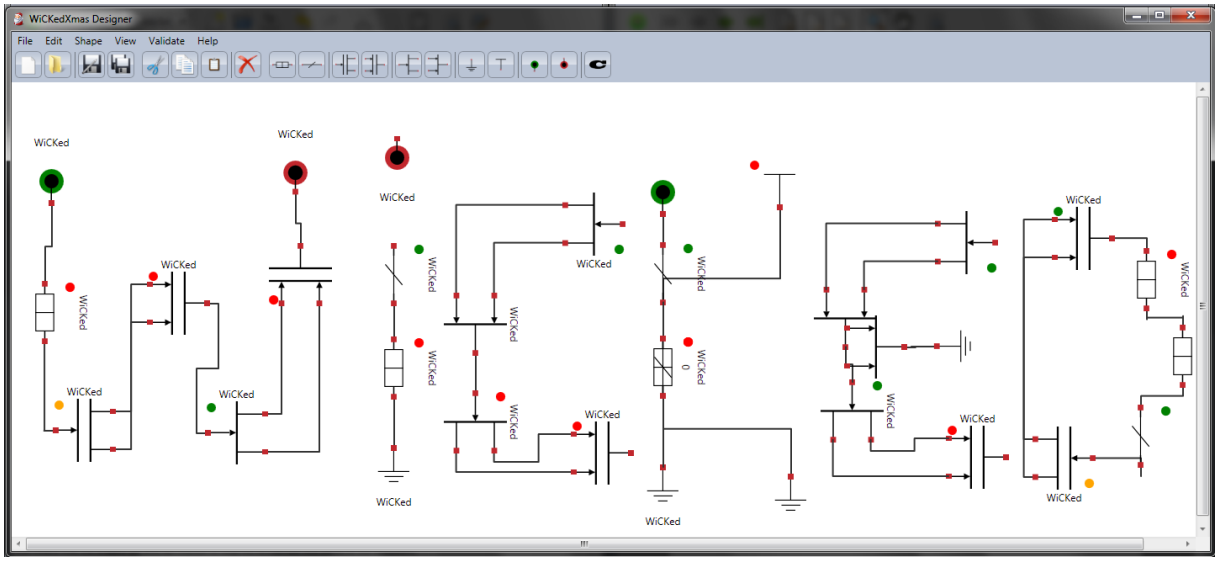
WiCKedXmas Editor
Technical Documentation

Kevin Reintjes (s0814954) k.reintjes@student.ru.nl
Christiaan Thijssen (s0814970) christiaan.thijssen@student.ru.nl
Willem Burgers (s0814830) willemburgers@student.ru.nl

February 1, 2012



Contents

1 Introduction

WiCKedXmas is the revolutionary editor for the xMAS language. This document is the technical documentation for the WiCKedXmas editor describing important information one should know when continuing developing the editor. It includes a description of the most important features, design choices, application structure, classes, known problems and future expansions.

2 Features

In this section, we will discuss the basic features of the WiCKedXmas Editor briefly.

2.1 Extensible canvas

The WiCKedXmas Editor consists of two main visual components when it is started. One is the menu structure with a toolbar and the second component is the (white) canvas in which you can place shapes freely. This canvas is automatically expanding when more shapes are added. When the canvas is empty, the canvas originally has the size of approximately one shape. When you add a shape, the shape will be displayed in the upper left corner. When you drag this shape more to the right or to the bottom, the canvas size will automatically be adjusted. When the canvas gets bigger than the window itself, scrollbars will be shown to scroll through the document. So, when you want to create a big canvas, you can simply drag a shape outside the window (on the right and bottom side).

2.2 Connect, drag and rotate xMAS shapes

As you may expect, shapes that you place on the canvas can be dragged around in the canvas freely. It is also possible to rotate the shapes by dragging the rotation bullet which appears when you select a shape, but we have fixed the angle that the shapes can be positioned at. This means that the shapes can only be rotated 90 degrees to maintain the clarity of the document.

Another feature is that you can connect shapes by dragging with the mouse from one connector that is attached to a shape to another. We have also implemented a check that an output has to be connected to an input so the designer cannot mistakenly connect an input to an input or an output to an output.

