KGraph Text (KGT)

Project Information

Responsible:

- Christian Schneider

The KGraph Text (KGT) project provides a textual syntax and editor for specifying KGraphs. While editing the KGraph in the textual editor, we provide a visualization of the graph in a KLighD view. This project was started with two use cases in mind:

1. Unit testing: KGT is a light-weight format that test graphs can be stored in.
2. Layout algorithm development: KGT is easy to use for testing layout algorithms while developing them.

This page documents the basic syntax of the KGT format and how to use the editor together with the KLighD view.

Content

- Quick Start Tutorial
- The KGraph Text Format
  - Basic Syntax
  - Types of Objects in the KGraph
  - Object Parameters
  - Object Properties
- A More Complex Example

Quick Start Tutorial

To quickly get you up to speed on how to write basic KGT files, we have a full-blown tutorial available for you learning pleasure. Once you've completed that, the rest of this page will provide a more comprehensive description of the KGT syntax.

The KGraph Text Format

In this section, we will take a closer look at the KGT syntax. We will start with the basic syntax. We will then take a look at the different kinds of objects available in KGT and at specifying parameters and properties of these objects.

> **Shortcut**
>
> If you prefer reading Xtext grammars, you will find it over here.

Basic Syntax

In KGT, the KGraph is specified by listing the nodes in the graph, their ports, edges, and labels. The specification of each type of object follows the same basic syntax:

```
type [id] <details>
```

An ID is only necessary if an object needs to be referenced later. For example, if you define a port that you want to connect an edge to, the port better have an ID or you won't be able to specify which port the edge should connect to. The `details` field doesn't always contain information. In fact, it is only necessary for the text of labels and for the source and target points of edges (see the next section for more details).

With this basic syntax, the following text specifies a complete KGraph with three nodes with the IDs `n1`, `n2`, and `n3`:

Example

```
knode n1
knode n2
knode n3
```

The KGraph itself is simply a list of objects. Some objects can be (and are sometimes expected to be) nested in other objects. For instance, a port must always belong to a node. To be able to add objects to other objects, we extend the basic syntax a little:

```
type [id] <details> {
    <declarations>
}
```

If nodes `n2` and `n3` should be nested inside `n1`, the previous example must be changed a little:

Example

```
knode n1 {
    knode n2
    knode n3
}
```

Types of Objects in the KGraph

Now that we know the basic way of specifying objects, it's time to see what types of objects there are, which other objects they can appear in, and whether they require any details.

Type	Keyword	Details	To appear in	Remarks
Node	`knode`	Empty	Top level, nodes	
Port	`kport`	Empty	Nodes	
Label	`klabel`	String (label text): `"My gigantically useful label text"`	Nodes, ports, edges	
Edge	`kedge`	Source and target: `(:sourcePortId -> targetNodeId: targetPortId)`	Nodes	Edges must always be defined in their source node. That is why the source node is not defined in the details.

When specifying edges, each port ID and the colon preceding it is optional. After all, edges don't have to connect ports, but can also connect nodes directly.

Object Parameters

So far, we have learned how the objects in a KGraph can be specified in the KGT format. However, this doesn't help us much since we don't know yet how to add basic parameters to the objects. This includes, for instance, the size of nodes.

Parameters are the first thing inside a block (the curly braces) of an object. There are three basic parameters that can be attached to a nodes, ports, and labels:

```
pos: x=<xPosition> y=<yPosition>
size: width=<width> height=<height>
insets: top=<top> bottom=<bottom> left=<left> right=<right>
```

A node with a predefined size could be written like this:

Exampe

```
knode n1 {
    size: width=100 height=50
}
```

Edges are somewhat different. Position, size, and insets aren't meaningful to edges, so edges have a different parameter that defines their list of bend points:

```
points: <source>;<bend>*;<target>
```

Each of the points is defined by their x and y coordinate as such:

```
<x>,<y>
```

Object Properties

If you have already used a layout algorithm through KIML, you will know that layout algorithms are configured through properties attached to the different graph objects. In KGT, this is done through the `properties` section that follows the parameters:

```
properties:
    <propertyId> = <propertyValue>
```

