

## Creating a Publication-Quality Graph Embedded in Another Graph

Warren F. Kuhfeld, SAS Institute Inc., Cary, NC  
John King, Ouachita Clinical Data Services, Inc.

### ABSTRACT

Several recent papers in the *New England Journal of Medicine (NEJM)* display graphs inside of graphs (Marso et al. 2016b, a; Zinman et al. 2015; Holman et al. 2017). Typically, both graphs are Kaplan-Meier plots. The larger graph has a Y axis that ranges from 0 to 100. The failure curves occupy a relatively small part of the graph and extend perhaps no more than 10%–20% of the way up the Y axis. Inside this graph, in the open white space, you can place another graph in which the Y axis extends from 0 to slightly above the maximum value. The outer graph shows the failure probability in the full 0%–100% context, whereas the inner graph zooms in to the results and provides additional text and annotation, such as curve labels, the hazard ratio, and  $p$ -values. The outer graph also displays the number of subjects at risk. The format for the *NEJM* is quite specific. This paper illustrates how to use ODS Graphics to create a graph inside a graph that is suitable for publication in the *NEJM*.

### INTRODUCTION

Figure 1 shows a basic Kaplan-Meier failure plot. Figure 2, Figure 3, and Figure 6 show successive refinements to the basic graph. This paper shows you how to put a graph inside a graph, customize the axes, and customize the axis table. You can use SG annotation to display additional text in the graph. You will learn how to do the following:

- determine which templates a procedure uses
- delete old (possibly modified) templates before modifying the templates that SAS provides
- write the templates that SAS provides to a file
- use PROC LIFETEST to create a failure plot
- read templates in a DATA step and use the TRANWRD function to modify them
- submit modified templates by using the CALL EXECUTE statement
- create a small graph to use as an inset in another graph
- modify a graph template so that it displays the inset graph
- modify template statements that span one or multiple lines
- create a graph that has an inset graph
- convert the Y axis from proportions to percentages
- change the range of the Y-axis variable
- omit legends, titles, and borders
- create a style that suppresses the graph frame
- change axis labels
- change offsets to make the origin (0,0)
- modify a template so that it can be used along with an SG annotation data set
- create ODS output data sets
- create a single SG annotation data set from two ODS output data sets
- display the table of number of subjects at risk by using SG annotation and add a title above the row labels
- use SG annotation to add curve labels and display a  $p$ -value
- capture dynamic variables and save them in a SAS data set
- use PROC SGRENDER to create a graph from an ODS output data set, captured dynamic variables, and an SG annotation data set

The first examples draw heavily on the techniques for template modification and creating highly customized graphs that are discussed by Kuhfeld (2015). The last example shows how to do the same thing without template modification.

## PRELIMINARY EXAMPLES

This section has some contrived examples designed to introduce concepts that are needed in later sections. These concepts are introduced in a simple (albeit unrealistic) context to make them easier to understand when they are subsequently used in more complex (and realistic) examples. Imagine that you want to print the **Sashelp.Class** data set and compute means. You would never want to do anything beyond the obvious:

```
proc print data=sashelp.class; run;

proc means data=sashelp.class; run;
```

However, you can do this in a different way. You could read the code in a SAS DATA step and use a SAS CALL routine to execute the code:

```
data _null_;
  input;
  call execute(_infile_);
  datalines4;
proc print data=sashelp.class; run;

proc means data=sashelp.class; run;
;;;
```

You can use the DATA \_NULL\_ step when you want to do DATA step processing without creating a SAS data set. The INPUT statement does not list any variables. When you are processing text files, it is often convenient to directly use the input buffer, **\_infile\_**, which exists whenever you use an INPUT statement. **\_infile\_** is an automatic variable. Like **\_N\_**, variables named in the POINT= option in SET statements, variables named in the IN= data set option, and other automatic variables, it is never written to a SAS data set.

The CALL EXECUTE statement submits to a buffer all the code that is read in the DATA step. The DATALINES4 statement indicates the beginning of instream data that might contain semicolons. The instream data contain the SAS code and end with four semicolons. The CALL EXECUTE statement is called three times, once for each line of instream data. When the DATA step finishes, SAS runs the code in the buffer and generates the listing and means. The CALL EXECUTE statement is handy when a data set contains information about a program that you want to write. In this case, it contains exactly the program that you want to write. In a real example, you might need to change the input before submitting it to SAS. The following step uses the same instream input data, but it changes the data set name to **Sashelp.Heart** and adds an option to print only the first 10 observations of the **Sashelp.Heart** data set:

```
data _null_;
  input;
  _infile_ = tranwrd(_infile_, 'class',
                    ifc(find(_infile_, 'print'), 'heart (obs=10)', 'heart'));
  call execute(_infile_);
  datalines4;
proc print data=sashelp.class; run;

proc means data=sashelp.class; run;
;;;
```

This DATA step contains a new assignment statement. This statement translates the PROC PRINT step into

```
proc print data=sashelp.heart(obs=10); run;
```

The PROC MEANS step is translated into

```
proc means data=sashelp.heart; run;
```

The TRANWRD function does the translation. For the PROC MEANS step, the translation is equivalent to

```
_infile_ = tranwrd(_infile_, 'class', 'heart');
```

The actual TRANWRD function is more complicated because of the other functions. The FIND function returns a true value when 'print' is found in **\_infile\_**. When the FIND function returns a true value, the IFC function returns the first argument; otherwise it returns the second argument. Ultimately, the TRANWRD function translates 'class'

to `'heart (obs=10) '` for PROC PRINT, translates `'class'` to `'heart'` for PROC MEANS, and leaves `_infile_` unchanged for the blank line.

In more realistic examples, you read the input from either an external file or a SAS data set, not from instream data. You use a DATA step to generate code. The code might exactly match the input, or it might be translated from the input by using the TRANWRD function or another method.

The TRANWRD function seamlessly replaces longer strings with shorter strings and replaces shorter strings with longer strings. But whenever you add text to a character variable, you need to be aware of the length of that variable and be certain that you do not generate a string that is too long and is truncated. The following steps show how to check the length and also illustrate techniques that are used in subsequent examples. This example begins by writing a graph template to a file named `tpl.tpl`:

```
proc template;
  source Stat.Lifetest.Graphics.ProductLimitFailure / file='tpl.tpl';
quit;
```

You can view the file by opening it in an editor, or you can read it by using a SAS DATA step. The following step reads the file, finds the length of the longest record, and determines the maximum length of the variable `_infile_`:

```
data _null_;
  infile 'tpl.tpl' end=eof;
  length str $ 32767;
  input;
  retain i maxl 0;
  maxl = max(maxl, length(_infile_));
  if _n_ = 1 then do i = 1 to 32767 until(str ne _infile_);
    str      = repeat('x', i - 1);
    _infile_ = repeat('x', i - 1);
  end;
  if eof then put i= maxl=;
run;
```

The INFILE statement names the file to read. The END=EOF option sets a flag to true when the DATA step processes the last record and sets it to false otherwise. The name **EOF** (for end of file) is arbitrary. When the variable is true, the PUT statement at the end of the DATA step is executed. **EOF** is an automatic variable and is never written to a SAS data set. The statement `maxl = max(maxl, length(_infile_))` finds the length of the longest record in the file. The results (which are displayed by the PUT statement) show that the longest record is 250 bytes. In fact, the maximum length possible is 256. Because `_infile_` is an automatic variable, you cannot save it to a SAS data set and use the CONTENTS procedure to find its length. Instead, a DO loop sets its value and sets the variable **Str** to the same value; it quits if the value of `_infile_` is truncated and no longer matches the longer variable. This step shows that the length of `_infile_` is 32,767, which is the maximum for a SAS character variable. No realistic translation of a 256-character input line will exceed the maximum length of 32,767.