2.3 Select multiple shapes by dragging

It often occurs that you want to move a subset of the shapes on the canvas in order to create space in your design for adding new shapes or to reorder the circuit connection lines. To prevent that the designer has to drag all shapes one by one, we have added a feature to select more than one object. When you hold the control key on the keyboard and click on a shape, it will be added to the set of selected shapes. When there are connections between two selected shapes it also becomes selected so it will also move around when moving the selected shapes. When a connection exists between a selected shape and an unselected shape, it will adapt his route. Multiple shapes can also be selected by starting a new drag beginning at some white space of the canvas and create a selection rectangle over the shapes you want to have selected. When you subsequently drag one of the selected shapes with your mouse, all selected shapes will move.

2.4 Setting shape details

The shapes represent building blocks for the xMAS language. This means that some of the shapes have properties that need to be set. For instance, the function shape requires some function that changes the input. This function can be set as C code in a textbox. This C code will be saved when the design is send to the xMAS validator. Also for the queue shape, it is possible to set the number of packets that can be in the queue (the queue size). These details can be set by right clicking on the shapes. All the shapes can also have a label for identification. This can also be set with use of the right mouse button. The labels are shown above the shape when they are set.

Some shapes will also initially have a red dot. This is an indicator that tells you if the required attributes are set for this particular shape. When the functions or queue sizes are not set, the indicator will be red. When some details are set, but not all, the indicator will be orange. When all the required details are set, the indicator will be green.

2.5 Composite objects

Because the circuits that can be created with the xMAS language can grow very big, we have added the possibility to abstract from some complexity of the network. It is possible to build your own building blocks that can be reused in documents later. You can create a network with inputs and outputs (the shapes with the green and red circles) and save this document as a .wck file. Next, you can import this piece of network in another document by clicking the shape button with the big C (of composite objects). Then you will get a window where you can navigate to the document with the network you created earlier. The WiCKedXmas editor will read in this document file and create a new shape based on the information of the inputs and outputs in this file. The basic shape will be a rectangle with a number of inputs and outputs as they are defined in the document saved earlier. In this way, you are able to use a very detailed network in your new document that still uses only the space of one shape. This document can be imported in the same document as much as you like, so it also makes the networks more reusable.

2.6 Save and open document

As we have mentioned before, it is also possible to save a document and open it to continue your work. The information is saved in a JSON format, but we gave the output files our own .wck extension. The documents that are saved can in principle all be used as a composite object, but it is required that the network is fully connected when you use it as composite object. Saving can also be initiated with the shortcut keys that are commonly used in other programs which makes the interface more powerful.

2.7 Cut, copy, paste shapes

It is also possible to duplicate a selection of shapes by using the copy paste functionality. The WiCKedXmas editor has his own internal clipboard on which you can add a selection of shapes by selecting the shapes you want to copy in the canvas and then click on the copy icon in the toolbar or pressing the Ctrl+C key combination. When you want to place the copy, you click the paste icon (Ctrl+V) and the set of selected shapes will be copied and also all the intermediate connection lines are copied. The same happens when you cut some selection, but this will also result in deleting the original selection of shapes.

2.8 Export to PNG image or XPS document

When the design is finished and you want to show the design to other people or use the design in other applications, it can be useful to export the view as you see it in the editor. Therefore, we added the possibility to export the canvas to a PNG image or to a XPS document. The view that is exported is an exact snapshot of the canvas. So if you do not want the indicators or labels to be visible in the exported image or XPS document, you have to uncheck these in the view menu to hide these elements before you export the canvas.

2.9 Change size of shapes

When the standard size of the shapes is not as you desire, you can select another size of the shapes in the view menu of the editor. When you select another shape size to work with, the size of the shapes are changed, but the position of the shapes will stay the same. This means that the room between the shapes will not be rescaled and the size of the connections will also not change. This could be some future expansion of the editor. See Section 10 for some ideas about this.

2.10 Send to xMAS Validator

For this feature, we only create the menu options and the method to check if all the connectors of all the shapes are connected before the document will be send to the validator. The main idea is that the xMAS validator can be executed with one push on this menu item. When the xMAS validator is fully developed, this feature can be developed further.

