- Graphview: a Kahn Process Network visualization tool

o Abstract

Compaan is a compiler that automatically generates Kahn Process Networks (KPNs) from small sequential programs written in Matlab. A designer can maps these KPNs onto different compute platforms like FPGAs, the Intel IXP network processors, or the IBM Cell processor. Neverthless, a KPN generated by Compaan contains lots of complex data and the designer may get overwhelmed by this amount of data. Therefore, to help the designer in the mapping process, we want to use visualization to organize the complex data present in KPNs. In this presentation, we present the Graphview editor that is a visualization tools for Kahn Process Networks (KPN). We build Graphview as an Eclipse plugin on top of the Graphical Editing Framework (GEF) as it allows developers to take an existing application model and quickly create a rich graphical editor. We will explain how we realized the Graphview editor and show results.[1]

o Introduction

The compaan tool was developed to automatically transform matlab nested loop programs into process networks, suitable for implementation on (embedded) multiprocessor architectures. It focuses on systems that consist of a microprocessor and several dedicated coprocessors. It distiguishes itself from other compilers for multiprocessor architectures by being not only taking advantage of the instruction level parallelism but also using the coarse-grained parallelism offered by the coprocessors.

Compaan achieves this goal by generating a Kahn Process Network (KPN) from a nested loop progam. This process is depictures in Figure 1. It shows a Matlab program that Compaan subsequently converts into a process network using a number of tools. The tools are MatParser that converts the Matlab into Single Assignment code, DgParser that converts the Single Assignment Code into a Polyhedral representation, and finally Panda that creates the Process Network. All information about the Kahn Process Network (KPN) is captures in a XML format. The KPN is a graph structure where each process is represented by a node and the data path between processes by an edge. The problem of an application to architecture now becomes the problem of assigning each process to either a microprocessor or a coprocessor.

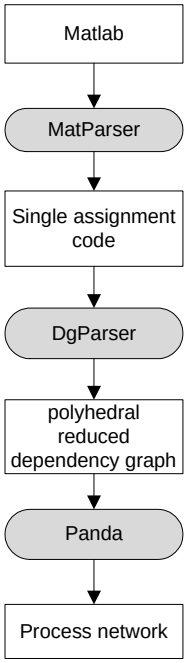


Figure 1: Compaan consists of three tools that transform a Matlab specification into a process network specification.

The KPNs generated by Compaan contains a lot of complex data. A designer confronted with the mapping problem gets overwhelmed by this amount of data. How to help this designer to structure all that data. A very effective way is to use visualization to organize the data present in the KPNs. In this thesis we will present the Graphview editor. This editor provides the designer with a visual KPN editor and viewer

o Problem description

A KPN is a complex datastructure that cannot easily be interpreted by a human designer. Our goal is therefore to present this data to a designer of multiprocessor platforms in a visually structured way. To improve the understanding of the KPN generated, the designer also needs the ability to play around with the KPN structure and view the results of this manipulation. We focus in this thesis on solving the visualization problem by creating a graph editor for the KPN datastructure, and thus using a graph like visual structure that contains, among other things, nodes connected by edges.

Our problem statement is twofold. First, there is the problem of the representing the KPN structure in a graph like form. The second part is concerned with manipulating this structure by interacting with the visual representation.

▪ Representation

The problem of representing a KPN like a graph can be divided into the subproblems of how to represent each element of the KPN in visual form, and how to position these elements in relation to each other. In particular these elements will be:

- The KPN's entities (processes) and the number of firing based on graph-scope parameters
- (Data) links between processes, the amount of data transferred of these links, and the linearization of this link.
- The in- and output variables of the entities, and which variable a link uses.
- The node and port domains.

Figure 2: a graph viewer visualizes the KPNshows what global structure is needed to represent the KPN visually. A graph viewer program takes a KPN as input and produces a graphical visualization of nodes (ellipses) connected by edges (arrows).

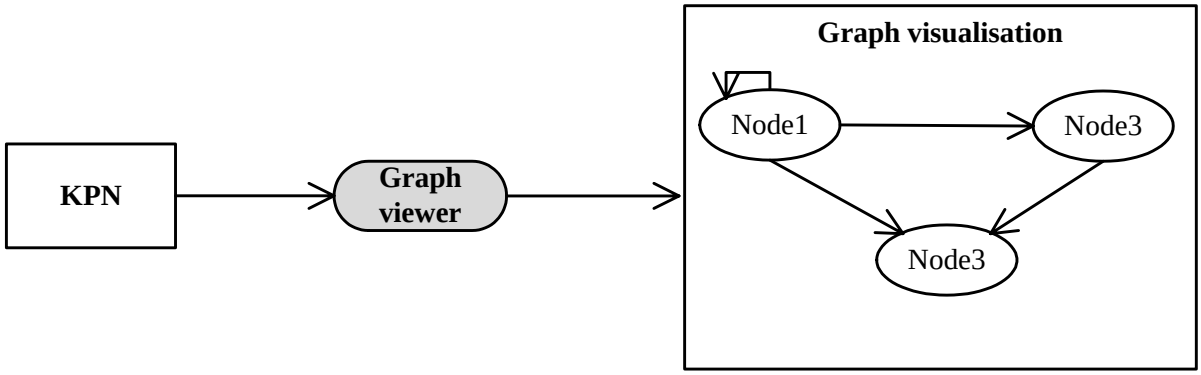


Figure 2: a graph viewer visualizes the KPN

- **Manipulation**

To adjust the KPN to the designer needs a way to manipulate the KPN. This also implies that the editor requires methods to change the KPN model. These methods can be divided into two categories. The first category consists of actions that leave the KPN structure consistent, and just adjust it to the designers needs.

Figure 3: the designer interacts with the graph interface, thereby changing the KPN. shows the basic program structure for a graph editor that allows interaction with the graph. The arrows back and forth indicate that there is an interaction between the parts. The graph editor generates the interface, but interactions from the designer with the interface will change the underlying KPN model, via the graph editor.

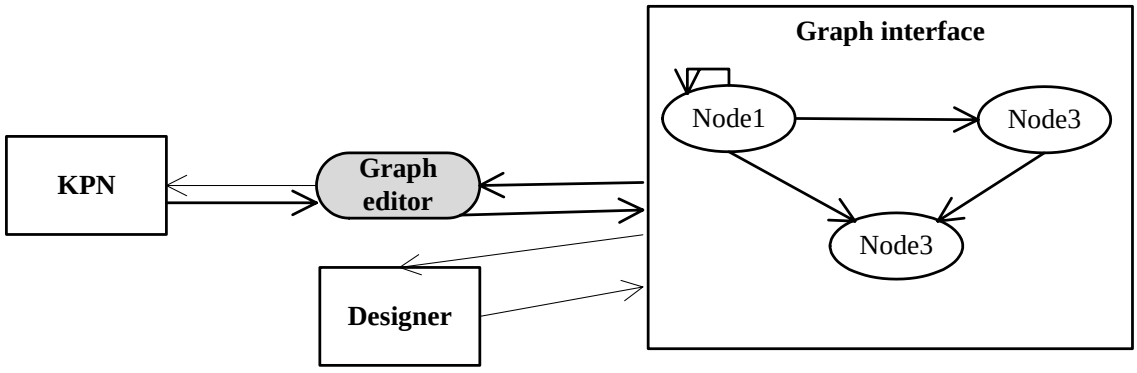


Figure 3: the designer interacts with the graph interface, thereby changing the KPN.

Normal manipulations

The actions that fall into the first category are:

- Loading and saving a KPN.
- Moving nodes and edges
- Laying out the graph.
- Annotating the graph with pieces of text.
- Changing node, edge, port, variable names.
- Add textual properties and editing their values.
- Outputting the graph to a printer.
- Saving the graph as an image

Consistency breaking manipulations

The second category consists of methods that compromise the consistency of a KPN edited. The implication of these methods on consistency is unclear, and are food for thought, and subject for further research.

- Adding/removing nodes, edges, ports and variables
- Editing node and port domains
- Decompose nodes into clusternode

Adding or removing elements from a KPN compromises the consistency of this KPN. A KPN describes a program, and each element in a KPN is a reflection of some element of this program. When, for example,

we add an entity to a KPN, it is no longer clear what the matlab program corresponding to this KPN would be. The entity has to represent a piece of code in this program, but it is on first sight unclear what piece of code this would. To put it more clearly and concisely: while each matlab program maps to a KPN, we, as of yet, have no way to map each KPN a matlab program. This might however be possible and we recommend it for an followup project

Explanations of KPN- specific terms can be found in Appendix A The KPN datastructure.

- **Development Environment Integration**

The KPN visualization and representation needs to be performed in a particular environment. Because compaan is an Eclipse plugin, so it is desirable to, again, use Eclipse as environment.

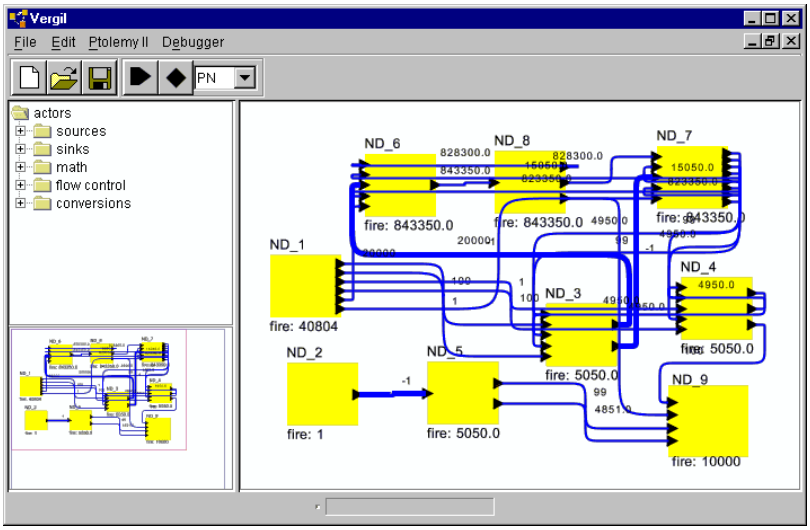
o Related Work

Visualization of graphs is done in some existing programs. The two programs that come closest to meeting our requirements are PtolemyII and Simulink. In the next two paragraphs we will discuss these programs and show where they fail to meet our requirements.

- **Ptolemy**

"The Ptolemy project studies modeling, simulation, and design of concurrent, real-time, embedded systems. The focus is on assembly of concurrent components... "[2]

Ptolemy provides the Vergil user interface to edit models. Since a KPN is a model of an embedded system, so PtolemyII would be a logical choice to use to visualize KPN's. Unfortunately the main intention of PtolemeyII is not to visualize models. This is particularly obvious when large models are edited with Vergil. PtolemyII provides an automatic layout mechanism for its connections. Whenever this layout is performed, the links are rerouted using a manhatttan router. As we see in Figuur 1: KPN model visualized with PtolemyII, this mechanism fails to keep the visualization clear, and actually hides the structure of the model.



Figuur 1: KPN model visualized with PtolemyII

- **Simulink**

"Simulink, developed by The MathWorks, is a tool for modeling, simulating and analyzing multidomain dynamic systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries."[3]

Simulink is a tool for modeling dynamic systems. Simulink suffers from similar layout problems as Ptolemy. All entities are manually placed, and all links are drawn by the user: there is no automatic layout mechanism. In figure 3 we see how the visualization of a large model seems very clear. However this layout is done manually, while we want to automatically place and route graph elements.

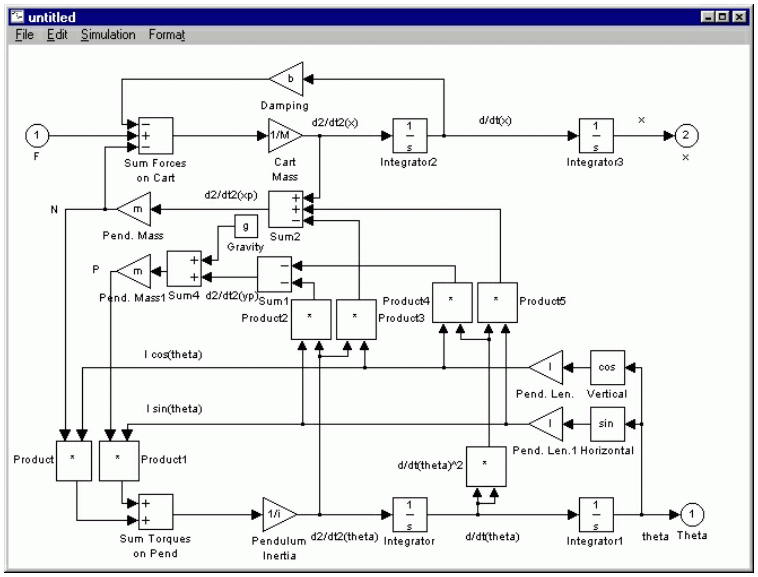


Figure 4: large model visualized in Simulink.

- **Development environment integration**

Another downside of using previously mentioned programs, PtolemyII/Vergil and Simulink, is that they are stand alone programs, and can therefore not be integrated in the Eclipse framework, used as the compaan development environment.

- **Conclusion**

The two most logical choices to visualize our KPN: *PtolemyII* and Simulink, both suffer from two flaws: they lack proficient layout functionality and can not be integrated into the eclipse framework. We have therefore created a new KPN editor that does meet our requirements.

o Solution Approach

The first decision that follows from our requirements is that we create a program that is integrated in the Eclipse environment. This is done by creating an Eclipse plugin. An eclipse plugin is a program that adds functionality to the Eclipse Framework. Our plugin Graphview adds to the Eclipse Framework by:

- Supplying a wizard in which a new .kpn file can be created.
- Supplying an editor for .kpn files.

An added advantage of integrating our editor into the eclipse environement is that we can use the Graphical Editing Framework(GEF) to create our editor. GEF is a toolkit designed too easily create rich graphical editors withing Eclipse [4] As mentioned before, the KPN datastructure is graphlike and is therefore well suited as a model for an GEF application.

An application built upon the GEF toolkit uses the Model-View-Controller pattern for visualization. The Model-View-Controller(MVC) is a software architecture that separates the model (in our case the KPN network) from the view (in our case the visualisation of the KPN file). The controller bridges the model and view, by providing the communication between the two. A direct result of this seperation of code is that both the model and the view can easily be replaced by a different one.

In Figure 5: JDot serves as the model in the MVC pattern. A KPN file is used to store the model. we see the three parts of the MVC pattern, as used in our program:

- The model: JDot
- The controller: multiple EditParts
- The view or visualization of the model.

A KPN file is used to store the model.

We use a JDotGraph object as the model. JDot is a graphlike structure as well, but a built- in layout mechanism. Whenever a .kpn file is opened in the Graphviewer editor, the KPN is loaded and converted to a JDotGraph. Upon saving, the JDotGraph is converted back to a KPN network, and the KPN is saved to a .kpn file.

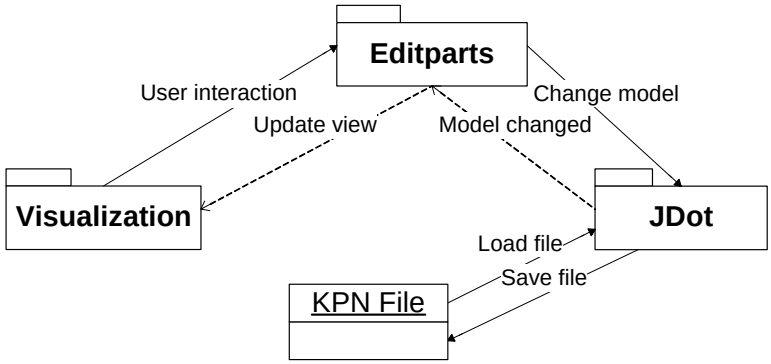


Figure 5: JDot serves as the model in the MVC pattern. A KPN file is used to store the model.

The built- in layout functionality of JDot is what seperates Graphview from programs such as Simulink and Ptolemy. JDot is a Java- port of a part of the Graphviz project[2] Graphviz is build around the dot algorithm. The dot- algorithm lays out graphs to have a compact but clear structure. JDot is explained in more detail in section .

o *Requirements Definition*

We have stated in our solution approach that we will solve the KPN visualisation problem by creating a grapheditor using Eclipses GEF framework. Now knowing the implementation context of our solution we can define the requirements this graph editor must meet.

The first part of the requirements list states that each element of the KPN must a have a visual representation in the editors editpane and the structure of this representation must represent the structure of the KPN. Each element does not have the same importance to the KPN, and so we make a

choice which information must be directly available for the user, and which information will be a few clicks away.

We define three levels of importance

1. Elements that must be represented by a seperated visual object; KPN's, Entites and Links
2. Elements or properties that do not have to have to be represent by an entire object, but do hold information that must be quickly available; in and outputvariables and a links linearization
3. Elements which appear by name and/or value in other parts of the visualization; IoPorts and Entity- and PortDomains and a KPN's parameters

To put this in requirement form:

Requirement 1.1.1: A KPN must be represented as the main editpane
Requirement 1.1.2: An Entity must be represented by a seperate object in the main editpane
Requirement 1.1.3 A Link must be represented by a seperate object in the main editpane.

Requirement 1.2.1: The name of the in- and outputvariable of a link should be quickly available to the user
Requirement 1.2.2: The linearization type of a link should be quickly available to the user
Requirement 1.2.3: The name of an entity's assignstatement should be quickly available to the user.

Requirement 1.3.1: The names of the ports of an edge must be available to the user somewhere in the editor.
Requirement 1.3.2 The value of a KPN's parameters must be available to the user.
Requirement 1.3.3: The domain of an entity should be available to the user.
Requirement 1.3.4: The domain of a port should be available to the user.

Apart from these KPN elements, there are some properties of KPN elements that are of special interest to the user, namely an entity's workload (also called its size) and a links data transfer amount (again also called its size). These are vital in deciding which entity should be mapped to which processor or copressor[6]. These sizes do not have a given value, but are calculated from a size equation property ,determined for each node and entity by the compaan tool, by substituting the KPN's parameter values in these equations. It's essential to the user not only which size an entity or link has, but also how big this size is compared to that of the other entities or links:

Requirement 1.4.1: The user must be able to access the size of each node and edge quickly
Requirement 1.4.2: The must be able to view the relative size of each node and edge quickly

As stated before, the structure of the KPN must be represented in the editor. In particular, each link is, as the name suggests, a link between two nodes, that is:

Requirement 1.5.1: The visual representation of a link must connect the visual representations of two entities.

The overal layout of the edit pane should be clear and compact:

Requirement 1.6.1: The editor must provide an automatical layout for the visual representation of the KPN.
Requirement 1.6.2: Nodes may not overlap and edges should cross nodes and other edges as little as possible.
Requirement 1.6.3: The layout of the graph must imply the direction of dataflow in the process network.

The second part of the requirements list will deal with the possible user interactions with the editor. The first requirements are straightforward and deal with the persistency of the KPN: where the kpn is stored.

Requirement 2.1.1: A KPN can be loaded from a .kpn file.
Requirement 2.1.2: A KPN can be saved to a kpn file.

When the user is not content with automatic layout the editor provides, he should be able to manually place an entity at other locations.

Requirement 2.2.1: The user must be able to place a node, anywhere within the editpane.

The user might want to add some textual notes to the editpane, for future reference. He might or might not want these to appear in the editpane:

Requirement 2.3.1: The user should be able to add annotations to the visual representation of a KPN, and edit these texts.
Requirement 2.3.2: The user should be able to add notes to an entity or link. These notes must be accessible somewhere in the user interface, but should not be displayed inside the editpane.

All KPN elements are already named, but a user may want to change it.

Requirement 2.4.1*: The user should be able to change an entity's name.*
Requirement 2.4.2*: The user should be able to change a links name.*
Requirement 2.4.3*: The user should be able to change a ports name*
Requirement 2.4.4*: The user should be able to change a variables name.*

The generated visual representation may be used to clarify design choices to other people and should therefore not only be available within the editor:

Requirement 2.5.1: The user should be able to print out the contents of the main editpane.
Requirement 2.5.2: The user should be able to save the contents of the main editpane in several image formats.

The third part of the requirement list deal with the manipulations of the KPN that compromise the KPN's integrity:

Requirement 3.1.1: The user should be able to add an entity to a KPN.
Requirement 3.1.2: The user should be able connect two entities with a link.
Requirement 3.1.3: The user should be able to add a variable to a KPN.

A node in a KPN is one of three types, a source, transformer or sinkentity. A sourceentity is an entity that represents the input of a program. Therefore it can have no incoming links. In the same a sinkentity represents the output of a progam and can have no links going out of it.

Requirement 3.1.4 A sourceentity may not be the writeentity of a link.
Requirement 3.1.5 A sinkentity may not be the readentity of a link.

For the same reasons a user will want to be able to remove elements from a graph:

Requirement 3.2.1: The user should be able to remove an entity from a KPN.
Requirement 3.2.2: The user should be able to remove a link from a KPN.

Requirement 3.3.1: The user should be able to edit nodedomains.

Requirement 3.3.2: The user should be able to edit portdomains.

Our KPN editor must handle hierarchical graphs: it can contain entities that represent an entire KPN. We will call such an an entity a clusterentity. The KPN it represents is called a clusterKPN. Each input and outputvariable on a clusterentity representents a source or sinkentity in its clusterKPN respectively.

As we will see later when we perform a case study (see paragraph

Case study), sometimes there is already a KPN present that performs a certain entity's process. Therefore we define an operation that is called decomposing an entity: this loads a KPN, converts the entity into a clusterentity, and then sets the KPN as its clusterKPN.

This leads to the following requirements.

