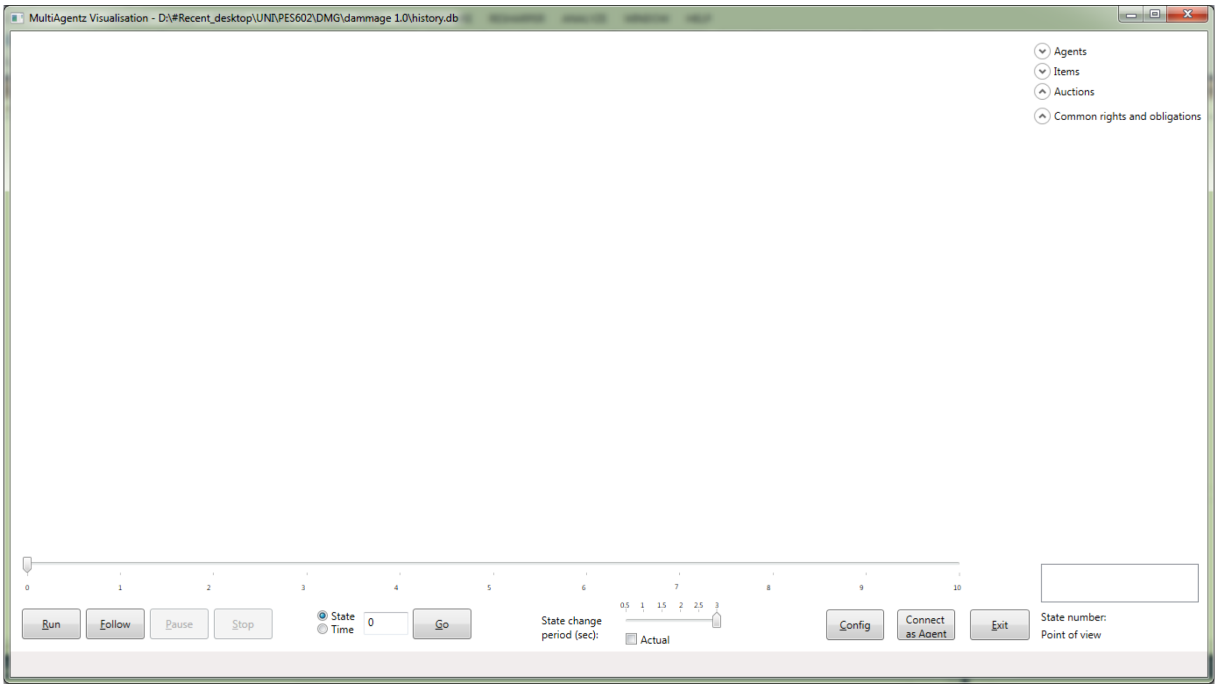
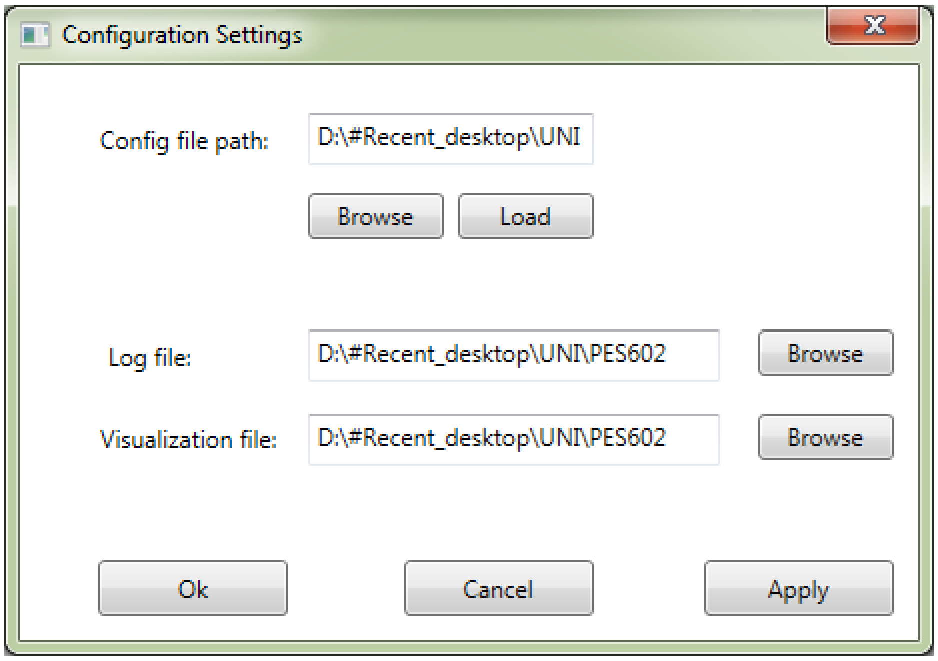

MultiAgentz Visualisation - D:\#Recent_desktop\UNI\PES602\DMG\dammage 1.0\history.db
Agents
Items
Auctions
Common rights and obligations
0
1
2
3
4
5
6
7
8
9
10
Run
Follow
Pause
Stop
State
Time
0
Go
State change period (sec):
0.5
1
1.5
2
2.5
3
Actual
Config
Connect as Agent
Exit
State number:
Point of view

Image E.2.1. Main window of the application, showing the navigation section in the bottom left corner.