3 System Requirements

- Microsoft Windows Vista / 7
- Microsoft .NET framework 3.5
- 1 gigahertz or faster processor

- 1 GB RAM memory
- 10 MB free Hard Disk space

4 Development Environment

4.1 Software

We have created the editor with the following tools and software:

Operating System	Microsoft Windows 7 Ultimate
Integrated Development Environment	Microsoft Visual Studio 2010 Ultimate 10.0
Framework	.NET framework 3.5
Graphical subsystem	Windows Presentation Foundation (WPF)
Graphical Editor (for creating the XAML shape images)	Inkscape 0.48.2
Graphical Editor (for creating raster images)	Paint.NET v3.5.10
Versioning system	SVN 1.7.0

4.2 References

To compile the program successfully, make sure you have the following references added to the reference list:

- Newtonsoft.Json (for exporting to JSON format) `http://james.newtonking.com/pages/json-net.aspx`
- ReachFramework (for exporting images)
- System.Printing (for exporting to XPS)

4.3 Used Information Sources

For developing the WiCKedXmas Editor, we have used a lot of documentation, examples and tutorials that we have found on the Internet. We will give a list of the most important information sources here:

- Data Binding in WPF by *John Papa*, `http://msdn.microsoft.com/en-us/magazine/cc163299.aspx`, 2007.
- WPF Diagram Designer Part 1: "Drag, resize and rotate elements on a Canvas" by *sukram*, `http://www.codeproject.com/KB/WPF/WPFDiagramDesigner_Part1.aspx`, 2008.
- WPF Diagram Designer Part 3: "Connecting Items" by *sukram*, `http://www.codeproject.com/KB/WPF/WPFDiagramDesigner_Part3.aspx`, 2008.
- Tutorial on Toolbars: "WPF RichTextBox with standard Text-formatting possiblities" by *GregorPross*, `http://www.codeproject.com/KB/miscctrl/WPFRichTextEditor.aspx`, 2010.

5 Project layout

For this project we used the Windows Presentation Foundation framework. We picked this framework because of the rendering speed which uses the GPU and does not redraw the whole canvas the whole time when moving a single object. The rendering speed of this framework is a lot higher than for example the Windows Forms Application framework.

The WPF framework uses separate files for the prebuilt view of classes in a separate language (XAML). The model of the classes is written in C# and can be found in the various .cs files. Possibly these classes have a prebuilt view which can be found in the various .xaml files. There is no real separation between model, view, controller but some classes act more as controllers (for example the MainWindow and DesignerCanvas class) and some more as models (for example the Fork class). The view of the classes is mostly described in the XAML files, but some non-prebuilt view functionality is in the C# files as well.

For the course this project was part of (System Development Research), we had to use agile development. We worked in so called sprints to develop parts of the functionality quickly. This means that we did not foresee some design issues. Therefore the code structure is not as neat as could be. Some functions are propagated through the program (like the commands in the toolbar/menu are propagated to the canvas, because the main window cannot see the shapes on the canvas). The most tricky propagations will be described in

