OpenModelica Interactive Simulation using an OPC UA client

Christoffer Fors Johansson

Supervisors: Adeel Asghar, Martin Sjölund
Examiner: Bernhard Thiele
Upphovsrätt


Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Christoffer Fors Johansson
Abstract

Simulating a model of a complex physical system can be useful in order to obtain a deeper understanding of the system. OpenModelica is an open-source, modeling and simulation environment in which such a simulation could be performed. This thesis presents a design and implementation of an interactive simulation prototype for the OpenModelica Connection Editor. After an investigation of the possibilities in the current OpenModelica environment, several requirements were determined about how interactive simulation should look and work. A prototype was developed based on these requirements. As a result, the prototype can act as a base for applying further functionality to the interactive simulation in both parallel and future works.
Acknowledgments

I would like to thank my supervisor Martin Sjölund for the opportunity to do this thesis. Your constructive comments and shared knowledge throughout the work have been greatly appreciated. I would also like to express my gratitude to my supervisor Adeel Asghar for your expertise and ideas. Your thoughts and comments during our meetings was always great, even if the weather somehow almost always were horrible. Last but not least I would like to thank my examiner Bernhard Thiele for your ability to always be there, solving problems and pointing me in the right direction. A big thank you to all three.

In addition, I would like to acknowledge my opponent Ivar Grahn for your dedication and insightful comments about this thesis.
# Contents

Abstract iii
Acknowledgments iv
Contents v
List of Figures vi
List of Tables viii

1 Introduction 1
   1.1 Justification ............................................ 1
   1.2 Thesis purpose ........................................... 3
   1.3 Research question ...................................... 3
   1.4 Scope .................................................. 3

2 Theory and Related Work 4
   2.1 Simulation and modeling of systems .................... 4
   2.2 Usability ................................................ 5
   2.3 User experience ........................................ 6
   2.4 Evaluation techniques ................................ 6
      2.4.1 Heuristic evaluation ............................... 7
      2.4.2 Cognitive walk-through .......................... 7
      2.4.3 Questionnaires .................................... 7
   2.5 OpenModelica environment .............................. 7
      2.5.1 OPC Interfaces .................................... 9
      2.5.2 OMCompiler ....................................... 11
      2.5.3 OMPlot ........................................... 12
      2.5.4 OMEdit ........................................... 13

3 Method 16
   3.1 Pre-study .............................................. 16
   3.2 Implementation ........................................ 17
   3.3 Evaluation ............................................. 17

4 Results 19
   4.1 Pre-study .............................................. 19
   4.2 Implementation ........................................ 23
   4.3 Evaluation ............................................. 33

5 Discussion 34
   5.1 Results ................................................ 34
   5.2 Method ................................................ 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Source criticism</td>
<td>35</td>
</tr>
<tr>
<td>5.4</td>
<td>The work in a wider context</td>
<td>36</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>38</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Diagram view of a NAND gate in OMEdit. ........................................... 2
1.2 Simulation result, plotted values of the variables $x_1$, $x_2$ and $y$. .......... 2

2.1 Simulation overview. ................................................................. 5
2.2 Usability overview. ................................................................. 6
2.3 An OPC UA client and server communicating over a network. ................. 9
2.4 UaExpert connected to the embedded simulated server, simulating the model Switch. 10
2.5 Running the simulation executable with the flag \texttt{-embeddedServer=opc-ua} from command line. ........................................... 12
2.6 Result red from a .mat file, plotted using OMPlot. ............................. 12
2.7 An overview of OMEdit and options available for the \textit{BouncingBall} model. ........ 13
2.8 Variables Browser, a graphical representation of the red variables in the .mat result file after a successful simulation of the model \textit{BouncingBall}. ........... 14
2.9 Simplified overview of the relationship between the OpenModelica tools OMC, OMEdit and OMPlot. Thin arrows denotes flow control and data. Fat arrows denotes generation. ........................................... 15

4.1 Functionality added to OMPlot. ...................................................... 20
4.2 QwtPlotPicker used during an interactive simulation pause. .................. 20
4.3 UaExpert connected to the embedded simulation server, browsing the time node. 22
4.4 Interactive simulation icon. ......................................................... 23
4.5 Fetched variables from the embedded simulation server presented to the user. ... 23
4.6 OMEdit possible memory management improvement. ........................... 24
4.7 Y-axis auto scale mechanism. ..................................................... 25
4.8 Simulation speed box expanded with suggested values. ........................ 25
4.9 Difference between a standard and an interactive plot window. ................. 26
4.10 Subscription back-end. .............................................................. 27
4.11 Step driven back-end. ............................................................... 28
4.12 Value interaction. ................................................................. 30
4.13 Relationship between the interactive plot window, the variables browser item and the simulation output widget. ........................................... 31
4.14 Interactive simulation option integrated in the simulation setup. ................. 32
List of Tables

4.1 Prototype requirements evaluated one by one. . . . . . . . . . . . . . . . . . . . . . 33
One great benefit of our time is the fast development of technology. One can argue that major progress in areas like robotics, biomechanics, automotive and aircrafts have and will change the way humans live their lives. However, technology itself has in some cases become so complex that sometimes studying how to study is a viable approach in order to fully understand how things work. Some researchers may even ask the question why things work, which has also proven to be a difficult and important question to answer.

For obtaining a deeper understanding of a complex physical systems, simulation can be applicable. Simulation gives the opportunity to investigate situations which might be too dangerous, too expensive or taking too much time or in real life. It is also possible to invent and test completely new, not yet existing systems while avoiding unwanted disruption from the surroundings.

How can one create simulations for complex physical systems? One way is Modelica. Modelica is a declarative, object-oriented, modeling language mainly focused on describing complex physical systems. OpenModelica is an open-source, modeling and simulation environment based on Modelica. The environment aims to provide helpful tools for studying those complex physical systems in detail.

Simulation is not a strict definition. Exactly how a simulation should work and what information to display, as well as when and how, is not regulated. One approach is interactive simulation. The idea is that the user should be able to interact with the simulation while running and display relevant information and do modifications to it.

1.1 Justification

It is fully possible to manage models and run simulations through OMEdit (OpenModelica Connection Editor). A model of a complex system can be simulated. Once the simulation is finished, values of variables are ready to be plotted and fetched to see how they changed during the simulation. Let us take an example from the Modelica Standard Library [13]. Figure 1.1 illustrates a graphical representation of the electrical component NAND.
1.1. Justification

Figure 1.1: Diagram view of a NAND gate in OMEdit.

Depending on the input signals $x_1$ and $x_2$, $y$ will change its value according to the definition of a NAND gate. This situation can also be simulated to show that the model behaves as expected. Figure 1.2 plots the values $x_1$, $x_2$ and $y$ over time. However, at the time when the values are plotted, the simulation is over.

A more intuitive way of learning and studying how a NAND gate works would be to follow the values in real-time. This way gives the opportunity to see when values change and why. Adding an interactive part would be useful as well. When a value is starting to change, the ability to pause the simulation could be helpful to catch up the ongoing scenario.

Figure 1.2: Simulation result, plotted values of the variables $x_1$, $x_2$ and $y$.

Remember that the previous example given is for a very specific electronic component. More complex systems involving different physical mediums may therefore increase the need to interact with the simulation. Furthermore, offering the ability to change a value in the middle of a running simulation and observing how other values take affect could also be an interesting feature for the user.
1.2 Thesis purpose

The current user interface for managing models in OpenModelica, the OpenModelica Connection Editor (OMEdit) has various support for managing models, including support for simulation as well as plotting the simulation results. However, it is not yet possible for a user to run the simulations and interact with them in real-time. The purpose of this thesis is to design and present a real-time interactive simulation prototype for OMEdit. The user should be able to start, stop and pause simulations, change parameter values and display relevant information in a user friendly way.