Requirement 3.3.1: The user should be able to add a clusterentity.
Requirement 3.3.2: The user should be able to view the KPN corresponding to a clusterentity.
Requirement 3.3.3: Adding a port to a clusternode adds an entity to the KPN.
Requirement 3.3.4: Adding a source or sinkentity to a clusterKPN adds a port to the clusternode.
Requirement 3.3.5: The user should be able to decompose a clusternode.

Having defined the requirements the KPN editor should meet, we will show how they are met in the next paragraphs.

o Background

In the next paragraph we will discuss the implementation of the design choices presented in the previous paragraphs, but before we get into details, we will first present a short introduction to our three most important parts of our software structure: the Model-View-Controller pattern, GEF's implementation of this design pattern, and the JDot datastructure.

▪ Model-View-Controller pattern

The Model-View-Controller(MVC) pattern is an software design pattern, that is, it specifies the roles of high-level parts of a software package[7] . The main feature of the MVC pattern is that the model (the datastructure) is seperated from the view (the visual representation of the data), that is: the model's code and the view's code do not reference each other directly; all communication between the model and the view is handled by the controller.

The advantages of this pattern come from the fact that the view and the model are seperated by the controller, so that they can be changed independently. This means that we can have multiple views (visual representation) of a single model. When interaction with one of the view occurs, the underlying is changed, and the controller updates all of the views. It also means that the model can be changed without having to change the views.

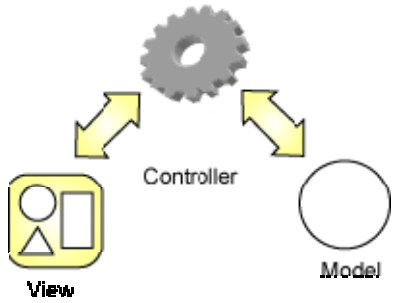


Figure 6: The view and model do not communcate directly, but through the controller.

In short, the roles of the three parts of the MVC- pattern can be described as follows.

Model
The representation of the information that is used by the application. Whenever the model changes it notifies all controllers that work on this model.

View
The representation of the model as a user interface element. The user can interact with the view ia the user interface. These interactions are handled by the controller to change the model.

Controller
The role of the controller is twofold, firstly the model is updated in reaction to user interaction with the view. Secondly the views are updated in reaction to changes of the model. [5]

- **The Graphical Editing Framework**

GEF is a toolkit designed too easily create rich graphical editors[5]. It does this by specifying a global structure for the model, view and controller, and the methods for their interaction. GEF assumes that the model consists of several seperate distinguishable parts. Therefore there are several controllers, called EditParts. An EditPart exists for each part of the model. In the next three paragraphs we will describe what structure GEF implies on the model, view and controller, and how these three parts communicate.

Model
The structure of the model is the least predefined by GEF. When a model consists of several distuingishable parts (such as the nodes of a graph), GEF specifies that a PropertyChangeListener must listen to every seperate part of the model, and each part notifies all its listening PropertyChangeListeners when it changes.

Although it does not strictly comply to the specifcation of a JavaBean[8], we have chosen to call such a seperate part of the model a bean. A bean implements java.beans package to provide these notification mechanisms.

The controller
A controller EditPart exists for each model bean. A controller typically knows about two things: its model and its view. An EditPart must implement the PropertyChangeListener interface, so that it can listen to changes of its model. Every time the model changes, its corresponding EditPart's propertyChange method is called. There are some standard EditPart baseclasses for common graphical editing objects such as nodes, edges and ports.
When an EditPart is signalled by it's view that some action from the user has occured, the controller does not change the model itself, but instead creates a Command to do this, and places it on an execution stack. The command updates the model, and does not know about anything but the model. After the

model is updated, and the model has signalled the controller about it's change, the controller updates the associated views accordingly.

The View

The model is usually implemented by a canvas on which figures are displayed. The canvas corresponds to the entire model, and the figures on this canvas correspond to distinguishable parts of the model. The view reacts to user input by creating and running an action. This action then notifies the controller to create a command to change the model.

- **Graphviz, Dot and JDot**

Graphviz is a c-program that provides layout mechanims for graphs. To make this functionality available to Java we have created JDot. JDot is a Java Port of a part of the c-program Graphviz. Among other things, Graphviz specifies a graph structure, and methods to layout these graphs. JDot makes this functionality available to the Java programming language by using the JNI framework.

The basic structure of a JDotGraph is quite simple, as shown in figure 6. A JDotGraph consists of a list of nodes and edges. All nodes have a position, and all edges have a path (described by a bezier spline list). Apart from this all graphs, nodes and edge have a list of string properties.

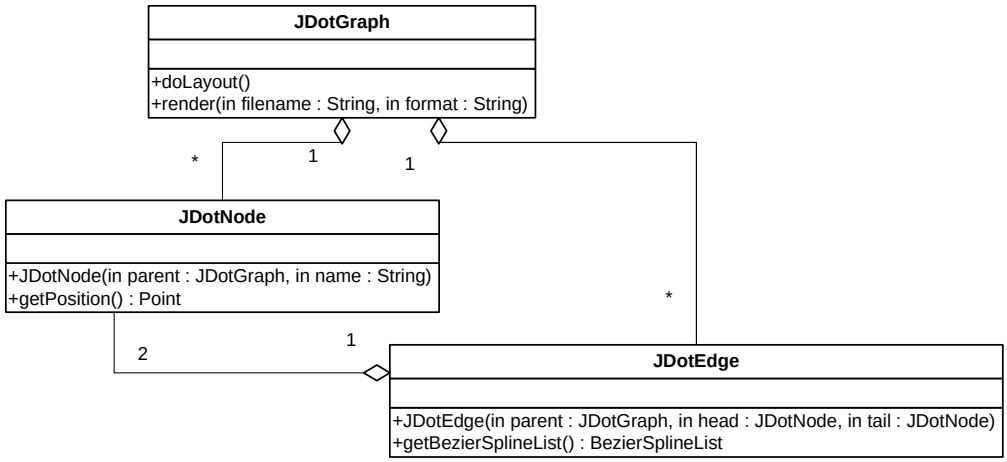


Figure 7: The JDot Api.

The doLayout method calculates new positions and paths for nodes and edges, by calling the dot layout algorithm implemented in the graphviz framework:[1]

dot - makes ``hierarchical'' or layered drawings of directed graphs. The layout algorithm aims edges in the same direction (top to bottom, or left to right) and then attempts to avoid edge crossings and reduce edge length."

An example of the results can be seen in figure 6 : the structure of this graph is presented in a clear and compact way; no edge crossings appear and the nodes are placed in a logic order.

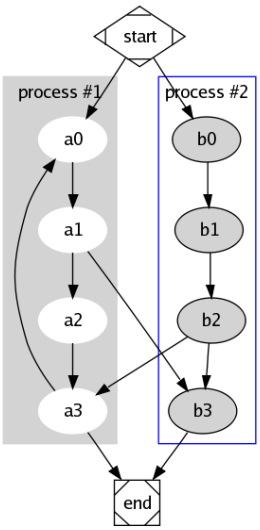


Figure 8: a simple graph laid out with Graphviz, using the dot- algorithm

o Solution implementation

In the next two paragraphs we will discuss the implementation of our solution to the KPN visualisation problem: we have created a KPN editor called Graphview. The first paragraph discusses how we use the GEF implementation of the MVC pattern, disucssing the three distinct software parts: the model, view and controller. The second paragraph shows how we used visualisation techniques to meet the requirements defined in paragraph o.

o Using GEF in Graphview

Using the Model-View-Controller pattern requires a clean seperation of these three parts. In the following paragraph we will discuss the structure of each of these parts, and how it fulfills its role

▪ The Model

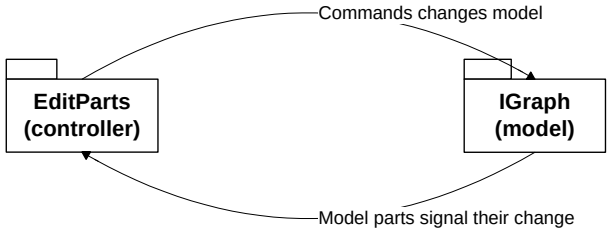
As mentioned before, we have chosen JDot as our model, mainly because of its layout capabilities. Unfortunately, JDot as a model is not enough to fulfill all our functional wishes, as it misses a notification mechanism and lacks the elements to succesfully mimic the KPN model. Therefore we have extended the JDot model with some additional functionality.

Notification

In the MVC pattern, the model has the responsibility of notifying the controller of changes. JDot provides no notification functionality. We have created the model.igraph package to solve this problem.
... shows several of the interfaces in the igraph package and the implementation files of the jdotdomain package.

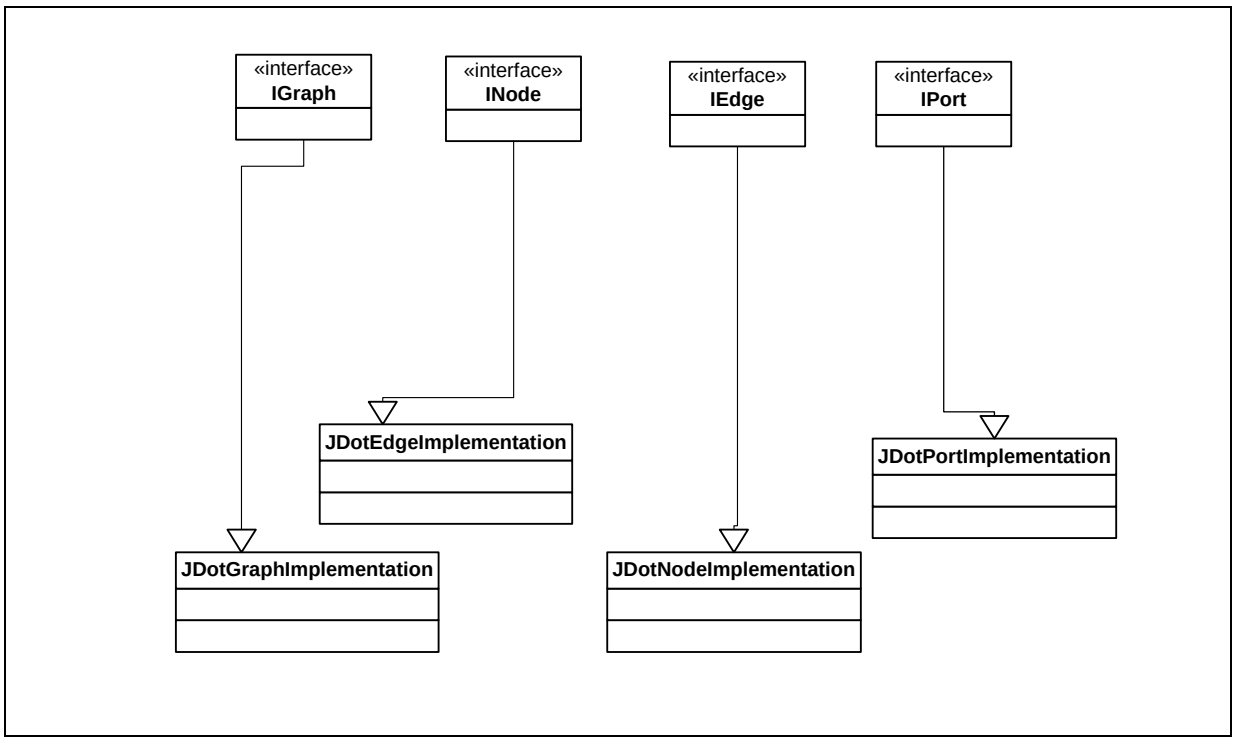
The igraph package consists of several interface classes, each providing an interface to a certain part of the model. For example the INode interface is the interface to a JDotNode object. The IBean interface is the base interface, and nearly every interface in the graphinterface package extends this interface, thus inheriting its notification functionality.

The interfacing to the JDot object is achieved via the model.jdotdomain package. Each class in this package is an implementation of a class in the igraph package. For example the JDotNodeImplementation implements the INode interface, and uses an JDotNode object to do this.



UML diagram 1: parts of the IGraph model signal the controller when changes in their structure occurs.

The controller is provided with implementations of the interfaces, yet it knows nothing about the specific implementation. For example, a GraphPart (EditPart for a graph) receives change notifications from an IGraph object, yet it knows nothing about the specific implementation. This gives us the possibility to change the underlying implementation of the interfaces. At the end of this chapter we will see that this comes in handy to change from the current JDot model to the KPN model.



UML diagram 2 (not all classes are shown)

In Code fragment 1 we see a new node being created. Note that this piece of code knows nothing about what implementation of INode is used.

```
...
IGraph graph = getBean();
INode newNode = graph.addNode(id);
...
```

Code fragment 1

In the next code fragment we look under the hood of the addNode method of the class JDotGraphImplementation:

```
public INode addNode(String name)
{
        JDotNode jdotNode =  new JDotNode(graph, name);
        INode node = JDotNodeImplementation.getInstance(jdotNode);

        firePropertyChange(NODE_ADDED);

        return node;
}
```

Code fragment 2

We see that, a new JDotNode is created and a JDotNodeImplementaton instance is created for this JDotNode (JDotNodeImplementation is a factory class). By using this construct we can use the functionality of the JDotNode object, without actually having to use Javas extend construct to inherit its functionality. The use of the extend construct is impossible since each implementation class already extends the DefaultBeanImplementation.

After the new implementation object is created, the controller is notified of the change to the model via the firePropertyChange(NODE_ADDED) call. How the controller handles this change will be discussed in the controller paragraph.

JDot must mimic KPN

Graphviewer editor takes an KPN file as input and constructs from the generated KPN structure a JDotGraph. Some parts of the KPN structure do not have counterparts in the JDot datastructure. We therefore augment JDot with additiontal functionality. In the paragraphs we will discuss these additional model elements.

We add these functions in a general way. For each element of JDot that lacks a certain property, the corresponding igraph interface specifies get/and or set methods for this property. The corresponding JDotImplementation class then usually has a field where this property is stored. We'll start with a straightforward example and continue by listing the rest of the missing functionality.

In the next paragraphs we will explain how we added the missing funcionality. Each paragraph presents a piece of missing functionality, and discuss why we need it. It will then present a list that says:

- What KPN element does not have a counterpart in JDot (and so we have to mimic it)
- What KPN methods use this KPN element.
- What IGraph interface specifies the functionality the mimicing JDotImplementation class must implement.
- What methods in the IGraph package classes use this mimicing method.

After we will show a short code fragment to give an idea how the method is implemented.

Parameters

The size of an edge is calculated based on the values of the parameters specified in the KPN. Since we wan't to display the size of an edge, we need these parameters and their values to be available to the JDot datastructure.

- KPN element that does not have a counterpart in JDot: Parameter
- Used by KPN.addParameter(Parameter parameter) and KPN.getParameters()

- IGraph interfaces that specify datastructure: IParameter and IParameterList interface
- IGraph interface methods that uses the datastructure: IGraph.getParameterList()

The next code fragment shows the implementation in the jdotdomain package.

```
public class JDotGraphImplementation extends JDotBeanWithStringPropertiesImplementation
implements IGraph
        ...
        private JDotParameterListImplementation parameterList;
        ...
        public JDotParameterListImplementation getParameterList()
        {
                if(parameterList == null)
                        parameterList = new JDotParameterListImplementation();
                return parameterList;
        }
        ...
}
```

Code fragment 3

Nodetype

An Entity has a method getType() to retrieve the entity's type. It is one of three values Source, Sink and Transformer. When an entity is a sourceentity there may be no link going into it, and likewise a sinkentity may have no links going out of it. To check if a certain link may be created this information must available to the editor.

- KPN element that has no counterpart in JDot: Entity.Type
- Used by Entity(String name, Type type) and Entity.getType()
- IGraph enum that specifies datastructure : NodeType
- IGraph interface methods that uses the datastructure : INode.setType(NodeType type) and INode.getType()

Code fragment showing the implementation:

```
public class JDotNodeImplementation extends JDotBeanWithStringPropertiesImplementation implements INode
{
        ...
        private NodeType nodeType = NodeType.transformer;
        ...

        public NodeType getNodeType()
        {
                return nodeType;
        }


        public void setNodeType(NodeType nodeType)
        {
                this.nodeType = nodeType;
        }
        ...
}
```

Code fragment 4

Arguments

A KPN entity has a list of arguments. These are the in and output variables of the function the entity represents. A link is connected to an entity via an IoPort (which we will discuss in the next paragraph). Since a link represents the dataflow between entites, an IoPort object specifies which variable it 'transports'. IoPort.getArgument() returns the name of this argument. In JDot, edges connect nodes directly without the use of ports.

- KPN element that has no counterpart in JDot: Argument(a simple string)
- Used by IoPort.getArgument();
- IGraph interface that specifies dastructur: IPort
- IGraph interface methods that uses the datastructure: INode.getPortList() (among others)

```
public class JDotPortListImplementation extends JDotBeanImplementation implements IPortList
{
	private Hashtable<String, JDotPortImplementation> ports = new Hashtable<String, JDotPortImplementation>()
	...
	public IPort addSourcePort(String name, IPortLocation location)
	{
		return addPort(createPort(name, location, true));

	}

	protected JDotPortImplementation addPort(JDotPortImplementation port)
	{
		ports.put(port.getName(), port);

		firePropertyChange(SOURCE_PORT_ADDED);

		return port;
	}

	protected JDotPortImplementation createPort(String name, IPortLocation location, boolean isSourcePort)
	{
		return new JDotPortImplementation(name, location, false);
	}
	...
}
```

Code fragment 5

The previous code examples need some clarification since there are some difference in naming between KPN and the igraph package. What is called an argument in KPN, is called a part in IGraph. The best way to understand these new names is with a naval analogy. A ship docks in a port with by using anchor. Multiple ships can dock in a single port. In the same way an IEdge docks on an IPort, via an IAnchor. Multiple IEdges can dock to a single IPort, but the IAnchor object is unique for each Edge.

Anchors

As said before an Link anchors to an Entity through an IoPort.

- KPN element that has no counterpart in JDot: IoPort, and subclasses WritePort and ReadPort.
- Used by Entity.getReadPorts(), Entity.getWritePorts(), Link.getReadPort() and Link.getWritePort()

- IGraph interface that specifies datastructure : IAnchor
- IGraph interface methods that uses the datastructure: IEdge.getInAnchor() and IEdge.getOutAnchor() (amongst others).

Code fragment 6 shows the methods needed to connect two nodes, using anchors.

```
public class JDotPortImplementation extends JDotBeanImplementation implements IPort
{
...
public IAnchor createAnchor()
{
	JDotAnchorImplementation anchor = new JDotAnchorImplementation(this);

	firePropertyChange(ANCHOR_CREATED);

	return anchor;

}
...
}


public class JDotGraphImplementation extends JDotBeanWithStringPropertiesImplementation
implements IGraph
{
	...
	public IEdge addEdge(INode tail, INode head, IAnchor outAnchor, IAnchor inAnchor)
	{
		...
		IEdge edge = ...

		outAnchor.connectTo(edge);
		inAnchor.connectTo(edge);
		...

	}
	...
}


public class JDotAnchorImplementation extends JDotBeanImplementation implements IAnchor
{
	...
	public void connectTo(IEdge edge)
	{
		...
		this.connectedTo = (JDotEdgeImplementation) edge;
		getPort().connectTo(edge, this);
		if(getPort().isSourcePort())
			connectedTo.setOutAnchor(this);
		else
			connectedTo.setInAnchor(this);
		...

	}
	...
}

public class JDotEdgeImplementation extends JDotBeanWithStringPropertiesImplementation
implements IEdge
{
	private JDotAnchorImplementation inAnchor;
	...

	public IAnchor getInAnchor()
	{
		return inAnchor;
	}

	protected void setInAnchor(JDotAnchorImplementation inAnchor)
	{
		this.inAnchor = inAnchor;
	}
	...
```

Code fragment 6

To connect two nodes, two anchors are created: the port on the tailnode and the port on the headnode is chosen and IPort.createAnchor() is called on these ports. Then IGraph.addEdge(...) is called, which creates an edge and anchors it to these two created ports.

Domains

"A node domain is a collection of polytopes, a function, and a set of port domains." In the KPN format, a Domain has a list of Constraints matrices, a Mapping matrix and a context matrix. The user will want to view and possibly edit these matrices in the editor, so a nodes (and its ports) domain should be part of the model. The IDomain interface specifies what a domain implementation must provide. This interface is shown in Code Fragment 7*. An implementation of the IDomain interface mimics KPN Domain class, so that the Domain functionality is available to our model.*

```
public interface IDomain extends IBean
{
        public List<IMatrix> listConstraints();
        public IMatrix getMapping();
        public IMatrix getContext();

        public void setMapping(IMatrix mapping);
        public void setContext(IMatrix mapping);
        public void setConstraint(int index, IMatrix constraint);
        public void addConstraint(IMatrix constraint);

        public List<String> listIndices();
        public void addIndex(String index);
        public List<String> listParameters();
        public void addParameter(String parameter);
        public IMatrix addNewConstraint();

        public void copyDomain(IDomain otherDomain);
}
```

Code Fragment 7

Labels

Textlabels are a tool to provide textual feedback to the user. We'll see when discussing the view that labels can take on al sorts of roles, but its basic behaviour will always be to display textual information in the graph. The two most relevant properties of a label, the text it will display and the position it will be placed are shown in Code Fragment 8.

```
public abstract class Label {

        ...
        private String text;
        Point position;
        ...
}
```

Code Fragment 8

Textlabels are not a part of the KPN datastructure, so when a KPN is saved we can not convert a label to its KPN counterpart, simply because there is no counterpart. We solve this by converting the label to several string properties, with a special prefix. When the model is loaded, the FromKPNConvertor

recognizes these properties by their prefix and uses them to recreate the Label. We will discuss the FromKPNConvertor class in more detail when discussing the controller in paragraph 0.