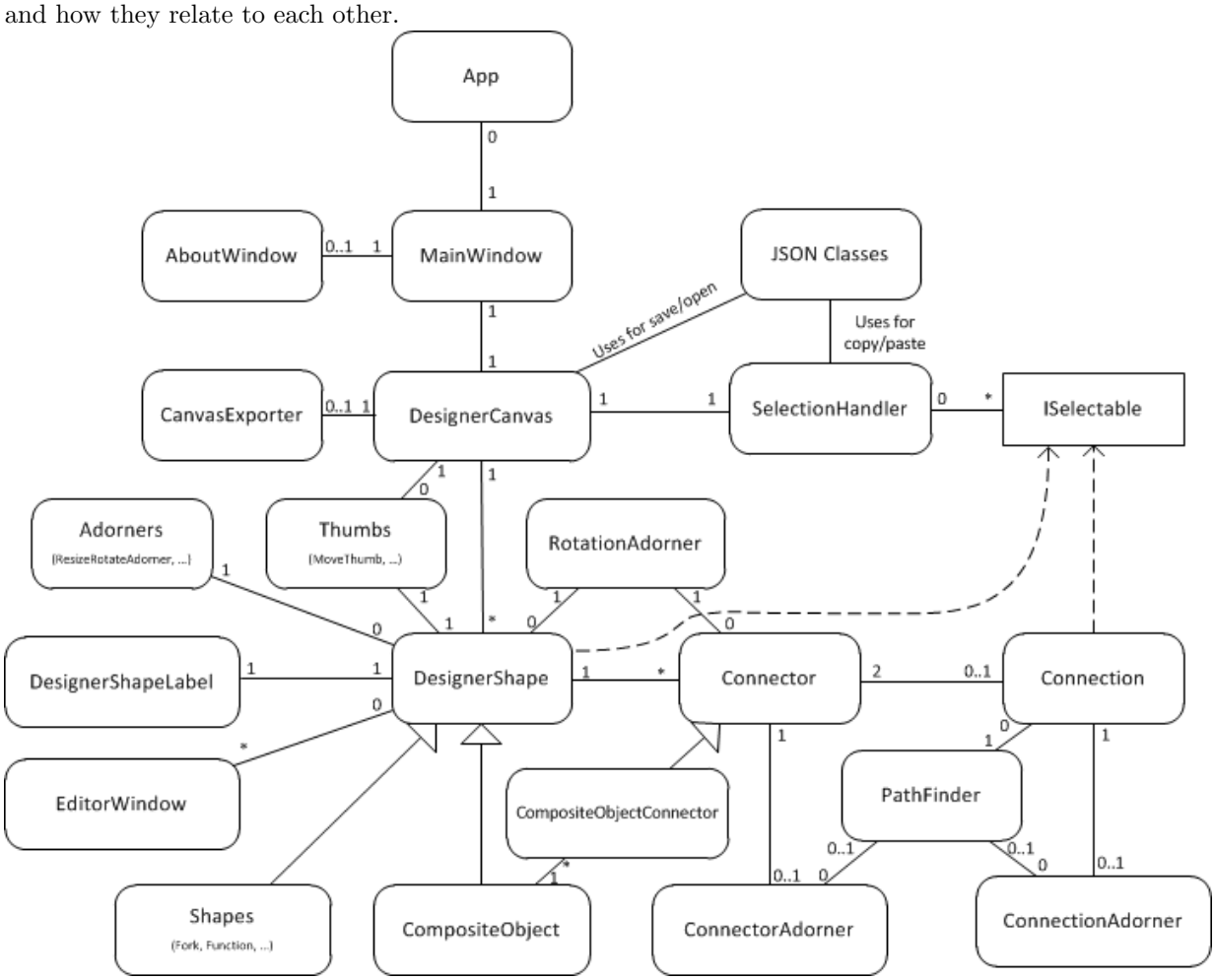
the documentation of the class itself.

The project is divided in a couple of folders. The root folder holds the most important and commonly used classes, like MainWindow and DesignerCanvas. Then there are some separate folders for parts of the project that belong together:

- Adorners: This folder holds all the Adorner classes for specific (view) functionality, like the selection box around shapes when they are selected.
- Images: This folder contains all the images used in the program, for example for the images for the toolbar and the program icon.
- JSON: This folder contains all the classes used for the JSON representation which is used for the copy/paste, save/open and export for validator functionality.
- Resources: This folder contains some of the most important XAML files, for example the basic DesignerShape view XAML file.
- StdShapes: This folder contains all the specific shapes (which extends the base DesignerShape class) like the Fork and Function, but also the CompositeObject.
- Thumbs: This folder holds the Thumbs classes for specific actions, like Move and Rotate.

6 Overall Design

Now we give a diagram describing the basic structure of the design of the program. It includes all classes and how they relate to each other.



7 Classes

7.1 App

App is the first class that starts when loading the program. It creates a MainWindow and loads all XAMLs. When a new shape is added, the XAML of that shape should be included in App.xaml.

7.2 MainWindow

The MainWindow class represents the main window of the program. It contains all visible elements when you first load the program. It contains the menu, toolbar and canvas. These are loaded in the MainWindow.xaml. All bindings of the menu items and toolbar buttons are done in the MainWindow.xaml.cs file. These functions mostly propagate to the DesignerCanvas, because the canvas handles the shapes on it.

7.3 DesignerCanvas

This is the white area of the main application window. The canvas holds all the shapes and implements some basic functionality for the application. The creation of new shapes, save and open a document are implemented here as well as most menu items.