E.2.1 Changing Configuration Settings

If instead of running with the present configuration user would like to change configuration settings, s/he can press Config button. The configuration window, as pictured on Image C.2.2, will appear.



Config file path: D:\#Recent_desktop\UNI

Log file: D:\#Recent_desktop\UNI\PES602

Visualization file: D:\#Recent_desktop\UNI\PES602

Image E.2.1.1 Configuration settings window

On this window, a number of relevant settings are available. First of all, it is path, to which the current configuration is saved. By saving settings to different configuration files, user can create profiles for different domains or simply different versions of visualization.

There is also an option to select the path to the visualization customization file, that stores the paths for icon files which are to be used in the visualization for the relevant item classes and agent ids, as well as the names to be associated with agent ids. The format of the file should comply with requirements described in Subsection 4.1.2.2 Visualization Customization File of Section 4.1 Input Format.

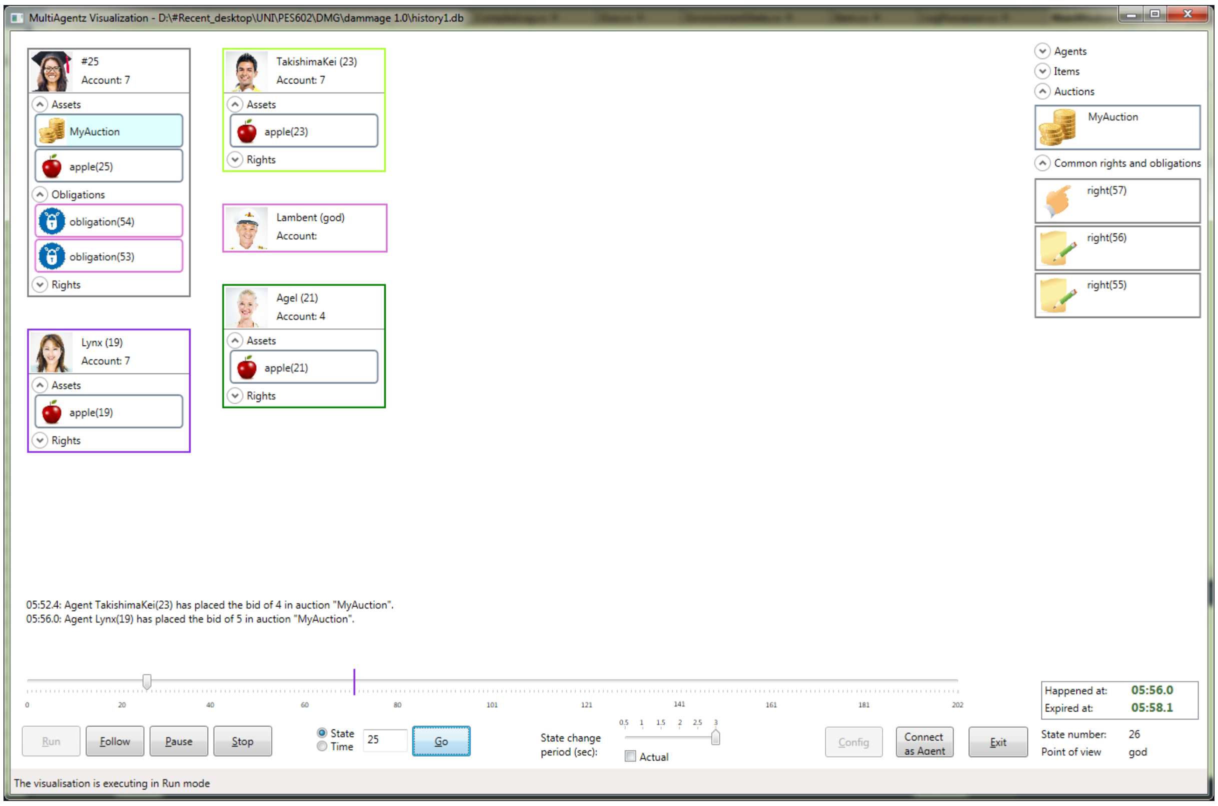
For example, for the element having the name (apple(23), [(owned_by, 23), (held_by, 23), (weight, 6), (instance_of, apple)]) the application would use the path from element definition (apple, "..\visualization\item_images\apple.jpg"). If such were absent, the application would use the path from default element definition, and if such is absent as well — the default item supplied by the application itself.