The research question will address the core essence of this thesis.

1.3 Research question

- \textit{How can interactive simulation of physical system models be designed using an OPC UA client regarding usability?}

Interactive refers to changing parameter values at any given time and plotting selected variables.
Usability refers to the user’s ability to achieve their goals in an effective, efficient and satisfactory way.

OPC UA (OPC Unified Architecture) is a machine to machine communication protocol.

1.4 Scope

OpenModelica is open source which means the tools have to follow the same principles. It already exists an OPC UA server in OpenModelica generated simulations which uses open62541 [20]. This means that the server side will be using open62541, while the OPC UA third party library on the client side could be using any other implementation of the OPC UA standard which is compatible with the open62541 interface.

The focus on the design will be finding ways of integrating interactive simulation functionality as a part of OMEdit. OMEdit is an advanced user friendly GUI and the prototype will be aiming to extend its existing functionality.
This chapter aims to summarize most of the important concepts relating to the thesis. It also include some important definitions and explanations.

2.1 Simulation and modeling of systems

Some important definitions regarding simulation, modeling, experimenting and systems are made in the literature [6]. A good start is to define what a simulation is intended to study. "A system is an object or collection of object whose properties we want to study." [6, p. 3]. Then, what is the plan of studying a system? What do we want to accomplish by studying a system? The idea is to get some output of the systems which might be able to answer questions regarding the system.

By controlling the inputs of a system together with observing the outputs it is potentially easier to make conclusions and learn more about the system. Such a process is called experimentation. "An experiment is the process of extracting information from a system by exercising its inputs." [6, p. 5].

There are a some drawbacks with the experimental approach according to the definition. Since the experiment is operating close to the system, the system itself must be in-place, providing accessible and insensitive inputs. This leads to the need of something wrapping up the system to allow experiments being done harmlessly while still providing interesting outputs. This something can be called a model. "A model of a system is anything an "experiment" can be applied to in order to answer questions about that system." [6, p. 6].

To summarize the chain of definitions, one more is to be made regarding simulations. Simulation is Latin for pretend which is the underlaying purpose of defining a model. "A simulation is an experiment performed on a model." [6, p. 7].

Figure 2.1 illustrates an overview of the previous definitions and how they are related to each other.
2.2 Usability

This thesis is about real-time interactive simulation. That implies time is involved in the simulation as well as interaction. Sometimes, ignoring time dependencies in a system can be helpful to focus on more important issues. This is called a static model while including time into the model is referred to as a dynamic model [6].

Simulation in a virtual environment offers some prosperity compared to experiments in the real world. Real-life experiments can be dangerous, expensive, have bad time scaling (taking years to finish) while virtual simulation offers a clean simulation environment and easy manipulation of models [6]. However, there are also some pitfalls to take into consideration. Forgetting that a simulation is not real or trying to force reality into the simulation [6] can be possible issues as well as making incorrect assumptions or using an invalid model [10].

One can conclude that simulation is heavily dependent on modeling. Looking into how modeling of physical systems can be performed is therefore interesting. One way is to write the modeling code directly in the Modelica language [11]. A more convenient way of managing models is to make use of a graphical modeling tool. There are several commercial tools available. In terms of open source, SimForge [27] and OMEdit are two options available. Nevertheless, one of the reasons for developing OMEdit was because of the SimForge stability and performance issues [1].

2.2 Usability

As previously mentioned, there already exist a GUI for managing models in OpenModelica, OMEdit [2]. As this thesis aims to further extend the GUI with a new feature, some existing interface design has to be taken into consideration. However, user interface design in general with the objective of usability include various aspects. In fact, usability is a wide concept covering some more areas. The ISO guidance on usability reads "A product is usable if the intended users can achieve their goals with effectiveness, efficiency and satisfaction in a specified context of use." ISO 9241-11:1998, Guidance on usability. This is a way of specifying the meaning of usability and identify relevant information when it comes to measuring user performance (effectiveness, efficiency) and satisfaction [3].

Three keywords correlated to the definition of usability is effectiveness, efficiency and satisfaction. One way of decipher usability is to consider those keywords in the perspective of the user experience regarding the system quality. How successful are the users to achieve their goals (effectiveness), within an acceptable time frame (efficiency) in a way which satisfies the users? However, in system development usability tend to be referred to as ease of use, which is a more narrow interpretation of usability [4]. Although ISO 9241-11:1998 has been rather successful in providing a common ground for understanding and applying the term usability, some further understanding has been accomplished after several years of widespread use in both industry and research [5]. One important aspect to look at is the user experience. Bevan, Carter and Harker [5] points out that the subjective feelings and reactions a user gets from a system has been given increasing awareness of the importance. It is worth
2.3 User experience

Since user experience was previously identified as an important aspect, included in the satisfactory part of usability according to Bevan, Carter & Harker [5], it is rather important to specify the meaning. Hassenzahl [7] does it in two different versions. The first one define user experience (UX) as a "...momentary, primarily evaluative feeling (good-bad) while interacting with a product or service." [7, p. 12]. It is quite straight forward to define user experience without any valuation of the quality. The next challenge is to evaluate how the good feeling can be accomplished.

The second definition made by Hassenzahl [7] states “Good UX is the consequence of fulfilling the human needs for autonomy, competency, stimulation (self-oriented), relatedness, and popularity (others-oriented) through interacting with the product or service (i.e., hedonic quality). Pragmatic quality facilitates the potential fulfilment of be-goals.” Here, hedonic quality refers to why someone would use a product. It also refers to the products ability to support accomplishment of “be-goals”, such as “be competent” or “be related to others”. On the other hand, pragmatic quality refers to a products ability to accomplish “do-goals”, i.e. “start a simulation” or “save plotted variables” [7]. Thus, a good user experience is a consequence from fulfilling human needs through interaction with a product or service and i.e., good quality on starting a simulation can assist the progress of being competent.

2.4 Evaluation techniques

Attempts to construct evaluation techniques for usability have been made. This section describes two inspection methods with at least one thing in common, evaluation of user interface usability is preferable including several people in the process. Furthermore, this section describe a usability test method involving end users.

Some of the usability evaluation techniques may be demanding in time, expertise, equipment or user base numbers. There might also be applicability limitations because of the current development phase [8].
2.4.1 Heuristic evaluation

Heuristic evaluation is a technique which has proven to be applicable with good results compared to similar methods [9]. Jeffries et al. described a slightly modified way of performing heuristic evaluation. The main difference between their technique and the one earlier proposed by Nielsen and Molich [15] is that the evaluation is performed by user interface specialists, rather than software developers themselves. One may ask what the requirements are for qualifying as a user interface expert. Nielsen and Molich [15] identify experience as an important factor. Experience in both usability and empirical user tests serves as a good background for evaluating usability.

People with much experience can be both costly and hard to find, can heuristic evaluation be performed with about equal results with less experienced evaluators? Heuristic evaluation performed by less experienced evaluators can be performed, but the result may not be as good. However, depending on the available participants, doing evaluation sessions can still be appropriate [8]. During a heuristic evaluation session, the evaluator goes through the interface several times and inspects different parts. The findings are then compared to specific usability principles, i.e. Nielsen's usability heuristics [14]. It is of importance that the usability principles compared to are suited for the specific system [8].

The idea of heuristic evaluation of a user interface is to identify usability problems in the user interface, sometimes even without providing an advice on how to solve them [15].