7.4 SelectionHandler

The SelectionHandler class manages all selected shapes. You can add a shape to the selection or put the selected items on the clipboard (cut or copy) and retrieve items from the clipboard (paste). The SelectionHandler also contains the keyboard shape movement handling (arrow keys). The selected shapes are updated when an arrow key is pressed. DesignerShapes are an extension on ISelectables, so that is all the ISelectable class does.

7.5 CanvasExporter

In the CanvasExporter class, the exporting of the canvas to PNG and XPS is done. By using an encoder other than the one we used, it is easy to export to other file formats like JPG.

7.6 DesignerShape

DesignerShape is the base class for all xMAS objects. This is where all the basic methods of a shape are found and basic attributes are set. Things like position, rotation, label (name for the object) and the context menu are implemented here. Many functions are overridable, because specific shapes require different handling of some functions. Any exception on the basic functionality of a DesignerShape will be discussed in the specific shape sections. Each shape has multiple connectors. They are stored in the DesignerShape class, but each shape is different and therefore we implemented creation of connectors in the XAML of the specific shapes. Per shape dragging is also handled in this class. To set labels we created a ContexMenu, which is automatically opened when clicking the right mouse button.

7.6.1 Most Shapes

Most shapes are not really different from the basic DesignerShape, though DesignerShape does not implement a canvas representation (only some thumbs for dragging and adorners to show when a shape is selected). Each shape has its own XAML in which the prebuilt view can be created. The XAML creates the connectors for each shape, because most shapes are different. The XAML is based on the SVG representation of the shape, which was created using InkScape. These representations are vector based, so scaling of objects won't be an issue. The Shapename.cs file mostly only contains a constructor. In the constructor, the right click menu is extended with an option to edit shape functions (C code) if the shape should contain these functions according to the xMAS language.

7.6.2 Queue

The Queue is based on DesignerShape, but has some specifics about it. The queue does not have any function (C code), but the queue is the only object with a size attribute. The editor for this size value is implemented like a function. There were some requirements for this queue to not only edit and save this queue size, but also to view it on the canvas. This means that the queue has an extra label in which we place the size value. The validation of the queue is also different from a normal shape.

7.6.3 CompositeObject

A composite object is an xMAS network that can be used as a single shape. It is based on DesignerShape, so you can drag and copy and cut and stuff like that. The representation of a composite object is pretty basic now. It just shows a rectangle with some in and out connectors. We did not have time to implement an SVG representation of the composite object. Also labels for in and out connectors are not yet shown. This should be easy to implement though. The CompositeObject.cs file is mainly used to create a correct representation on the canvas.

7.7 Adorners

Adorners are used to show some fancy box around a shape or connection when it is selected. The rotation handle is only shown when a shape is selected and is part of the adorner. More important are the pieces of code in the ConnectorAdorner that will draw the path when dragging/creating a new connection. The same goes for the ConnectionAdorner that will show the path when moving a connection to an other connector.

7.8 Thumbs

Thumbs implement the code that actually manipulate objects on the canvas. When an object is dragged, the MoveThumb is called. When the rotation handle is clicked, the RotateThumb is called. The thumbs handle the mouse input to check the difference in mouse position. They move/rotate the object according to that difference.

7.9 Connectors and Connections

A connector is a thing you can draw a connection from and to. The actual connectors are only shown when the mouse is hovering over a shape. The red dots that are always visible are not the connectors. If you draw a connection from a connector on shape A and hover over shape B, shape B will only show the connectors that can be connected to. This means In connectors can't be connected to other In connectors and connectors can not have more than one connection. This was one of the most tricky functions to implement, because shapes do not know of the existence of other shapes. Especially the hiding of connectors can't be done from the connection, connector or shape class itself. This is propagated to the canvas, so it can be done for each shape.
Composite objects use a different kind of connector. It is based on the normal connector class, but it implements the use of a label for the in and out connectors for the composite object. Though we do save the label for in and out connectors, we do not yet show the labels when a composite object is loaded into the program. This can be a future expansion of the program.

