Drawing ER diagrams with Ti*k*Z

Claudio Fiandrino

Abstract

The paper will illustrate some techniques to represent Entity-Relationship (ER) diagrams with Ti*k*Z. In particular, it will focus on the standard internal library `er`, on the external package Ti*k*Z-er2, on the external tool Graphviz and on the object-oriented approach provided by the `er-oo` library.

Sommario

L'articolo illustrerà alcune tecniche per rappresentare i diagrammi Entià-Relazione (ER) con Ti*k*Z. In particolare si concentrerà sulla libreria standard interna `er`, sul pacchetto esterno Ti*k*Z-er2, sul programma esterno Graphviz e sull'approccio orientato agli oggetti fornito dalla libreria `er-oo`.

1 Introduction

The Entity-Relationship (ER from now on) model is a common way to model and represent databases. Peter Chen proposed the model specification in (Chen, 1976). The model is built using three main blocks:

- entities;
- relationships;
- attributes.

Entities are real-world items or concepts that can exist independent of one another and are uniquely identified. Examples of *physical* entities are "computer" or "car", while *concept* entities are "customer order" or "payment". Their standard notation is a rectangle. The entity "student" is represented in figure 1.

Figure 1: Entity graphical representation

Entities are linked by relationships, which describe the relation the entities share. Such relations can be classified according to the so-called degree of the relationship, an index of relevance that represents how many entities the relationship involves. Relationships are represented with diamonds as shown in figure 2, where a "student" and a "professor" are linked by a *thesis*. According to common diagram conventions, a relation in which entities participate more than once is called *total* or *surjective* or *recursive* and the visual link is double line. The relation "supervision" in a "team", shown in figure 3, is *total* because some team members will be supervisors, others will be supervisee.

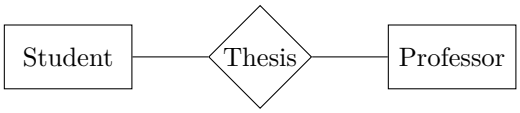

Figure 2: Relationship graphical representation

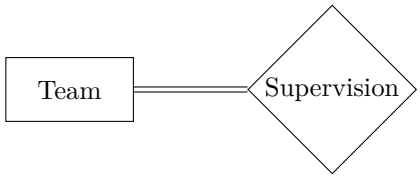

Figure 3: Total relation graphical representation

At last, both entities and relationships can have attributes to describe particular properties. Notice that attributes themselves can have attributes and are called *composite attributes*; for example, the "address" attribute for a "person" entity could be described by "street" and "city", two attributes in this case. Attributes are represented with ovals which border changes according to the type of the attribute:

solid border for *simple* attributes;

dashed border for *derived* attributes. Attributes are derived when we infer them from entities or relationships (e.g., the "age" attribute could derive from the entity "person");

double border for *multi*-attributes. Likewise total relationships, a multi-attribute has more than one value per entity or relationship (e.g., the attribute "phone" for the entity "person").

Figure 4 shows every possible attributes in a single example.

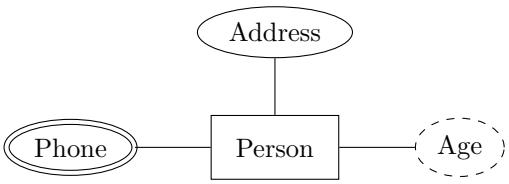

Figure 4: Attributes graphical representation

While this short introduction to ER models is not exhaustive, it provides the sufficient

background to understand this paper. Further references are http://wofford-ecs.org/dataandvisualization/ermodel/material.htm and http://users.informatik.uni-halle.de/~brass/db04/c3_ermod.pdf.

The remainder of the paper is organized as follows: section 2 will focus on the er library, section 3 will describe the Tikz-er2 package, section 4 will show how Graphviz works and section 5 will mention the er-oo module. Every sections will show the same example programmed (drawn) using the discussed tool. At last, section 6 will conclude the paper.