## EXAMPLES

The following step creates simulated data:

```
data simdata(drop=n);
  Treatment = 'Drug  ';
  do until(n = 500);
    Month = -100 * log(uniform(368));
    status = not (uniform(368) > 0.10 or (Month > 15 and uniform(368) > 0.2));
    if Month < 18 then do; output; n + 1; end;
  end;
  n = 0;
  Treatment = 'Placebo';
  do until(n = 500);
    Month = -150 * log(uniform(368));
    status = not (uniform(368) > 0.16 or (Month > 15 and uniform(368) > 0.1));
    if Month < 18 then do; output; n + 1; end;
  end;
run;
```

The following step deletes all compiled templates that you need to modify in subsequent steps. It is good practice to delete compiled templates at the beginning to ensure that you are using the templates that SAS provides. You want to modify the templates that SAS provides, not the ones that you previously modified.

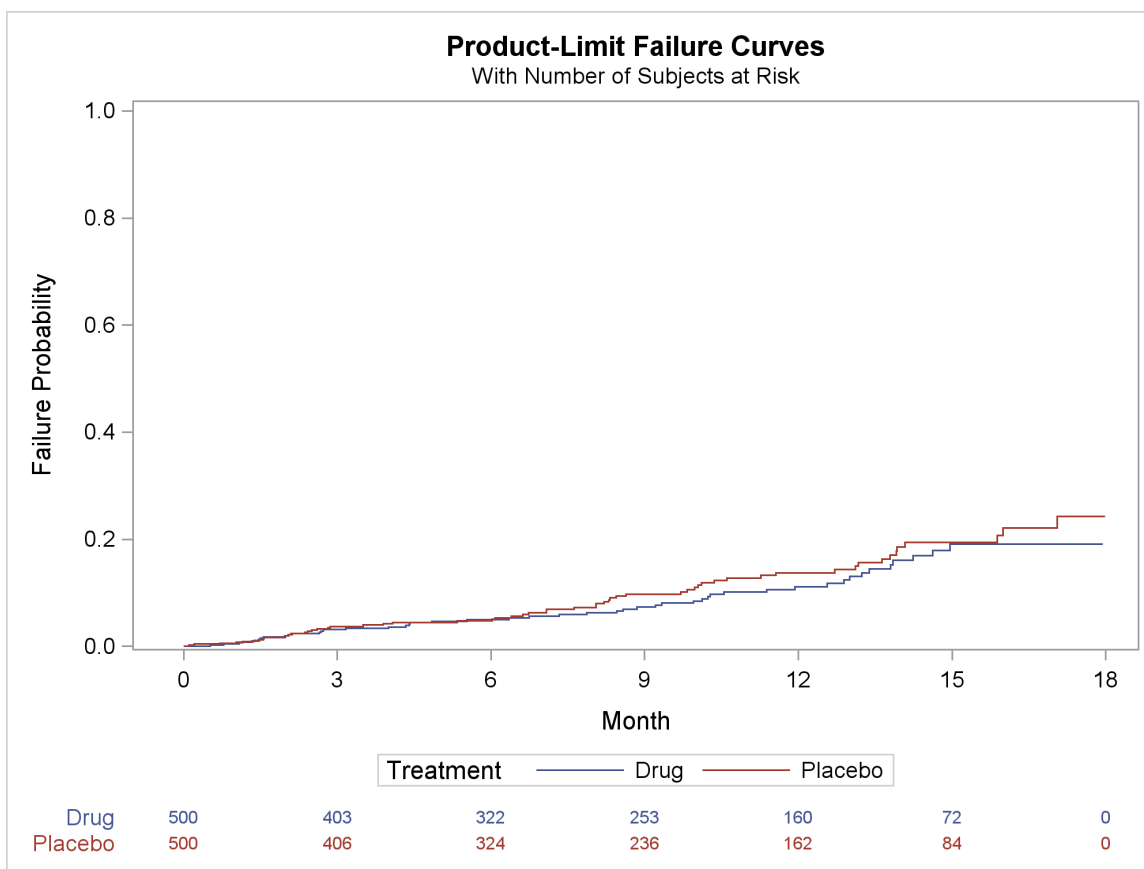
```
proc template;
  delete Stat.Lifetest.Graphics.ProductLimitFailure / store=sasuser.templat;
  delete Stat.Lifetest.Graphics.ProductLimitFailure2 / store=sasuser.templat;
quit;
```

### BASIC FAILURE PLOT

The next step uses the simulated data and creates the failure plot displayed in [Figure 1](#):

```
ods graphics on;
proc lifetest data=simdata notable
  plot=survival(failure nocensor atrisk(outside atrisktickonly)=0 to 18 by 3);
  time Month * status(0);
  strata treatment / notest;
run;
```

**Figure 1** Failure Plot



### GRAPH IN A GRAPH

The examples in this section create graphs from pairs of graphs. (See [Figure 2](#), [Figure 3](#), and [Figure 6](#).) In each pair, the first graph is smaller than the default size, and it has a restricted range for the Y axis. The second graph is the default size, and it has the full Y-axis range; this graph also displays the number of subjects at risk. The larger graph has available white space, and the first graph is inserted into that white space. Neither graph has a legend, because the names and colors in the at-risk table show how to identify each step function. You must change two templates before you can create a graph in a graph.