2.4.2 Cognitive walk-through

A cognitive walk-through aims to divide the interface into smaller tasks. The variant of the cognitive walk-through used in the comparison by Jeffries, Miller Wharton and Uyeda [9] provided the group of evaluators with selected walk-through tasks. The idea behind this method is to focus on how new users experience the user interface. This can be done by going through each subtask of the interface and for each subtask, asking user experience related questions.

2.4.3 Questionnaires

Questionnaires is a usability test method which intend to query the users after their opinions about the features. It can also fetch information about how users use the system. A user opinion can be of different quality. Therefore, questionnaires is not collecting hard facts about the system. It gather opinions on the user interface rather than evaluating the actual user interface [8].

2.5 OpenModelica environment

The OpenModelica environment is a suite of tools. The whole project is open source and is available on github [21]. But first, let us have a look at the Modelica language [12] in general and how it works. Modelica is a language with several features that makes modeling of complex physical systems convenient. The language is strongly typed. This means that the compiler is more likely to generate an error if, for example, the passed type to an argument does not match the expected type. It is also object-oriented and everything in the language is a class. Furthermore, Modelica is equation-based which means that it is possible for the code generator to convert equations related to the model to a form which is more suitable for solving large scale non-linear equations. Modelica is also a declarative programming language. This means that the programmer is not supposed to explicitly state how the program operates, which is situation for an imperative language such as C/C++. Instead, programming a declarative language focuses on describing what the program should accomplish, rather than how. This means, in a declarative language is how a task is accomplished a part of
the language implementation [11]. Let us study a simple model of a Switch, available in the OpenModelica project [21]. Contemplate the following snippet:

```modelica
model Switch
    Real v;
    Real i;
    Real i1;
    Real itot;
    Boolean open;

    equation
        itot = i + i1;
        if open then
            v = 0;
        else
            i = 0;
        end if;
        1 - i1 = 0;
        1 - v - i = 0;
        open = time >= 0.5;
    end Switch;
```

The model is encapsulated within `model Switch` and `end Switch;`. Variables are declared at first with their data type followed by name. Each variable value is determined by the equations stated after the `equation` keyword. The equation part also contains a flow control structure depending on the boolean variable `open`. A closer look at the line `open = time >= 0.5;` indicates that `open` becomes true as the variable `time` is equal to or greater than 0.5. Time is a variable accessible by all models. However, the value of the variable `open` determines if `v` or `i` is equal to 0. Thus, the following, simple equation becomes interesting:

\[ 1 - v - i = 0; \]

For the equation above to remain correct must the value of the variable not set to 0 be 1. This means that before the time value 0.5, `open` is false, `i` has the value of 0 and therefore, the value of `v` is 1. The switch occurs at the time value 0.5 and remain so as long as time variable value increases. It is also possible to conclude the value of `i1` and `itot` due to the following equation:

\[ 1 - i1 = 0; \]

The value of `i1` will always be 1 to ensure that the equation is correct. Furthermore, `itot` holds the value 1 plus the value of `i`, which has the value 0 when the switch is closed and 1 when it is open. Thus, `itot` holds the value 1 at closed and 2 in open state. How a simulation of the model `Switch` described above is thus not explicitly specified. Instead, the result of a simulation should satisfy the stated equations as well as possible [12].

Some common data types are predefined in the Modelica language. Real, Boolean, Integer, String and enumeration types. Variables declared like in the Switch model example above are declared inside the model and they are assumed to have a solution owing to the equations. Another interesting keyword in the Modelica language is `parameter`. A variable can be declared with the `parameter` keyword as the following example shows:

```modelica
parameter Real gravity="9.82" "gravity acceleration";
```

A parameter is constant during a running simulation but can easily be changed between simulations if it is desirable for the user [12].
The above given example of a simpler Modelica model aimed to give a short review of how a model can work. A reasonable next step is to find out how to work with the model itself. For this thesis, three of the existing tools in the OpenModelica environment are of certain interest and will therefore be examined.

2.5.1 OPC Interfaces

As simulation has been mentioned earlier, it may be useful to further investigate how a simulation can be performed. In OpenModelica a model can be simulated through the OPC interface. It is possible to choose between two similar interfaces, OPC Data Access (DA) and OPC Unified Automation (UA). Both interfaces offer about the same functionality. However, OPC DA is a part of OPC Classic which is an older interface based on Microsoft Windows technology [16]. OPC DA is a standardized way of communicating data in terms of values, time and other quality information, bound to Windows platforms. The communication takes place in a client-server model which means that the OPC server and client communicate over a computer network. Figure 2.3 illustrates this typical scenario with an OPC UA client and a server communicating over a network. It is fully possible to connect several OPC clients to an OPC server. However, the OPC DA server in OpenModelica is, according to the documentation, currently broken. Instead, let us have a closer look at OPC UA which is currently in an experimental state [22].

![Figure 2.3: An OPC UA client and server communicating over a network.](image)

OPC UA is a standardized protocol for industrial communication in the ISO/IEC 62541: 2015, OPC Unified Architecture. The standard aims to define information exchange between clients regardless of the hardware in use. The OPC Foundation [17] released OPC UA in 2008 and it integrates the previous OPC Classic specification functionality into one unified architecture. OPC UA is backward compatible with OPC Classic as well as hardware independent. It can run on several platforms such as Windows, Linux, Android and Mac. In other words, OPC UA can be used as a communication protocol between smaller embedded devices as well as between large network infrastructures. OPC UA can be used in both closed networks and over the internet. Authentication, access control and encryption is built into the protocol for security measures in case it is intended for usage outside a closed network [17].

OPC UA is functionality equivalent to OPC Classic but also extended with further capabilities. For example, it is possible to invoke a Remote Procedure Call (RPC) which makes it possible to call functions and execute programs on the server side, from a client. Another interesting feature is subscriptions. A client can subscribe to a server and monitor interesting data. If the interesting, monitored data item changes, it will be reported back from the server to the client [17]. This feature can therefore reduce the amount of network traffic by only sending data as it becomes interesting for the client. It can be valuable in different aspects in terms of reduced overhead cost for a small embedded device or reduced network traffic for a large scale cloud infrastructure.

Another new functionality added to OPC UA is the ability for a OPC UA client to discover OPC UA servers on a local computer or network, or both. All data on a OPC UA server is represented hierarchy using folders. A folder can contain other folders and data items [17].
Furthermore, an important difference between OPC DA and OPC UA is the address space model [18]. The primary object for the address space is to provide a standardized way for the server to represent objects to clients. It contains metadata about the server as well as data items, referred to as nodes. Each node is assigned to a node class which in turn represent a different object in the object model [18]. Nodes are especially interesting because they are described by attributes. They can also contain information about the relation to other nodes. However, attributes contains information about each node such as the node id, value and data type. They can be accessed by a client using the read, write, query or subscription services [19]. There is also possible to protect attributes of a node from being written by the WriteMask attribute [18]. Can be useful in cases where read only nodes are exposed to clients.

An example can help to obtain further knowledge about OPC UA and how it works. The following example uses UaExpert [28] which is an OPC UA test client with a GUI. Figure 2.4 depicts UaExpert connected to the simulation of the model Switch. The address space to the left contains the exposed nodes in the address space model hierarchy. Attributes related to the variable node step have been read and the response is shown in the attributes area to the right. Several attributes are available as Figure 2.4 illustrates, including the attribute NodeClass which defines that step is a variable node.