Clusters

A KPN partitions a program into subprograms. A subprogram itself, again can be partitioned into subprograms. A program or algorithm is process which takes a number of inputs, performs some operations, and provides a number of outputs.
The model is provided with a datastructure for hierarchical graphs, or clustergraphs as we call them, to represent hyrarchical subprograms. In a graph, a clusternode represents a entire KPN. Functionality for hierarchical graphs is not (yet) present in the KPN model, but it is provided by the IGraph interface package.

```
public interface IClusterNode extends INode
{
        public IClusterGraph getClusterGraph();
        public void createClusterGraph();

        public IClusterPortList getPortList();

        ...
}

public interface IClusterGraph extends IGraph
{
        public IClusterNode getClusterNode();
        public void setClusterNode(IClusterNode clusterNode);
}

public interface IClusterPort extends IPort
{
        public INode getRepresentedNode();
}
```

Code Fragment 9

In Code Fragment 9, the IClusterNode interface specifies that the corresponding ClusterGraph must be available. For convenience, a new IClusterGraph can be created if none is present yet. While not explicitely required, the IClusterNode's id must equal the IClusterGraph's id, and the IClusterNode's ports names must equal the names of the nodes in the IClusterGraph. To help enforce this last requirement the IClusterNode's portlist contains IClusterPorts's. These clusterports know about the corresponding INode in the clustergraph.

Replacing the model

The igraph package only specifies how the graph model can be manipulated. An implementation of these interfaces can be made for any graph-like datastructure. This is one of the reasons for choosing the Model-View-Controller pattern: each of it's three parts can be replaced by another implementation without affecting the other two.

As the model was developed it has also becane more and more clear that the choice for JDot as the model might have been unwise. It would have been better to use the KPN as model, and to augment this

with placement properties for entities and links. Then at 'layout time' a temporary JDot structure could be built to perform the layout. The generated layout- position info could then be copied to the KPN model.

Now the jdotdomain package in many ways a rewriting of the KPN model; such redundancy could be avoided by choosing the proper model. Changing the model would also help in keeping the model consistent. As it is now, lot's of work is done to keep the JDot model consistent with the KPN model structure. When a KPN model is used, this is no longer needed.

- **The Controller**

" The controllers *bridges the view and model (see Figure 1). Each controller, or* EditPart *as they are called here, is responsible both for mapping the model to its view, and for making changes to the model."[86]*

The control flow of the Graphview editor is shown in UML Diagram 2, and can be characterized as follows:

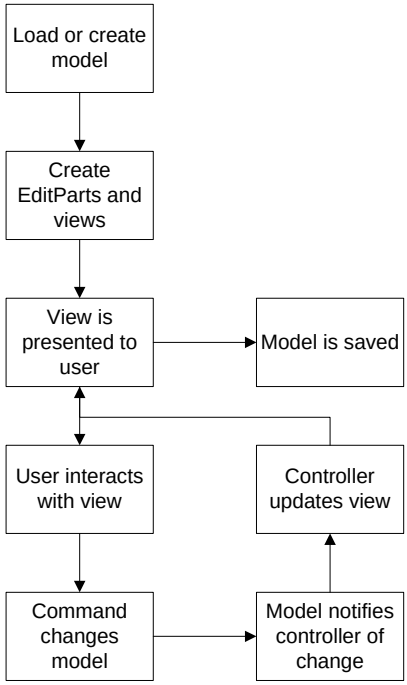
1. Load or create the model.
2. Create editparts and views (controllers) for the different parts of the model.

Repeat:

3. The user interacts with the view
4. The controller creates a command which updates the model.
5. The model notifies the controller that it has changed.
6. The controller maps the model to the view (that is, to reflect the changes in the model).

End repeat

7. Save the model.



UML diagram 3 : Graphview flowchart.

In the following paragraphs we discuss these steps in order and clarify them by using examples.

Load or create the model

The Graphview is activated when a .kpn file is openend in Eclipse. The loading of the model is done by the Graphviewer class, which creates the main graph EditPart, but is not part of the controller itself.

First the Graphviewer loads a KPN object from a .kpn file. Then the FromKPNConvertor is used to convert the loaded KPN object into a JDot model. It extends the KPNExtendedVisitor class, which is based on the

visitor pattern to visit each part of the KPN model. As the Visitor pattern dictates, each KPN element can visited by having the element accept a visitor class.

In Code Fragment 10 we see an example of one of these visitStructure methods. Here an entity is converted to an INode object. First we retrieve the parent graph (by using stack in which store previously created beans). Then we use a command to add a node to the graph (why we use commands is explained in 0). Then we use a helper class to convert the properties of the entity. Lastly, the nodetype is set. The we visit the nodes children elements, which, in this case, is its AssignStatement.

```
public class FromKpnConvertor extends KPNExtendedVisitor
{
	...
	public void visitStructure(Entity entity)
	{
		IGraph graph = (IGraph) createdBeanStack.peek();

		AddNodeCommand addCommand;
		boolean isClusterEntity = isClusterEntity(entity);
		if(isClusterEntity(entity))
			addCommand = new AddClusterNodeCommand();
		else
			addCommand = new AddNodeCommand();

		addCommand.setParent(graph);
		addCommand.setId(entity.getName());
		addCommand.execute();

		INode node = addCommand.getCreatedNode();

		FromKpnPropertyConvertor.convertProperties(entity, node);

		if(isClusterEntity)
		{
			LoadClusterGraphCommand loadCommand = new LoadClusterGraphCommand();
			loadCommand.setClusterNode((IClusterNode) addCommand.getCreatedNode());
			loadCommand.execute();
		}

		ChangeNodeTypeCommand changeTypeCommand = new ChangeNodeTypeCommand();
		NodeType nodeType = determineNodeType(entity);

		changeTypeCommand.setNode(node);
		changeTypeCommand.setNewNodeType(nodeType);
		changeTypeCommand.execute();

		createdBeanStack.push(node);
		entity.getAssignStatements().get(0).accept(this);
		createdBeanStack.pop();
	}
	...
}
```

Code Fragment 10

For each KPN element such a method is implemented. From an elements visitStructure() method the accept method on its childrens elements is called. This way all elements in a KPN are converted to igraph elements.

Create editparts and views

When the model is loadded the controllers are created. This is done in the Graphviewer.initializeGraphicalViewer() method as we can see in Code fragment 11. This method calls the setContents(), which then creates an GraphPart editpart for the graph, and sets this editpart as the viewers contents. This results in creating and setting a view for the graph's editpart. Also, it's children's (such as nodes and edges) EditParts are created and provided with a view.

For the graph this view is a canvas on which the views of it's childrens editparts are painted. A node is presented as a ellipseshape with a label, and an edge is a curve between two nodes., implemented programmatically by instances of the Draw2D IFigure interface. When an EditPart is created and activated, these figures are updated to reflect the contents of the model.

```
public class GraphViewer extends GraphicalEditorWithPalette {

        ...
        protected void initializeGraphicalViewer() {

                ...
                getGraphicalViewer().setContents(graph);
                ...

        }
        ...
}


public abstract class AbstractEditPartViewer
        implements EditPartViewer
{
        ...
        public void setContents(Object contents) {
                ...
                setContents(getEditPartFactory().
                                createEditPart(null, contents));
        }
        ...
}
```

Code fragment 11

User interaction with the view

In the previous paragraph we saw that an EditPart was created, and that it was provided with a view object. When this step is complete, the views are painted to the screen. In the case of the Graphviewer editor, the user is presented with a canvas containing nodes, edges and some labels.

The canvas provides a number of ways to interact with it, which we will discuss in paragraph. . Every time the user performs an action on the view a Command is created by an EditPart to modify the model. A user action can be as simple moving a node, or as complex as editing domain matrices.

Updating the model: commands

We will discuss at a small example: moving an existing node. The user interaction to trigger this update are straightforward:

1. The user selects a node by clicking it (the view remembers which figures are selected)
2. The user drags the node to new location

The moving of a node is performed by an EditPolicy. EditPolicies provide support for standard editor operations , such as adding nodes to a graph, moving these nodes and connecting by edges. Policies are installed for an EditPart in the EditPart.createEditPolicies() method. GraphPart uses the GraphLayoutPolicy that is a subclass of the XYLayoutPolicy. Basically this policy provides the functionality to add and move elements within a graph. In Code fragment 12, we see the GraphLayoutPolicy creates a ChangeNodeConstraintCommand to move a node a and sets the appropriate fields.

This command is then executed by the EditPart (this is done by the GEF package using a command stack). In Code fragment 13 we see that when the execute() method of this command is called, the command changes the position of the node. Every command can be undone (unless specified otherwise), in this particular case we remember the old constraints to do this.

```
public class GraphLayoutPolicy extends XYLayoutEditPolicy{

	...
	if(child instanceof NodePart)
	{
		INode node = ((NodePart) child).getBean();

		ChangeNodeConstraintCommand command = new ChangeNodeConstraintCommand();
		command.setNode(node);
		command.setConstraints((Rectangle) constraint);

		return command;
	}
	...
}
```

Code fragment 12

```
public class ChangeNodeConstraintCommand extends Command
{
	...
	public void execute() {

		oldSize = node.getSize();
		oldLocation = node.getLocation();

		node.setLocation(constraints.getLocation());
		node.setSize(constraints.getSize());

	}

	public void undo() {

		node.setLocation(oldLocation);
		node.setSize(oldSize);


	}
	...
}
```

Code fragment 13

Change notification

A command makes one or more changes changes to the model. As discussed before, changes to the model trigger the notification mechanism. Each model object signal it's PropertyChangeListeners about the changes made. Code Fragment 14 shows how a GraphPart (or any EditPart for that matter) is registered as a ChangeListener with it's model object in the EditPart.activate() method.

```
public class GraphPart extends AbstractGraphicalEditPart implements ChangeReactor,
                PropertyChangeListener, IElementPart<GraphPropertySource, GraphPropertyAdapter, GraphProperty
        ...
        public void activate() {
                ...
                getBean().addPropertyChangeListener(this);
                ...

        }
        ...
}
```

Code Fragment 14

In Code Fragment 15 we see that when, for example, the location of a node is changed, the bean (by subclassing JDotBeanImplentation) calls its firePropertyChange() method, which in turn calls the propertyChange() method on all its PropertyChangeListeners.

```
public class JDotNodeImplementation extends JDotBeanWithStringPropertiesImplementation
implements INode
{
        ...
        public void setLocation(Point location)
        {
                java.awt.Point position = new java.awt.Point(location.x, location.y);
                node.getAttributes().setPosition(position);

                firePropertyChange(LOCATION, location);
        }
        ...
}
```

Code Fragment 15

Updating the view

The model notifies the controller that the view must be updated. As we saw in the previous paragraph, this is done by calling the propertyChange method. When the propertyChange method is called it is provided with a PropertyChange event. This event contains among other things a propertyName, the name of the property of the model that has been changed.

We can divide these properties into two types. The first type is the structural property. For example if a node has been added to a graph, the graph will pass the NODE_ADDED constant as propertyname.

The other type of property is the string property. The parts of the graph model that have string properties (graphs, nodes and edges), will pass the name of the property when the value of this string property changes.

```
public class NodePart ...
{
	public void propertyChange(PropertyChangeEvent evt)
	{
		...
		if (propertyName.equals(NodePropertySource.INEDGE))
			refresh();
		else if (propertyName.equals(NodePropertySource.OUTEDGE))
			refresh();
		else if (propertyName.equals(NodePropertySource.LAYOUT))
			refresh();
		else if (propertyName.equals(NodePropertySource.EDGE))
			refresh();
		else if(propertyName.equals(NodePropertySource.SOURCE_PORT))
			refresh();
		else if(propertyName.equals(NodePropertySource.TARGET_PORT))
			refresh();
		else
			getPropertySource().nodeProperties.performReaction(this,
					propertyName);
	}
}
```

Code Fragment 16

In Code Fragment 16 we see that the first tests if a structural change was executed on a Node. In that case, the refresh method is performed. The refresh method first updates the Controller, creating EditParts for newly created model parts, and removing EditParts for removed model parts, and the updates the view to match these structural changes.

For a stringproperty the reaction to the model change is stored in the properties table. The performReaction method first retrieves the appropriate property, and then performs its changeReaction. This usually results in calling the refreshVisuals method on the current EditPart.

For all EditParts we have implemented the refreshVisuals method to delegate the updating of the figures to a FigureUpdater class. In Code fragment 17, we see that the NodeFigureUpdater uses the node's propertyTable to change some of the NodeFigures properties, so that it reflects the state of the model. The same type of updating occurs for all other EditParts.

```
public class NodeFigureUpdater ...{

	...

	public void updateFigure(NodeFigure figure)
	{
		updateFigureData(figure);
		...
		figure.repaint();
	}

	protected void updateFigureData(NodeFigure figure)
	{
		NodeProperties properties = getPropertyTable();

		figure.setName(properties.labelProperty.getValue());
		updateShape(figure);
		updateColors(figure);

		figure.setLabelFont(properties.fontProperty.getValue());
		figure.setBounds(properties.boundsProperty.getValue());
		figure.setIsClusterNodeFigure(getBean().isClusterNode());
	}
}
```

Code fragment 17

We must note that by using this updating scheme, all properties of a figure are updated whenever the corresponding model object changes. A better solution would be to only update the appropriate figure properties. Especially in large KPN's this can speed up the editor considerably. The updating of the view should be made more efficient, and changing the code to achieve could be part of a follow up project.

Save the model.

After continuously manipulating the model, the user will want to save the new version of the model. This is done in two steps, as shown in Code fragment 18. Firstly, the JDot model is converted 'back' to a KPN, using the ToKpnConvertor, which we'll discuss shortly. Following that, Graphviewer uses compaan's built-in functionality to write the KPN model to a file.

```
public class KpnFileManager
{
	...

	public static void saveGraph(IGraph graph, String fileName)
	{

		KPN kpn = convertGraph(graph);

		saveKpn(kpn, fileName);
		...

	}

	private static KPN convertGraph(IGraph graph)
	{
		ToKpnConvertor convertor = new ToKpnConvertor();
		graph.accept(convertor);

		return convertor.getKpn();
	}

	private static void saveKpn(KPN kpn, String fileName)
	{
		PrintStream printStream = new PrintStream(...);

		new kernel.visitor.KPNModelVisitor(printStream).visitStructure(kpn);
		...
	}

}
```

Code fragment 18

The ToKpnConvertor is in many ways the opposite of the FromKpnConvertor class. In Code fragment 19, we see how the ToKpnConvertor object visits all the seperate object of the JDot model and convert them to their KPN counterparts. We see that a new Entity is created and added to the graph, representing a INode. In the same way we saw in Code fragment 29, an INode was created to mimic an Entity. If the model hasn't changed inbetween saving and loading, the new Entity will hold the same information as the old one.

```
public class ToKpnConvertor extends AbstractGraphVisitor
{

	public void visitBean(IGraph graph)
	{
		kpn = kpnFactory.makeKPN(graph.getId());
		...

		pushKpnElement(kpn);
		graph.getParameterList().accept(this);
		visitNodes(graph.listNodes());
		visitEdges(graph.listEdges());
		popKpnElement();
	}

	public void visitBean(INode node)
	{
		KPN kpn = (KPN) kpnElementStack.peek();

		Entity entity = kpnFactory.makeEntity(node.getId());
		kpn.addEntity(entity);

		...

		pushKpnElement(entity);
		node.getDomain().accept(this);
		node.getPortList().accept(this);
		popKpnElement();
		...
	}



	...
}
```

Code fragment 19

Properties

Before we go on to the view, we will talk about properties, a special part of the controller. In our IGraph model graphs and nodes and edges provide a list of properties. Here, you can store the values of string properties. The properties can be retrieved by name. JDot itself has some methods to store and retrieve integer and floating point properties, by converting them from and to strings. Unfortunately there is no way to determine the type of a specific property, so if we were to use these properties we would always need to specify the specific type of the property.

Properties server a lot of purposes. Firstly, properties are displayed and can be edited in the Properties View. This is designed in such a way that we need to have a PropertyDescriptor for each property. Also we need to retrieve and set the value of a property solely on the base of its name. This means that we need to build and if/else loop choosing from the list possible property names. Such a list is generally considered bad coding.

To solve this problem, a Property object was created for each property. The list of these property objects for each graph, node and edges is stored in a PropertyTable. Each node has the same Property objects (and so have all graphs and edges). The property object stores all information about the property in one object. It therefore contains:

- Its id, title in the Properties view, and default value.
- Methods to set and get the value of the property (this is done using the models string properties)
- Methods to perform the reaction an EditPart must perform when this property changes.
- Methods to create a PropertyDescriptor and retrieve the value for the propertyEditor (and set the value from the propertyEditor). It also specifies if it can be edited, or if it even should be shown in the editor.

There are four basic types of properties:

- JDot property
- Composite property
- Custom property
- Calculated property

JDotProperty

JDotProperty mimic properties of any type by saving them as String properties in an IGraph bean. Examples are JDotColorProperty, JDotIntegerProperty, but of course also the trivial JDotStringProperty.

CompositeProperty

The CompositeProperty provides for a property that combines several JDotProperties, called PartProperties. For example a FontProperty combines a IntegerProperty(size) and a StringProperty(name).

CustomProperty

Custom property are used to store the values of user defined notes. All custom properties have a special prefix to their name so that they can be identified as being a custom property when loading and saving.

CalculatedProperty

A calculated property does not store a value itself, but mere calculates its value from the value of another property. Examples of this are the caculated sizes of nodes and edges.

We must note, that properties are NOT part of the model. When we say that the property changes the value of an IGraph object's string property, we mean that it creates a command that does this. sometimes though properties are used in Commands to perform a change to the model (which is a contradiction to the motto that the Command may only know about the model). Again resolving this contradiction can be an interesting followup project.

This concludes the paragraph about the controller. In this paragraph we have shown how we used GEF to create a collection of EditParts as well as some helping classes to implement the Controller in the Model-View-Controller pattern. In the next paragraph we will discuss how the View is implemented.

- **The View**

The view represents the information stored in the model, but also the serves as the user interface to manipulating the model. Our view consists of five distinct parts:. These five parts area:

- The editing area
- The palette
- The properties view
- Dialogs
- Menu's and toolbars

Each of these parts will be discusses in a seperate paragraph.

The editing area

The editing area is the first and most important part of the editor's view. In Figure 9 we see this area displaying a small graph. The entire pane is the view for a graph, the ellipses are the views for nodes, and the lines represent the edges.

Examples of the information shown by the editing area are the graph's name (at the top of the pane) and the nodes names (shown as a label inside the ellipse). Also, we see a label on the edges showing their 'calculated size'.

The interaction level of this pane is fairly limited. Nodes, labels and edges can be moved about. With the use of the Palette more can be done.

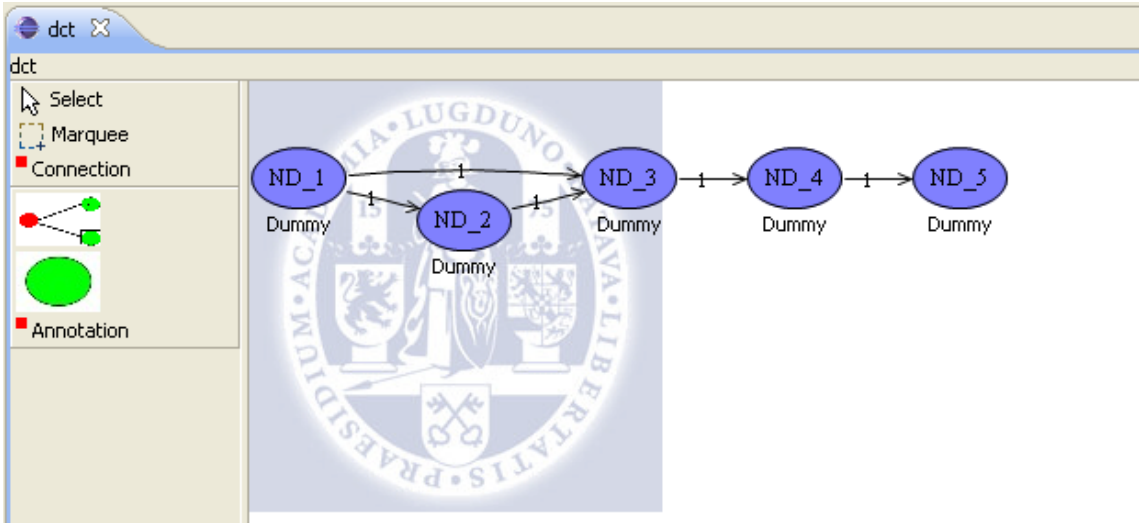


Figure 9

Figure 9 shows The editpane(in the middle) with the Palette on the left. The palette contains two groups, which contain several buttons. One group is for selecting items, and one for adding objects to the graph.

We see that on the editpane the line from ND_1 to ND_3 is not a straight line, but a curve. When we discussed JDot in paragraph we saw that laying out the graph results in a spline list for each edge. Simply said, a spline list is a list of points that describes a a polynomial function. The curve is the visual representation of this polynomial. When a node is moved, we are presented with a problem: we now

have a spline list, but the ends of the path that this spline list defines are not at the correct positions anymore.

A smart solution seems to be to move the spline points with the node. This works nicely when only one node of an edge is moved, but whene both nodes are no longer at their original place, we have a problem: we can not move the splines in two directions. A possible solution would be to average between the movement of the two nodes, so that spline points close to the tail node move more with the tail node, and spline points closer to the headnode move more with the head node. For followup projects it would be interesting to try implement this correctly.

Another interesting option to look into would be to use the dot algorithm to pin the nodes position, and only layout the edges. This option is present in graphviz, but not yet implemented in JDot.