### Finding the Template Names

Before either graph is created, you need to figure out which templates PROC LIFETEST uses. The previous step enables ODS Graphics, and the following step enables ODS trace output. This step creates a graph without displaying the number of subjects at risk.

```
ods trace on;
proc lifetest data=simdata notable plot=survival(failure nocensor);
  time Month * status(0);
  strata treatment / notest;
run;
```

The trace output displays the name of the template and other information:

#### Output Added:

```
-----
Name:      FailurePlot
Label:     Failure Curves
Template:  Stat.Lifetest.Graphics.ProductLimitFailure
Path:     Lifetest.FailurePlot
-----
```

ODS Graphics and ODS trace output are enabled for the rest of this paper. The following step displays the failure plot and the number of subjects at risk:

```
proc lifetest data=simdata notable
  plot=survival(failure nocensor atrisk(outside atrisktickonly)=0 to 18 by 3);
  time Month * status(0);
  strata treatment / notest;
run;
```

The trace output shows that a different template is used this time:

#### Output Added:

```
-----
Name:      FailurePlot
Label:     Failure Curves
Template:  Stat.Lifetest.Graphics.ProductLimitFailure2
Path:     Lifetest.FailurePlot
-----
```

You can see that PROC LIFETEST uses ProductLimitFailure2 and ProductLimitFailure for when the at-risk table is and is not displayed, respectively.

### Storing the Templates in Files

The following steps delete any modified templates and then write the two templates to files:

```
proc template;
  delete Stat.Lifetest.Graphics.ProductLimitFailure / store=sasuser.templat;
  delete Stat.Lifetest.Graphics.ProductLimitFailure2 / store=sasuser.templat;
  source Stat.Lifetest.Graphics.ProductLimitFailure / file='tpl.tpl';
  source Stat.Lifetest.Graphics.ProductLimitFailure2 / file='tpl2.tpl';
quit;
```

Again, the code deletes (potentially modified) templates before they are modified; the deletion is unnecessary. But this style of defensive programming might save you some time and head scratching when you are repeatedly modifying graph templates, by ensuring that you are starting with the templates that SAS provides.

## First Approximation

This section shows how to put the graph inside a graph. Most refinements are deferred until the next two sections. The file *tpl.tpl* contains the template that you need to modify first. It is the template that you will use to make the inset graph (which does not display the number of subjects at risk). It is too long to display here. To see it, you can run PROC TEMPLATE and the SOURCE statement. **You need to look at it before writing code such as that in the next step.** The next step uses a DATA step to modify the graph template. You could use an editor, but then you lose the chain of reproducibility in your program. The following program modifies the graph template:

```
* Modify template for inset graph;
data _null_;
  infile 'tpl.tpl' end=eof;
  input;
  if _n_ = 1 then call execute('proc template;');

  * Skip the legend by using a nonexistent name;
  _infile_ = tranwrd(_infile_, 'DiscreteLegend "Failure"', 'DiscreteLegend "skipit"');

  * Change the Y-axis tick value list, specify an ad hoc range;
  _infile_ = tranwrd(_infile_, 'tickvaluelist=(0 .2 .4 .6 .8 1.0)',
                    'tickvaluelist=(0 .1 .2 .3)');
  * Make corresponding change to the Y-axis maximum;
  _infile_ = tranwrd(_infile_, 'viewmax=1', 'viewmax=0.3');
  call execute(_infile_);
  if eof then call execute('quit;');
run;
```

The preceding step uses a DATA \_NULL\_ step to read, modify, and run PROC TEMPLATE along with the modified template. The INFILE statement specifies the input file that contains the template. The record length of this file is 256. This is long enough that most statements do not wrap to a separate line; however, the longest statements do wrap. The INPUT statement does not mention any variables. When you are processing text files, it is often convenient to process the input buffer, **\_infile\_**, which exists whenever you use an INPUT statement. The template in *tpl.tpl* does not contain a PROC TEMPLATE statement, so when **\_N\_** is 1, the DATA step writes one. You need to omit the legend, which is generated by the following statement:

```
DiscreteLegend "Failure" / title=GROUPNAME location=outside;
```

It generates a legend for a statement named "Failure". The following statement specifies NAME="Failure", so this is the statement that provides the source for the legend:

```
stepplot y=eval (1-SURVIVAL) x=TIME / group=STRATUM index=STRATUMNUM
  name="Failure" rolename=(_tip1=ATRISK _tip2=EVENT)
  tiplabel=(y="Failure Probability" _tip1="Number at Risk"
           _tip2="Observed Events") tip=(x y _tip1 _tip2);
```

You can omit the legend in at least two ways. You can delete the DISCRETELEGEND statement, but that requires you to ensure that you delete it both when the statement appears on one line and when it wraps to more than one line. An easier approach is to change the name in the DISCRETELEGEND statement to something nonexistent:

```
DiscreteLegend "skipit" / title=GROUPNAME location=outside;
```

The following statement makes that change:

```
_infile_ = tranwrd(_infile_, 'DiscreteLegend "Failure"', 'DiscreteLegend "skipit"');
```

Again, you must ensure that you are specifying the correct case and the correct translation. For example, in another part of the template, the statement name is lowercase:

```
discretelegend "Censored" / location=inside autoalign=(topleft bottomright);
```

The DATA step contains the following statement:

```
_infile_ = tranwrd(_infile_, 'tickvaluelist=(0 .2 .4 .6 .8 1.0)',
                  'tickvaluelist=(0 .1 .2 .3)');
```

It changes the option `tickvaluelist=(0 .2 .4 .6 .8 1.0)` to `tickvaluelist=(0 .1 .2 .3)`. It does this twice, because `tickvaluelist=(0 .2 .4 .6 .8 1.0)` appears twice in the template. Changing the tick value list is not sufficient. You must also change the maximum value for the Y axis. The following statement changes the maximum from 1 to 0.3:

```
_infile_ = tranwrd(_infile_, 'viewmax=1', 'viewmax=0.3');
```

The next step creates the smaller inset graph:

```
ods _all_ close;
ods html image_dpi=300;

* Inset graph scaling factor specified as a quotient;
%let s = 5 / 10;

* Enable graphics, set imagename for inset, reset graph name index,
  disable border, scale height and width;
ods graphics on / imagename="Inset" reset=index border=off noscale
                  height=%eval(480 * &s)px width=%eval(640 * &s)px;
ods listing style=htmlblue image_dpi=300;
ods html exclude all;
proc lifetest data=simdata notable plot=survival(failure nocensor);
  time Month * status(0);
  strata treatment / notest;
run;
ods html exclude none;

* Close listing destination, because you are done capturing inset graph;
ods listing close;
```

The name of the image is specified in the ODS GRAPHICS statement. This name is used in a subsequent step. The RESET=INDEX option ensures that an integer index number is not appended to the name. The first graph is stored in the file *Inset.png*; subsequent graphs are stored in *Inset2.png*, *Inset3.png*, and so on. Specifying RESET=INDEX is important when you need to completely control the image name as you run and rerun the code to develop and refine your graph. BORDER=OFF suppresses the border around the graph. The NOSCALE option ensures that fonts are not shrunk when the image is created at a size smaller than the default. The height and width are set to 50% of their default value. The macro variable `&s` contains the scaling factor. It is specified as a quotient so that it can easily be used in a %EVAL macro function and produce an integer. One key to making all this work well is to use a consistent scaling factor both when you create the inset graph and when you add it to the outer graph. The HTML destination is open (only because that is the default destination for many SAS users). This graph is excluded from that destination, it is stored in a file when you specify the LISTING destination, and the graphs are created at 300 dots per inch. The code ensures that only one destination is open so that only one version of the graph is produced. PROC LIFETEST is run and the destinations are restored.