After setting the desired values to the indicated fields, the configuration can be saved by pressing either "Apply" or "Ok". In both cases, the configuration is saved in xml file in the indicated folder. Pressing "Cancel" discards all the changes made since the last save and closes the window without prompting user about saving those.

E.2.2 Running the Visualization

The Visualization can be run by pressing “Run” or “Follow” button. For that, it shall use the settings as they are set in the configuration. The system checks if such log file exists. In case of being unable locate the file, the application shall give the appropriate message to the user and prompt him to indicate the valid one.

In case of successful connection to the source, in the Run mode, the application starts to depict the virtual market states in the following manner: the states retrieved from the log files shall be reflected in “slide-show” mode, showing the next state after a number of seconds set by State Change Period slider. If the checkbox “actual” is set, then the states will be reflected as they were initially produced. In case of Follow, the application will keep the distance from the end of file and will show the next state when the new state is added to a log file. The distance is set when the Follow button is pressed. The visualization field during the run might look as pictured on image E.2.2.1.



05:52.4: Agent TakishimaKei(23) has placed the bid of 4 in auction "MyAuction".
05:56.0: Agent Lynx(19) has placed the bid of 5 in auction "MyAuction".

State: 25

Happened at: 05:56.0
Expired at: 05:58.1
State number: 26
Point of view: god

The visualisation is executing in Run mode

Image E.2.2.1. Main application window during the run of an application

The application assumes the following depicting conventions:

1. The list of items adjacent to the agent frame header represents the list of items that are possessed by the agent at the current step.

2. Each agent is randomly assigned a unique color that can be seen during all period of the run as border framing the agent frame header.

3. Items currently held by the agent are pictured with neutral grey border if they belong to the same agent and with the color assigned to the other agent otherwise.

4. Clicking on agent or item header collapses / expands the frame.

5. Yellow background marks the new element, blue marks the item, whose attributes has been changed.

6. Grey background marks the removed element (it would be actually removed from the list during the next stage). In the case of Reference panel the grey background marks the elements not present in the simulation environment in the current state.

7. Agents and items are assigned icons and names (only applicable to agents) as indicated in customization file. The name of the special items section and its contents are also defined in this file.

8. Actions performed by the agents are displayed in the action log.

9. Error messages are displayed in red in status bar in the bottom of application window. The current visualization mode is displayed in the status bar in black.

10. Current state, point of view and time values of the current state are displayed in the clock panel in the bottom right corner of an application.

E.2.3 Available Functionality

There are following application control elements:

1. “Run” button, already mentioned before. Starts application execution in Run mode. Also resumes the visualization after it has been paused.

2. “Follow” button. Starts application execution in Follow mode. Also resumes the visualization in the Follow mode after it has been paused. The distance from the end of file in case of resuming is set to the current value.

3. “Pause” button. Pauses the application in the current state

4. “Config” button. It allows the user to change the configuration, as described above.

5. “Stop” button. Stops the visualization run.

6. “Exit” button. Closes the application.

7. “Go” button. Restarts the visualization starting from state or time as provided by the user in the adjacent text box.

8. Control+left click on agent header. Restarts the visualization from the current state from the clicked agent point of view. For it to work, the agent log with the name complying with the proper format conventions should be available in the same directory with the main log file (see Subsection 4.1.3.1 Simulation Log File of Section 4.1.3. Input processing for details). The example of the naming format: for main log with name history.db, agent with id 25 should have the log named history_25.db

9. “Connect as Agent” button. Connects as an agent to the running simulation system. The proxy should be running for it to work. If connected successfully to the simulation, opens a console window, letting the user to enter the commands.