2 The library er

2.1 Usage

Ti*k*Z provides plenty of standard libraries, including er (Tantau, 2010, section 31 Entity-Relationship Diagram Drawing Library). As every Ti*k*Z libraries, the user has to load it in the preamble:

```
\usepackage{tikz}
\usetikzlibrary{er}
```

The library defines those keys necessary to represent standard entities, relationships and attributes. Those keys are:

- entity to represent the entity nodes;
- relationship to represent the relationship nodes;
- attribute to represent the attribute nodes;
- key attribute to represent a key attribute node;

and they are applied to Ti*k*Z nodes by the command

```
\node[key type] (label) at (position)
 {text};
```

The user can customize nodes style modifying the every entity, every relationship and every attribute keys. The ways to globally customize nodes of an ER diagram, according to Ti*k*Z habits, are:

- ```
  \tikzset{every entity/.style={
      ...
      customization
      ...
   },
  }
  ```

- ```
  \tikzstyle{every entity}=[
      ...
      customization
      ...
  ]
  ```

The latter way is discouraged in favor of the former.

The main advantage of using the er library is that users do not need to use external packages or tools: standard TEX Live or MiKTEX let them immediately operate. Unfortunately users are always requested to spatially organize the diagram and, optionally, to customize elements style. When the diagram is large, finding a good layout can be tricky and time-consuming.

2.2 A real example with er

We will now see how to draw a simple diagram composed by two entities, "person" and "tool", linked by the only relationship "uses". Notice that more than one person may use the same tool and a tool can use other tools. This diagram will be taken as reference to compare the code of all the techniques shown in the paper.

Listing 1 shows a complete minimal example which result appears in figure 5.

As described in subsection 2.1, the library only provides keys for basic elements, so it has been necessary setting up the multi attribute, derived attribute and total relationships styles. Notice that defining these style is very simple: it is just needed to use the already present attribute style along with the necessary customization. That is:

```
\tikzset{multi attribute/.style={
  attribute,
  double distance=1.5pt
  }
}
```

This particular procedure carries a very important advantage: the new attribute type will inherit its *father* style attribute. Indeed, it is perfectly possible to manually set up the multi attribute:

```
\tikzset{multi attribute/.style={
  ellipse,
  minimum size=1.5\baselineskip,
  draw,
  double distance=1.5pt,
  every multi attribute
  }
}
```

which prevents a later automatic style customization.

A specific command \key has been created to distinguish the key attribute. Indeed the er library does not provide any methods to underline a key attribute and just emphasizes it using italic.

Notice how all the elements have been manually positioned with keys above, below, left and right; they could have even been combined for finer results (e.g., above right). This explains why it is important that each node has its own *name*: in this way, it is possible to place it relatively to already defined nodes (e.g., above of=prevnode). The elements are 7em distant from each other

```
\documentclass[a4paper,11pt,x11names]{article}

\usepackage{tikz}
\usetikzlibrary{er}
\tikzset{multi attribute/.style={attribute,double distance=1.5pt}}
\tikzset{derived attribute/.style={attribute,dashed}}
\tikzset{total/.style={double distance=1.5pt}}
\tikzset{every entity/.style={draw=orange, fill=orange!20}}
\tikzset{every attribute/.style={draw=MediumPurple1, fill=MediumPurple1!20}}
\tikzset{every relationship/.style={draw=Chartreuse2, fill=Chartreuse2!20}}
\newcommand{\key}[1]{\underline{#1}}

\begin{document}
\begin{tikzpicture}[node distance=7em]
\node[entity] (person) {Person};
\node[attribute] (pid) [left of=person] {\key{ID}} edge (person);
\node[attribute] (name) [above left of=person] {Name} edge (person);
\node[multi attribute] (phone) [above of=person] {Phone} edge (person);
\node[attribute] (address) [above right of=person] {Address} edge (person);
\node[attribute] (street) [above right of=address] {Street} edge (address);
\node[attribute] (city) [right of=address] {City} edge (address);
\node[derived attribute] (age) [right of=person] {Age} edge (person);
\node[relationship] (uses) [below of=person] {Uses} edge (person);
\node[entity] (tool) [below of=uses] {Tool} edge[total] (uses);
\node[attribute] (tid) [left of=tool] {\key{ID}} edge (tool);
\node[attribute] (tname) [right of =tool] {Name} edge (tool);
\end{tikzpicture}
\end{document}
```