![Figure 2.4: UaExpert connected to the embedded simulated server, simulating the model Switch.](image)

Remember, OPC UA is a protocol. The protocol can be implemented in different ways using different languages, techniques etc. OpenModelica’s OPC UA server is implemented in open62541 which is an open source implementation of OPC UA written in C. Palm et al. [23] discusses why an open source implementation of OPC UA is important as well as the reasons for starting the project. As mentioned earlier, OpenModelica is open source which means that the third party libraries available is limited to the open source domain. So it is not possible for an open source project to use closed source libraries, which further emphasizes the reasoning made by Palm et al. [23]. However, the protocol is intended to be hardware and platform independent. A proper implementation of OPC UA should therefore be able
to communicate with clients and servers independent of their actual implementation. Thus, it is possible to choose another third party implementation of OPC UA for the client even if OpenModelica’s OPC UA server is implemented using open62541.

2.5.2 OMCompiler

A central part of the whole environment is the Advanced Interactive OpenModelica Compiler (OMC). One of several areas of responsibility for OMC is to translate Modelica code of models to C code. This process includes several steps such as a translator, analyzer, optimizer and code generator. The code generator outputs C code which can be compiled by a C compiler to generate a simulation executable. Calling the simulation executable is equivalent to running the simulation. The simulation executable is typically linked with numerical solvers for the numerical integration of Ordinary Differential Equations (ODE) or Differential Algebraic Equations (DAE) [22]. This can be useful if the nature of the equations requires ODE or DAE to be solved [22].

The generated simulation executable can be called with a number of different flags. Flags can be used to specify a value or set an option. This applies to both OMC and the simulation executable. For example, models can be simulated using different integration methods. Depending on the model, a simulation can benefit from choosing a suitable integration method since different methods can be good at solving different type of equations. The default solver is dassl, which is able to solve equations of the order 1 to 5. However, the integration method can be changes by using the flag \(-s=value\) or \(-s value\), were value is the name of the desired solver [22]. The following example shows a command for how the simulation executable of the model BouncingBall can be executed using the euler integration method:

```
./BouncingBall -s=euler
```

A more detailed overview over all the available integration methods and other flags as well, regarding both OMC and the simulation executable are available in the OpenModelica User’s Guide [22].

As mention earlier in section 2.5.1, a model can be simulated using the OPC UA interface. This can be controlled similar as before with the help of flags. Running a simulation executable generated by OMC with success and without defining how the model should be simulated, will result in an output file with the simulation data. This particular data can be presented with the help of the 2D plotting tool OMPlot [22]. However, running the executable with the flag \(-embeddedServer=value\) or \(-embeddedServer value\) results in the simulation instead being using en embedded server. Value can be opc-ua, opc-da, filename or none. The value filename can be a path to a shared object implementing the embedded server interface. Furthermore, the value opc-da or opc-ua will as expected enable an embedded simulation server using the corresponding OPC UA interface over the standard port 4841 [22]. For this thesis, the value opc-ua is of certain interest.

Figure 2.5 illustrates a running OPC UA embedded simulation server. The server was started from the command line using the following command:

```
./BouncingBall -embeddedServer=opc-ua
```
2.5. OpenModelica environment

Figure 2.5: Running the simulation executable with the flag `-embeddedServer=opc-ua` from command line.

Enabling an embedded simulation server means that simulation flow controls are required. The client has to contact the server to somehow make the simulation run. There are two ways of controlling the simulation. RPC and writing values to variables via OPC UA. A client can use the OPC UA interface for writing values to specific simulation control nodes on the server side or call remote functions. For example, asking the OPC UA server to take a step forward in the simulation through RPC would require the client to call the procedure `step()`. The other approach for the client would be to write the value `true` to the value attribute associated with a simulation control node. Currently, the embedded OPC UA simulation server supports the writing to simulation control nodes way of simulation flow control. The reason behind is that OPC DA do not have support for RPC and some commercial systems are just a bridge between OPC UA and OPC DA. Taking the approach supported by both OPC UA and OPC DA could therefore include a wider scope [22].

2.5.3 OMPlot

The results from a successful simulation with the embedded server disabled are stored in a result file. A simulation result file can be visually represented with help of OpenModelica’s 2D plotting tool OMPlot. Currently supported simulation results output formats are `.mat`, `.csv` and `.plt` [21].

OMPlot is implemented using the third party library Qwt [26]. A typical plot situation is illustrated in Figure 2.6, which plots the result from a `.mat` file generated by a three seconds long simulation of the model `BouncingBall`.

![Figure 2.6: Result from a .mat file, plotted using OMPlot.](image)

The plotted curve is based on data points and a line is drawn between each set of points. This means that the amount of data points can have an affect the smoothness of the curve. Too few data points can therefore make the curve appear jagged. However, this can be adjusted by
setting the interval number of a simulation. Figure 2.6 was plotted using the default interval value of 500. Note that the simulation interval is used by the selected integration method, which means that there can be a difference between the number of internal data points and points in the result file [22].

2.5.4 OMEdit

The tool used for managing models, running simulations and plotting the results is the advanced GUI OMEdit. It also supports the Modelica Standard Library which is a part of a OpenModelica installation. OMEdit has support for creating and designing models. These models can be checked and compiled using OMC. A successfully compiled model can also be simulated through the GUI [22]. This is depicted in Figure 2.7, which shows one way of simulating the model BouncingBall. Furthermore, this section contains a brief description on how OMEdit currently is implemented and working.

![Figure 2.7: An overview of OMEdit and options available for the BouncingBall model.](image)

Under the hood, OMEdit uses the same technique for starting a simulation as described previously in this chapter. Simulation settings set through the GUI are collected and passed to the executable as flags. The code example below illustrates this phenomenon [21]. A list of strings with flags and values is created:

```cpp
// setup simulation flags
QStringList simulationFlags;
simulationFlags.append(QString("-override =%1=%2,%3=%4,%5=%6,%7=%8,%9=%10,%11=%12,%13=%14")
    .arg("startTime").arg(simulationOptions.getStartTime())
    .arg("stopTime").arg(simulationOptions.getStopTime())
    .arg("stepSize").arg(simulationOptions.getStepSize())
    .arg("tolerance").arg(simulationOptions.getTolerance())
    .arg("solver").arg(simulationOptions.getMethod())
    .arg("outputFormat").arg(simulationOptions.getOutputFormat())
    .arg("variableFilter").arg(simulationOptions.getVariableFilter()));
simulationFlags.append(QString("-r=").append(simulationOptions.getResultFileName()));
```

13
Flow control structures can append several flags similar based on the configuration set by the user. However, these flags together with the name of it is later used to run the simulation executable. The following code shows the main part of the code responsible for starting the simulation, where `fileName` is the name of the targeted executable and `args` is a String list of arguments, containing all the flags [21].

```cpp
mpSimulationProcess = new QProcess;
.
.
.
mpSimulationProcess->start(fileName, args);
```

When a simulation is finished, the process int the snippet above is terminated and OMEdit reads the result file. Variables in the file gets inserted into the Variables browser and appears in the Variables browser in the OMEdit plotting page, depicted in Figure 2.8.

![Variables Browser](image)

Figure 2.8: Variables Browser, a graphical representation of the red variables in the .mat result file after a successful simulation of the model `BouncingBall`.

From here, results can be plotted on the plotting surface as desired by the user. Results and variables from different result files can be plotted in the same plotting window and there currently exists two types of plotting windows, ordinary and parametric. Several new plotting windows of both types can be added and switched between. However, the amount of simultaneously displayed plotting windows is limited to one. Selected variables in the variables browser is connected with their targeting plot window. Switching between plotting windows means that the selected variables changes correspondingly [21].