The following step modifies the second template:

```
* Modify template for full-size graph;
data _null_;
  infile 'tpl2.tpl' end=eof;
  input;

  * Add PROC TEMPLATE statement;
  if _n_ = 1 then call execute('options nosource; proc template;');

  * Skip the legend by using a nonexistent name;
  _infile_ = tranwrd(_infile_, 'DiscreteLegend "Failure"', 'DiscreteLegend "skipit"');

  * Find beginning of LAYOUT OVERLAY statement;
  if find(_infile_, 'layout overlay / xaxisopts') then lo + 1;
```

```

* Find end of LAYOUT OVERLAY statement;
if lo and index(_infile_, ';') then do;
  * Reset layout flag;
  lo = 0;

  * Write original or modified template line;
  call execute(_infile_);

  * Add DrawImage statement after the LAYOUT OVERLAY statement;
  _infile_ = 'drawimage "Inset.png" / width=%eval(640 * &s) x=99 y=99 ' ||
    ' anchor=topright drawspace=wallpercent sizeunit=pixel;';
end;

* Write original, new, or modified template line;
call execute(_infile_);

* End PROC TEMPLATE, display source again;
if eof then call execute('quit; options source;');
run;

```

This time, the NOSOURCE and SOURCE options are specified to suppress displaying the lines that CALL EXECUTE generates. You can remove the OPTIONS statements if you want to see that code. It is good practice to look at the generated source code when you are developing the code. This is the template that is used when the graph contains an axis table of the number of subjects at risk. Some of this template modification is the same as in the previous template modification program. Here is some of the code from the second template:

```

define statgraph Stat.Lifetest.Graphics.ProductLimitFailure2;
  ...
  if (NSTRATA=1)
    ...
  else
    ...
    layout lattice / rows=2 rowweights=ROWWEIGHTS columndatarange=union rowgutter=10;
    cell;
      layout overlay / xaxisopts=(shortlabel=XNAME ...
        tickvaluelist=(0 .2 .4 .6 .8 1.0));
      ...
      DiscreteLegend "Failure" / title=GROUPNAME location=outside;
      ...
    endlayout;
  endcell;
  cell;
    layout overlay / walldisplay=none xaxisopts=(display=none);
    axistable x=TATRISK value=ATRISK / ...;
    endlayout;
  endcell;
  endlayout;
endif;
...
end;

```

The template has two parts. The first part is used when there is a single stratum, and the second part is used when there are multiple strata. Because you have multiple strata, you need to be concerned only about the second part. Although it is not shown here, much of the first part is similar to the second part. It is acceptable if your template modifications change the first part, as long as the changes are syntactically valid. The following DATA step statement searches the template for the beginning of the main LAYOUT OVERLAY statement:

```

if find(_infile_, 'layout overlay / xaxisopts') then lo + 1;

```

When the DATA step finds that line, it sets the `lo` variable (layout flag) to 1 and retains it by using the sum statement: `lo + 1`. When the flag is 1 (true) and the line contains a semicolon, then you are looking at the end of the LAYOUT OVERLAY statement, which might (and in this case does) span more than one line:



```
if lo and index(_infile_, ';') then do;
```

When the end of the LAYOUT OVERLAY statement is found, the flag is set back to false, the current line (the last part of the LAYOUT OVERLAY statement) is submitted to the buffer, and a new statement is submitted to the buffer:

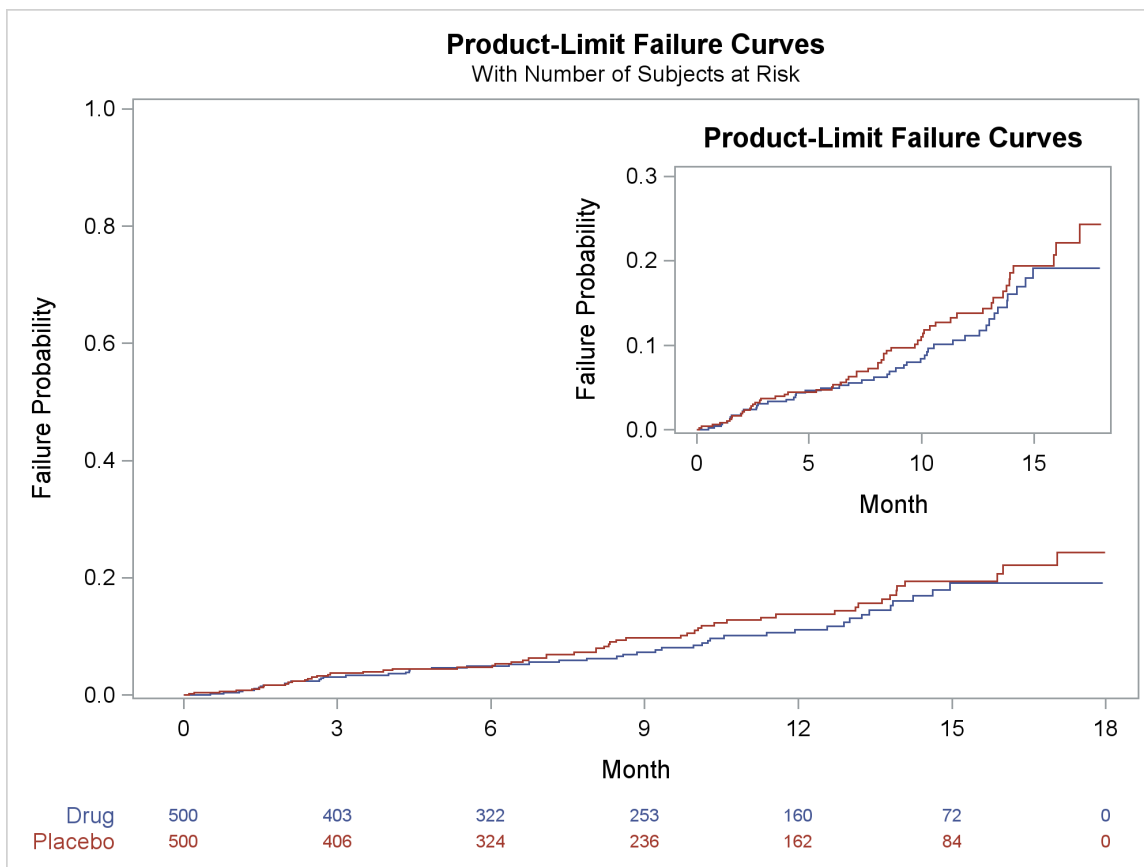
```
if lo and index(_infile_, ';') then do;
  lo = 0;
  call execute(_infile_);
  _infile_ = 'drawimage "Inset.png" / width=%eval(640 * &s) x=99 y=99 ' ||
    'anchor=topright drawspace=wallpercent sizeunit=pixel;';
end;
```