The `properties` section can of course contain an arbitrary number of lines, each specifying the value of one property. For example, if we want to add ports and a label to a node, we will often have to set certain properties on it as well to tell the layout algorithm what to do:

Example

```
knode n1 {
    properties:
        de.cau.cs.kieler.sizeConstraint="PORTS NODE_LABELS"
        de.cau.cs.kieler.nodeLabelPlacement="INSIDE V_TOP H_CENTER"
        de.cau.cs.kieler.portConstraints=FIXED_SIDE

    klabel "Node 1"

    kport p1 {
        size: width=5 height=5
        properties:
            de.cau.cs.kieler.portSide=WEST
        klabel "West port 1"
    }

    kport p2 {
        size: width=5 height=5
        properties:
            de.cau.cs.kieler.portSide=WEST
        klabel "West port 2"
    }

    kport p3 {
        size: width=5 height=5
        properties:
            de.cau.cs.kieler.portSide=EAST
        klabel "East port 1"
    }
}
```

Often enough, we also have to set properties on the graph itself (for instance, which layout algorithm to use). In that case, simply include a `properties` section at the top of the file:

Example

```
properties:
    de.cau.cs.kieler.algorithm="de.cau.cs.kieler.klay.tree"

knode n1 {
    size: width=50 height=50

    kedge (-> n2)
    kedge (-> n3)
}
knode n2 {
    size: width=50 height=50
}
knode n3 {
    size: width=50 height=50
}
```

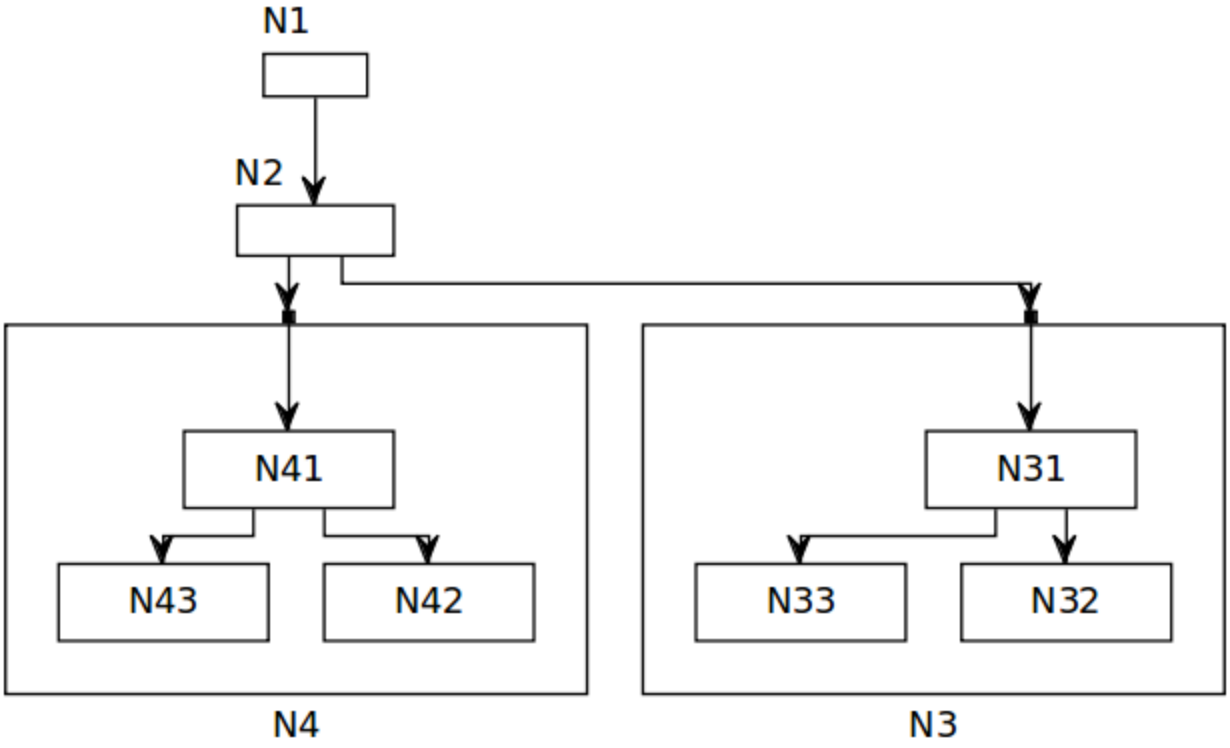
A More Complex Example

We finish with a more complex example that highlights some of the features of the KIELER graph layout technology and how to use it through the KGT format.