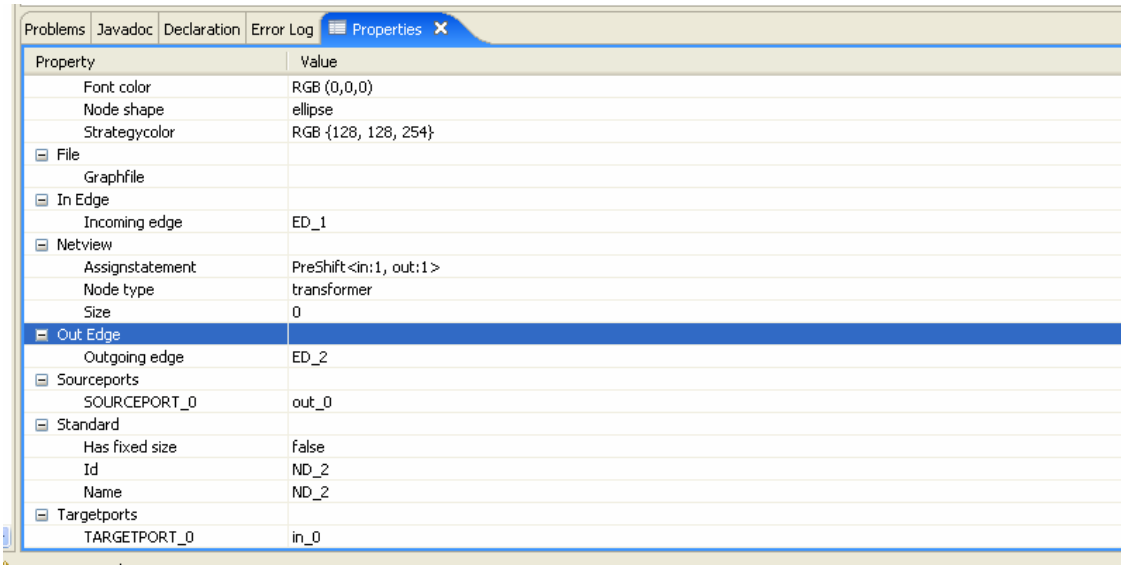
We have therefore decided that edges of nodes that have moved will just be displayed by straight lines. An exception to this are self loops. Displaying self loops as straight lines paints them over the node. Therefore when moving a node with a selfloop the splines are not affected.

Palette

On the left of Figure 9 we see the palette. The palette is a list of tools for manipulating the objects in the panel. Our pallete is are divided into two parts. The first part contains the 'Selection' and the 'Marquee' tool. These tools are used for selection a single and multiple graph parts respectively. The second parts contains tools for adding new elements to the graph. Elements that can be added to a graph are a (cluster)node, an edge, and an annotation. Adding an element is done by selecting its tool in the palette and then clicking at the spot in editing area where you want it placed. Placing an edges is connecting two nodes: first click the tailnode, followed by the headnode.

Properties View

The properties view is a builtin view for eclipse. When opened, this view displays information about the properties of a selected object. Usually this is a file, or a java method, but we modified it to display information about a selected graph object. The properties view is not a view that is automatically opened, it can be activated by choosing the Window -> Show View -> Others -> General -> Properties.



The Properties View displays the properties of the part of the model the selected EditPart represents. Code Fragment 20 shows that when a graphelement is selected, it requests a IPropertySource Adapter

instance from the relevant GraphPart. The class diagram of the IPropertySource interface can be seen in class diagram 1.

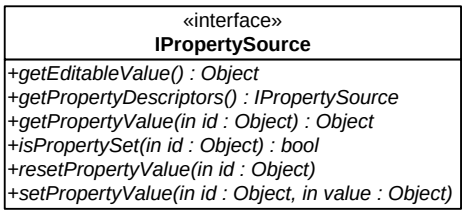
The Properties View uses the retrieved IPropertySource instance to determine the properties it needs to display. For this it uses the getPropertyDescriptors method. A propertydescriptor describes a property, specifying it's identifier, the title to be displayed in the properties view and in which category it belongs. Then, for each propertydescriptor, it retrieves the property value, with the getPropertyValue() method.

```
public class GraphPart extends AbstractGraphicalEditPart implements ChangeReactor,
                PropertyChangeListener, IElementPart<GraphPropertySource, GraphPropertyAdapter, GraphProperty
{

        ...
        public Object getAdapter(Class adapter) {
                ...
                if (adapter == IPropertySource.class)
                        return getPropertySource();
                ...
        }

        public GraphPropertySource getPropertySource() {
                return GraphPropertySource.getInstance(getBean());
        }
        ...
}
```

Code Fragment 20



class diagram 1

The value of a property can also be edited in the Properties View. If the user selects a property in the properties view, an editor is opened for the property. Usually this means that the value field changes into a textfield where the value can be edited, but more complex editor like a colorselector are not uncommon. When the editor is closed the new propertyvalue is set by calling the setPropertyValue() method. The propertySource retrieves the appropriate property and sets it's value by calling the setValueFromPropertySheet() method, as shown in Code Fragment 22.

One of the main reasons to create the properties package was to make the implementation of this interface simpler, as explained in paragraph 0. For example, when we have a simple string properties the getPropertyValue() method looks like Code Fragment 21: we'd have to write an if/else statement for each property, since there is no other way to associate the setting methodof a property with its identifier. When properties are objects, we can store these objects in a hash like structure, and retrieve them by name, and call a getValue() method on this object. Using such a strategy, this long list of statements is replaced by the short method in Code Fragment 22. We see that each property is an instance of (a

subclass of) the AbstractProperty class. This class specifies that each property must have certain methods such as getValue(), setValue() and getPropertyDescriptor().

```
public Object getPropertyValue(Object id) {

        if(id.equals("name"))
                return getName();
        else if(id.equals("fillcolor"))
                return getFillColor();
        else
        ...

}
```

Code Fragment 21

```
public Object getPropertyValue(Object id) {

        PropertyTable properties = getProperties();
        AbstractProperty property = properties.getProperty((String) id);

        return property.getValueForPropertySheet();
}
```

Code Fragment 22

Menu's and toolbars

An indispensible part of any user interface are menu's and toolbars. They are commonly used for more complex operations, which might require wizards or other dialogs. Eclipse is equipped with menu's and toolbars and any plugin can add operations to them.
Graphviewer overrides the menu options save, load, undo, redo and print functions to perform the expected operations. The standard toolbar is augmented with the zoom option, to zoom in and out of a particular part of the graph.
While this is all fairly straightforward, a more interesting piece of Graphviewer is the contextmenu. The contextmenu is brought up by right- clicking an object in the graph. The user can then select the desired action from the context menu.

GEF contains the ContextMenuProvider class to write the context menu fast and effective. The buildContextMenu() method is overriden to add options to the menu. Code fragment 23 shows that for each menu option an action must be created. This action is run when the menu option is selected. The layout action executes immediately, but most context actions require some additional information from the user which we will talk about in the next paragraph.

As stated before, the model may only be altered with the use of commands, so a specific menu action must create a command to be executed. Therefore we created the abstract class GraphViewerAction (see Code Fragment 24). Each Action class subclasses this class, so that it must create a Command. Also an EditPart must be passed to the Action's constructor, so that the created command can be placed on the correct command stack.

```
public void buildContextMenu(final IMenuManager menu) {

	...
	menu.add(new GroupMarker(GEFActionConstants.GROUP_EDIT));

	IAction layoutAction = new LayoutAction(graphPart);
	menu.appendToGroup(GEFActionConstants.GROUP_EDIT, layoutAction);
	...
}
```

Code fragment 23

```
public abstract class GraphviewerAction extends Action{

	private EditPart editPart;

	public GraphviewerAction(EditPart editPart, String text)
	{
		super(text);
		this.editPart = editPart;
	}

	protected abstract Command doRun();

	public void run()
	{
		Command command = doRun();
		if(command != null)
			editPart.getViewer().getEditDomain().getCommandStack().execute(command);
	}

}
```

Code Fragment 24

In Code fragment 25, we see a simple example of a GraphviewAction; the LayoutAction(). This action simply creates a LayoutCommand, sets the graph field, and returns it, letting the superclass put the command on the command stack.

```
public class LayoutAction extends GraphviewerAction
{
	...
	public Command doRun() {

		LayoutCommand command = new LayoutCommand();
		command.setGraph(graphPart.getBean());

		return command;
	}
}
```

Code fragment 25

Code fragment 26 shows the AddParameterAction, an Action that has a little bit more body to it. This action uses a Wizard to retrieve additional information (name, initial value, lower- and upper bound) about the parameter to be added to the graph. If the user finishes the wizard by providing all this information; a AddParameterCommand is created and returned. If the user cancels the action no command is returned or executed.

```
public class AddParameterAction extends GraphviewerAction
{
	...
	protected Command doRun()
	{
		AddParameterWizard wizard = new AddParameterWizard();
		wizard.init(graphPart);

		WizardDialog dialog = new WizardDialog(null, wizard);
		   dialog.create();
		   if(dialog.open() == WizardDialog.OK)
		{

			Parameter newParameter = wizard.getResult();

			AddParameterCommand command = new AddParameterCommand();
			command.setGraph(graphPart.getBean());
			command.setParameter(newParameter);

			return command;
		}
		else
			return null;

	}


}
```

Code fragment 26

The context menu is, as the name suggest, context sensitive. When different graphelements are selected, different actions are available in the contextmenu. The AddParameterAction, for example, is only available if the Graph is selected. The DecomposeNodeAction, on the other hand, is only available for nodes. There are also some actions that are available at all times such as the HideMultipleEdgesAction.

Dialogs

In the previous paragraph we discussed that some GraphViewerActions need additional info from the user to run. These actions use dialogs to retrieve this data from the user. Code fragment 26 shows that AddParameterAction is such an action, using a WizardDialog. A wizarddialog is a dialog that gathers a list of data, by requiring the user to fill out several forms.

In the case of the AddParameterWizard we only use one form, on the account the amount of data is too small for multiple pages. In fact, the use of a normal dialog would have been sufficient, but using a wizard has the advantage of providing the user with a familiar environment: the wizard is a commonly used way of getting data from the user, especially in the Eclipse environment.

Without going into the details of how wizards work, we will stress that wizards may not and do not change the model. They will only retrieve data from the user. This data is then processed by the Action that used the wizard to construct a command. This command will then change the model.

o Visualisation and Interaction with the view

In the previous chapter we have discussed the View as a part of the MVC-pattern. In this chapter we will discuss how the view and the interaction with the view meets the requirements we set in our introduction.

As we have seen in the paragraph we have chosen to let our KPN editor use a general graph model as model layer. This means that in our implementations we will talk about visualizing "edges" and "nodes" in stead of "link" and "entities". So for example, we willt talk about adding a node to graph, and not about adding an Entity to a KPN.

▪ File Management

Requirement 2.1.1: A KPN can be loaded from a .kpn file.
Requirement 2.1.2: A KPN can be saved to a kpn file.

Implementation:
Loading a model is done by first loading a .kpn file into a KPN datastructure. This KPN is then converted into a JDot model. Saving a model is exactly the reverse operation: the JDot model is converted (back) into a KPN datastructure, and the KPN is saved to a file. The conversion is done by visitor classes.

Argumentation: Using visitor classes seperates the saving and loading of the model from the model itself, this is also know as a seperation of concerns. An added benefit is that a visitor object has a state, e.g. the conversion of a node must know about the converted parent graph, which is stored in the visitors state.

Implementation details:
In Code fragment 27 we see the that the editor receives the inputfile. The Graphviewer class uses the loadgraph method to load an IGraph implementation object from the inputfile. The loadgraph method is shown in Code fragment 28. Here we see that loading the model has two steps. First, the KPN is loaded from the .kpn file. If the loading fails (the input file was empty or had an incorrect format) an empty graph is created. If the loading succeeded, the loaded KPN is converted into the IGraph model with the use of the FromKpnConvertor class.

```
private IGraph graph;


public class GraphViewer extends GraphicalEditorWithPalette {

        ...
        protected void setInput(IEditorInput input) {

                super.setInput(input);
                IFile file = ((IFileEditorInput) input).getFile();

                graph = KpnFileManager.loadGraph(file.getRawLocation().toOSString(),
false);
                ...

        }
        ...
}
```

Code fragment 27

```
public class KpnFileManager
{
	public static IGraph loadGraph(String fileName, ...)
	{
		IGraph graph = null;
		KPN kpn = loadKpn(fileName);

		if (kpn == null)
		{
			...
			graph = createNewGraph("graaf");
			...
		}
		else
		{
			FromKpnConvertor convertor = new FromKpnConvertor(...);
			kpn.accept(this);
			graph = convertor.getGraph();
		}
		...

		return graph;
	}
}
```

Code fragment 28

This convertor copies the information from the KPN to the IGraph datstructure. It does this part for part, using the KPNExtendedVisitor visitor class. In Code fragment 29, we see that a new JDotGraphImplementation object is created (this is the only part of the controller that knows which implementation of the igraph package is used). The graph is then put on the stack of created beans, and then all nodes are visited. Because the graph was just put upon the stack, then entity can now retrieve this graph. It then uses the CreateNodeCommand to add a node to the graph. The created node is then put on the stack, followed by the conversion of its parts. The same story holds for all other parts of the KPN.

We have mentioned in the paragraph that the model may only be manipulated by the use of commands. Although the editparts have not yet been created, and so technically the conversion of the model is not a strict part of the controller we still follow this requirement. Almost all modifications performed in the convertor are used somewhere in the controller as well, so doing otherwise would result in code duplication.

```
public class FromKpnConvertor extends KPNExtendedVisitor
{
	...
	private IGraph graph;

	@Override
	public void visitStructure(KPN kpn)
	{
		graph = JDotClusterGraphImplementation.createNewGraph(kpn.getName());
		...

		createdBeanStack.push(graph);
		...
		for (Entity entity : kpn.getEntities())
			entity.accept(this);
		...
		createdBeanStack.pop();
	}
	...
	@Override
	public void visitStructure(Entity entity)
	{
		IGraph graph = (IGraph) createdBeanStack.peek();

		CreateNodeCommand createCommand = new CreateNodeCommand();

		createCommand.setParent(graph);
		createCommand.setId(entity.getName());

		createCommand.execute();

		INode node = createCommand.getCreatedNode();
		...

		createdBeanStack.push(node);
		entity.getAssignStatements().get(0).accept(this);
		createdBeanStack.pop();
	}
	...
}
```

Code fragment 29

- **Displaying graphs, nodes and edges**

Requirement 1.1.1: A KPN must be represented as the main editpane
Requirement 1.1.2: An Entity must be represented by a seperate object in the main editpane
Requirement 1.1.3 A Link must be represented by a seperate object in the main editpane.
Implementation:
When an EditPart for a bean in the model is created, a figure for this bean is created by the FigureFactory. The Figure for a graph is a FreeFormLayeredPane and this pane is added to the main editpane of Graphview.

The figure for a node can have several shapes, but is an ellipse by default. An edge is displayed as a straight line between two nodes, or a curve if a layout has been generated. The figures for the children of an EditPart are automatically added to its parent's figure.

Argumentation:

Because the GraphFigure is the only element in the main editpane, it will cover the entire EditPane. Nodes and Edges have seperate objects as views and are automatically displayed in the graph.

Implementation details:

In Code fragment 30, we see that a GraphFigure is nothing more than a FreeFormLayeredPane (a pane that is easily extendable in all four directions). A small image is painted in the left upper corner, and the graph's name is displayed at the top of the pane.

```
public class GraphFigure extends FreeformLayeredPane{

	...
	protected void paintFigure(Graphics graphics) {
		super.paintFigure(graphics);

		Image rulImage = Activator.getDefault().getRulImage();
		if(rulImage != null);
			graphics.drawImage(rulImage , 0, 0);
		graphics.drawText(_name , new Point(getSize().width/2.0-10.0,0));

	}
	...
}
```

Code fragment 30

Code fragment 31 shows that a NodeFigure does not have its own visual representation (the only function is that it can be selected) It does have a shape childfigure. This construction is used so that the shape of a node can be changed to, for example a rectangle, without having to change the NodeFigure itself. Every time the NodeFigure is repainted, the shape is repainted as well, so functionally a node is represented by a seperated visual object.

```
public class NodeFigure extends Figure{
	...
	protected void paintFigure(Graphics graphics) {

		graphics.setForegroundColor(getForegroundColor());
		graphics.setBackgroundColor(getBackgroundColor());
		shape.paintShape(graphics);

		graphics.setForegroundColor(ColorConstants.black);
	}
}
```

Code fragment 31

The drawing of on edge object is done in one of two ways. When no layout has been performed, or the layout has been lost because of node movement, a straight is drawn representing the edge (the EdgeFigure is a PolyLine with with two points). When a layout is still present, the edge uses the generated splinelist to draw a path or curve between two nodes. The SplineDrawer class, shown in Code Fragment 32, creates a Path object which can be painted on a graphics object.

```
public class EdgeFigure extends PolylineConnection {

	...
	public void outlineShape(Graphics graphics)
	{

		graphics.setLineStyle(getLineStyle());

		if(getPoints().size() > 2)
		{
			new SplineDrawer().drawPath(graphics, getPoints());
		}
		else
			super.outlineShape(graphics);
	}
	...
}
```

Code Fragment 32

- **Creating graph elements**

Requirement 3.1.1: The user should be able to add an entity to a KPN.

Implementation:
The user presses the 'add node' button in the pallette and selects the place and size of the new node in the editing panel. The target graph's editpolicy creates a CreateNodeCommand, which is executed to create a new node.

Argumentation:

Adding the basic blocks of a graph should be intuitive and easy to use. The palette is a kind of toolbar, a user interface element very familiar to most computer users.

Implementation details:

A GraphLayoutPolicy (instance of a XYLayoutPolicy) is installed for each graph. Every time the 'create node' button in the Palette is pressed, The ModelFactory of Graphview creates a dummy Node. When the user chooses the appropriate size and position for the node, by selecting an area in the editing area. The EditPolicy then creates and executes a CreateNodeCommand, thereby adding a node to the model graph, with the chosen size and position. Code Fragment 33 shows the code for creating an CreateNodeCommand.

```
protected Command getCreateCommand(CreateRequest request) {

        ...
        JDotNode templateNode = (JDotNode) templateObject;
        CreateNodeCommand create = new CreateNodeCommand();

        create.setParent(getGraphPart().getBean());
        create.setConstraint((Rectangle) getConstraintFor(request));

        return create;

}
```

Code Fragment 33

Requirement 3.1.2: The user should be able connect two entities with a link.
Requirement 3.1.4 A sourceentity may not be the writeentity of a link.
Requirement 3.1.5 A sinkentity may not be the readentity of a link.

Implementation:
The user presses the 'create edge' button. Then, he clicks somewhere on the node that is to be the tailnode, followed by clicking the headnode. Clicking a SinkNode does not start the edge creation process, clicking a SourceNode does not end it. Edges are placed on the port closest to the clicked position on the node. Feedback is provided if a node is illegal to server as a source or targetnode by changing the curser to an disallowed sign. Clicking anywhere outside a node cancels the process.

Argumentation:
Again, the edge is a basic block of a graph and should be intuitively created. An edge points from the tailnode to the headnode, so the order of execution is logical. A manual selection of the source and target port (preferrably through a dialog) might be more clear, yet it does slow down a basic operation. The source and targetport can later be changed via the properties view.

Implementation details:
Although we treat the creation of an edge as adding an edge to a graph, the concept of connecting two nodes in GEF is slightly different: a connection is added to two nodes. The created edge is a child of these two nodes, and not of the graph. Therefore the policy that governs the creation of an edge is installed on the nodes, and it is called the NodeConnectionPolicy.

Code Fragment 34 shows the getConnectionCreateCommand(), which creates a ConnectNodesCommand, and sets the tailNode and sourcePort fields. This command is then saved, by calling the setStartCommand() method on the policy. The command is stored in the policy until a second node is clicked. The getConnectionCompleteCommand() then sets the remaining fields and returns this completed command, which is then executed to create a new edge.

The ConnectNodesCommand is a combination of three other commands. In Code fragment 35 shows that when this command is executed, first two anchors are created, then a new edge is created and lastly the edges is connected to these anchors. Undoing the commands results in undoing all these five commands.

```
public class NodeConnectionPolicy extends GraphicalNodeEditPolicy
{
	protected Command getConnectionCreateCommand(CreateConnectionRequest request)
	{

		ConnectNodesCommand command = new ConnectNodesCommand();
		NodePart hostNode = getHostNode();

		if(!hostSource.getPropertyViewer().isSink())
		{
			IPort sourcePort = ...

			command.setSourcePort(sourcePort);
			command.setTailNode(hostNode.getBean());
			request.setStartCommand(command);

			return command;
		}
		else
		{
			return null;
		}

	}


	protected Command getConnectionCompleteCommand(CreateConnectionRequest request)
	{

		ConnectNodesCommand command = (ConnectNodesCommand) request.getStartCommand();
		NodePart hostNode = getHostNode();

		if(!hostSource.getPropertyViewer().isSource())
		{
			IPort targetPort = ...

			command.setHeadNode(hostNode.getBean());
			command.setTargetPort(targetPort);

			return command;
		}
		else
		{
			return null;
		}

	}

}
```

Code Fragment 34

```
public class ConnectNodesCommand extends Command
{
	public void execute()
	{

		createInAnchorCommand = new CreateAnchorCommand();
		...
		createInAnchorCommand.execute();

		createOutAnchorCommand = new CreateAnchorCommand();
		...
		createOutAnchorCommand.execute();

		addEdgeCommand = new AddEdgeCommand();
		...
		addEdgeCommand.execute();

		connectInAnchorCommand = new ConnectAnchorCommand();
		...
		connectInAnchorCommand.execute();

		connectOutAnchorCommand = new ConnectAnchorCommand();
		...
		connectOutAnchorCommand.execute();



	}
	...
}
```

Code fragment 35

Requirement 3.1.3: The user should be able to add a variable to a KPN.