OMEdit contains many interesting functionalities regarding several aspects of a simulation. Creating and managing a model for it, running it as well as presenting the results. This is possible with the help of other tools in OpenModelica. Figure 2.9 shows a simplified overview of the relationship between OMC, OMEdit and OMPlot. OMEdit is dependent on OMPlot but OMPlot is not dependent on OMEdit [21]. Thus, OMPlot could be used to plot simulation result files without OMEdit.
Figure 2.9: Simplified overview of the relationship between the OpenModelica tools OMC, OMEdit and OMPlot. Thin arrows denotes flow control and data. Fat arrows denotes generation.
This chapter aims to describe how the work was planned as well as the different methods applied in the different stages of this thesis. Depending on the reader’s background, simulation of physical models might be more or less known. Regardless of background, the focus in this thesis is to provide an answer to the research question:

- *How can interactive simulation of physical system models be designed using an OPC UA client regarding usability?*

The nature of the question supports a divided approach to it. First of all, a pre-study is applicable in order to understand both the wider perspective as well as the specific context. A prototype is about to be implemented together with OMEdit using an OPC UA client, how does OMEdit and OPC UA work and how is it currently implemented?

Second, the prototype should be usable. The theory behind usability might have an impact on the implementation as well as limitations or possibilities in the context. This leads up to the need of a structured approach to the implementation phase.

Third, the prototype should be evaluated. The evaluation section of this chapter describes the method used for evaluating the prototype.

### 3.1 Pre-study

The pre-study phase in this thesis will include four major areas, the existing OMEdit implementation, the nature of simulation, needs of users and what OPC UA is. It is also interesting how OPC UA works. Creating something useful for the end users will be dependent on a good understanding in all four subjects.

This led to the decision to study related research in the areas of model driven simulation and usability because of the relation between this thesis and those areas. When a broader perspective of simulation and usability were accomplished, tools in the OpenModelica environment related to this thesis was investigated in further detail. Studying simulation and OpenModelica also resulted in further reading about the Modelica language in general as well as the purpose of it.

The next step was to get more familiar with the concrete implementation of OMEdit. This led up to the need of a further investigation of tools connected to the OMEdit source.
3.2 Implementation

Implementation of the prototype was primarily done by focusing on meeting the requirements set in the pre-study phase. Before any implementation could be made the tools had to be further examined.

Implementing the small test program described in the pre-study section ended up to be a time consuming task. The initial thought was to start exploring qtopcua and Free OPC-UA because of their C++ interface. The main problem was the lack of documentation. Another problem while testing Free OPC-UA were also detected. The provided example client was able to connect to the example server. However, changing the clients remote address to OpenModelica’s embedded server turned out to cause problems and the client were not able to connect. The cause of the problem were not found during the limited time spent on debugging before the decision to proceed with an OPC UA client implemented using open62541 were made.

Furthermore, the small test program was implemented using open62541 with the rather old C interface. An OPC UA server could be contacted and the client was able to fetch variable values from the server. The small test program could also, after further development, write variables values to the server as well as plot a simple sinus curve in real-time using the Qwt library.

Once requirements on the prototype were decided, they were implemented. The implementation was done in C++ using Qt, OMPlot, Qwt and open62541.

3.3 Evaluation

The approach taken in this thesis was to implement the prototype based on the requirements. Implementation was performed when they were decided. A natural step in the evaluation code. OMEdit is based on C++ and the Qt library. It is dependent on OMPlot which is also developed in C++ using the plotting library Qwt [26]. In order to obtain a good knowledge of the possibilities and limitations of primarily Qt and Qwt, different code examples were studied. Furthermore, reading the OpenModelica User’s Guide [22] gave important hints on how to enable the embedded OPC UA server.

In order to get an idea of what OPC UA is, the OPC Foundation’s website [17] was used. Furthermore, an open-source implementation of OPC UA had to be chosen. Implementations considered were the Qt module qtopcua [25], as well as the libraries Free OPC-UA and open62541 [20]. Example code and available documentation were determination factors for deciding a suitable library to continue to work with.

To get more experience with the Qt library and acquire the required knowledge, small code examples from the available documentation were studied. Further knowledge was obtained by continuing to write small programs and exploring different features of the library.

Before proceeding with the implementation phase, a small test program using Qt, Qwt and a third party was developed. The purpose of the program was to get a deeper understanding of how the libraries could be used together and also investigating plotting possibilities in Qwt. The small test program was developed with the Integrated Development Environment (IDE) Qt Creator [24]. The reason was to familiarize with the environment since OMEdit was developed with the same IDE.

When further knowledge about the existing tools and libraries was obtained a natural step was to examine the prototype look. At first, the design was outlined on a piece of paper. This paper was used as a base for providing ideas to the prototype requirements. Then, prototype requirements were added after consultation with the supervisors. Requirements were decided before they were implemented.
phase of this theses was to examine if the prototype fulfilled the requirements. The list of requirements were examined one by one.

Requirements were evaluated with the help of different, smaller example models available in the OpenModelica project and the Modelica Standard Library.
4 Results

This chapter presents the results. The chapter structure follows the method chapter to make the connection between method and result easier to follow.

4.1 Pre-study

The pre-study phase resulted in knowledge about existing tools as well as areas where new functionality could be added. This information together with knowledge from related work is the basis for the following subset of requirements:

1. Selected variable values should be visually displayed during simulation.
2. Include interactive simulation controls such as play, pause and simulation speed and make them tied to the plotting window.
3. Support for both subscription and step driven back-ends.
4. Value interaction, changing parameter values during a simulation.
5. Allow multiple interactive simulation instances with appropriate instance management.
6. Integration of interactive simulation: differentiate interactive simulation from existing tools.

Meeting these requirements meant that OMPlot had to be extended with a suitable real-time plot possibility. At the moment, OMPlot included support for plotting from results files. Since interactive simulation values does not utilize files in the process OMPlot should be able to plot values directly as they appear in main memory. Instead, values will be fetched from the server directly to the main memory and eventually be plotted with help of the added functionality in OMPlot.

OMPlot was therefore extended with functionality to provide OMEdit with positions to data vectors as a variable gets selected for plotting. This is illustrated by Figure 4.1. The red arrows indicate added functionality to OMPlot.
Storing incoming variable values in vectors provided by OMPlot has the following advantage. The QwtPlotPicker used in OMPlot is based upon these vectors. Interactive simulation will be using the same mechanism as plotting from a result file would. Figure 4.2 shows how QwtPlotPicker works. A small window with information pops up as the mouse pointer moves across the curve. However, this mechanism will cause Qwt to redraw the plot. Redrawing the plot can be a rather expensive operation and should not be done if not necessary. Therefore, will the QwtPlotPicker feature only be active during simulation pauses.
OMEdit utilizes the axis vectors illustrated in Figure 4.1 when a user wants to plot a variable. The following code example shows how a PlotCurve object is created and a data sample attached to the curve. Finally, the vectors used for temporary variable value storage is returned to OMEdit. They will be appended with results as they eventually become available:

```cpp
QPair<QVector<double>*, QVector<double>*> plotInteractive(PlotCurve *pPlotCurve) {
    QString variableName = mVariablesList.at(0);
    pPlotCurve = new PlotCurve(mInteractiveModelName, variableName, getUnit(),
                                getDisplayUnit(), mpPlot);
    // clear previous curve data
    pPlotCurve->clearXAxisVector();
    pPlotCurve->clearYAxisVector();
    mpPlot->setSamples(mpInteractiveData);
    mpPlot->addPlotCurve(pPlotCurve);
    mpPlot->replot();
    return pPlotCurve->getAxisVectors();
}
```

Making the continuously appended points appear to the user without issuing expensive replot operations is the motivation behind using the class QwtPlotDirectPainter. Every curve instantiates its own version of the class. The code example below illustrates how the member function `drawSeries` is utilized to paint each attached curve in the list incrementally.

```cpp
void updateCurves() {
    for (auto & p : mpPlot->getPlotCurvesList()) {
        // append the last point to the plotting curve
        p->getPlotDirectPainter()->drawSeries(p, p->getSize() - 2, -1);
    }
}
```

The further investigation of OPC UA led to the finding of the general purpose OPC UA test client UaExpert [28]. Figure 4.3 illustrates the GUI of UaExpert when connected to the embedded simulation server. The address space area to the left reveals variables exposed to the OPC UA interface. Information about a variable such as its name, if it is writable or not, data type and value can be fetched and is also shown by the attributes area to the right in Figure 4.3.