Example

```
properties:
    de.cau.cs.kieler.edgeRouting=ORTHOGONAL
    de.cau.cs.kieler.direction=DOWN
    de.cau.cs.kieler.separateConnComp=FALSE

// The node label of N1 is placed outside. Edges connect directly to
// the node, without any ports in between
knode N1 {
    properties:
        de.cau.cs.kieler.nodeLabelPlacement="OUTSIDE H_LEFT V_TOP"
        de.cau.cs.kieler.sizeConstraint="NODE_LABELS PORTS"
    klabel "N1"

    kedge (-> N2)
}

// Compared to N1, this node will be slightly higher because the size
// constraints also take a default minimum size into account
knode N2 {
    properties:
        de.cau.cs.kieler.nodeLabelPlacement="OUTSIDE H_LEFT V_TOP"
        de.cau.cs.kieler.sizeConstraint="MINIMUM_SIZE NODE_LABELS PORTS"
        de.cau.cs.kieler.sizeOptions=DEFAULT_MINIMUM_SIZE
    klabel "N2"

    kedge (-> N3:N3Port)
    kedge (-> N4:N4Port)
}

// N3 is a compound node with three nodes nested inside. The layout
// algorithm uses the default node placement algorithm
knode N3 {
    properties:
        de.cau.cs.kieler.edgeRouting=ORTHOGONAL
        de.cau.cs.kieler.direction=DOWN
        de.cau.cs.kieler.nodeLabelPlacement="OUTSIDE H_CENTER V_BOTTOM"
    klabel "N3"

    kport N3Port {
        size: width=5 height=5
        pos: x=10 y=-5
```

```
    }
    kedge (:N3Port -> N31)

    knode N31 {
        size: width=80 height=30
        properties:
            de.cau.cs.kieler.nodeLabelPlacement="INSIDE H_CENTER V_CENTER"
        klabel "N31"

        kedge (-> N32)
        kedge (-> N33)
    }

    knode N32 {
        size: width=80 height=30
        properties:
            de.cau.cs.kieler.nodeLabelPlacement="INSIDE H_CENTER V_CENTER"
        klabel "N32"
    }

    knode N33 {
        size: width=80 height=30
        properties:
            de.cau.cs.kieler.nodeLabelPlacement="INSIDE H_CENTER V_CENTER"
        klabel "N33"
    }
}

// Another a compound node with three nodes nested inside. The layout
// algorithm uses the linear segments algorithm for node placement
knode N4 {
    properties:
        de.cau.cs.kieler.edgeRouting=ORTHOGONAL
        de.cau.cs.kieler.direction=DOWN
        de.cau.cs.kieler.klay.layered.nodePlace=LINEAR_SEGMENTS
        de.cau.cs.kieler.nodeLabelPlacement="OUTSIDE H_CENTER V_BOTTOM"
    klabel "N4"

    kport N4Port {
        size: width=5 height=5
    }
    kedge (:N4Port -> N41)

    knode N41 {
        size: width=80 height=30
        properties:
            de.cau.cs.kieler.nodeLabelPlacement="INSIDE H_CENTER V_CENTER"
        klabel "N41"

        kedge (-> N42)
        kedge (-> N43)
    }

    knode N42 {
        size: width=80 height=30
        properties:
            de.cau.cs.kieler.nodeLabelPlacement="INSIDE H_CENTER V_CENTER"
        klabel "N42"
    }

    knode N43 {
        size: width=80 height=30
        properties:
            de.cau.cs.kieler.nodeLabelPlacement="INSIDE H_CENTER V_CENTER"
        klabel "N43"
    }
}
```


N1
N2
N41
N43
N42
N4
N31
N33
N32
N3