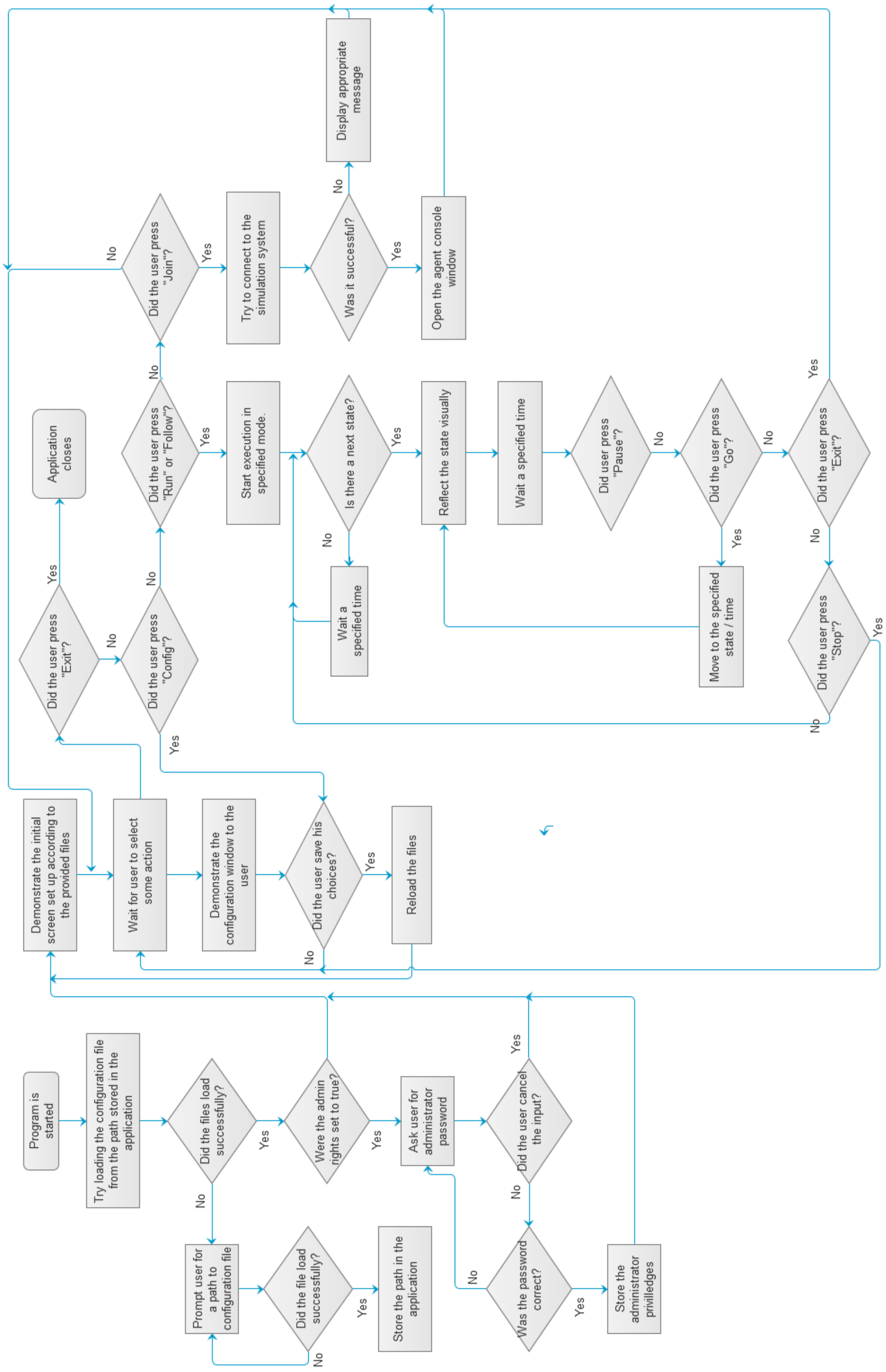
10. “Admin” button. Only available if option “AdminRights” in configuration file is set manually to “true”. In this case user will be prompted for a password upon the start of an application. In this case the application will serve as a replacement of original administration module and should be run after the communication module has been started. By clicking “Admin” button, application opens a console that lets user enter the commands for administration module.

Appendix F

Flow Chart of the Application

F.1 Flow Chart

Here we provide the flow chart of the application. Here one can observe typical sequences of the processes that take place when the application is run.


Program is started
Try loading the configuration file from the path stored in the application
Did the files load successfully?
Yes
No
Prompt user for a path to configuration file
Did the file load successfully?
Yes
No
Store the path in the application
Were the admin rights set to true?
Yes
Ask user for administrator password
Did the user cancel the input?
Yes
No
Was the password correct?
Yes
No
Store the administrator privilledges
Demonstrate the initial screen set up according to the provided files
Wait for user to select some action
Demonstrate the configuration window to the user
Did the user save his choices?
Yes
No
Reload the files
Did the user press "Exit"?
Yes
No
Application closes
Did the user press "Config"?
Yes
No
Did the user press "Run" or "Follow"?
Yes
No
Start execution in specified mode.
Is there a next state?
Yes
No
Wait a specified time
Reflect the state visually
Wait a specified time
Did user press "Pause"?
No
Did the user press "Go"?
Yes
No
Move to the specified state / time
Did the user press "Exit"?
Yes
No
Did the user press "Stop"?
Yes
No
Did the user press "Join"?
Yes
No
Try to connect to the simulation system
Was it successful?
Yes
No
Display appropriate message
Open the agent console window