The new statement is a DRAWIMAGE statement. It provides the same capabilities as SG annotation in PROC SGPLOT, but it uses a different syntax. This statement inserts the image *Inset.png*, which you made in the previous PROC LIFETEST step. The width of the inset graph is based on the scaling factor stored in the macro variable **&s** and is specified in pixels. The graph is located in the top right portion of the graph. The wall is the area inside the axes. The coordinates are based on the percentage of the wall and are 99% of the way across the X axis and 99% of the way up the Y axis. The top right corner of the graph is placed at these two coordinates. The following step runs PROC LIFETEST and uses the modified template:

```
* Capture final graph at original size;
ods graphics on / reset=imagename reset=border height=480px width=640px;
proc lifetest data=simdata notable
  plot=survival(failure nocensor atrisk(outside atrisktickonly)=0 to 18 by 3);
  time Month * status(0);
  strata treatment / notest;
run;
```

The graph is displayed in [Figure 2](#).

**Figure 2** Graph in a Graph



## Second Approximation

Now that you know how to generate both graphs and position one inside the other, you can start to refine them. You do not need to start by deleting the templates and writing them to files. Those files still exist from the section “[Storing the Templates in Files](#)” on page 5. You can still use them as the starting point for your template modifications. You might want to convert the Y axis from proportions to percentages. You might want the minimum offsets to be zero. That is, you might want the origin to be precisely at (0,0). The first graph does not need a title, axis labels, or a full frame or box around it. The next steps eliminate those components and create the graph shown in the top right corner of [Figure 3](#):

```
* Modify template for inset graph;
data _null_;
  infile 'tpl.tpl' end=eof;
  input;

  * Add PROC TEMPLATE statement;
  if _n_ = 1 then call execute('options nosource; proc template;');

  * Skip the legend by using a nonexistent name;
  _infile_ = tranwrd(_infile_, 'DiscreteLegend "Failure"', 'DiscreteLegend "skipit"');

  * Convert Y axis to percentages;
  _infile_ = tranwrd(_infile_, '(1-SURVIVAL)', '(100-100*SURVIVAL)');

  * Change the Y-axis tick value list, specify an ad hoc range;
  _infile_ = tranwrd(_infile_, 'tickvaluelist=(0 .2 .4 .6 .8 1.0)',
                    'tickvaluelist=(0 10 20 30)');

  * Make corresponding change to the Y-axis maximum;
  _infile_ = tranwrd(_infile_, 'viewmax=1', 'viewmax=30');

  * Add a bogus undefined dynamic variable;
  _infile_ = tranwrd(_infile_, 'dynamic', 'dynamic notdefined');

  * Remove offset, make the Y axis start precisely at zero;
  _infile_ = tranwrd(_infile_, 'yaxisopts=(', 'yaxisopts=(offsetmin=0 offsetmax=0 ');

  * Remove offset, make the X axis start precisely at zero;
  _infile_ = tranwrd(_infile_, 'offsetmin=.05', 'offsetmin=0 offsetmax=0');

  * Drop all titles by specifying undefined dynamic;
  if index(_infile_, 'entrytitle') then _infile_ = 'entrytitle notdefined;';

  * Drop the label from both the X and Y axes;
  _infile_ = tranwrd(_infile_, 'axisopts=(',
                    'axisopts=(display=(line ticks tickvalues) ');

  * Remove the border;
  _infile_ = tranwrd(_infile_, 'BeginGraph;', 'BeginGraph / border=false;');

  * Write original or modified template line;
  call execute(_infile_);

  * End PROC TEMPLATE, display source again;
  if eof then call execute('quit; options source;');
run;

* Create a style that does not have a border;
proc template;
  define style noframe;
    parent=styles.htmlblue;
    style graphwalls from graphwalls / frameborder=off;
  end;
run;
```

```

* Inset graph scaling factor specified as a quotient;
%let s = 6 / 10;

* Enable graphics, set imagename for inset, reset graph name index,
  disable border, scale height and width;
ods graphics on / imagename="Inset2" reset=index border=off noscale
                  height=%eval(480 * &s)px width=%eval(640 * &s)px;

* Open only LISTING destination, for capture, and use new style;
ods _all_ close;
ods listing style=noframe image_dpi=300;
* Create inset graph;
proc lifetest data=simdata notable plot=survival(failure nocensor);
  time Month * status(0);
  strata treatment / notest;
run;

* Close listing destination, because you are done capturing inset graph;
ods listing close;

```

For the outer graph, you might want to change both axis labels. The following steps create the graph displayed in [Figure 3](#):

```

* Modify template for full-size graph;
data _null_;
  infile 'tpl2.tpl' end=eof;
  input;

  * Add PROC TEMPLATE statement;
  if _n_ = 1 then call execute('options nosource; proc template;');

  * Skip the legend by using a nonexistent name;
  _infile_ = tranwrd(_infile_, 'DiscreteLegend "Failure"', 'DiscreteLegend "skipit"');

  * Add a bogus undefined dynamic variable;
  _infile_ = tranwrd(_infile_, 'dynamic', 'dynamic notdefined');

  * Drop all titles by specifying undefined dynamic;
  if index(_infile_, 'entrytitle') then _infile_ = 'entrytitle notdefined;';

  * Convert Y axis to percentages;
  _infile_ = tranwrd(_infile_, '(1-SURVIVAL)', '(100-100*SURVIVAL)');
  _infile_ = tranwrd(_infile_, 'tickvaluelist=(0 .2 .4 .6 .8 1.0)',
                    'tickvaluelist=(0 20 40 60 80 100)');
  _infile_ = tranwrd(_infile_, 'viewmax=1', 'viewmax=100');

  * Remove the border;
  _infile_ = tranwrd(_infile_, 'BeginGraph;', 'BeginGraph / border=false;');

  * Change Y-axis label;
  _infile_ = tranwrd(_infile_, 'label="Failure Probability"',
                    'label="Patients with an Event (%)"');

  * Change X-axis label;
  _infile_ = tranwrd(_infile_, 'xaxisopts=(',
                    'xaxisopts=(label="Months since Randomization" ');

  * Find beginning of LAYOUT OVERLAY statement;
  if find(_infile_, 'layout overlay / xaxisopts') then lo + 1;

  * Find end of LAYOUT OVERLAY statement;
  if lo and index(_infile_, ';') then do;