7.10 Pathfinder

The Pathfinder class is used by the adorner classes to find the best possible path from point A to point B on the canvas without getting in the way of objects. When dragging a new connection, the pathfinder searches for the best path from the start connection to the mouse pointer. When a connection is made, the path is recalculated. When moving an object, the path is also recalculated. The PathFinder class likely needs the most processing power. When dragging the whole network, the paths do not change. All the paths between the dragged objects are however recalculated. This is unavoidable for now, but can be improved in the future. Especially when a large network is built, it can become a bit slow.

8 Format Saved Document

As we have mentioned before, the documents are saved in a JSON format. The extension of the files are .wck, but these files are readable when you open them with a standard text editor.

8.1 Basic layout of JSON

The JSON file has the following main layout: One object containing two attributes. One attribute is a list called "shapes" and contain information about all the shapes that are visible on the canvas. The second attribute is a list called "connections" which defines all the connection lines between the shapes. When you save an empty canvas, it will result in the following JSON:

```
{
 "shapes": [],
 "connections": []
}
```

8.2 Simple standard shape example

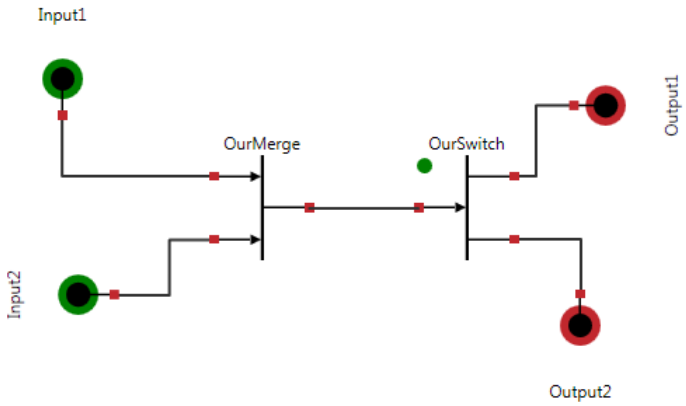
In the next example, we are going to add two shapes to the canvas and create one connection to explain the manner in which our actions are represented in the resulting JSON file. We have added some comments to make it more clear. The network we designed looks as following:

Which resulted in the following JSON output:

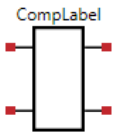
```
{
 "shapes": [
     {
      "oldShapeID": 0,                The old unique identifier of the shape.
      "posX": 176,                    The X coordinate of the middle of the shape.
      "posY": 117,                    The X coordinate of the middle of the shape.
      "orientation": 1,               The orientation of the shape where 1 is North.
      "sizeMultiplier": 1.0,          The magnification of the visual representation of a shape.
      "type": "Merge",                The type of the shape, as class name.
      "label": "OurMerge",            The label the user gave this shape.
      "functions": [                  The collection of shapes that apply to this shape.
         "",                          No functions need to be set for this shape,
         ""                           So these function placeholders are not set.
      ]
     },
     {                    The same attributes are saved for the second shape
      "oldShapeID": 1,
      "posX": 406,
      "posY": 117,
      "orientation": 1,
      "sizeMultiplier": 1.0,
      "type": "Switch",
      "label": "OurSwitch",
      "functions": [
         "do A if B else C",          For a switch, one function needs to be set.
         ""
      ]
     }
 ],
 "connections": [                Now one connection is defined
     {
      "oldSourceDesignerShapeID": 0,      The old unique ID of the "From" shape of this connection
      "oldSinkDesignerShapeID": 1,        The old unique ID of the "To" shape of this connection
      "sourceConnectorName": "Out1",      The connected port of the "From" shape of this connection
      "sinkConnectorName": "In1",         The connected port of the "From" shape of this connection
     }
 ]
}
```

8.3 Composite object example

All the JSON output of the other possible standard shapes is reasonably the same. The only exception that is interesting to take a look at, is the way the composite objects are saved in the JSON format. Therefore, we are going to import a composite object that represents the following network. This network is basically the same as the network we have threated in the previous section, the only thing we added are input and outputs which are required to create the connection points of the resulting composite object. As you can see, we have now two inputs and two outputs, so the composite object we will have as a result will have four connections, two inputs (on the left side) and two outputs (on the right side) as well.