Implementation: The user can choose the 'add parameter' action from the context menu and choose a name, starting value, and lower and upper bound for the variable with the help of a wizard.

Argumentation: The requirement is trivially met.

Implementation details:

In Code fragment 36 we see that AddParamterAction retrieves a Parameter object from the AddParameterWizard. The addParameter method that is called by the AddParameterCommand, as shown in Code Fragment 37, adds a parameter to the list with the same name, value, lower and upperbound as the specified parameter, but will not store the passed parameter itself. This prevents the AddParameterAction from changing an Parameter that is a part of the model.

```
public class AddParameterAction extends GraphviewerAction
{
	...
	protected Command doRun()
	{
	   AddParameterWizard wizard = new AddParameterWizard();
	   wizard.init(graphPart);

	   WizardDialog dialog = new WizardDialog(null, wizard);
	   dialog.create();
	   if(dialog.open() == WizardDialog.OK)
	   {

		Parameter newParameter = wizard.getResult();

		AddParameterCommand command = new AddParameterCommand();
		command.setGraph(graphPart.getBean());
		command.setParameter(newParameter);

		return command;
	   }
	   else
		return null;

	}
}
```

Code fragment 36

```
public class AddParameterCommand extends Command
{
	...
	public void execute()
	{
		graph.getParameterList().addParameter(parameter);
	}
	...
}
```

Code Fragment 37

- **Moving graph elements**

Requirement 2.2.1: The user must be able to place any node, anywhere in the editpane.

*Implementation:*The user can move a node by dragging it anywhere in the editpane. The graphs LayoutPolicy then creates a command to change the node's constraints in the model.

Argumentation

Dragging the node will execute a ChangeNodeConstraintCommand, which updates the node's position parameter. The NodePart will be notified of the change and will update the node's figure to be positioned at the appropriate spot, so that the user will effectively have moved the node.

- **Removing graph elements**

Requirement 3.2.1: The user should be able to remove an entity from a KPN.

Implementation:
The user selects a node using the select tool from the palette. Then, the user invokes the delete action (either by pressing the delete key or selecting the delete option from the edit menu, or in the main toolbar). The DeleteNodeCommand that is created every time a nodes is selected is now executed, removing the node from the model. When a node is removed, all edges connected to this node are removed as well.

Argumentation:
The GraphComponentPolicy extends the ComponentEditPolicy, this is another built- in feature of GEF, which is easy to use and implement.

Implementation details:

Every time a user selects a node, the GraphComponentPolicy creates a new DeleteNodeCommand(), by calling it's createDeleteCommand method() (see Code fragment 38). This command removes the node, by calling the removeNode() on the nodes parent graph (see Code fragment 39). In the specification of IGraph, it is stated that the removeNode() must also implement the removal of all the nodes edges. We see in Code fragment 40 that the JDotGraphImplementation meets this requirement, and that the controller is notified of the change in the model.

```
public class GraphComponentPolicy extends ComponentEditPolicy{

        protected Command createDeleteCommand(GroupRequest deleteRequest) {
                ...
                NodePart node = (NodePart) getHost();

                DeleteNodeCommand command = new DeleteNodeCommand();
                command.setNode(node.getBean());
                ...

        }
}
```

Code fragment 38

```
public class DeleteNodeCommand extends Command{

        ...
        public void execute()
        {
                IGraph graph = node.getGraph();

                graph.removeNode(node);
        }
}
```

Code fragment 39

```
public class JDotGraphImplementation extends JDotBeanWithStringPropertiesImplementation implements IGraph
{
	...
	public void removeNode(INode node)
	{
		JDotNodeImplementation augNode = (JDotNodeImplementation) node;

		for(IEdge edge : augNode.listEdges())
			removeEdge(edge);

		graph.getNodeList().remove(augNode.getJDotNode());
		JDotNodeImplementation.removeInstance(node);

		firePropertyChange(NODE_REMOVED);
	}
}
```

Code fragment 40

Requirement 3.2.2: The user should be able to remove a link from a KPN.

Implementation:
Every time the user selects an edge, it's associated EditPart controller uses the creates a DeleteEdgeCommand(). When the user invokes the delete action, this command is executed. This command removes the edge from the graph. This operations also involves removing the associated anchors.

Argumentation:
Again, this EditPolicy is a built-in feature of the GEF-framework which simplifies implementation.

Related work:

Implementation details:
Every time the user selects an edge, the edges EdgeEditPolicy is requested for a DeleteEdgeCommand (see Code fragment 41). When the user invokes the delete action, this action is executed. Executing this command involves calling the deleteEdge() method on the edges graph (see Code fragment 42). In Code fragment 43 we see that in the implementation of IGraph that we use, JDotGraphImplemenation, removes the edge and triggers the removal of the edges anchors.

```
public class EdgeEditPolicy extends ConnectionEditPolicy{

	protected Command getDeleteCommand(GroupRequest request) {
		...
		IEdge edge = (IEdge) getHost().getModel();

		DeleteEdgeCommand command = new DeleteEdgeCommand();
		command.setEdge(edge);

		return command;
		...
	}
}
```

Code fragment 41

```
public class DeleteEdgeCommand extends Command{

	...
	public void execute()
	{
		IGraph parentGraph = edge.getGraph();
		parentGraph.removeEdge(edge);
	}
}
```

Code fragment 42

```
public class JDotGraphImplementation ...
{

	public void removeEdge(IEdge edge)
	{
		JDotEdgeImplementation augEdge = (JDotEdgeImplementation) edge;

		augEdge.getInAnchor().removeMe();
		augEdge.getOutAnchor().removeMe();

		graph.getEdgeList().remove(augEdge.getJDotEdge());
		JDotEdgeImplementation.removeInstance(edge);

		firePropertyChange(EDGE_REMOVED);
	}
}
```

Code fragment 43

- **Displaying properties in labels**

Requirement 1.2.1: The name of the in- and outputvariable of a link should be quickly available to the user
Requirement 1.2.2: The linearization type of a link should be quickly available to the user
Requirement 1.2.3: The name of an entity's assignstatement should be quickly available to the user.
Requirement 1.4.1: The user must be able to access the size of each node and edge quickly

Implementation:

We meet this requirements by providing nodes and edges with a textlabel. In this label, the value of a property can be displayed. For a node this is the assignstatement name. For an edge, the user can choose one of four options: the inputvariable, the outputvariable, the linearization type or the size.This choice can be made in Eclipse's preference page, which can be accessed via the View->Preference menu option.
The value of the label that is displayed is updated each time the value of the property changes.

Argumentation:
The amount of a nodes in graph is usually limited, and the space between them large enough to fit multiple labels. Yet it is often the case that there is a multitude of edges. If all these edge and nodes had multiple labels, the view would be severely cluttered.

Note that although this preference is not strictly speaking a part of the model, changing it does have an influence on the model, namely the label's text. It is therefore the question if the ILabel interface should be in the igraph interface or that it should be part of the view. The position of the label, on the other hand, is a property of the model, since it is independant of the view used. As for now we will allow this small breach of the seperation between the view and the model.

Implementation details:

In Code fragment 44 we see that all labels are returned as modelchildren of the graph. Although this is conceptually incorrect (the labels are children of a node or edge) this is necessary: when a object is a model child in GEF, it's image will also displayed within the image of its parent. In the code of a node, the textlabel would have to be displayed inside the node, which we do not want.

```
public class GraphPart ...{

	...
	protected List getModelChildren() {

		List children = new ArrayList();

		children.addAll(getBean().listNodes());
		children.addAll(getBean().listEdges());
		for(INode node : getBean().listNodes())
			children.add(node.getLabel());

		return children;
	}
	..
}
```

Code fragment 44

A LabelPart is automatically created for this label, and every time its text or position changes, it's visuals are refreshed (see Code fragment 45). The label's position changes due to user input, or because of the layout mechanism, as we will see in paragraph . A label's text is changed when either the value of the corresponding property changes, or when the property displayed changes.

```
public class LabelPart extends AbstractGraphicalEditPart implements PropertyChangeListener
{
	...
	public void propertyChange(PropertyChangeEvent evt)
	{
		String propertyName = evt.getPropertyName();

		if(propertyName.equals(ITextLabel.POSITION))
			refreshVisuals();
		else if(propertyName.equals(ITextLabel.TEXT))
			refresh();
		else
			Activator.logErrorMessage("Illegal property name");

	}
	...
}
```

Code fragment 45

In Code fragment 46, we see that everytime the propertyChange of an EdgePart (something similar happens for a NodePart) method is executed (some property of the edge has changed) this method checks if the property changed matches the property displayed. If so, the labels text is updated to reflect the new value, which, because of the notification mechanism, updates the labels text in the view.

```
public class EdgePart ...
{
	...
	public void propertyChange(PropertyChangeEvent evt) {

		...
		if(propertyName.equals(GraphviewPreferencePage.getShownOnEdge()))
		{
			ChangeLabelTextCommand command = new ChangeLabelTextCommand();
			AbstractProperty property = getPropertyTable().getProperty(propertyName);
			String propertyValue = property.getValue();

			command.setLabel(getBean().getLabel());
			command.setText(propertyValue);

			command.execute();
		}
		...
	}
	...
}
```

Code fragment 46

Requirement 2.3.1: The user should be able to add annotations to the visual representation of a KPN, and edit these texts.

*Implementation:*A graph can have multiple textlabels, called annotations. These labels are manually placed by a user. The text of an annotation can be changed with a dialog.

Argumentation: A user may want to add all kinds of additional information to a graph, such as a short explanation of its contents or an author statement. Annotations are a good way to do this.

*Implementation details:*The user can add an annotation by using the 'add annotation' tool from the pallete. When the button is clicked a dummy textlabel is created. When the user places this dummy label somewhere on the graph, a CreateAnnotationCommand is executed. Code fragment 47 shows this command being created by the graph's GraphLayoutPolicy.

```
public class GraphLayoutPolicy extends XYLayoutEditPolicy{

	...
	protected Command getCreateCommand(CreateRequest request) {

		Object templateObject = request.getNewObject();

		...if(templateObject instanceof ITextLabel)
		{
			CreateAnnotationCommand create = new CreateAnnotationCommand();

			create.setGraph(getGraphPart().getBean());
			Rectangle constraint = (Rectangle) getConstraintFor(request);
			create.setPosition(constraint.getCenter());

			return create;
		}
		...
	}
	...
}
```

Code fragment 47

The new label wil be supplied with a default text. To change this text, the user can select the 'change annotation' option from the label's context menu. A dialog will appear, where a new text can be entered. When the user selects the 'change annotation' method a ChangeAnnotationTextAction is executed. Code Fragment 48shows that this action first retrieves the new text from the user using a ChangeLabelTextWizard. Then a ChangeLabelTextCommand, which actually changes the text, is created and executed.

```
public class ChangeAnnotationTextAction extends GraphviewerAction {

        ...
        public Command doRun()
        {
           ChangeLabelTextWizard wizard = new ChangeLabelTextWizard();
           String oldValue = labelPart.getBean().getText();

           wizard.init(oldValue);

           WizardDialog dialog = new WizardDialog(null, wizard);
           dialog.create();

           if(dialog.open() == WizardDialog.OK)
           {
                 ChangeLabelTextCommand command = new ChangeLabelTextCommand();

                 command.setLabel(labelPart.getBean());
                 command.setText(wizard.getNewValue());

                 return command;
           }
           else
                 return null;

        }

}
```

Code Fragment 48

- **Label positions**

Now that we have chosen to display some properties in labels, it has to be clear to which graphobject a label belongs. This introduces a new requirement:

Requirement 3:
The user should be able to identify textlabels with the appropriate graph element intuitively.

Implementation:
Nodes and edges have exactly one label. The label is placed at a fixed position relative to the node or edge. For a node, this is directly under it. For an edge it's in the point. The label may be moved manually.

Whenever a node or edge moves, the label moves with it. However, the context menu option places all textlabels at their original fixed positions. Performing a LayoutAction on a graph will also have this result.

Argumentation:

Having the labels at a fixed places close to the node or edge quickly teaches the user that the label belongs to this object. Because textlabels are not a part of the base JDot datastructure, the layout algorithm knows nothing about these labels and does not take these labels into account when determining the layout. This means that the layout algorithm may place a label on top of another graph element. Since we have not algorithm to avoid the overlay of labels, we leave the possibility to place the textlabel at a different spot.

Implementation details:

Code fragment 49 shows that after laying out of a graph, the labels of all nodes and edges are put at their standard positions, using ResetLabelPositionCommands. Code fragment 50 shows that the ResetLabelPositions method positions a node's label directly under the lower bound of the node. The label of an edge is positioned at it's center. Note that the position of a label denotes the center of this label.

```
public class LayoutCommand extends Command
{
	...
	public void execute()
	{
		graph.doLayout();

		resetLabelPositions();
	}

	private void resetLabelPositions()
	{
		for(INode node : graph.listNodes())
		{
			ResetLabelPositionCommand command = new ResetLabelPositionCommand();
			command.setParentBean(node);

			command.execute();
		}
		for(IEdge edge : graph.listEdges())
		{
			ResetLabelPositionCommand command = new ResetLabelPositionCommand();
			command.setParentBean(edge);

			command.execute();
		}
	}
}
```

Code fragment 49

```
public class ResetLabelPositionCommand extends Command
{

	private ChangeLabelPositionCommand delegateCommand;
	...
	public void execute()
	{
		delegateCommand = new ChangeLabelPositionCommand();
		delegateCommand.setTextLabel(getTextLabel());
		delegateCommand.setNewPosition(getDefaultPosition());

		delegateCommand.execute();

	}

	private Point getDefaultPosition()
	{
		if(parentBean instanceof INode)
		{
			INode node = (INode) parentBean;
			Point position = node.getLocation();

			ITextLabel label = node.getLabel();
			Point labelPosition = position.getTranslated(0, node.getSize().height/2 + 10);

			return labelPosition;
		}
		else if(parentBean instanceof IEdge)
		{
			IEdge edge = (IEdge) parentBean;
			Point position = determineEdgeMiddlePoint();

			return position;
		}
	}
	...
}
```

Code fragment 50

When the user drags a label, the GraphLayoutPolicy (see Code fragment 51) creates and executes a ChangeLabelPositionConstraint. Note that this can be a label for a node or an edge, but also an annotation, since they use the same ITextLabel interface.

```
public class GraphLayoutPolicy extends XYLayoutEditPolicy{

	protected Command createChangeConstraintCommand(EditPart child, Object constraint) {
		...
		 if(child instanceof LabelPart)
		{
			LabelPart labelPart = (LabelPart) child;
			ChangeLabelPositionCommand command = new ChangeLabelPositionCommand();
			command.setTextLabel(labelPart.getBean());
			command.setNewPosition(((Rectangle) constraint).getCenter());

			return command;
		}
		else
			return UnexecutableCommand.INSTANCE;

	}
	...
}
```

Code fragment 51

When a node or edge moves, and its label is still at its default position, the label must move with the node or edge, so that it is still in the default position after the move. In Code fragment 52, we see that the ChangeNodeConstraint first checks if the node's label was still at its original position. If so, it resets the position of this label after repositioning the node. The same is done for the edges. A similar method is used when an edge is moved.

```
public class ChangeNodeConstraintCommand extends Command{
	...
	public void execute() {

		...
		createResetLabelCommands();
		.. //change node constraint
		executeResetLabelCommands();

	}
	private void executeResetLabelCommands()
	{
		for(ResetLabelPositionCommand command : resetLabelPositionCommands)
			command.execute();
	}
	private void createResetLabelCommands()
	{
		ResetLabelPositionCommand command = new ResetLabelPositionCommand();
		command.setParentBean(node);
		if(command.hasDefaultPosition())
			resetLabelPositionCommands.add(command);

		for(IEdge edge : node.listEdges())
		{
			command = new ResetLabelPositionCommand();
			command.setParentBean(edge);
			if(command.hasDefaultPosition())
				resetLabelPositionCommands.add(command);
		}

	}
	...
}
```

Code fragment 52

- **Graph layout**

Requirement 1.6.1: The editor must provide an automatical layout for the visual representation of the KPN.
Requirement 1.6.2: Nodes may not overlap and edges should cross nodes and other edges as little as possible.
Requirement 1.6.3: The layout of the graph must imply the direction of dataflow in the process network.

Implementation:

Normally, node and edge placement is unmanaged. Executing the 'Perform Layout' action from the graph's context menu trigger the Layout Command, which calls JDot's layout method. As we have seen in paragraph this layout method meets the above requirements.The JDotGraphImplementation doLayout method notifies the graph and it's nodes and edges that their layouts have changed.

If the amount of edges clutters the graph, the user can also execute the 'HideMultipleEdgesCommand'. This command checks if more than one edge is present between two nodes, and then updates the view to show only one edges per nodepair. This edge can be seen as the representing edge.

Argumentation:

This requirement is vague and intentionally defined subjectively. Any implementation suffices as long as the layout of the graph clarifies the structure of the graph. In figure 6 we saw that the JDot layout functionality matches this requirement quite nice. An added benefit of JDot is that dot also imposes a direction on the graph, thereby illustrating the data flow.

Although the JDot package produces nicely structured layouts, the number of different edges can be so large that it obstructs the understanding of the dataflow in the KPN (see Figure 10) Hiding multiple edges effectively solves this problem as we can see in Figure 11.

The current implementation of the HideMultipleEdgesCommand chooses one of the multiple edges to display. Unfortunately this has the side effect that this particular edges label and color are shown as if the were the 'average' label and color. As an followup project we recommend to change the code so that the editor refrains from edge coloring and hides the labels when in hide multiple edges mode.

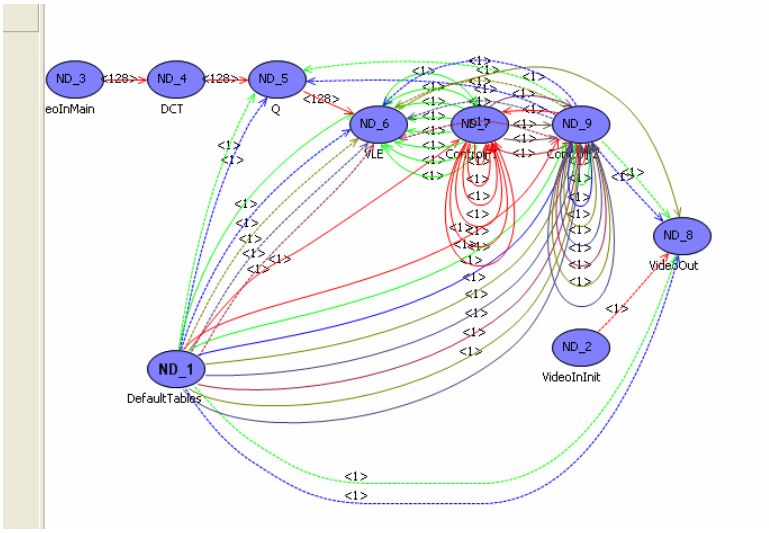


Figure 10

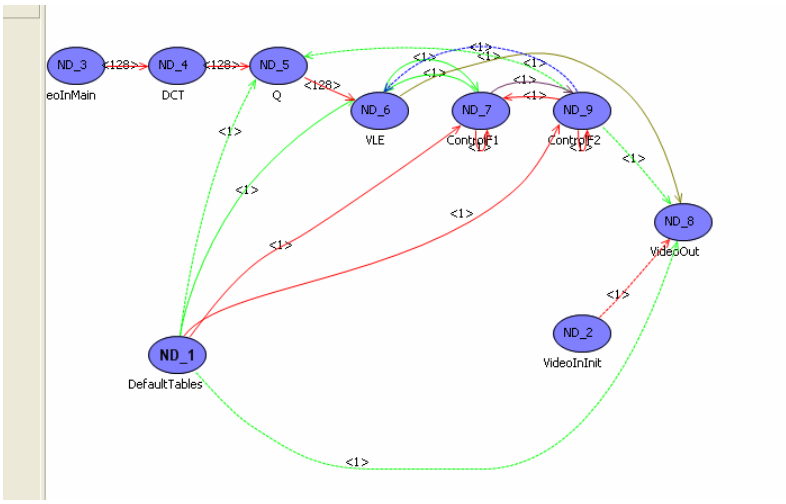


Figure 11

Implementation details (doLayout):

When the user executes the 'Perform Layout' action the PerformLayoutCommand is executed (see Code fragment 53). This performs the doLayout method on the appropriate IGraph bean. Code fragment 54 shows that the JDotGraphImplementation calls the doLayout method on the JDot graph and that it notifies the graph and its nodes and edges that the layout has been performed.

```
public class GraphviewerContextMenuProvider extends ContextMenuProvider {
	...
	private void addEditActions(IMenuManager menu) {
		...
		IAction layoutAction = new LayoutAction(graphPart);
		menu.appendToGroup(GEFActionConstants.GROUP_EDIT, layoutAction);
		...
	}
	...
}

public class LayoutAction extends GraphviewerAction
{
	...
	public Command doRun() {

		LayoutCommand command = new LayoutCommand();
		command.setGraph(graphPart.getBean());

		return command;
	}
}
```

Code fragment 53

```
public class LayoutCommand extends Command
{
	...
	public void execute()
	{
		graph.doLayout();
	}
}

public void doLayout()
{
	graph.doLayout();

	for(IEdge edge : listEdges())
		edge.firePropertyChange(IEdge.POINT_LIST_CREATED);
	for(INode node : listNodes())
	{
		node.firePropertyChange(INode.SIZE);
		node.firePropertyChange(INode.LOCATION);
	}
	firePropertyChange(IGraph.LAYOUT);
}
```

Code fragment 54

The NodePart reacts to this change by refreshing its visuals. This will paint the node at the new place. The same holds for the EdgeParts. Because the edge now has a spline list, the EdgeFigureUpdater uses this list of points to update the figure. This means that the edges will now be painted as a curve. Details about how this is done can be found in paragraph.