```

```

* Reset layout flag;
lo = 0;

* Write original or modified template line;
call execute(_infile_);

* Add DrawImage statement after the LAYOUT OVERLAY statement;
_infile_ = 'drawimage "Inset2.png" / width=%eval(640 * &s) x=102 y=105 ' ||
           'anchor=topright drawspace=wallpercent border=false sizeunit=pixel;';
end;

* Write original, new, or modified template line;
call execute(_infile_);

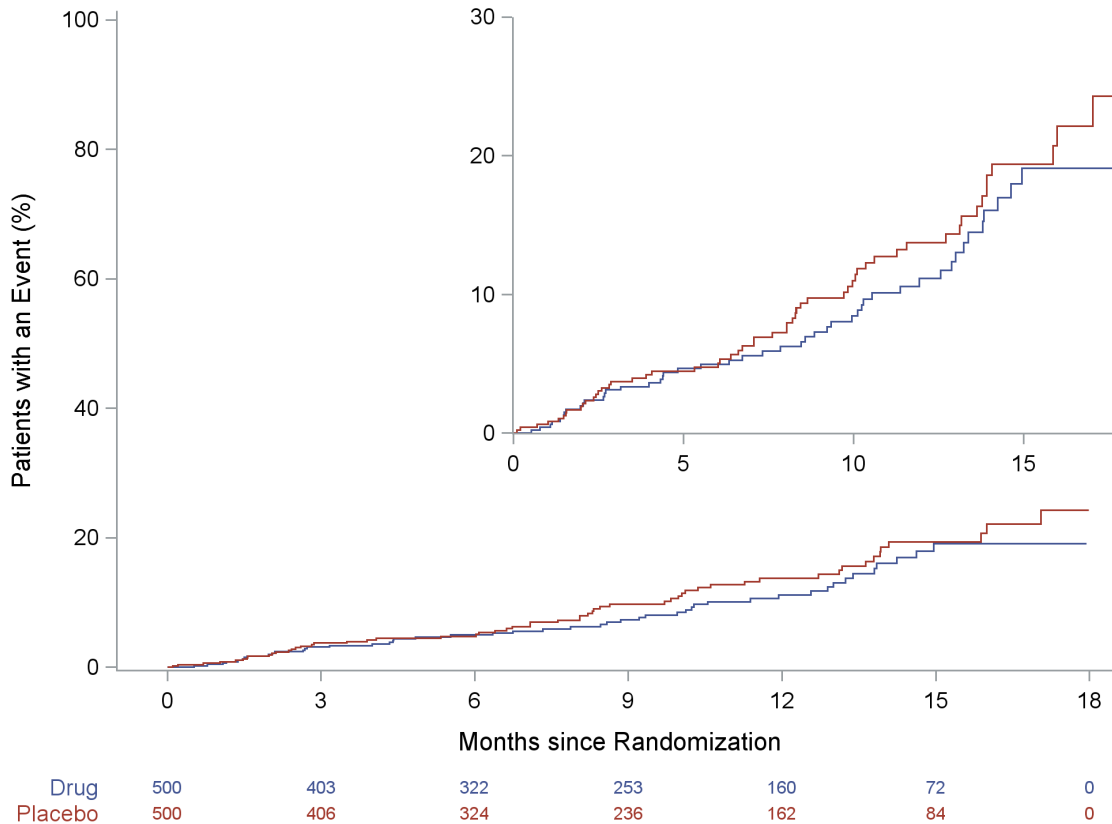
* End PROC TEMPLATE, display source again;
if eof then call execute('quit; options source;');
run;

* Enable HTML capture of final graph at original size;
ods html image_dpi=300 style=noframe;
ods graphics on / reset=imagename reset=border height=480px width=640px;

* Create graph using modified template and inset image;
proc lifetest data=simdata notable
  plot=survival(failure nocensor atrisk(outside atrisktickonly)=0 to 18 by 3);
  time Month * status(0);
  strata treatment / notest;
run;

```

**Figure 3** First Set of Refinements



Notice in [Figure 3](#) that you have not yet set the origin of the outer graph to (0,0).

## Final Customized Results

For the final graph, you might want the minimum offset to be zero for both the outer and inset graphs. You cannot use the axis table if you do this. There is currently no option that enables the axis table to flow into the left area below the Y-axis ticks and label. However, you can display the axis table by using SG annotation. You can also use SG annotation to display other information, such as a header over the axis table (“No. at Risk”) and curve labels. Next, you need to modify the template for the inset graph. This step is the same as the step in the section “[Second Approximation](#)” on page 10.

The following steps create the inset graph:

```
* Inset graph scaling factor specified as a quotient;
%let s = 57 / 100;

* Enable graphics, set imagename for inset, reset graph name index,
  disable border, scale height and width;
ods graphics on / imagename="Inset3" reset=index border=off noscale
                  height=%eval(480 * &s)px width=%eval(640 * &s)px;

* Open only LISTING destination, for capture, and use new style;
ods _all_ close;
ods listing style=noframe image_dpi=300;
* Create inset graph;
proc lifetest data=simdata notable plot=survival(failure nocensor);
  time Month * status(0);
  strata treatment / notest;
run;

* Close listing destination, because you are done capturing inset graph;
ods listing close;
```

The next step modifies the graph template for the outer graph:

```
* Modify template for full-size graph;
data _null_;
  infile 'tpl.tpl' end=eof;
  input;

  * Add PROC TEMPLATE statement;
  if _n_ = 1 then call execute('options nosource; proc template;');

  * Skip the legend by using a nonexistent name;
  _infile_ = tranwrd(_infile_, 'DiscreteLegend "Failure"', 'DiscreteLegend "skipit"');

  * Add a bogus undefined dynamic variable;
  _infile_ = tranwrd(_infile_, 'dynamic', 'dynamic notdefined');

  * Remove offset, make the Y axis start precisely at zero;
  _infile_ = tranwrd(_infile_, 'yaxisopts=(', 'yaxisopts=(offsetmin=0 offsetmax=0 ');

  * Remove offset, make the X axis start precisely at zero;
  _infile_ = tranwrd(_infile_, 'offsetmin=.05', 'offsetmin=0 offsetmax=0');

  * Drop all titles by specifying undefined dynamic;
  if index(_infile_, 'entrytitle') then _infile_ = 'entrytitle notdefined;';

  * Convert Y axis to percentages;
  _infile_ = tranwrd(_infile_, '(1-SURVIVAL)', '(100-100*SURVIVAL)');
  _infile_ = tranwrd(_infile_, 'tickvaluelist=(0 .2 .4 .6 .8 1.0)',
                    'tickvaluelist=(0 20 40 60 80 100)');
  _infile_ = tranwrd(_infile_, 'viewmax=1', 'viewmax=100');

  * Remove the border;
  _infile_ = tranwrd(_infile_, 'BeginGraph;', 'BeginGraph / border=false;');
```

```

* Remove the at-risk table, change Boolean expression to unconditional false;
_infile_ = tranwrd(_infile_, 'PLOTATRISK=1', '0');

* Change Y-axis label;
_infile_ = tranwrd(_infile_, 'label="Failure Probability"',
                  'label="Patients with an Event (%)")');

* Change X-axis label;
_infile_ = tranwrd(_infile_, 'xaxisopts=(',
                  'xaxisopts=(label="Months since Randomization" ');

* Find beginning of LAYOUT OVERLAY statement;
if find(_infile_, 'layout overlay / xaxisopts') then lo + 1;

* Find end of LAYOUT OVERLAY statement;
if lo and index(_infile_, ';') then do;

    * Reset layout flag;
    lo = 0;
    * Add padding;
    _infile_ = tranwrd(_infile_, ';', 'pad=(bottom=10% left=5%)');

    * Write original or modified template line;
    call execute(_infile_);

    * Add DrawImage statement after the LAYOUT OVERLAY statement;
    _infile_ = 'drawimage "Inset3.png" / width=%eval(640 * &s) x=102 y=106 ' ||
              'anchor=topright drawspace=wallpercent sizeunit=pixel border=false;' ||
    %* Add ANNOTATE statement to LAYOUT OVERLAY and give it an ID;
              'annotate / id="id";';

end;

* Write original, new, or modified template line;
call execute(_infile_);

* End PROC TEMPLATE, display source again;
if eof then call execute('quit; options source;');
run;