Listing 1: Exploiting the er library

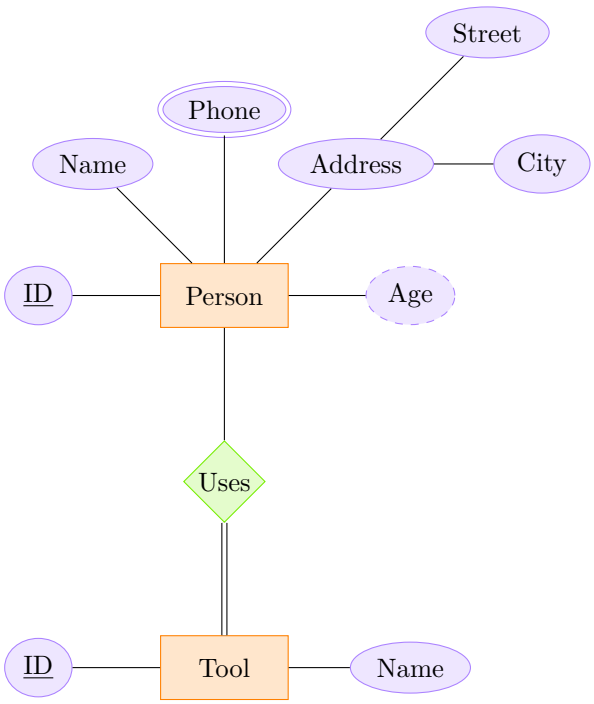

Figure 5: The reference ER diagram

thanks to the key node distance: it is the distance between the anchor center of each pair of nodes. This distance applies to every nodes in tikzpicture, but it could be locally redefined in case one element should be shifted a bit; the right keys to use for that are xshift and yshift.

```
\node[multi attribute, xshift=1cm,
  yshift=1cm]
  (phone) [above of=person] {Phone}
  edge (person);
```

It is not important where we place the options, i.e., we can set them *before* or *after* the type of the node. However, what it is important to highlight is that the syntax

```
\node[options]
  (name) [position] {label}
  edge (destination);
```

makes it possible to put and connect a node to an already existing destination node. Users usually write nodes in Ti*k*Z all together and then create the links using draw or path. That particular procedure allows to be fast, but the destination has to be already defined when a new node is attached to it. Thus:

```
\node[multi attribute] (phone)
  [above of=person] {Phone}
  edge (person);
\node[entity] (person) {Person};
```

will not work, but rather it will arise the following error:

```
! Package pgf Error:
No shape named person is known.
```

3 The package *Tikz-er2*

3.1 Usage

The *Tikz-er2* package (https://www.assembla.com/wiki/show/tikz-er2) provides a more detailed set of styles than the er library. Unfortunately, the package is not part of CTAN and thus does not come along with TEX Live or MiKTEX. The users wishing to use it have to install it by themselves.

A closer look at the package unveils its good structure: not only it provides the same styles er provides, it also has different types of attributes, entities and connections. Indeed it distinguishes among simple and total relationship with styles link and total styles respectively.

Just like er, the user is the only customizer of elements. She will do it in the same way she did with er because the *Tikz-er2* defines the same styles. These styles are in the form every *current-style*, thus customizing them could involve, again, tikzset. Positioning of elements is left to the user too. Ti*k*Z library positioning could be of help, but do not expect stumbling results: only GraphViz, introduced in section 4 can save time and some user effort in placing the nodes.

3.2 A real example with *Tikz-er2*

Tikz-er2 allows to obtain the same result already shown in figure 5. The listing 2 shows that it is not necessary to write new styles and the users can only concentrate in customizing and placing elements.

4 GraphViz

4.1 Why GraphViz?

GraphViz is a graph-deployment program. Since an ER diagram *is* graph which vertices are diagram elements (entities, relationships and attributes) and edges are links between elements, it is straightforward to think of drawing an ER diagram with Graphviz. Readers can learn the basics about GraphViz-Ti*k*Z interaction in Fiandrino (2012). In this paper we will use dot2texi (Fauske, 2008) already present in TEX Live and MiKTEX as it simplifies the needed interaction.