Implementation details (hideMultipleEdges):
Code fragment 55 shows that the hideMultipleEdgesAction is available from the context menu. If the multiple edges are already hidden, the unhideMultipleEdgesAction is available. The hideMultipleEdges property is a preference: it is applicable to all instances of the Graphviewer editor. Code fragment 56 shows that the (Un)HideMultipleEdges command first sets the preference value, followed by running a command that (un)hides the edges for each graph that is openend in a editor.
Since the code for hiding multiple edges in a graph is fairly long, we omit it here. It suffices to say, that the style attribute of a lot of edges is set to invisible. This in turn notifies the EdgePart to update the edge figure, making it either visible or invisible, according to it's style property. The label is also updated, so that it is visibility matches that of the edge.

```
public class GraphviewerContextMenuProvider extends ContextMenuProvider
{
	...
	private void addViewActions(IMenuManager menu) {


		if(GraphviewPreferencePage.getHideDuplicateEdges())
		{
			IAction unhideDuplicateEdgesAction = new UnhideEdgesAction(graphPart);
			menu.appendToGroup(GEFActionConstants.GROUP_VIEW,
							unhideDuplicateEdgesAction);
		}
		else
		{
			IAction hideDuplicateEdgesAction = new HideEdgesAction(graphPart);
				menu.appendToGroup(GEFActionConstants.GROUP_VIEW,
							hideDuplicateEdgesAction);
		}
	}
	...
}
```

Code fragment 55

```
public class HideMultipleEdgesCommand extends Command
{
	...
	public void execute()
	{
		GraphviewPreferencePage.setHideDuplicateEdges(true);
		for (GraphViewer graphViewer : GraphViewer.getInstances())
		{
			IGraph graph = graphViewer.getGraph();
			HideMultipleEdgesForGraphCommand command =
				new HideMultipleEdgesForGraphCommand();
			command.setGraph(graph);
			command.execute();
			...
		}
	}
	...
}
```

Code fragment 56

- **Displaying information through color**

Requirement 1.2.1: The name of the in- and outputvariable of a link should be quickly available to the user
Requirement 1.2.2: The linearization type of a link should be quickly available to the user
Requirement 1.4.2: The must be able to view the relative size of each node and edge quickly

Implementation:
The user can select a 'coloring property' on the preferences page. The user can choose one of the list: 'size', 'inargument', ' outargument' and 'linearization type' The edges are then colored according to the chosen property.

Argumentation:
In the case of size, we are dealing with a property where it is important where the data bottlenecks are, that is, which edges transport a lot of data. The large values are colored with a bright red, and the small values with a blue color. The “hot” red color draws attentation, whereas the “cold” does not.

In the case of the arguments and linearization properties, we are dealing with a way to display similarities of edges. Edges with the same values are given the same the same color, thereby visualing grouping them. Also, the whole spectrum is used to color these edges, thereby seperating the groups from each other as much as possible.

Related work:

Implementation details:

Coloring of edges happens with the use of ColoringStrategies. Each ColoringStrategy must implement the 'performColoring' method, implying that it calculates a color value for each element the strategy works on, and then apply this colorvalue to these element.We use three kinds of strategies for edges: the RangeColoringStrategy, StringColoringStrategy and LinearizationColoringStrategy.

Size: When the selected property is size, the a RangeColoringStrategy is used. In Code fragment 57 we see the definition of a RangeColoringStrategy. The subclass used is the EdgeMinMaxColoringStrategy, which must implement the three abstract methods. In Code Fragment 58 we see that the MinMaxColoringStrategy implements two of these. The specific code is omitted; but calculateExtrema calculates the minimal and maximum value for the property the strategy works on (in this case the 'size' of the edge) of the list of elements. The calculateColor method then determines the appropriate color by determining the relative value according to these extrema. That is: closer to the miminum means this color is more blue, closer to the maximum makes it more green. Code fragment 59 shows that the added functionality by EdgeMinMaxColoringStrategy's is the setting of the appropriate color property: it sets the edges 'color' property.

```
public abstract class RangeColoringStrategy<PropertySourceClass extends PropertySource>
		extends ColoringStrategy<PropertySourceClass> {


	public RangeColoringStrategy(String propertyName, List allSources)
	{
		super(propertyName, allSources);
	}

	public void performStrategy()
	{
		calculateExtrema();
		for (PropertySourceClass sourceObject : this.allSources)
		{
			Color color = calculateColor(sourceObject, propertyName);
			applyColor(sourceObject, color);
		}
	}

	protected abstract void calculateExtrema();
	protected abstract Color calculateColor(PropertySourceClass propertySource, String propertyName);
	protected abstract void applyColor(PropertySourceClass properytSource, Color color);
}
```

Code fragment 57

```
public abstract class MinMaxColoringStrategy extends RangeColoringStrategy{
	...
	@Override
	protected Color calculateColor(PropertySource propertySource, String propertyName) {
		...
	}

	@Override
	protected void calculateExtrema() {
		...

	}
}
```

Code Fragment 58

```
public class EdgeMinMaxColoringStrategy extends MinMaxColoringStrategy{
	....
	protected void applyColor(PropertySource properytSource, Color color) {
		((EdgePropertySource) properytSource).strategyColor().setValue(color);
	}
}
```

Code fragment 59

In- or outvariable: When the selected property is the in- or outvariable, the used strategy is the EdgeStringColoringStrategy. This is a direct subclass of the ColoringStrategy class. This strategy is pretty straightforward. Code fragment 60 shows that in the the performStrategy() method, the differentvalues for the appropriate property (in this case the in- or outargument) are determined by calling determineDifferentValues(). Then, in the performColoring method, the strategy mixes a color for each distinct valuea and the performColoring method is called, which colors all edges that have the same value for this property in a common color.

```
public class EdgeStringColoringStrategy extends ColoringStrategy<EdgePropertySource>
{
	...
	public void performStrategy()
	{
		determineDifferentValues();
		performColoring();

	}

	private void determineDifferentValues()
	{
		...
	}

	private void performColoring()
	{
		List<Color> mixedColors = mixColors(differentValues.size());
		Map<String, Color> valueToColorMap = new Hashtable<String, Color>();
		for(int index = 0; index < differentValues.size(); index++)
		{
			String value = differentValues.get(index);
			Color color = mixedColors.get(index);
			valueToColorMap.put(value, color);
		}

		for(EdgePropertySource edgeSource : allSources)
		{
			String value = valueForElement(edgeSource);
			Color color = valueToColorMap.get(value);
			applyColor(edgeSource, color);
		}
	}
	...
}
```

Code fragment 60

Linearization type: when the strategy property is the Linearization type, the value of the property is limited to one of four values: IOM-, IOM+, OOM-, OOM+. We assign a hardcoded color to each of these values. Although this strategy can be generalized (to, say, a MapValueToColorStrategy) we have not

done this: the class is too small to require this step. Code fragment 61 shows that this strategy just 'calculates', for each edge, the correct color using an if-else list. This color is then applied to the edge.

```
public class EdgeLinearizationColoringStrategy extends ColoringStrategy<PropertySource>{

	...
	public void performStrategy()
	{
		for (PropertySource sourceObject : allSources)
		{
			Color color = calculateColor(sourceObject, propertyName);
			applyColor(sourceObject, color);
		}
	}
	protected void applyColor(PropertySource properytSource, Color color) {
		((EdgePropertySource) properytSource).strategyColor().setValue(color);
	}

	protected Color calculateColor(PropertySource propertySource, String propertyName) {
		EdgePropertySource edgeSource = (EdgePropertySource) propertySource;
		Linearization linearization = edgeSource.linearization().getValue();
		if(linearization.equals(Linearization.IOM_MINUS))
			return ColorConstants.blue;
		else if(linearization.equals(Linearization.IOM_PLUS))
			return ColorConstants.green;
		else if(linearization.equals(Linearization.OOM_PLUS))
			return ColorConstants.cyan;
		else if(linearization.equals(Linearization.OOM_MINUS))
			return ColorConstants.red;
		else
			throw new RuntimeException("Illegal linearization type: " + linearization);
	}
}
```

Code fragment 61

In the previous alinea's we have seen how the edges and nodes can be colored based on certain properties. The following requirements leads to another kind of coloring: coloring based on node and edge 'size'.

Requirement: The user must be able to view the size of the node. This size indicates the amount of work the associated entity represents (calculated based on some the graphs parameters).

Implementation:
The NodeColoring property can be set on the preferences page. The user has the choice of size and none. When size is chosen, a RangeColoringStrategy is used to calculate the fillcolors of the nodes. When none is selected, the node is filled with a standard color (which can also be determined on the preferences page).

Argumentation:
Information about the 'size' of the nodes is needed to determine which part of algorithm the KPN represents should be changed, to improve performance. Again we use the red color to denote nodes with a big 'size' and a cool blue for the nodes with a small size, thereby attracting the users attention to the nodes with the big 'size' property.

Implementation details:

As with coloring the edges with the EdgeMinMaxColoringStrategy, coloring the nodes according to their sizes is done by using a subclass of the MinMaxColoringStrategy: the NodeMinMaxColoringStrategy. As said before, this strategy calculates the minimum and maximum values for the size and assigns a relative value to each node. The node is then colored according to this relative value.

- **Editing non- visual information**

In this paragraph we we will list the requirements that are met by giving the user the option to change property values in the Properties View. Information about how the Properties view is used to change properties can be read in paragraph 0.

Properties view of graph:
Requirement 1.3.2 The value of a KPN's parameters must be available to the user.

Properties view for node:
Requirement 2.4.1*: The user should be able to change an entity's name.*
Requirement 2.4.4*: The user should be able to change a variables name.*

Properties view for edge:
Requirement 1.3.1: The names of the ports of an edge must be available to the user somewhere in the editor.
Requirement 2.4.2*: The user should be able to change a links name.*
Requirement 2.4.3*: The user should be able to change a ports name*

- **Notes:**

Requirement 2.3.2: The user should be able to add notes to an entity or link. These notes must be accessible somewhere in the user interface, but should not be displayed inside the editpane.

Implementation:

The user can add a property to a node or edge, by executing the 'Add property' action from the context menu. This property can then be edited in the properties view.

Argumentation:
Sometimes the user might want to add a note that can not be seen in the editor. These notes can be edited in the properties view, so that they are grouped with the other textual properties.

Implementation details:

The controller adds a property by running an AddPropertyAction. In Code fragment 62 we see that this action uses a PropertyNameWizard to get the name and defaultValue from the user. These values are retrieved as a propertyTemplate. When multiple EditParts are selected, a property is added to each of the EditParts. Therefore we need to return multiple commands that add a property to each of these editparts' bean. This is done by using a CommandListCommand. This is a command that executes a list of commands.
Code fragment 63 shows that a AddCustomPropertyCommand creates a custom property and adds it to the propertyTable of the corresponding bean. Then it sets the value of the property to the defaultValue.

```
...
public class AddPropertyAction extends GraphviewerAction {

	...
	public Command doRun() {
	   PropertyNameWizard wizard = new PropertyNameWizard();

	   WizardDialog dialog = new WizardDialog(null, wizard);
	   dialog.create();
	   if(dialog.open() == WizardDialog.OK)
	   {

		CustomPropertyTemplate template = wizard.getResult();

		CommandListCommand listcommand = new CommandListCommand();
		for(IElementPart elementPart : selectedEditParts)
		{
			AddCustomPropertyCommand command = new AddCustomPropertyCommand();
			listcommand.addCommand(command);

			command.setName(template.getName());
			command.setValue(template.getDefaultValue());
		}
		return listcommand;
	   }
	   else
		return null;
	}
	...
}
```

Code fragment 62

```
...
public class AddCustomPropertyCommand extends PropertyCommand
{

	...
	public void execute()
	{
		JDotCustomProperty customProperty = new JDotCustomProperty(bean, name, defaultValue);
		table.addProperty(customProperty, PropertyTable.CUSTOM_CATEGORY);
		customProperty.setValue(value);
	}
	...

}
```

Code fragment 63

- **KPN Specific**

Node domain

Requirement 1.3.3: The domain of an entity should be available to the user.

Implementation: The context menu of a node contains the option 'Edit node domain'. Clicking this, the user will be presented with a dialog with three matrix editors; for the constraints, context, and mapping matrix. If there are multiple constraint matrices, they each have their own tab in the constraints editor

area.The user can edit these matrices. Pressing ok saves the changes, cancel returns the domain to its original state.
In addition to the editing functionality the user can copy each matrix to the clipboard, making the matrix available to other programs as a csv(comma-seperated-version) string.

Argumentation:
On of the plugins of Eclipse is het jface package. This package contains amongst a lot of other things, the TableViewer class. A matrix is like a table, so we use a tableviewer to view and edit the matrices of the node domain.

Implementation details:

Code fragment 64 shows the EditNodeDomainAction. This action first gets the current domain, and creates a copy of it. It then creates an EditDomainDialog, provides it with the copied domain, and presents it to the user. The user edits this domain at will. When the user presses 'Ok' to save the changes, the correct domain is copied into the nodes domain.

```
public class EditNodeDomainAction extends GraphviewerAction
{
	protected Command doRun()
	{
		IDomain domain = nodePart.getBean().getDomain();

		EditDomainDialog dialog = new EditDomainDialog(null);
		dialog.setSource(domain);
		dialog.create();
		dialog.setTitle("Edit entity domain");
		dialog.open();

		CopyDomainCommand command = new CopyDomainCommand();

		command.setSourceDomain(dialog.getResult());
		command.setTargetDomain(nodePart.getBean().getDomain());

		return command;
	}
}
```

Code fragment 64

Port domain
Requirement 1.3.4: The domain of a port should be available to the user.

Each IoPort on a node has a port domain. If there are ports present on a node, their domains can be viewed by selecting the 'Edit writeports ' or 'Edit readports' actions in the contextmenu. Selecting such an action will show the same matrix dialog as for the 'Edit node domain' actions, with an added combo box from which the user can select which read/writeport he wants to edit.

View source files
Requirement:
The user should be able to view the c or java code file that describes the Kahn Process a node represents.

Implementation:

In the context menu, the user can select the open *filename*.c or *filename*.java action (if these files are present). This opens the appropriate file in the default Eclipse editor (that is the default for this filetype)

Argumentation:
Each time a SAC is transformed into a KPN, it also generates a codefile for each describing the Kahn Process this node represents. This action provides Graphview with a quick way to access this code.

Implementation details:
The OpenCodeFileAction uses the Graphviewer to open a file in it's appropriate editor. In Code fragment 65 we see that GraphViewer accomplishes this by retrieving the current workbench and opening the file in this workbench.

```
public class GraphViewer extends GraphicalEditorWithPalette {
        ...
        private static IWorkbench getWorkBench() {
                IWorkbench workbench = PlatformUI.getWorkbench();
                return workbench;
        }

        public static void openFile(IFile file) {

                FileEditorInput editorInput = new FileEditorInput(file);


                getWorkbenchPage().openEditor(
                                editorInput,
                                getWorkBench().getEditorRegistry().getDefaultEditor())
                ...
        }
        ...
}
```

Code fragment 65

- **Outputting the graph to other media**

Requirement 2.5.1: The user should be able to print out the contents of the main editpane.

Implementation: The Eclipse print option automatically outputs the contents of the current pane to the printer. This is default Eclipse functionality therefore we will not show any implementation details.

Requirement 2.5.2: The user should be able to save the contents of the main editpane in several image formats.

Implementation:
The user can select the 'Save As' option from the file menu. The user can then select the name and the image format of the file to be saved. The image format can be bitmap(.bmp) or JPEG. The image file is created by painting the current graph to a image canvas.

Argumentation
The reuqirement is trivially met.*:*

Implementation details:

In Code fragment 66 we see the SaveAsAction. The save action first gets a filename and filetype from the user, using a FileDialog. Then, it creates a new Image and creates a Graphics object that paints to this Image. After painting the current graph to this Graphics object the image is converted to bytes, and saved to the appropriate file.

```
public class SaveAsAction extends Action
{
	...

	public void run()
	{
		FileDialog dialog = new FileDialog(Display.getCurrent().getActiveShell());
		dialog.setFilterExtensions(new String[]{"*.bmp", "*.jpeg"});

		String fileName;
		if((fileName = dialog.open()) != null)
		{
			IFigure figure = determineFigure();

			Image image = createImage(Display.getCurrent(), figure.getBounds());
			GC gc = createGraphicalContext(image);
			Graphics graphics = createGraphics(gc);

			paintGraph(graphics);

			saveImage(image, fileName);

			image.dispose();
			gc.dispose();
			graphics.dispose();
		}
	}
	..

}
```

Code fragment 66

- **Clusternodes and clustergraphs**

Requirement 3.3.1: The user should be able to add a clusterentity.
Requirement 3.3.2: The user should be able to view the KPN corresponding to a clusterentity.
Requirement 3.3.3: Adding a port to a clusternode adds an entity to the KPN.
Requirement 3.3.4: Adding a source or sinkentity to a clusterKPN adds a port to the clusternode.
Requirement 3.3.5: The user should be able to decompose a clusternode.

Implementation:
When specified in the KPN file (by the isClusternode property being true) a node is a clusternode. In the editor, the variables of a clusternode are represented by a triangles attached to the node (pointing inward for invariable, outward for outvariable). Every clusternode has a clustergraph associated with it. The JDotClusterGraphImplementation class is a subinterface of the JDotGraphImplementation class and therefore inherits all it's functionality. It only adds the method to retrieve its representing node.

The contextmenu of a clusternode has the option 'Descend into clustergraph'. Performing this action changes the view to display the clustergraph. The parent graph can be accessed by selecting the 'Ascend into parentgraph' option. If a kpn is saved, all of its childrens kpn's are saved as well.
Every time a port is added to a clusternode, a source or sinknode with the same name is added to the clusternode's clustergraph. A node represents a part of a program, and the in and output of this program are represented by the in and outputports. A KPN is also a program, and its in and outputs are represented by the source and sinknodes. Each in and input of a clusternode must therefore have a corresponding in and outputnode (sink or sourcenode) in its clustergraph.
The jdotdomain package enforces this requirement by making sure that every time a port is added to a clusternode, a source or sinknode with the same name is added to it's clustergraph. In the same, every time a source or sinknode is added to a clustergraph (or a node is set to be a source or sinknode) a input or outputvariable is added to the clusternode.

Aside from this, a user can decompose a node, by selecting this action from the nodes contextmenu. Graphview asks for the file location of a KPN and then analyzes the graph for fitting in the node (that is, the number of source and sinknodes match the number of in and outputvariables of the node). If the graph fits, the nodes is converted into a clusternode, and takes the selected KPN as its clustergraph.
An additional gimmick is in the coloring of the ports in the subgraph. Whenever the edgecoloring property is set to in- our outputvariable, the in- or outputvariables on the nodes are colored the same color as it's attached edges.

Argumentation:

JDot provides functionality for adding JDotClusterGraph to a JDotGraph. Unfortunately, all nodes in a clustergraph are still drawn in the parentgraph. That is, there is a box around them to identify the contained nodes to comprise a clustergraph.
In our editor, we want the clusternodes to remain nodes. The clustergraph is further refines the process the clusternode represents. The information in this clustergraph is therefore on a more fine- grain level than the information in the parentgraph. It would therefore inappropriate to show it a the same level in the editor.
Therefore we create an entire new graph for each clusternode, and let it be represented by a . The clustergraph can be viewed on its own level, if so requested by the user.