```

Unlike previous examples, here you modify the same template for both graphs. This is because you are not relying on PROC LIFETEST to make the at-risk table. Instead, you make it by using SG annotation. The previous step also adds an ANNOTATE statement to the template. Without it, your SG annotations are ignored. The next step also deviates from the previous example in several ways. This time, PROC LIFETEST does not directly make the final graph. Rather, it creates the underlying data object and stores it in an ODS output data set. The ODS document captures the dynamic variables. You can use the output data object to provide the data set for the graph and to provide the input for the SG annotation data set. The following step runs PROC LIFETEST:

```

* Create data object for graph (failureplot=fp),
  data set that contains the p-value (homtests=p),
  and an ODS document, from which you will get the dynamic variables;
ods document name=MyDoc (write);
ods graphics on / reset=imagename reset=border height=480px width=640px;
proc lifetest data=simdata notable
  plot=survival(failure nocensor atrisk(outside atrisktickonly)=0 to 18 by 3);
  time Month * status(0);
  strata treatment;
  ods output failureplot=fp homtests=p;
run;
ods document close;

```

This step lists the contents of the ODS document, which are displayed in [Figure 4](#):

```
* List document contents, need to copy the path from the output;
proc document name=MyDoc;
  list / levels=all;
quit;
```

Figure 4 ODS Document Contents

Obs	Path	Type
1	\Lifetest#1	Dir
2	\Lifetest#1\CensoredSummary#1	Table
3	\Lifetest#1\StrataHomogeneity#1	Dir
4	\Lifetest#1\StrataHomogeneity#1\HomogeneityNote#1	Note
5	\Lifetest#1\StrataHomogeneity#1\HomStats#1	Table
6	\Lifetest#1\StrataHomogeneity#1\LogrankHomCov#1	Table
7	\Lifetest#1\StrataHomogeneity#1\WilcoxonHomCov#1	Table
8	\Lifetest#1\StrataHomogeneity#1\HomTests#1	Table
9	\Lifetest#1\FailurePlot#1	Graph

You must specify the failure plot path in the OBDYNAM (dynamic object) statement:

```
* Using failure plot path from the ODS document, output dynamic variables;
proc document name=MyDoc;
  ods exclude dynamics;
  ods output dynamics=dynamics;
  obdynam \Lifetest#1\FailurePlot#1;
quit;
```

This creates a SAS data set, **Dynamics**, that contains the dynamic variables. These are needed if you want to reproduce the graph by using PROC SGRENDER. The following step creates annotations:

```
* SG annotation data set;
data anno(drop=atrisk tatrisk stratum stratumnum left probchisq);
  retain Function 'Text' /* All annotations add text */
  X1Space 'DataValue' /* Most X coordinates are data values */
  Y1Space 'GraphPercent' /* Most Y coordinates are graph percentages */
  ID 'id' /* Constant ID. Matches ANNOTATE stmt ID. */
  Width 100 /* Width=100% ensures no splitting */
  left -3; /* Extrapolated data value for label column */
/* of axis table */

* Extract axis table from graph data object;
set fp(drop=_: censored event survival time);
by stratumnum;
if _n_ eq 1 then do;
  Label = 'No. at Risk'; /* Header for axis table row labels */
  x1 = left; /* Ad hoc X coordinate */
  y1 = 9; /* Ad hoc Y coordinate */
  Anchor = 'Left'; /* Left-justify header */
  TextWeight = 'Bold'; /* Bold text */
  TextStyleElement = 'GraphDefault'; /* Text style element */
  output; /* Write header */
  textweight = ' '; /* Back to normal font */
  TextSize = 8; /* Text size for p-value */
  set p(keep=probchisq); /* Get p-value from ODS output data set */
  label = 'P=' || put(probchisq, pvalue6.4); /* p-value inset */
  x1 = 7.5; /* Ad hoc X coordinate */
  y1 = 90; /* Ad hoc Y coordinate */
  output; /* Write p-value */
end;
```

```

        TextSize = .;          /* Restore default text size          */
    end;
                                /* Set colors for axis table rows      */
    textstyleelement = cats('GraphData', stratumnum);
    textweight = ' ';         /* Normal font (not bold)            */
    if first.stratumnum then do; /* Set up axis table row labels      */
        label = stratum;      /* Row label                          */
        x1 = left;           /* Ad hoc X coordinate                */
        y1 = (3 - stratumnum) * 3; /* Ad hoc Y coordinate                */
        anchor = 'Left';     /* Left-justify header                */
        output;              /* Write row label                    */
        anchor = 'Right';    /* Right-justify                      */
        y1space = 'DataValue'; /* Use data value Y coordinate        */
        x1 = 97.5;           /* Ad hoc X coordinate                */
        y1 = ifn(stratumnum eq 1, 75, 90); /* Ad hoc Y coordinates              */
        TextSize = 7;        /* Text size for ad hoc curve labels  */
        x1space = 'GraphPercent'; /* Graph percentage X coordinates    */
        output;              /* Write out ad hoc curve labels      */
        textsize = .;        /* Restore default text size          */
        x1space = 'DataValue'; /* Restore X coordinate space        */
    end;
    if n(tatrisk);           /* If this is part of the at-risk table */
    label = put(atrisk, 6. -L); /* Format numeric value to string     */
    x1 = tatrisk;           /* X coordinate from data set          */
    y1 = (3 - stratumnum) * 3; /* Ad hoc Y coordinate                */
    y1space = 'GraphPercent'; /* Y coordinate is graph percentage   */
    anchor = 'Center';      /* Center values relative to ticks    */
    output;                  /* Write out body of axis table       */
run;

* Display SG annotation data set;
proc print noobs; run;

```

The data set is displayed in [Figure 5](#).