GraphViz makes the user ignore how to place elements because it has specific algorithms to accomplish this task. They can be activated by options like circo or neato. However, elements styles are not foreseen and the user has to design styles by herself.

The compiler will compile the main file with the option -shell-escape because it has to translate the dot language with the underlying dot2tex application. For instance, let main.tex be the main file; the command to compile it is:

```
pdflatex -shell-escape main.tex
```

Of course, the user has to check for the presence of GraphViz and dot2tex in her system before using this approach. The paper Fiandrino (2012) describes the complete installation procedure for Ubuntu.

4.2 A real example with GraphViz

Listing 3 shows how to exploit GrapViz to draw the ER diagram and, at the same time, shows the major novelties of this approach.

As already mentioned, it is necessary to build all the styles describing every necessary elements. They are in the preamble and are always defined via tikzset. Notice that the styles multi attribute and derived attribute inherit from the already defined style attribute in this case too.

The major novelty and facility introduced by GraphViz is that each element is not placed in a given position, but its description is given by choosing the category it belongs to. We make the choice with the notation style="<category>" inside square brackets:

```
\documentclass[a4paper,11pt,x11names]{article}

\usepackage{tikz-er2}
\tikzset{every entity/.style={draw=orange, fill=orange!20}}
\tikzset{every attribute/.style={draw=MediumPurple1, fill=MediumPurple1!20}}
\tikzset{every relationship/.style={draw=Chartreuse2, fill=Chartreuse2!20}}

\begin{document}
\begin{tikzpicture}[node distance=7em]
\node[entity] (person) {Person};
\node[attribute] (pid) [left of=person] {\key{ID}} edge (person);
\node[attribute] (name) [above left of=person] {Name} edge (person);
\node[multi attribute] (phone) [above of=person] {Phone} edge (person);
\node[attribute] (address) [above right of=person] {Address} edge (person);
\node[attribute] (street) [above right of=address] {Street} edge (address);
\node[attribute] (city) [right of=address] {City} edge (address);
\node[derived attribute] (age) [right of=person] {Age} edge (person);
\node[relationship] (uses) [below of=person] {Uses} edge (person);
\node[entity] (tool) [below of=uses] {Tool} edge[total] (uses);
\node[attribute] (tid) [left of=tool] {\key{ID}} edge (tool);
\node[attribute] (tname) [right of =tool] {Name} edge (tool);
\end{tikzpicture}
\end{document}
```

Listing 2: Exploiting the *Tikz-er2* package

```
Person [style="entity"];
...
Phone [style="multi attribute"];
...
Uses [style="relationship"];
```

This is the standard way to locally customize the elements in GraphViz. Since the various categories are also the styles names, the elements will automatically inherith their properties once converted in TikZ code. The label outside the square brackets is used to later connect the elements and it also appears in the diagram by default. Identifying key attributes could be a problem but GraphViz makes it possible to customize even this label with the notation `label="<label>"` (always inside square brackets). For example:

```
pid [style="attribute",
     label="\underline{ID}"];
```

In this way, the picture will use the label set with the key `label`, but for the connection phase it is the label *outside* the square brackets that matters, the *name*. Notice that the *name*s should be unique inside the dot2tex environment, therefore it is possible to exploit the key `label` also to differentiate the elements that might assume the same *name*; in the example this property has been used for:

```
Name[style="attribute"];
...
tname[style="attribute",label="Name"];
```

Elements position is decided by GraphViz and it is activated with the option neato: this automatically locates the elements near to other elements to which they are connected to.

The connections creation phase is a very simple task: the syntax is `<element1> -> <element2>` and that's all. Automatically, these connenctions inherith the provided style with:

```
edge [style="simple relation"];
```

This definition allows to declare the style globally. It will hold for all the connections unless we locally override the global definition:

```
Tool -> Uses[style="total relation"];
```

that sets up the connection to be of type *total*.

The result obtained with this approach is shown in figure 6; the compilation with the option `-shell-escape` will also create the files

```
main-dot2tex-fig1.dot
main-dot2tex-fig1.tex
```

provided the main file is named `main.tex`. The first one contains just the dot code and it can be opened with any Dot viewer while the second contains the tikzpicture code. These files could be seen as auxiliary files created in the translation process from the dot syntax to the TikZ one.