Furthermore, the pre-study showed that parameters were not exposed by the embedded simulation server. If desired, it is possible to expose them by adding support for it on the server side and dismiss write actions to the parameters. This is important because the OPC UA interface instantiates the model before OPC UA commands can be used.

The embedded simulation server exposes more than just the variables themselves via the OPC UA interface. Nodes for steering the simulation is and also should be available as well. For example, writing the value `true` to the `step` variable causes the embedded simulation server to take a time step and calculate new variable values for the current step. In other words it is possible to move a simulation forward by stepping through all the time steps one by one. If the possibility of moving backwards in simulations is desired, it is something that also has to be implemented as a possible control node on the server side.
According to the requirement number 5 earlier in this section the user should be able to run several interactive simulations simultaneously. The pre-study showed that running the simulation executable with the flag `-embeddedServer=opc-ua` or `-embeddedServer opc-ua` will enable the embedded simulation server over the TCP port 4841. Thus, spawning more than one embedded simulation server will cause the servers to interfere over the same port. The solution was to add another flag to the simulation executable. `-embeddedServerPort=value` or `-embeddedServerPort value` specifies the port number used by the embedded simulation server. This flag makes it possible to run several simulation executables with embedded servers without causing the server to interfere. A code example is presented below to illustrate how the flags work together:

```c
int port = 4841;
/* If an embedded server is specified */
if (dllHandle != NULL) {
    if (omc_flag[FLAG_EMBEDDED_SERVER_PORT]) {
        port = atoi(omc_flagValue[FLAG_EMBEDDED_SERVER_PORT]);
        /* In case of a bad conversion, don’t spawn a server on port 0...*/
        if (port == 0) {
            port = 4841;
        }
    }
}

The port number will later be passed as a parameter to the embedded server initialization function which in turn uses it to make the server configuration:

```c
state->nl = UA_ServerNetworkLayerTCP(UA_ConnectionConfig_standard, port);
```
4.2 Implementation

In order to identify an interactive simulation in OMEdit, an icon was created. The icon is depicted in Figure 4.4.

![Interactive simulation icon](image)

Figure 4.4: Interactive simulation icon.

Requirement number 1 is about visually displaying variable values and states:

- **Selected variable values should be visually displayed during simulation.**

The user should therefore be able to somehow select variables to be plotted during the simulation. Figure 4.5 shows how the existing variables browser is used to expose interactive simulation variables red from the embedded server to the user. The motivation behind is that the user is previously familiar with managing variables in this way and it is natural to gather variables of different kinds in the variables browser.

![Fetched variables from the embedded simulation server presented to the user.](image)

Figure 4.5: Fetched variables from the embedded simulation server presented to the user.

Variables can be selected and deselected at any time during a simulation. Only selected variables will get their data vectors appended with new values. A deselected variable will be removed from the plotting surface. However, the memory used will not be freed. This is also the case when variables are plotted from results files. Reusing memory from deselected variables is therefore a possible area of improvement for OMEdit. Let us look at an example below. Figure 4.6 illustrates this issue by showing a normal plot situation in Figure 4.6a. Deselecting the variable \( y \) will remove the curve from the plotting surface as expected. The values will however still be present in main memory as depicted by the blue dashed line in Figure 4.6b.
4.2. Implementation

(a) Plotting the selected variables $x$ and $y$.

(b) Unreleased memory after deselection of the variable $y$. The blue dashed line represents the values of variable $y$ still stored in main memory.

Figure 4.6: OMEdit possible memory management improvement.

Another problem associated with requirement number 1 is to make sure that the plot curves fits within the plotting surface. The problem arises because values are calculated gradually. This means that the $y$-axis should be scaled to fit all the curves on the plotting surface when needed. Again, this should also be done with avoidance of the expensive replot operation in mind. The solution was to keep track of the global minimum and maximum value for all curves on the plotting surface and issue a replot only if the greatest or lowest value is out of bounds. Figure 4.7 illustrates how the plotting surface behaves as a curve is about to exceed the current lower bound. The red curve is about to exceed the lower bound in Figure 4.7a and Figure 4.7b illustrates how the plotting surface reacts immediately after the replot.

The scaling is done to avoid a situation where a replot is issued in every time step and also with respect to that the value can quickly be changed.
4.2. Implementation

(a) Curve approaching the lower boundary.  (b) Plotting surface after exceeding the lower boundary, causing a replot.

Figure 4.7: Y-axis auto scale mechanism.

Requirement number 2 is about how the user is able to interact with the simulation process and states:

- Include interactive simulation controls such as play, pause and simulation speed and make them tied to the plotting window.

The start button makes the sampling thread to run while the pause button puts the thread to sleep. A sampling thread runs a procedure for fetching values and plotting them at a given interval. This interval can be adjusted by the speed control mechanism. The code example below describes how a speed value is interpreted by the prototype:

```c++
void OpcUaWorker::setSpeed()
{
    bool isFloat = true;
    double value = mpSimulationSpeedComboBox->lineEdit()->text().toDouble(&isFloat);
    if (isFloat && value > 0.0) {
        setInterval(mSampleInterval / value);
    }
}
```

A high value shortens the time interval between each call and makes the simulation run faster. The other way around, a small value increases the time and causes the simulation to run slower. `mSampleInterval` is pre-set to 100 milliseconds. Speed values between 0.01 and 100 is allowed, making one step every millisecond the fastest possible interactive simulation speed. Figure 4.8 shows how the simulation speed box looks when expanded.

Figure 4.8: Simulation speed box expanded with suggested values.

It is required that the controls are tied to a simulation plot window. This is implemented by injecting the controls to the plot window if and only if a model is simulated using interactive simulation. Figure 4.9 illustrates how an interactive plot window differs from a
4.2. Implementation

standard plot window. The controls available in Figure 4.9a are the same for both standard and parametric plot windows while the controls in Figure 4.9b are exclusive for interactive plot windows.

(a) Functionality of a standard plot window.

(b) Simulation flow controls tied to an interactive plot window.

Figure 4.9: Difference between a standard and an interactive plot window.

The type of plot window is set at the creation stage. For example, a model is simulated using interactive simulation which causes OMEdit to request an interactive plot window from OMPlot. This is done by setting the requested plot window type in OMEdit and let OMPlot later examine the plot type and from there, make appropriate calls. The code example below illustrates how OMPlot examines the plot type and how the program flow structure was appended with support for interactive plot windows:

```cpp
//Plot
if(plotType.toLower().compare("plot") == 0)
{
    setPlotType(PlotWindow::PLOT);
    plot();
}
else if(plotType.toLower().compare("plotall") == 0)
{
    setPlotType(PlotWindow::PLOTALE); 
    plot();
}
else if(plotType.toLower().compare("plotparametric") == 0)
{
    setPlotType(PlotWindow::PLOTPARAMETRIC); 
    plotParametric();
}
else if(plotType.toLower().compare("plotinteractive") == 0)
{
    setPlotType(PlotWindow::PLOTINTERACTIVE); 
    plotInteractive();
}
```

Furthermore, requirement number 3 states:

- **Support for both subscription and step driven back-ends.**

The user can choose the desired back-end between subscription or step mode. Subscription is the default back-end because of the possible lower overhead. The more demanding back-end, simulation using steps, is therefore possible to enable with a short explanation about the pros and cons. A more detailed description of the subscription and step driven back-end is presented further in this section.

Contact with the remote embedded simulation server is implemented using a separate thread. Each client object has its own thread which is responsible for contacting the server and fetch variable values. The thread’s tasks varies depending on the used back-end. This is implemented using a common flow control structure.
Let us start with having a look at how simulation with subscriptions is implemented. The client starts by adding itself as a subscriber to the server. As the user selects and deselects variables, so called monitored items will be added and removed. When the publish request is sent to the server, monitored items on the server will get their new values sent to the client. Values of variables will only be sent back to the client if they have changed from the previous time step. This means that variable data will travel from the server to the client only if necessary. Figure 4.10 illustrates an interactive simulation using subscription. The numbers in Figure 4.10 are explained in detail below.

1. Subscription created.
2. monitoredItems can be added and removed during the simulation.
3. A publish request is sent to the server.
4. Changes to monitoredItems is sent to the client. The latest value gets updated.
5. The latest value is used to populate the curve data vectors.

Subscription as back-end entails low overhead. A possible downside is that some values might be missed. The reason is that the simulation background thread, running on the embedded server reads the copied values with a fixed time step. If a change is detected, the client gets notified about it at the next publish request. However, it might be the case that a value changes more than once during before the server checked the copied value and sends it over to the client. This leads up to the other approach taken, step driven simulation. The server will be contacted in each time step and explicitly ask for the value of all the selected variables. Figure 4.11 illustrates how step driven interactive simulation was implemented. Numbers in the Figure are explained in detail below.
4.2. Implementation

Figure 4.11: Step driven back-end.

1. Client tells the server to take a step.
2. Client contacts the server for each selected variable and reads the value.
3. The read value and the current time gets inserted to the plotting vectors.

The following snippet is the sample function used by the separate, OPC UA worker thread. It shows the flow control structure depending on the back-end as well as function calls to the different tasks:

```cpp
void OpcUsWorker::sample()
{
    // draw selected curve lines
    emit sendUpdateCurves();
    if (mSimulateWithSteps)
    {
        // append the new values
        appendVariableValues();
        // step the simulation by contacting the remote server
        stepSimulation();
    }
    else
    {
        UA_Client_Subscriptions_manuallySendPublishRequest(mpParentClient
            ->getClient());
        insertValues();
    }
}
```

Simulation using steps calls the function `appendVariableValues()` which will contact the remote for each selected variable and append the results to the corresponding data vectors, as expected. `stepSimulation()` writes the value `true` to the attribute of the simulation control node `step`. It is implemented using the high level functionality of open62541 and works as follows:
4.2. Implementation

```c
void OpcUaWorker::stepSimulation()
{
    // set the step variable to true
    UA_Variant *newValue = UA_Variant_new();
    UA_Boolean run = true;
    UA_Variant_setScalarCopy(newValue, &run, &UA_TYPES[UA_TYPES_BOOLEAN]);
    UA_Client_writeValueAttribute(mpParentClient->getClient(),
        UA_NODEID_NUMERIC(0, 10000),
        newValue);
}
```

The function `UA_Client_writeValueAttribute` writes the `newValue`, which is `true`, to the attribute belonging to the step node.

The else clause in the function `sample()` illustrates function calls to be made if the back-end is using the subscription approach. First, the client sends a publish request to start the OPC UA publishing cycle. Second, `insertValues` copies the cached values of each selected variable to the corresponding data vectors, as expected.

Each added `monitoredItem` is required by open62541 to have a pointer to a function which gets called upon a variable value change to deal with it. The following code illustrates how a `monitoredItem` is added as well as pointers to the functions `boolChanged()` and `realChanged()`:

```c
if (pVariable->isBool()) {
    UA_Client_Subscriptions_addMonitoredItem(mpParentClient->getClient(),
        mSubscriptionId,
        monitoredItem,
        UA_ATTRIBUTEID_VALUE,
        &boolChanged,
        mpParentClient->getClient(),
        &monitorId);
} else {
    UA_Client_Subscriptions_addMonitoredItem(mpParentClient->getClient(),
        mSubscriptionId,
        monitoredItem,
        UA_ATTRIBUTEID_VALUE,
        &realChanged,
        mpParentClient->getClient(),
        &monitorId);
}
```

The functions `boolChanged()` and `realChanged()` are as expected, called upon a value change. However, pointers to functions which is required by open62541 is by C, required to be static. Static functions can only access static data members which indicates that several instances of the code would use the same addresses for caching values if they were stored as data members to the class. This is a problem because different simulation instances should not be using the same data structure to read and write temporary variables since they would interfere with each other. Instead, values are cached using per-thread data storage. Each interactive simulation has one OPC UA client object, each having one worker thread where the variable values are temporary stored.

Value interaction belonging to requirement number 4:

- **Value interaction, changing parameter values during a simulation.**

An editable variable can be written to by entering the new value in the variables browser. The new value will be written to the embedded server associated with the model. The written value will become present to the solver on the server side which takes it into consideration when calculating values for future time steps.
4.2. Implementation

Different solvers on the embedded server side handle parameter changes slightly different. However, after a few bug fixes regarding how the solver reacts to a changed value, changes became present to the user at most two time steps forward. The reason for this behaviour is the different solver implementations on the embedded server side. The fixes made it possible for the user to change writeable variable values regardless of the selected solver.

Changes of variables values can be done regardless of whether the simulation is running or paused. An example is depicted in Figure 4.12. The simulation is paused in Figure 4.12a and the variable values for $h$ and $v$ are editable. The value 2.5 is entered to the variable value $h$ while the simulation is running. Figure 4.12b illustrates the running simulation with the result of the new value as well as the editable boxes in the variables browser.

The prototype is required to allow several interactive simulation instances according to requirement number 5:

- Allow multiple interactive simulation instances with appropriate instance management.

This requirement was based on the ability to determine the port number used by the embedded simulation server. Creating several instances of the same code requires proper
manage of those instances. The solution was to tie the plot window, the item in the variables browser and the output widget running the simulation executable together. A variable can only be attached to its corresponding plotting window. There can only be one interactive plot window per simulated model and also, only one instance of the same model. Trying to issue a second interactive simulation will remove the first one and continue executing the second.

Figure 4.13 illustrates how two instances of two different models are simulated simultaneously over different ports. The red areas show the three different items tied to each other. Figure 4.13a depicts the interactive simulated model *BouncingBall* while Figure 4.13b shows an interactive simulation of *NandGate*. Both simulations instantiated at the same time in OMEdit.

Finally, the last requirement implemented was:

- Integration of interactive simulation: differentiate interactive simulation from existing tools.
The option to simulate a model interactively is integrated into the simulation setup as depicted in Figure 4.14. It is a part of the General settings tab because the simulation procedure takes a rather different approach. For example, the results will never be stored in a result file and it offers a different area of use for the user.

The low overhead back-end is selected by default. If the user types in a specific port number, an attempt will be made to start the embedded server on that particular port. If no port number is entered, an available port will be searched for, starting from the default port number 4841 and up.

A model can be re-simulated. The user selects the targeted model from the variables browser and emits the re-simulate. Targeting an interactive simulation item results in a termination of the simulation executable. The executable is thereafter started again. Previous selected variables will be selected repeatedly in the new instance. The reason for restarting the whole simulation executable and not just the embedded server is because the embedded server does not offer the functionality needed. However, this be subject for a change in future works.

Clean the plot window button can also be applied to an interactive simulation item. It will deselect all the selected variables connected to the plot window, as expected.

Results from different result files can be plotted on the same plot window. This is however not the case for interactive simulation. A variable from an interactive simulation is restricted to its own plotting window. Selecting a variable from another interactive simulated model will make the window connected to the newly selected variable become focused. Mixing interactive simulated variables with variables from results files will result in an informative error message.
4.3 Evaluation

Evaluating how the prototype fulfilled the requirements lead to a table with information about areas where continuous work and improvements are applicable. Table 4.1 shows the current status of each requirement.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Selected variable values should be visually displayed during simulation.</td>
<td>Implemented.</td>
</tr>
<tr>
<td>2. Include interactive simulation controls such as play, pause and simulation speed and make them tied to the plotting window.</td>
<td>Implemented.</td>
</tr>
<tr>
<td>4. Value interaction, changing parameter values during simulation pauses.</td>
<td>Implemented.</td>
</tr>
<tr>
<td>5. Allow multiple interactive simulation instances with appropriate instance management.</td>
<td>Implemented. Possible stability improvements.</td>
</tr>
</tbody>
</table>

Table 4.1: Prototype requirements evaluated one by one.

Requirement 5 and 6 are implemented and functional. However, multiple interactive simulation instances together with previous functionality can in rare cases cause stability problems.
5 Discussion

This chapter aims to discuss the results, the methodology used and finally put the work in a wider context.

5.1 Results

The solution for fitting plot curves on the plotting surface was dependent on previous functionality to work properly. For example, simulating over a long period causes a long time scale on a limited plotting surface. Changes to variable values would therefore be hard to discern. The ability to zoom in on specific parts of the surface is consequently a crucial feature for the user in order to follow the changes of selected variable values. It is possible to zoom both in and out at any time during the simulation. Furthermore, a specific time interval could be selected which means that only variables in that particular time frame will be displayed. This could be useful if, for example, the user is only interested in the last part of the simulation.

Another already existing feature which could be favorably applied to an interactive simulation plot window is the control button Fit in View. This mechanism re-scale both axes in a way which respects all the selected curve boundaries. In other words, a good overview can be achieved and serve as starting point for further investigation. It can also be used to include all the curves in a situation where the user wants to save the plotted results and include all the selected curves. Furthermore, the possibility to save plotted results of an interactive simulation is another example of functionality derived from already existing plotting functionality as a result of how interactive simulation were integrated.

Exact result values can primarily be obtained by using the QwtPlotPicker feature. Another possible solution would be to make use of the variables browser and update the results for each plotted data point. However, one possible drawback could be that the user might feel overwhelmed by a constant value in the GUI, risking creating confusion rather than something useful. An alternative would be to update the variables browser with the latest value at a simulation pause to avoid the above mentioned problem.

Interactive simulation using an OPC UA client is as expected, dependent on the feature set of the OPC UA server. A clear example is how the data type boolean is handled in an interactive simulation compared to a normal simulation. The fact that each variable node on the server side holds one value at each time correct simulation step taken makes a difference
of the presentation to the user. A conventional simulation can append two or more values to a time step in order to make the change to a boolean variable appear immediate. This is not the case in an interactive simulation because of the one value per time stamp functionality previously discussed. However, if presentation of boolean variables should resemble the result from a normal simulation, it could be implemented. The solution would be to recognize a boolean variable value change and append an extra data point with the old value, together with the current simulation time value, before the new value is appended. This would make boolean variable value changes appear immediate, just like in a normal simulation, since the plotting tool simply draws lines between data points. Two different values at the same time stamp would therefore result in a straight line, perpendicular to the time axis.

There are pros and cons regarding such an implementation. A client side solution could be implemented without any major concerns but also takes over part of the executable’s responsibility. A modification on the server side might be more technically advanced and would also probably require some modifications in the interaction between the client and the server. However, the only responsible unit for providing data points would be the embedded simulation server.

5.2 Method

A thesis of good academic standard include concepts of replicability, reliability and validity in relation to the used method. Therefore, this particular thesis is evaluated with these concepts as a basis. Starting by looking at replicability. Another student, employee or interested person should be able to do this thesis again and get about the same results. If the thesis was conducted again using the same methodology, one of the most critical parts would be the pre-study section. It was in that section the prototype requirements were established. The implementation and in extension also the evaluation is heavily based on meeting these requirements. It is possible that the method used in the pre-study section could result in different requirements. It is also reasonable to assume that these requirements should not be very unlike and different from each other. Rather, the opposite, the actual implementation in terms of code would surely differ, but it is less interesting if the requirements are met.

With that said, some things could have been done differently to get different results. This thesis took the approach of studying theory and related work, engage a discussion with the supervisors to set requirements and finally implement them. Another applicable approach would be to do the pre-study, implement the prototype and then, evaluate the results. Holzinger [8], also described in the theory chapter 2, discuss that some evaluation techniques can be demanding in terms of resources such as time, expertise, equipment and user base numbers. However, spending resources on executing for example, one of the usability evaluation techniques in the theory chapter 2 could have some advantages. For example, the perspective of prospective users could provide measurable data about the prototype’s usability. Additional resources, mainly in terms of time would have been spent on a usability evaluation technique.

Computers have limited memory space. Another interesting approach for evaluating interactive simulation would be to simulate models of larger scale until the memory capacity is reaching its maximum. Furthermore, OMEdit should exhibit relevant information to the user when a simulation is about to reach the maximum memory capacity.

5.3 Source criticism

To begin with, the OpenModelica User’s Guide is a solid base for essential technical information about the OpenModelica environment. Sources in this thesis have been evaluated carefully with regards to their trustworthiness. This thesis contains firsthand scientific papers, books, technical reports and web links. Some used scientific papers are from the 20th
century. A review of their relevance today has been made in those cases. An advantage of the older sources is that the number of citations could act as an indicator of the credibility. It is harder to do a similar evaluation on newer publications. In those cases a stamp from well-known umbrella organizations such as ACM or IEEE were used as an indication of a trustworthy work.

5.4 The work in a wider context

It can be tricky to imagine social and ethical aspects related to a new feature for OMEdit. However, a rich OpenModelica feature set has the potential of increasing the understanding of complex physical systems. One can argue that it can be hard to point out one specific feature as a direct reason for the knowledge of the system to increase. In that case, interactive simulation has a possible potential to a small contribution regarding the obtained knowledge of a system.

A relevant and interesting question is what the simulated system is intended to do. It is fully possible that ethical questionable systems can be modeled and simulated in OpenModelica with different purposes. Assume that interactive simulation can contribute to a further knowledge of a system. In that case the knowledge could also be obtained for an ethical questionable system as well.
An analysis of the prototype together with thoughts about the theory and related work leads to some conclusions about the purpose of this thesis.

The research question, how can interactive simulation of physical system models be designed using an OPC UA client regarding usability, lead to the design and implementation of a prototype with requirements based on usability theory and knowledge about simulation in general, OMEdit in particular. This thesis presents a new area of use for OMEdit with the implemented prototype. A design can be outlined in many ways, the prototype presented is a concrete way of running interactive simulations in OMEdit.

For future works, it would be interesting to add functionality to the embedded simulation server. For example, expose parameters with write protection and enabling a step backwards in the simulation to allow the user to go back and forth in time.

It would also be interesting to improve the memory management in OMEdit by releasing memory after a variable deselection and ameliorate how changes in boolean values are presented to the user. Software is rarely perfect. However, the implemented prototype can act as a base for further improvements in both parallel and future works.