Implementation details:

Code Fragment 67 shows that the FigureUpdater for a clusternode extends that of a normal node. A clusternode is also visualized with a NodeFigure. The only difference is that margins are added to a ClusterNodeFigure by the ClusterNodeFigureUpdater. These margins are used to draw small dots that represent the ports of the clusternode. Because we draw the ports on ClusterNodeFigure, they automatically move with the node.

```
public class ClusterNodeFigureUpdater extends NodeFigureUpdater{

	public ClusterNodeFigureUpdater(IClusterNode clusterNode)
	{
		super(clusterNode);
	}

	@Override
	protected void updateFigureData(NodeFigure figure) {
		super.updateFigureData(figure);
		figure.addPortMargins();
	}
}
```

Code Fragment 67

The OpenClusterGraphAction is shown in Code Fragment 68. When it is activated by the user, it determines the represented graph of the selected clusternode. It then deactivates the GraphPart for the currently displayed graph. The ClusterGraphPart is then set as contents for the viewer, which has the effect that the ClusterGraph is now shown in the editor. The user can ascend into the original graph by selecting the 'Ascend into Parent" . This sets the parent of the current graph as the contents for the GraphViewer.

```
public class OpenClusterGraphAction extends Action{
	...
	public OpenClusterGraphAction(ClusterNodePart clusterPart)
	{
		super("Descend into cluster");
		this.clusterPart = clusterPart;
	}

	@Override
	public void run() {

		IClusterGraph clusterGraph = clusterPart.getClusterPropertyViewer().getClusterGraph();
		getCurrentGraphPart().deactivate();
		getViewer().setContents(clusterGraph);
	}
	...
}
```

Code Fragment 68

Code fragment 69 shows that every time a port is added to a clusternode (which has a IClusterPortList instead of a normal IPortList), a node is added to the clusternode's clustergraph with the same name. In the same way a representing port is created when the nodetype of a node is changed, and the node is in a clustergraph, a port is created in the representing node of the clustergraph, which we see in Code fragment 70.

```
public class JDotClusterPortListImplementation extends JDotPortListImplementation implements IClusterPortList
{

	...
	protected JDotPortImplementation createPort(String name, IPortLocation location, boolean isSourcePort)
	{
		INode representedNode = findOrCreateRepresentedNode(name, isSourcePort);
		JDotClusterPortImplementation clusterPort = new JDotClusterPortImplementation(name, location,
		clusterPort.setRepresentedNode(representedNode);

		return clusterPort;

	}

	private INode findOrCreateRepresentedNode(String name, boolean isSourcePort)
	{
		JDotClusterGraphImplementation clusterGraph = (JDotClusterGraphImplementation)
		JDotNodeImplementation node = clusterGraph.findNode(name);

		if(node == null)
		{
			node = clusterGraph.addClusterNode(name);
			if(isSourcePort)
				node.setNodeType(NodeType.sink);
			else
				node.setNodeType(NodeType.source);

		}

		return node;
	}
	...

}
```

Code fragment 69

```
public class JDotNodeImplementation extends JDotBeanWithStringPropertiesImplementation implements INode
{
	...
	public void setNodeType(NodeType nodeType)
	{
		this.nodeType = nodeType;
		firePropertyChange(INode.NODE_TYPE);

		if(getGraph() instanceof IClusterGraph)
		{
			IClusterGraph clusterGraph = (IClusterGraph) getGraph();
			IClusterNode representingNode = clusterGraph.getClusterNode();

			PortLocationManager manager = representingNode.getPortLocationManager();

			if(representingNode.getPortList().getPort(getId()) != null)
			{
				if(nodeType.equals(NodeType.source))
					representingNode.getPortList().addTargetPort(getId(),
						manager.getNextSourceLocation());
				else if(nodeType.equals(NodeType.sink))
					representingNode.getPortList().addSourcePort(getId(),
						manager.getNextTargetLocation());
			}
		}
	}
	...

}
```

Code fragment 70

The user decomposes a node by invoking the 'Decompose Node' action. In Code fragment 71 we see that the DecomposeAction first questions the user for the filename of the intended clustergraph for this graph. Then the doesKpnFitInNode method checks if the graph fits into this node, that is if there is a source/targetport for each sink/sourcenode in the clustergraph. (which we see in Code fragment 72). If this test succeeds the action returns a DecomposeCommand. In Code fragment 73 we see that this command is a merger of two actions. The first action turns the node into a clusternode, preserving all it's edges an properties (this quite an complex operation, so we won't show the details here). The second loads the chosen graph at sets it as the clusternode's graph.

```
public class DecomposeAction extends GraphviewerAction{
	...
	protected Command doRun() {

		String fileName = getFileNameFromUser();
		if(fileName != null)
		{
			INode node = nodePart.getBean();
			IGraph graph = node.getGraph();


			if(doesKpnFitInNode(fileName, node, true))
			{
				DecomposeNodeCommand command = new DecomposeNodeCommand();

				command.setNode(node);
				command.setFileName(fileName);

				return command;
			}
		}

		return null;
	}
	...
}
```

Code fragment 71

```
public class DecomposeAction extends GraphviewerAction{

	...
	protected static boolean doesKpnFitInNode(String fileName, INode node, boolean messageFailure)
	{
		IGraph graph = KpnFileManager.loadGraph(fileName, false);

		int noPorts = node.getPortList().listPorts().size();
		int noSourcePorts = node.getPortList().listSourcePorts().size();
		int noTargetPorts = node.getPortList().listTargetPorts().size();

		int noNodes = graph.listNodes().size();
		int noSourceNodes = countSourceNodes(graph.listNodes());
		int noSinkNodes = countSinkNodes(graph.listNodes());


		boolean kpnFitsInNode = true;
		kpnFitsInNode = kpnFitsInNode && (noSourceNodes == noTargetPorts);
		kpnFitsInNode = kpnFitsInNode && (noSinkNodes == noSourcePorts);

		... // show dialog with error message

		return kpnFitsInNode;
	}
	...
}
```

Code fragment 72

```
public class DecomposeNodeCommand extends Command
{
	...
	public void execute()
	{
		makeCommand = new MakeClusterNodeCommand();
		makeCommand.setNode(node);
		makeCommand.execute();

		loadCommand = new LoadKpnIntoClusterNodeCommand();
		loadCommand.setClusterNode(makeCommand.getClusterNode());
		loadCommand.setFileName(fileName);

	}
	...
}
```

Code fragment 73

- **Case study**

In this paragraph we present a case study. First we we handle a KPN called DCT, a small KPN which allow us to show the basic functionlity of the Graphview program Then we will move on to a larger example, a KPN generated from the M_JPEG algorithm. Since this algorithm uses the DCT algorithm as one of it's assignstatements, this is a perfect opportunity to show the cluster operations the editor provides.

We will start by providing the matlab sequential program that is the DCT (Discrete Cosine Transform) program. The matlab code for this program is shown in Code fragment 74. The structure of this program is quite simple. The Pixel array is first initialized with the source values (input to the program). The second part of the program consists of an outer loop, with three inner loops, on of which is conditional. The last part outputs the results to an output array.

In The DCT KPN: the edges have labels that display their size. No coloring strategies have been performed. we see this KPN visualized, with a coloring strategies set to none: the nodes and edges have a default color. On the edges we see a label, displaying the calculated edges size (the data transfer rate of the link). On first sight we note that there are two paths from the node labeled 'Source' to the node labeldd 'PixelsToBlock'. This represents the fact to under the condition k<=2 in the outer loop, the Preshift assignment is executed before the PixelsToBlock operation is performed.

Figure 13: The DCT KPN with the edges colored according to their size attribute. shows the same KPN, but now with the edges colored according to their sizes. The edge with the largest size immediately jumps out with its bright green color. The edges with the middle sizes (which are 1/4th of the largest size) are colored a medium blue. The edge with the really small size (1) is colored with the darkest blue.

```
for k = 1:1:4,
      for j = 1:1:64,
            [ Pixel(k,j) ] = Source;
    end
end

for k = 1:1:4,

  if k <= 2,
    for j = 1:1:64,
      [ Pixel(k,j) ] = PreShift( Pixel(k,j) );
    end
  end

  for j = 1:1:64,
    [ Block ] = PixelsToBlock( Pixel(k,j) );
  end

  for j = 1:1:64,
    [ Pixel(k,j) ] = BlockToPixels( Block );
  end
 end

for k = 1:1:4,
  for j = 1:1:64,
    [ ] = Sink( Pixel(k,j) );
  end
end
```

Code fragment 74:The DCT matlab progarm

Compaan

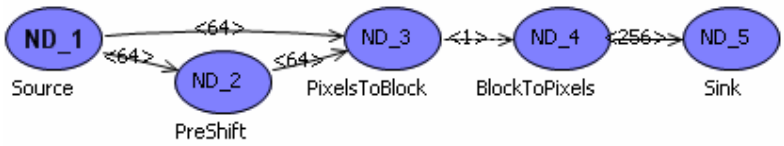


Figure 12 The DCT KPN: the edges have labels that display their size. No coloring strategies have been performed.

Compaan

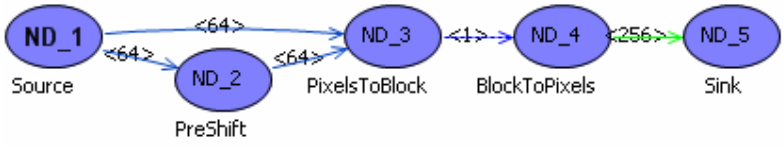


Figure 13: The DCT KPN with the edges colored according to their size attribute.

Now, lets add a textual note to the graph. In Figure 14: The DCT KPN annotated with some usefull information. we see how an annotation can be used to clarify something about a KPN. Now, the nodes are a bit cramped together, so we want to stretch the graph out a bit. We do this by moving the nodes around. The results are shown in Figure 15: The DCT KPN stretched out.. We see the labels remain at their appropriate positions. We did have to move the annotation seperately since this is not bound to the preshift node, or any other node.

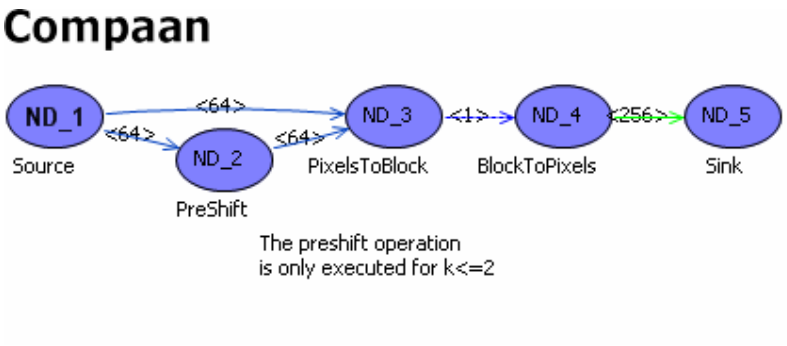


Figure 14: The DCT KPN annotated with some usefull information.

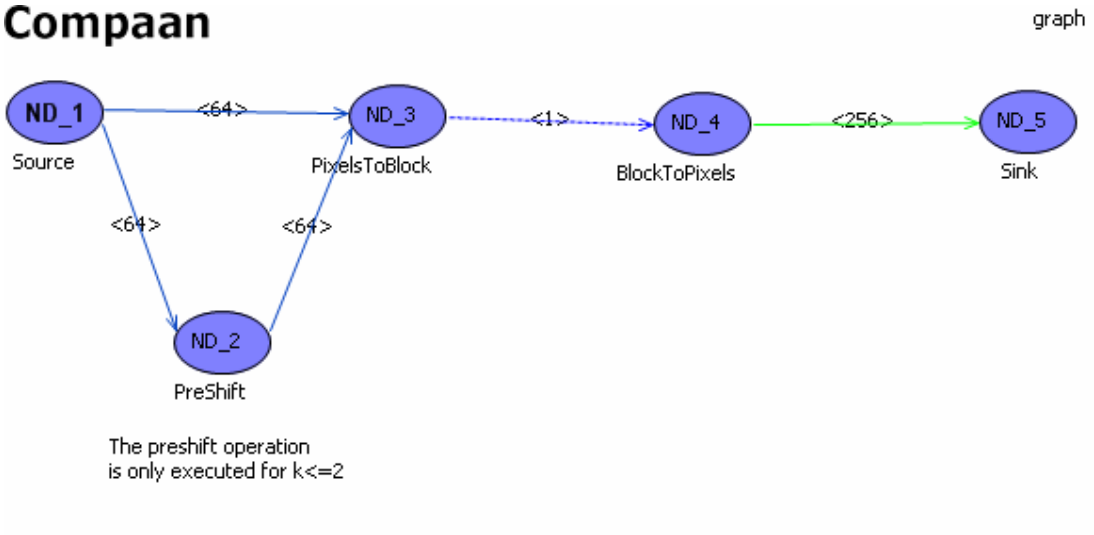


Figure 15: The DCT KPN stretched out.

We have shown that the editor performs simple tasks perfectly for a small example. We now move on to the more complex M_JPEG program and its corresponding KPN.

The M_JPEG matlab program is a bit large, so we won't show it here. Suffice it to say it analyzes a constant stream of JPEG images very quickly. Again we start with an image (see Figure 16: The M_JPEG KPN) of the graph without any labels, and with no coloring strategies performed. The first thing we notice is that there are a lot of edge crossings. In Figure 17: The M_JPEG KPN with large nodes. we see one way to solve this problem, we increase the node size. This gives the edges more space to attach to a node, and thus gives a clearer picture. The only problem is that we now have bigger nodes, and therefore a bigger picture in the editing canvas. This can off course be solved by zooming out of the node, but this make smaller parts of the graph (such as labels) lease easy readable.

A better solution therefore is to use the 'Hide multiple edges' action. In Figure 18: The M_JPEG KPN with multiple edges hidden. we see the same KPN with the multiple edges hidden. The basic data flow in the program now is a lot clearer.

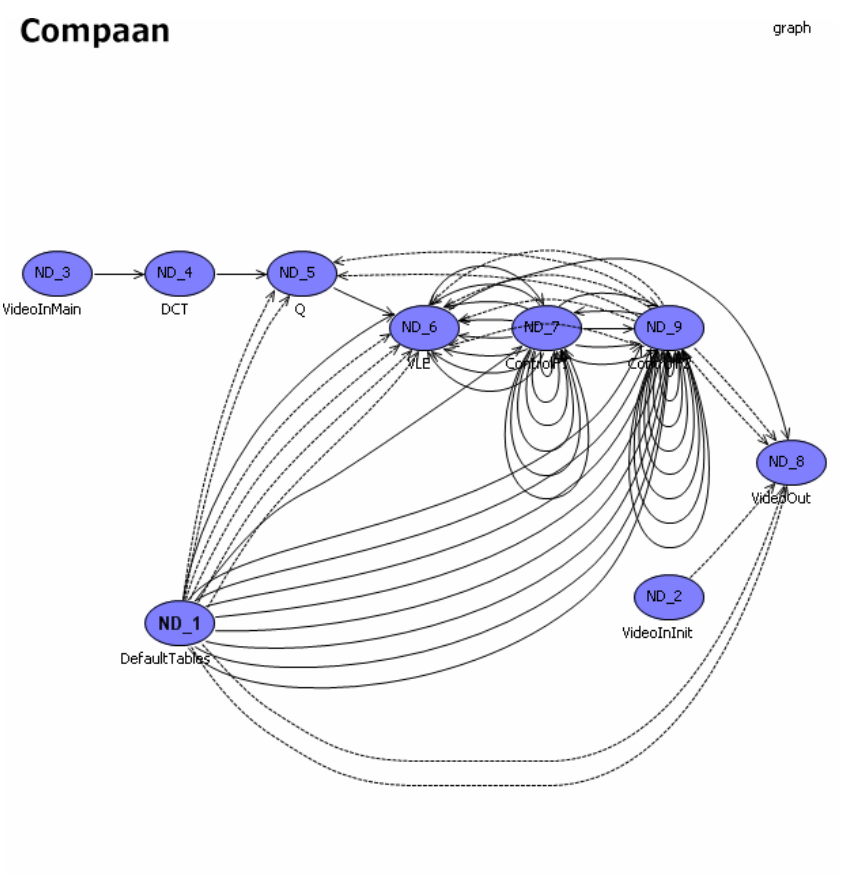


Figure 16: The M_JPEG KPN

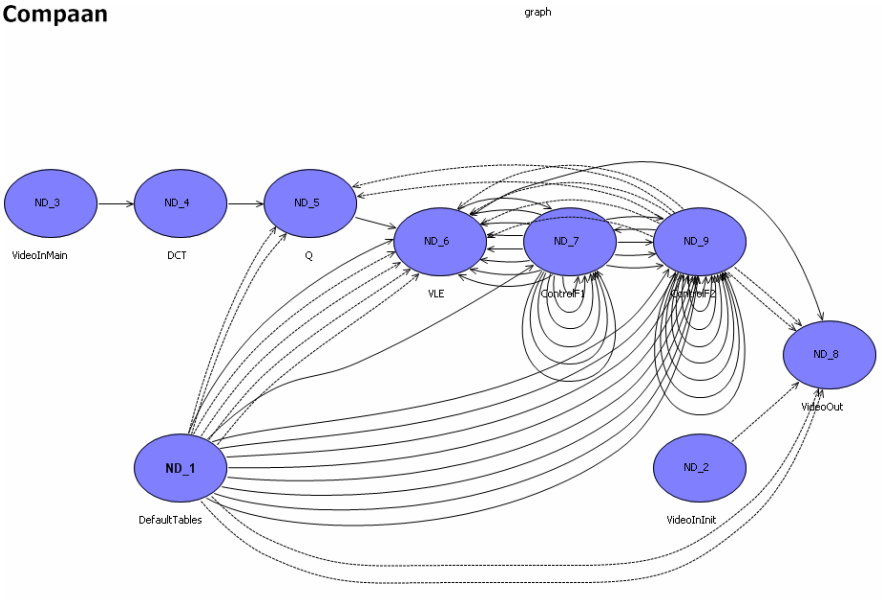


Figure 17: The M_JPEG KPN with large nodes.

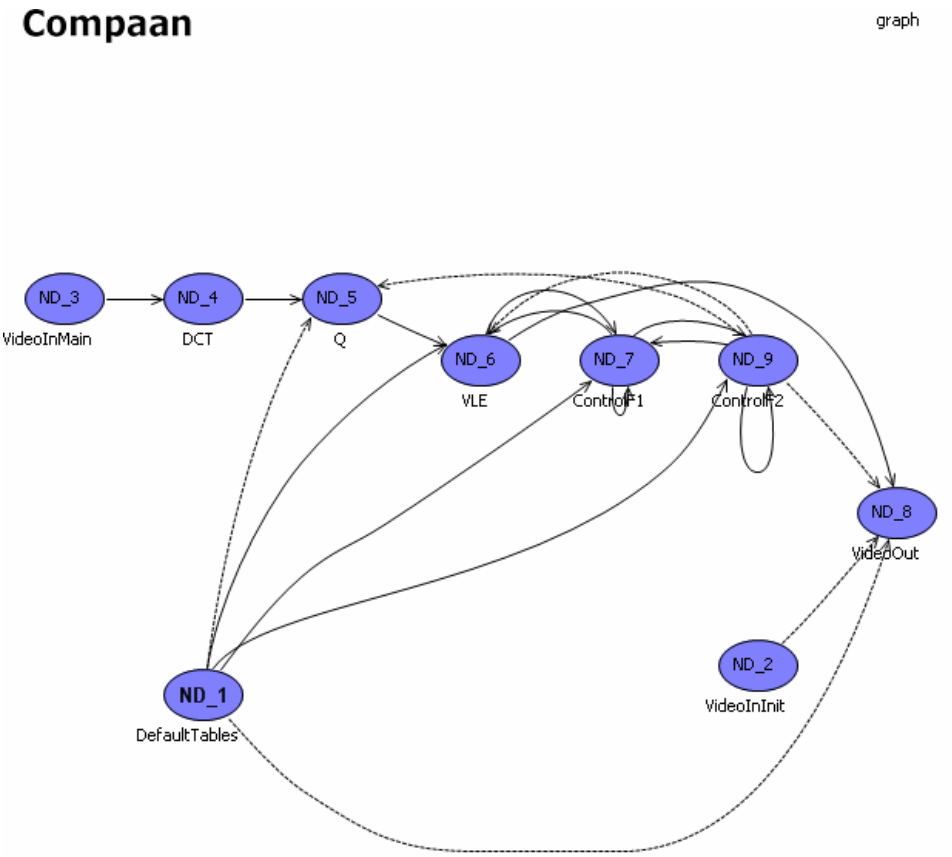


Figure 18: The M_JPEG KPN with multiple edges hidden.

Figure 19: The M_JPEG KPN with edges colored according to inputvariable. and Figure 20: The M_JPEG KPN with edges colored according to outputvariable. show the M_JPEG KPN with edges colored according to input and outputvariable. On first sight, the coloring based on in- or outargument gives us little information. For almost each pair of connected nodes the links have different colors, meaning that they work on different variables. The links from the ControlF1 node to the VLE node however al have the same input and outputvariables. This means that the same information is transmitted over different links to the same destination. This means that a designer can decide, based on this information, that not all of these data paths need to be implemented in the hardware, just one is enough.

The observant reader might have noticed the DCT node in this KPN. The DCT node represents some variable being assigned with the DCT of some block. In the M_JPEG matlab program, the implementation of this function is not further specified. We will replace this node with a clusternode which represents the DCT kpn. As expected this KPN 'fits' the DCT node, and the result of this decompose operation can be seen in Figure 21: The M_JPEG KPN with the DCT node decomposed. The result looks almost the same as figure 14, yet with the difference that the ports of the clusternode are shown. If we would descend into the clustergraph of the DCT clusternode, we would see the graph of figure 10.

The DCT nodes still holds the same properties as it did before the decompose operation, and so the meaning of the KPN is not lost. Since the clustergraph for the node represents the same ‘program’ as the clusternode, we now have redundant information. It would be interesting to research how w can change the KPN model, so that this information needn’t be stored twice.

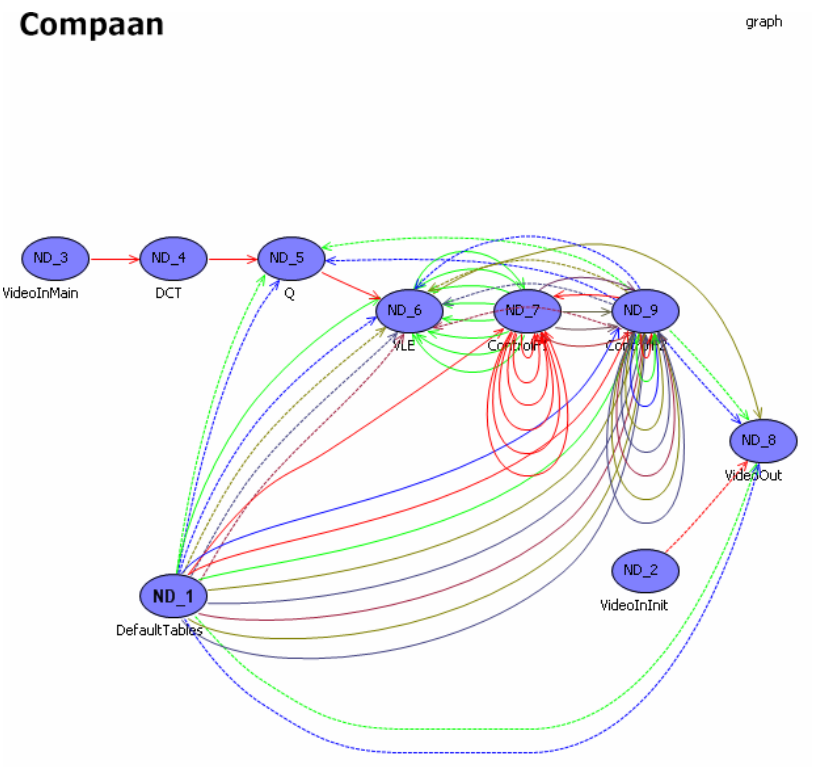


Figure 19: The M_JPEG KPN with edges colored according to inputvariable.