5 The object-oriented programming approach

5.1 A short introduction to object-orientation in TikZ

The concept of object-oriented programming in LaTeX graphics is still something quite new although LuaTeX seems to offer an interesting potentiality as per Giacomelli (2012).

```
\documentclass[a4paper,11pt,x11names]{article}

\usepackage{tikz}
\usetikzlibrary{automata,shapes}
\usepackage{dot2texi}
\tikzset{entity/.style={draw=orange, fill=orange!20}}
\tikzset{attribute/.style={ellipse,draw=MediumPurple1, fill=MediumPurple1!20}}
\tikzset{multi attribute/.style={attribute,double}}
\tikzset{derived attribute/.style={attribute,dashed}}
\tikzset{relationship/.style={diamond,draw=Chartreuse2, fill=Chartreuse2!20}}
\tikzset{simple relation/.style={-}}
\tikzset{total relation/.style={-,double,double distance=1.5pt}}

\begin{document}
\begin{tikzpicture}
\begin{dot2tex}[styleonly,mathmode,codeonly,neato,options=-s]
digraph G {
edge [style="simple relation"];
// nodes
Person [style="entity"];
pid [style="attribute",label="\underline{ID}"];
Attribute [style="attribute"];
Name [style="attribute"];
Phone [style="multi attribute"];
Address [style="attribute"];
Street [style="attribute"];
City [style="attribute"];
Age [style="derived attribute"];
Uses [style="relationship"];
Tool [style="entity"];
tid [style="attribute",label="\underline{ID}"];
tname [style="attribute",label="Name"];
// edges
Person -> pid;
Person -> Attribute;
Person -> Name;
Person -> Phone;
Person -> Address -> Street;
Person -> City;
Person -> Age;
Person -> Uses;
Tool -> tid;
Tool -> tname;
Tool -> Uses[style="total relation"];
}
\end{dot2tex}
\end{tikzpicture}
\end{document}
```