We saved this document and gave it the name "compositeExample.wck". To import this document as a composite object, we open the base document, that in our case will just be a new empty canvas. To import the composite object we just created, we click on the button with the big C in it and select the file "compositeExample.wck". After that, the following figure will be shown (we have also set a label for our composite object):



When we save this new document with the composite object, it will have the following layout:

```
{
 "shapes": [
     {                  The shape attributes are almost the same, only "filename" is added
      "fileName": "C:\\Users\\WiCKedUsahr\\Desktop\\compositeExample.wck",
      "oldShapeID": 0,
      "posX": 94,
      "posY": 67,
      "orientation": 1,
      "sizeMultiplier": 1.0,
      "type": "CompositeObject",
      "label": "CompLabel",
      "functions": [
          "",                Functions cannot be set for a composite object,
          ""                 so these function placeholders are not set.
      ]
     }
 ],
 "connections": []              We did not draw any connections so this is empty
}
```

8.4 Generation method saved document

For creating the JSON output, we used a commonly used package for JSON serialization. This package is the NewtonSoft JSON.net package which provides you with some static functions to serialize and deserialize objects. Serialization can easily be done with the following statement:

```
JsonConvert.SerializeObject(this, Formatting.Indented);
```

When we save a document, we want to serialize all the contents of the DesignerCanvas but because the DesignerCanvas has many more objects and attributes attached to it than we need to save to disk, we first created a JsonMain object. This JsonMain object will copy all the information we really want to save to disk from the original objects. We have also created such a serializable data object for the general DesignerShape object (called the JsonShape), Connection objects (called the JsonConnection), and a special dataobject for the Queue (JsonQueue) and the CompositeObject (JsonCompositeObject). This JsonMain object therefore consists of two public attributes: A list of JsonShape objects and a list of JsonConnection objects. The JsonQueue and JsonCompositeObject extend the JsonShape object.

This approach is chosen because serializing the original objects would give problems because we have a lot of coupling in them. This means that the objects have many pointers to other objects and there are some pointer loops possible. When you serialize an object, all the objects that are referred to are also serialized. So serializing the original objects itself would give some infinite loops. Creating the JSON data objects ourselves has as advantage that you have better control of which information is saved to files and which information is not. One disadvantage of this approach is that when you want to add a new attribute to the saved document file, you will need to create the code that copies the information from the original object to the JSON data object. Also the code for opening a document needs to be extended with the code that restores the value of the extra saved attribute based on the attributes in the JSON data object. So it is less expandable.

Note that all the attributes of the JSON data objects need to be "public" in order to serialize them with the NewtonSoft JSON.net serializer. When the attributes are not "public" they will be ignored.

In our approach, the following sequence is executed while saving a document:

- A JsonMain object is created and on creation it gets a pointer to the DesignerCanvas.
- For all the DesignerShapes and Connections in the canvas, the getJsonObject function is called.
- The getJsonObject will create a new JSON data object for the current item and the attributes that need to be saved are copied.
- Pointers to the JSON Objects that are created are added to the appropriate list in the JsonMain object.
- Using the standard JSON serialize method, the JsonMain is serialized (and all the objects where the JsonMain object contains pointers to).
- A String will be the result of the serialize method and this can be easily written to a file.

9 Known Issues

- Every saved network can be imported as composite object. There is no check if the network is valid or not. You can for example create a few shapes without a connection between them and save it. If you then import it as composite object, it will not be checked for any error. A simple workaround is to always save using the export for validator, which checks for connectors without connection and checks if all shape functions are set. A future expansion of the program would be to check composite objects when importing them. Also the export for validator should check inside composite objects.
- Paths to composite objects are absolute. This means opening a file with a composite object in it, won't work on any other computer than the one it is created on. (Unless you fix the path in the .wck file or it is already correct by coincidence.)
- Shapes can be moved out of the canvas. When using the arrow keys to move the shapes, no check is performed whether the shape is still in the canvas. Therefore it is possible to move a shape out of the canvas at the top / left side. This could cause the program to crash due to negative positions. Also the canvas will not be enlarged when moving the shape out of the canvas at the bottom / right side and thus the scrollbar will not appear or is too small.

10 Future expansions

This section describes some possible expansions which we would wanted to implement, but could not do due to time constraints.