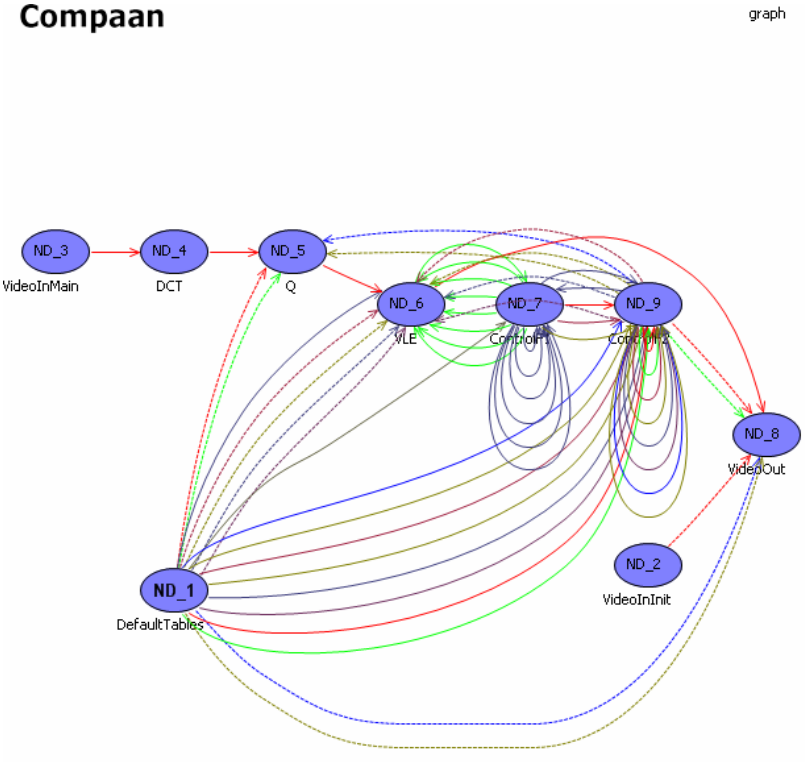


Figure 20: The M_JPEG KPN with edges colored according to outputvariable.

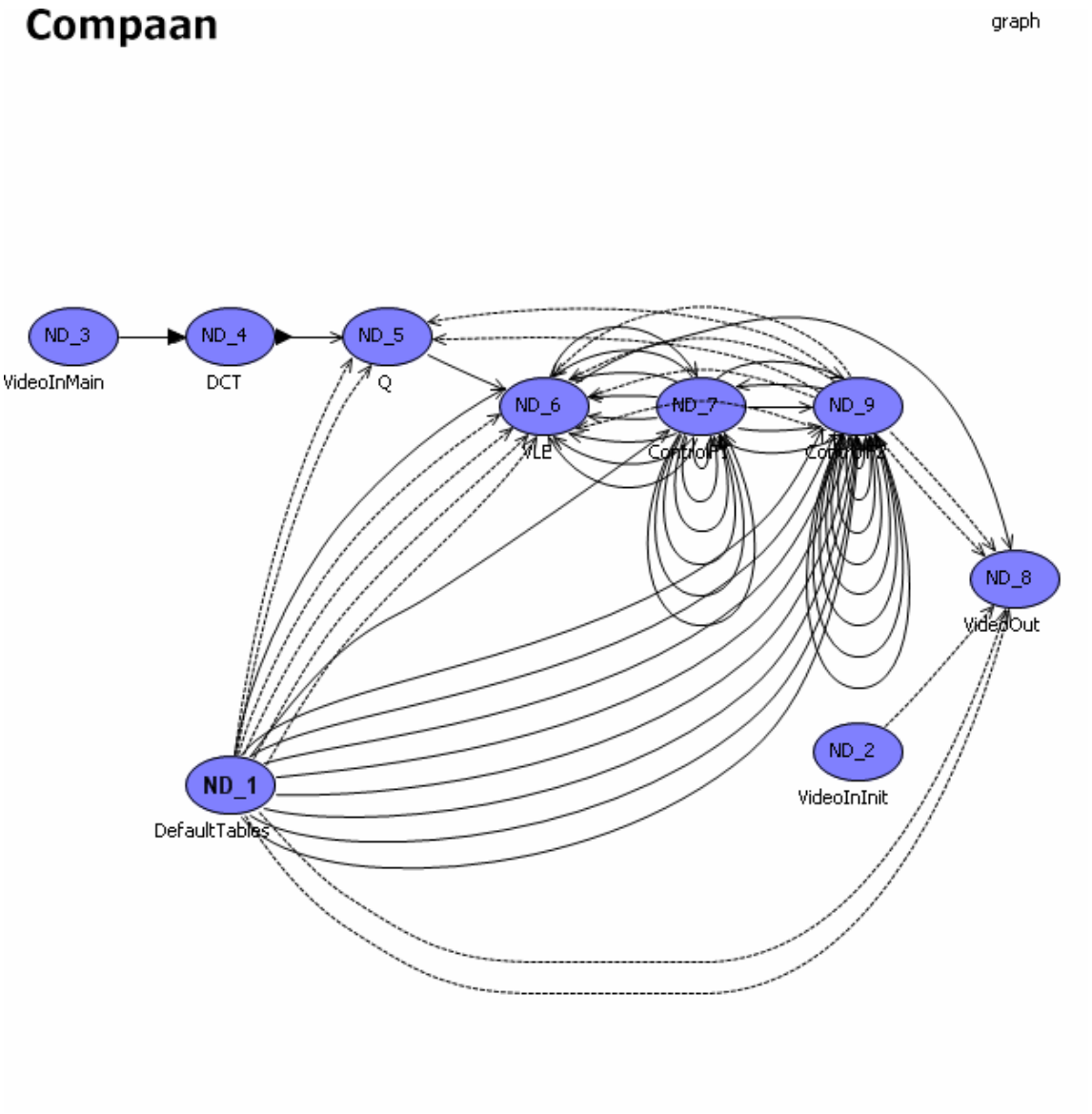


Figure 21: The M_JPEG KPN with the DCT node decomposed.

The last set of images shows the influences of the value of the parameters on the size of edges. Figure 22: Matlab code for QRvr program shows the code for the QRvr program. The program has two parameters, N (between 1 and 16) and K (between 1 and 1000). In Figure 23 to Figure 27 we see that varying the values of the graphs parameters has a profound impact on the sizes of the various node. It can therefore be concluded that choice of implementation for this KPN depends very much on the values of K an N.

```
%parameter N 1 16;
%parameter K 1 1000;

for j = 1:1:N,
  for i = j:1:N,
    [r(j,i)] = ReadMatrix_Zeros_64x64();
  end
end

for k = 1:1:K,
  for j = 1:1:N,
    [x(k,j)] = Read();
  end
end

for k = 1:1:K,
  for j = 1:1:N,
    [r(j,j), x(k,j), t ] = Vectorize( r(j,j), x(k,j) );
    for i = j+1:1:N,
      [r(j,i), x(k,i), t] = Rotate( r(j,i), x(k,i), t );
    end
  end
end

for j = 1:1:N,
  for i = j:1:N,
    [ Sink(j,i) ] = Pass( r(j,i) );
  end
end
```

Figure 22: Matlab code for QRvr program

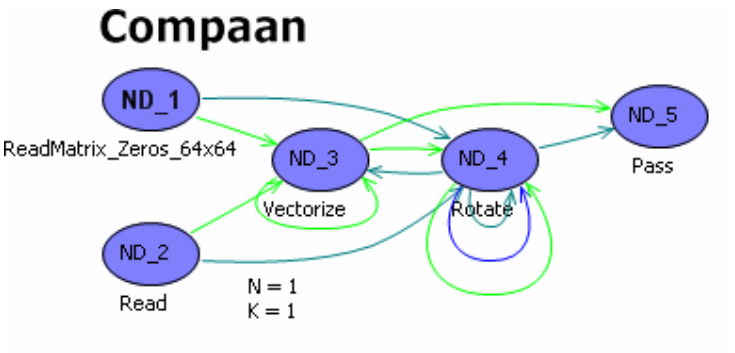


Figure 23: The QRvr KPN with parameters N=1 and K=1

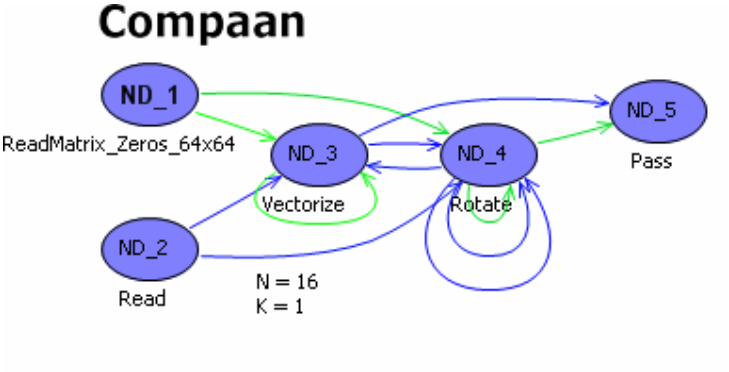


Figure 24: The QRvr KPN with parameters N=16 and K=1

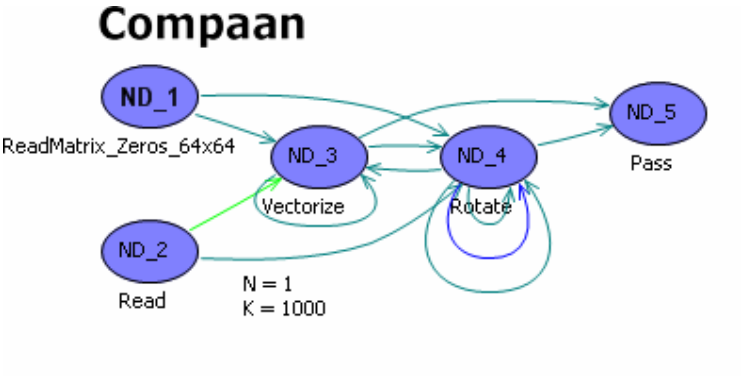


Figure 25: The QRvr KPN with parameters N=1 and K=1000

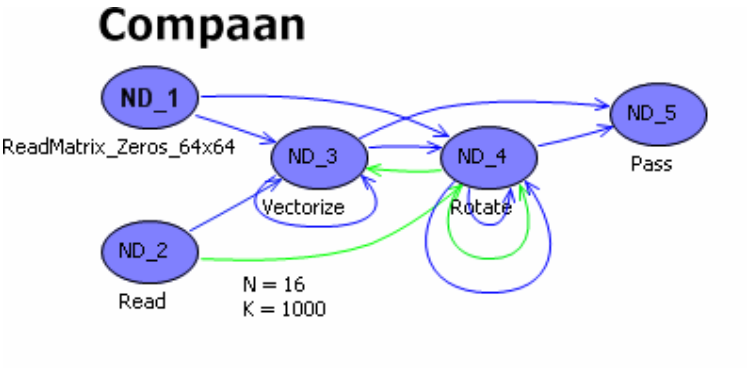


Figure 26: The QRvr KPN with parameters N=16 and K=1000

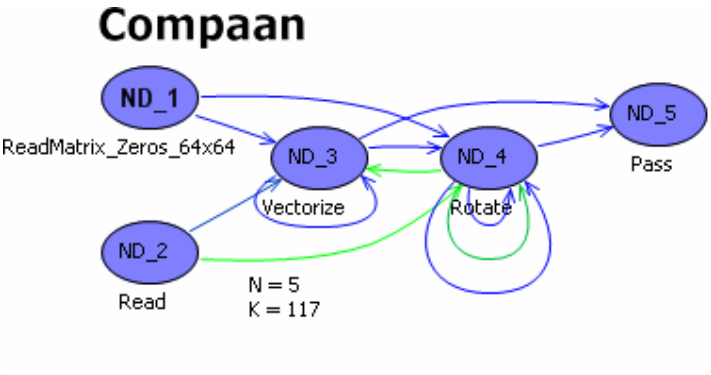


Figure 27 : The QRvr KPN with parameters N=5 and K=117

o Summary and conclusions

In this document we have described Graphview, a tool to visualize Kahn Process Networks. The tool is used to help a designer choose a implementation of a KPN onto an embbed multiprocessor system. Providing the user with a layout that illustrates the structure of the KPN is a quality that seperates it from other graph editors such as Simulink and Ptolemy. Furthermore, we have shown that visualizing the workload and data transfer rates helps the designer decide on the mapping of the entities and links to the set of (co)processors and datapaths of the embedded system.

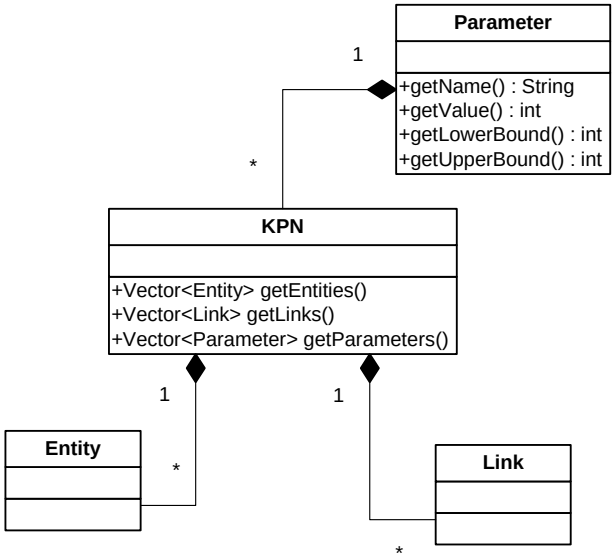
An new idea that has been proposed in this document is to use hierarchy in KPN's. As it is unclear how relating a node to a seperate KPN, 'partitioning the sequential process this node represents' influences the meaning of the domain of the node and it's ports, this might be a interesting point for further research.

o Appendix A The KPN datastructure

"The KPN model of computation assumes a network of concurrent autonomous processes that communicate in a point-to-point fashion over unbounded FIFO channels using a blocking-read synchronization primitive. Each process in the network is specifiifed as a sequential program that executes concurrently with other processes"[6]

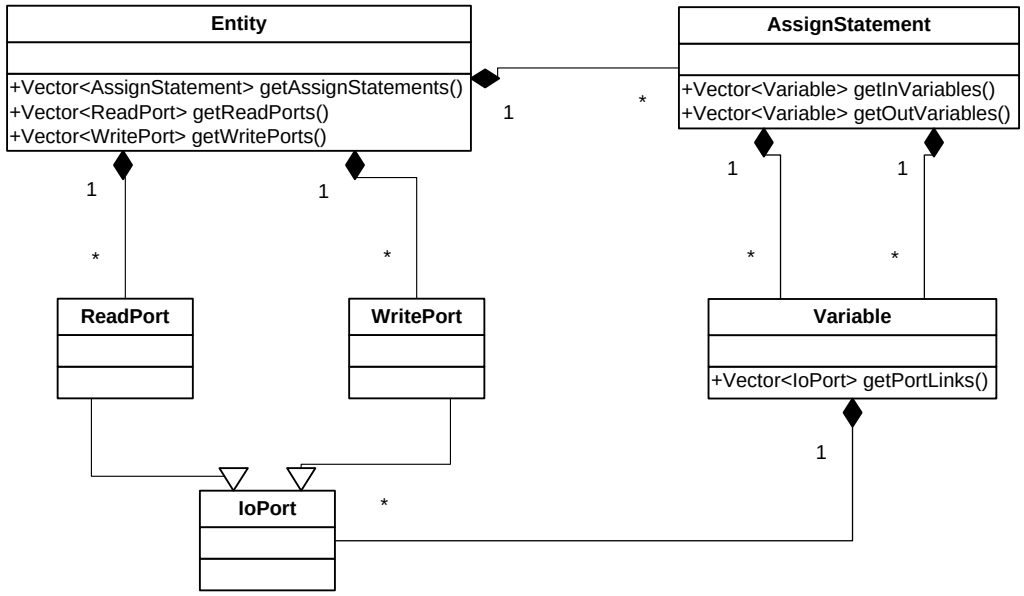
The KPN datastructure implements this model. The processes are implemented by the Entity class, the FIFO channels by the Link entity class. Each KPN has, as would be exepected, a list of both of these type of objects.

The compaan tool is used to generate KPN's from sequential programs and compaan calculates expressions for the amount of times a process is run, and the amount of data that is transferred over a fifo channel. These expressions are parametrized, that is, they depend on input variables to the sequential program being processed.

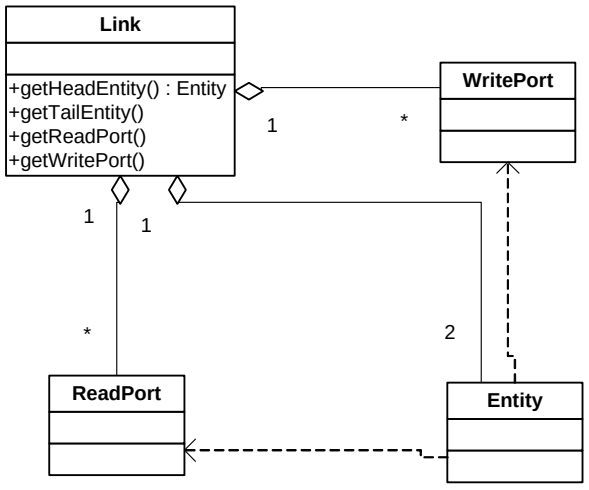


class diagram 2 : The KPN structure contains a list of Entities and a list of Links. Apart from this a list of Parameters is also available to evaluate the size expression of these nodes and edges.

The communication between Entities goes through Links. The output of an Entity is represented by outputvariables. The values of these outputvariables are transferred to become the value of the inputvariable of another link. In the datastructure, each Entity has one or more Assignstatements. These Assignstatements correspond with the assign statements in the sequential program. Each statement that is executed by the process represented by this Entity is reflected by an Assignstatement in the entity. This assignstatement has a number of in and outputvariables. Each link can use the value of these variables by creating an IoPort on the specific variable. Therefore, each variable can be associated with a number of read or writeports.

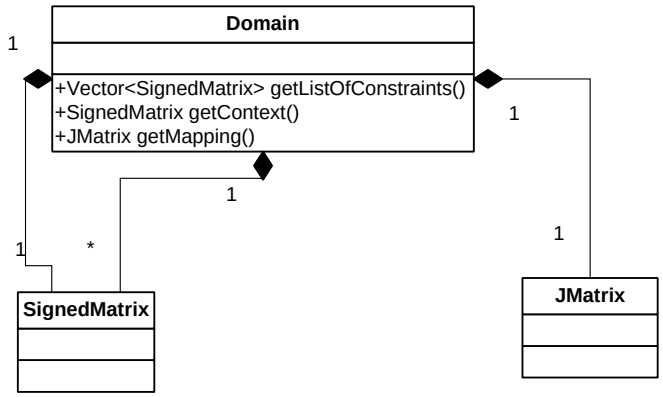


class diagram 3: Each entity has a number of assignstatements, that contain in and outputvariables. These variables can be accesed via IoPorts.



class diagram 4 : Links are bound to their read- and writeEntity via IoPorts. These ports are used to retrieve the value of the variables that the Link operates on.

Without going into specific details, each Entity and each Port has a Domain property. For an entity this Domain contains a number of matrices that describe which part of the original program is assigned to the process. The points of the domain a port reads from form the points for this ports domain.



class diagram 5: The domain contains, among other things a context , a list of constraints, a context and a mapping matrix.

- **References:**

1. Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, Ed Deprettere: System Design using Kahn Process Networks: The Compaan/Laura Approach, in proceedings of the Design, Automation and Test in Europe conference DATE2004, Feb 16-20 2004, Paris, France.
2. The Graphviz project website home page: http://www.graphviz.org/
3. The Ptolem project website: home page: http://ptolemy.berkeley.edu/ptolemyII/
4. Wikipedia: Simulink article,: http://en.wikipedia.org/wiki/Simulink. GEF's FAQ: http://wiki.eclipse.org/index.php/GEF_Developer_FAQ
5. Randy Hudson, Software developer,IBM:Create an Eclipse-based application using the Graphical Editing Framework:, http://www-128.ibm.com/developerworks/opensource/library/os-gef/
6. Edwin Rijpkema, Ed F. Deprettere and Bart Kienhuis:Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures, 8th International Workshop on Hardware/Software Codesign (CODES'2000), May 3 -- 5 2000, San Diego, CA, USA.
7. Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). Design patterns: Elements of reusable object-oriented software. Reading, MA: Addison Wesley.
8. Wikipedia: JavaBeans article: http://nl.wikipedia.org/wiki/JavaBeans

o Table of contents:

- Abstract 1
- Introduction 1
- Problem description 2
  - Representation 2
  - Manipulation 3
  - Development Environment Integration 4
- Related Work 4
  - Ptolemy 4
  - Simulink 5
  - Development environment integration 5
  - Conclusion 5
- Solution Approach 5
- *Requirements Definition* 6
- Background 9
  - Model-View-Controller pattern 9
  - The Graphical Editing Framework 10
  - Graphviz, Dot and JDot 11
- Solution implementation 12
- Using GEF in Graphview 12
  - The Model 12
  - The Controller 21
  - The View 31
- Visualisation and Interaction with the view 37
  - File Management 37
  - Displaying graphs, nodes and edges 39
  - Creating graph elements 41
  - Moving graph elements 45
  - Removing graph elements 46
  - Displaying properties in labels 48
  - Label positions 52
  - Graph layout 56
  - Displaying information through color 60
  - Editing non- visual information 64
  - Notes: 64
  - KPN Specific 65
  - Outputting the graph to other media 67
  - Clusternodes and clustergraphs 68
- Case study 74
- Summary and conclusions 83
- Appendix A The KPN datastructure 83
- References: 86
- Table of contents: 87