Listing 3: Exploiting GraphViz

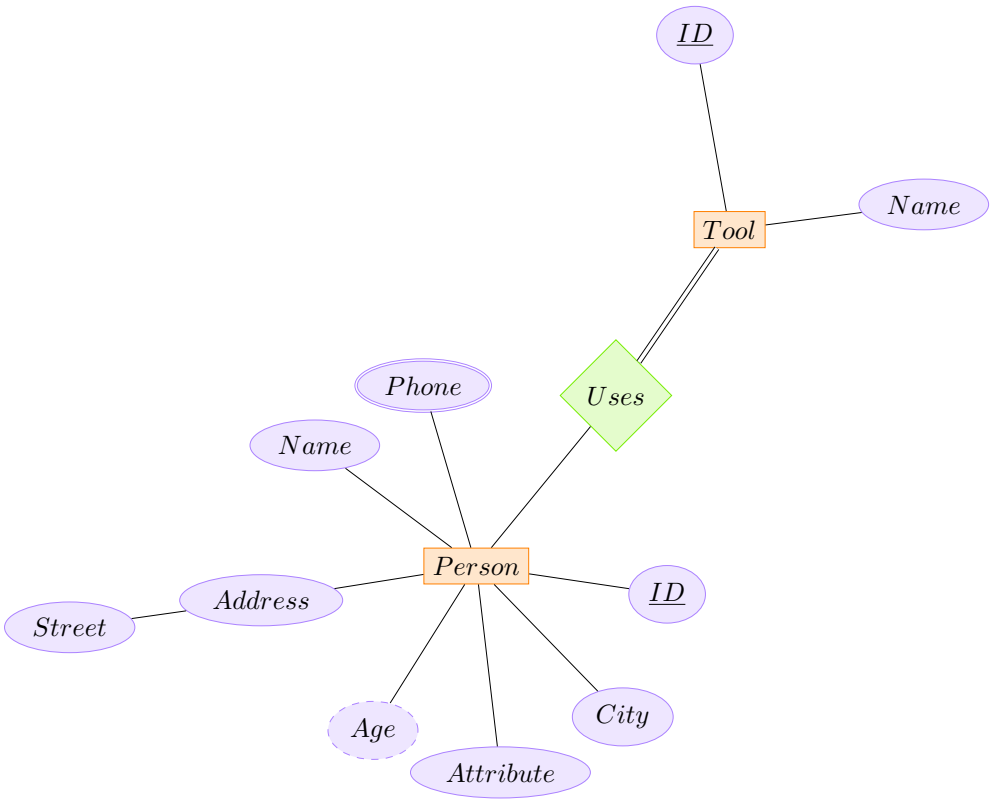

FIGURE 6: The ER diagram realized with GraphViz

On the contrary, the present work is focused on the module `oo` directly provided by TikZ. It can be loaded with

```
\usepgfmodule{oo}
```

in the preamble.

The module provides a generic macros set to build classes, methods, attributes and objects. At the moment, as far as I know, there are no libraries developed in this way. This approach merges the advantages of the object-oriented paradigm along with the TikZ syntax.

In my opinion, the object-oriented paradigm is extremely useful to draw pictures that have common features repeated several times. An ER diagram falls exactly in this category because its entities, relationships and attributes have common features: the set of rules to represent them to accomplish the standard. Moreover, these elements are repeated several times because in each diagram there are usually several entities, relationships and attributes.

From an object-oriented point of view, things with common features are called *objects*. The following command shows how to create a new object:

```
\pgfoonew \obj=new constructor()
```

where `constructor()` is a *method* of a given *class*. In particular, the *constructor* method is devoted to instantiate new objects. Each object belongs to a given class; classes are to be defined as:

```
\pgfooclass{c-name}{
  ...
  code
  ...
}
```

while methods will be defined with:

```
\method m-name(parameters) {
  ...
  code
  ...
}
```

A class is characterized by *attributes* that describe objects properties. It is only possible to customize or activate these properties with methods. Attributes can be defined with

```
\attribute a-name;
```

in case they do not have a predefined value, or with

```
\attribute a-name=value;
```

when they do have a predefined value. For example, an object which prints some text may have an attribute `text` without a predefined value while the attribute `color text` could be set to, e.g., `blue` if the text should be mainly printed in `blue`.

5.2 The `er-oo` library

To make this work significant I developed a TikZ library named `er-oo`. You can download it from

https://github.com/cfiandra/er-oo. As any other library, it will be loaded in the preamble with:

```
\usepackage{tikz}
\usetikzlibrary{er-oo}
```

after installing it. The library is ultimately a package, so the recommended way to install it is putting the files in the personal tree:

- `tikzlibraryer-oo.code.tex` and `er-oo.dtx` under `../texmf/tex/latex/er-oo/`
- `er-oo.pdf` under `../texmf/doc/er-oo/`

where `er-oo` is a directory to be created *ad hoc* and the `..` denotes the missing path which depends on the distribution and the operating system running on the machine. It also works copying `tikzlibraryer-oo.code.tex` in the directory of the main file. This solution, however, makes the library work only for that document while the former solution makes it work for all documents regardless of their position in the file system.

The library defines three classes for entities, relationships and attributes with a predefined look (at least in terms of colors), unlike the other techniques seen in this paper. The user is still free to change the basic look according to her personal preference with the apposite methods. Available methods are the same for all the classes though the class `attribute` has a further method named `set type`.

Before unveiling the code that draws the usual ER diagram, it might be useful a short introduction to the library. Specifically I will take the `attribute` class as the reference because it provides all methods.

The list of the currently defined attributes is:

```
\attribute text;
\attribute border color=er-purple;
\attribute fill color=er-purple!20;
\attribute text color=black;
\attribute label;
\attribute type;
\attribute width=1.5cm;
\attribute height=0.35cm;
```

Some attributes *have* a default value, related to the elements look; those attributes *without* a default value are those requiring an input from the user: `text` is the attribute devoted to set the label of the element inside the diagram, the `label` attribute is resposible to identify an element (by its *name*) and `type` allows to select the attribute category (standard, multi attribute or derived attribute). This set of attributes is very good since it provides a good customization potentiality, but could be improved. A new attribute could be:

```
\attribute text opacity=1;
```

to set the opacity of the text. A good idea, in defining new attributes, is to use names similar to the keys already provided by Ti*k*Z: in such a way, the attribute will set the correspondent key with the default value (1 in this case to indicate an opaque text) maintaining names consistence between the Ti*k*Z layer and the object-oriented upper layer.