10.1 Image in Composite Object

Something that we wanted to implement in the last sprint was that we could give the Composite objects their own image. The most preferred way of working would be that the user could right click on the composite object and select a menu item "select image". The user can subsequently browse the file system and select an image that he wants to use for the current composite object. The image is then depicted in the square of the composite object which makes the function of the composite object more clear. But unfortunately, we did not succeed in implementing this feature on time. This is due to the fact that this image also has to be resized when opened, copied when the object is copied and that we needed to extend the JSON format in which the composite objects are saved to save a reference to this image.

10.2 Drag shapes from menu

At first we had only a standard menu where we could click on the name of the shape we wanted to create and then it would be added to the DesignerCanvas on a fixed position in the top left position of the window. We thought that this approach was not very handy if you want to add lots of components to your canvas when you are really designing a circuit. Therefore, we added a toolbar to the window so you can select the shapes in a more intuitive way. But there is a even more intuitive design for selecting shapes: Displaying the shape figures in a shape container in the menu which can be dragged out of this container onto the canvas. When the drag is ended, the shape is placed on the canvas and a new copy of the shape figure is added to the container again (in his original place).

10.3 Method to check if composite objects changed

The addition of the feature of creating composite objects has introduced a lot of challenges. We did not found a proper solution of all these challenges yet. One of the challenges that still needs a solution is situation in which you have imported a composite object. It does not lock or check the original save file of the composite object during the designing of the circuit. Only the filename of the source of the composite object is saved in the current document and when you want to "flatten" the design to export it for the validator software, the editor has to read the original source file of the composite object again. In the meantime, another process could have changed this file. This could result in the situation that the input and output connections of the composite object do not comply with the input and output connectors defined in the composite object source file any longer. This could lead to a crash of the editor when you want to export it.

Therefore, some integrity checks need to be added to the editor. This could be done for example by hashing the source file of the composite object and save this hash file in the CompositeObject shape object. When the composite object source file is read again, we could first check its integrity by hashing the file a second time and check if the hash is the same as the hash that is stored. When the hash does not match, the composite object can be reloaded (but still displayed on the same position) and the connections to and from this object can be deleted so the user is "forced" to review the connections of the reloaded component. Another way of making sure that the file does not change when it is imported as a composite object is locking the file. A disadvantage of this second approach is that you cannot design different parts of a network at the same time. Once a file is locked, you cannot change it. So in principle, when you add circuits to the canvas, these composite objects cannot be changed until the editor is closed. There also need to be a solution for checking the integrity and relocking the files when opening a document with composite objects.

10.4 Use of windows clipboard

Another feature that we tried to implement (for a small period of time) but we did not succeed in, was the use of the windows clipboard so we could copy and paste instances of shapes and connections from one WiCKedXmas Editor window to another. When we have this feature, the designing experience is more flexible because you could copy or cut parts from one design to another without saving the document to another file, removing the undesired circuit parts, create input and output nodes, save it and import it as a composite object in the second document.

10.5 Undo and Redo

As we are used to undo and redo our mistakes in other editor software, we also wanted to create this ability in the WiCKedXmas Editor. Unfortunately, we did not have the time to implement this. The benefits of these functions are obvious but implementing an intuitive undo and redo model could be challenging.

10.6 Interfacing with validator software

The export for validator function can be expanded to actually validate the complete network and export to the compiled deadlock checker. We did not get the chance to implement this. Our JSON format is not the same as Freek's format. Therefore the export won't work completely yet. Though all information necessary for Freek's format is included in our JSON format, the structure is different.

As mentioned in the known issues section, there is no validation of networks when they are used/imported as composite objects. Also the export for validator function for a network which uses a composite object will not validate the internal structure of the composite object. So it does not dig into the hierarchy of a network. This should be relatively easy to fix in future versions of the program.

10.7 Automatically reorder shapes

Because the networks could become very big when designing real world processor structures, it is possible that the placing of the objects is not optimal at a certain point of the designing process and that some connection lines will cross needlessly. Another feature that could be added is an automatic restructure option which rearranges all the shapes and connections. When it does this, it will reduce the total length of the connection lines and place the shapes in the most compact way. Though, it needs to take into account that the crossing of connection lines has to stay unambiguous.