The user sets the attributes value with methods. The attribute `label`, for example, has a correspondent method:

```
\method set label(#1) {
 \pgfooset{label}{#1}
}
```

Methods, however, are not only designed to set attributes. Two methods can place one element: `draw` and `place`. The first one accepts as arguments a pair of coordinates (x, y) while the second one wants as argument a position relative to another element. Here is the definition of the method `draw`:

```
\method draw(#1,#2) {
\node [ellipse,
 attribute type={\pgfoovalueof{type}},
 draw=\pgfoovalueof{border color},
 fill=\pgfoovalueof{fill color},
 text=\pgfoovalueof{text color},
 minimum width=\pgfoovalueof{width},
 minimum height=\pgfoovalueof{height},
 ] (\pgfoovalueof{label})
 at (#1,#2) {\pgfoovalueof{text}};
}
```

Notice that it is the macro `\pgfoovalueof` that sets Ti*k*Z keys inside `\node` retrieving values of the corresponding attributes. This also holds for `text` and `label` of the element.

Three alternative methods link elements: `connect`, `multi connect` and `total relation`. The first method simply draws a link between two elements exploiting the usual `\draw` syntax; the second, instead, links a single element (specified in the first argument) to a list of elements (specified in the second argument), as you may see in its definition:

```
\method multi connect(#1,#2) {
  \foreach \i in {#2}{
    \draw[-] (#1)--(\i);
  }
}
```

Notice that the usage should be in this form:

```
\myobject.multi connect(1,{2,3,4})
```

Braces are needed to protect the list of items in the second argument, since the list has to be comma separated.

Finally, the `total relation` is the method needed in case the relationship has to be of type *total*. Unfortunately it is not possible to use these methods subsequently as it happens in the traditional object-oriented languages. This means that the following syntax

```
\myobject.method one().method two()
```

and so the following code

```
\myobject.connect(1,2).total()
```

are forbidden (i.e., wrong).

To make code writing an easier and quicker process, it is possible mixing methods and creating new methods as a composition of pre-existent ones; for example, it is the case of:

```
\method set and draw(#1,#2,#3,#4) {
 \pgfoothis.set label(#1)
 \pgfoothis.text(#2)
 \pgfoothis.draw(#3,#4)
}
```

Thanks to the macro `\pgfoothis` the *contained* methods are always applied to the current object using the *container* method; for instance:

```
\myobject.set and draw(a,b,0,0)
```

is translated into

```
\myobject.set label(a)
\myobject.text(b)
\myobject.draw(0,0)
```

5.3 A real example with oo-er

As listings 1 and 2, also listing 4 provides the result already shown in picture 5.

In order to use the oo-er library, at first it is needed to instantiate the objects using the special method constructor:

```
\pgfoonew \myentity=new entity()
\pgfoonew \myrel=new relationship()
\pgfoonew \myattr=new attribute()
```

From that point on, objects are able to invoke those methods defined by each class. For example, the first entity, "tool", is placed with the method `set and draw`:

```
\myentity.set and draw(tool,Tool,1,0)
```

Since no other elements are currently present in the picture, it is not possible to place it with `set and place` as its parameters refer to an already placed element. We can therefore use `set and place` for "tool" attributes:

```
\myattr.set and place(tool-id,
  \underline{ID},left of=tool)
\myattr.set and place(tool-name,
  Name,right of=tool)
```

Indeed, once the "tool" entity has been placed, it is possible to refer to it via its *name* `tool`.

After the attributes definition, there is the connection phase:

```
\myentity.multi connect(tool,
  {tool-id,tool-name})
```

because the recommended way to proceed is to define and immediately connect the entity with its own attributes.

Notice that it seems possible to optimize the `multi connect` or `connect` methods in order to use only one argument. Assuming

```
\method x-multi connect(#1) {
 \foreach \i in {#1}{
   \draw[-]
     (\pgfoovalueof{label})--(\i);
 }
}
```

the previous connection may be obtained with

```
\myentity.x-multi connect(
  tool-id,tool-name)
```

since the method refers to the last placed object. However, in this way, it becomes mandatory to connect attributes and entities as soon as they are located because

```
\myentity.set and draw(x,x,0,0)
\myattribute.set and draw(x1,x1,1,0)
\myattribute.set and draw(x2,x2,0,1)
\myentity.set and draw(y,y,0,0)
\myentity.x-multi connect(x1,x2)
```

will not connect the entity "x" to the attributes "x1" and "x2", but rather "y" to "x1" and "x2". This does not happens with the current definition of `multi connect`:

```
\myentity.set and draw(x,x,0,0)
\myattribute.set and draw(x1,x1,1,0)
\myattribute.set and draw(x2,x2,0,1)
\myentity.set and draw(y,y,0,0)
\myentity.multi connect(x,{x1,x2})
```

correctly connects the entity "x" to the attributes "x1" and "x2".

6 Conclusion

The paper presented several techniques to draw ER diagrams with TikZ. These techniques mainly differ in terms of the programming style: the usual TikZ syntax is the base of the er library and the Tikz-er2 package, the dot language is required to exploit GraphViz and the object-oriented programming style is the key feature of the er-oo.

A user may prefer one tool or another according to her personal preferences or programming style, but she could wisely pick the tool according to the ER diagram size. Indeed, for large ER diagrams GraphViz is recommended because of the dot capability to automatically place elements.

7 Acknowledgements

I would like at first to thank Paulo Massa Cereda: this paper originated from an answer I gave on TeX.SX after a talk in chat with him.

```
\documentclass{article}

\usepackage{tikz}
\usetikzlibrary{er-oo}

\begin{document}
\begin{tikzpicture}[node distance=2.75cm]
\pgfoonew \myentity=new entity()
\pgfoonew \myrel=new relationship()
\pgfoonew \myattr=new attribute()
\myentity.set and draw(tool,Tool,1,0)
\myattr.set and place(tool-id,\underline{ID},left of=tool)
\myattr.set and place(tool-name,Name,right of=tool)
\myentity.multi connect(tool,{tool-id,tool-name})
\myrel.set and place(rel,Uses,above of=tool)
\myrel.total relation(rel,tool)
\myentity.set and place(per,Person,above of=rel)
\myattr.set and place(per-id,\underline{ID},left of=per)
\myattr.set type(derived attribute)
\myattr.set and place(per-age,Age,right of=per)
\myattr.set type() % to reset the derived attribute style
\myattr.set and place(per-name,Name,above left of=per)
\myattr.set type(multi attribute)
\myattr.set and place(per-phone,Phone,above of=per)
\myattr.set type() % to reset the multi attribute style
\myattr.set and place(per-addr,Address,above right of=per)
\myattr.set and place(street,Street,above right of=per-addr)
\myattr.set and place(city,City,right of=per-addr)
\myattr.multi connect(per-addr,{street,city})
\myentity.multi connect(per,{per-id,per-age,per-name,per-phone,per-addr,rel})
\end{tikzpicture}
\end{document}
```

Listing 4: Exploiting the object-oriented library `er-oo`

Last but not least, I would also like to thank the reviewer of the paper for his precious help in making readable my bad english.

References

Chen, P. (1976). «The entity-relationship model—toward a unified view of data». *ACM Transactions on Database Systems (TODS)*, **1**(1), pp. 9–36.

Fauske, K. M. (2008). *The dot2texi package*. URL `http://www.ctan.org/pkg/dot2texi`.

Fiandrino, C. (2012). «Graphviz e Ti*k*Z». *ArsTEXnica*, (13), pp. 4–10. URL `http://www.guit.sssup.it/arstexnica/`.

Giacomelli, R. (2012). «Grafica ad oggetti con LuaTEX». *ArsTEXnica*, (14), pp. 53–71. URL `http://www.guit.sssup.it/arstexnica/`.

Tantau, T. (2010). *The Ti*k*Z and PGF Packages*. URL `http://www.ctan.org/pkg/pgf`.

▷ Claudio Fiandrino
`claudio dot fiandrino at gmail dot com`