**Figure 5** SG Annotation Data Set

Function	X1Space	Y1Space	ID	Width	Label	x1	y1	Anchor	TextWeight	TextStyleElement	TextSize
Text	DataValue	GraphPercent	id	100	No. at Risk	-3.0	9	Left	Bold	GraphDefault	.
Text	DataValue	GraphPercent	id	100	P=0.3940	7.5	90	Left		GraphDefault	8
Text	DataValue	GraphPercent	id	100	Drug	-3.0	6	Left		GraphData1	.
Text	GraphPercent	DataValue	id	100	Drug	97.5	75	Right		GraphData1	7
Text	DataValue	GraphPercent	id	100	500	0.0	6	Center		GraphData1	.
Text	DataValue	GraphPercent	id	100	403	3.0	6	Center		GraphData1	.
Text	DataValue	GraphPercent	id	100	322	6.0	6	Center		GraphData1	.
Text	DataValue	GraphPercent	id	100	253	9.0	6	Center		GraphData1	.
Text	DataValue	GraphPercent	id	100	160	12.0	6	Center		GraphData1	.
Text	DataValue	GraphPercent	id	100	72	15.0	6	Center		GraphData1	.
Text	DataValue	GraphPercent	id	100	0	18.0	6	Center		GraphData1	.
Text	DataValue	GraphPercent	id	100	Placebo	-3.0	3	Left		GraphData2	.
Text	GraphPercent	DataValue	id	100	Placebo	97.5	90	Right		GraphData2	7
Text	DataValue	GraphPercent	id	100	500	0.0	3	Center		GraphData2	.
Text	DataValue	GraphPercent	id	100	406	3.0	3	Center		GraphData2	.
Text	DataValue	GraphPercent	id	100	324	6.0	3	Center		GraphData2	.
Text	DataValue	GraphPercent	id	100	236	9.0	3	Center		GraphData2	.
Text	DataValue	GraphPercent	id	100	162	12.0	3	Center		GraphData2	.
Text	DataValue	GraphPercent	id	100	84	15.0	3	Center		GraphData2	.
Text	DataValue	GraphPercent	id	100	0	18.0	3	Center		GraphData2	.



The last step reads the data set of dynamic variables and creates a PROC SGRENDER step from it:

```

* Enable HTML capture of final graph at original size;
ods html image_dpi=300 style=noframe;

* Write SGRENDER code, populate dynamics;
data _null_;

  * Read dynamics;
  set dynamics(where=(label1 ne '___NOBS___')) end=eof;

  * Write PROC statement, start DYNAMIC statement;
  if _n_ = 1 then do;
    call execute('proc sgrender data=fp sganno=anno ' ||
      'template=Stat.Lifetest.Graphics.ProductLimitFailure;');
    call execute('dynamic');
  end;

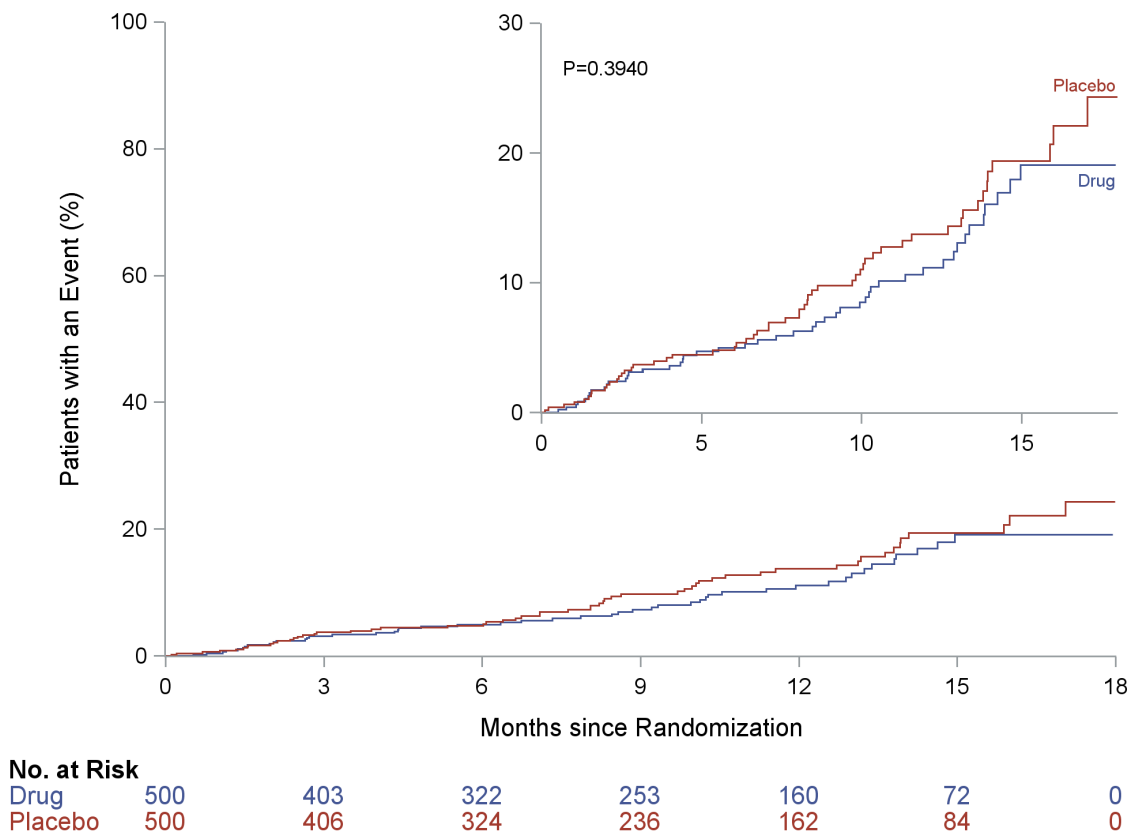
  * Write name/value pairs. Numeric: name=formatted-value.
  Character: name=quoted formatted-value;
  if cvalue1 ne ' ' then
    call execute(catx(' ', label1, '=',
      ifc(n(nvalue1), cvalue1,
        quote(trim(cvalue1)))));

  * End the DYNAMIC statement, end the PROC SGRENDER call;
  if eof then call execute('; run;');
run;
ods _all_ close;
ods graphics on / reset=all;

```

You need to use a DATA step and a CALL EXECUTE statement to populate a DYNAMIC statement with all the values of the dynamic variables. The final graph is displayed in [Figure 6](#).

**Figure 6** Final Customizations



The final step deletes the modified templates:

```
proc template;
  delete Stat.Lifetest.Graphics.ProductLimitFailure / store=sasuser.templat;
  delete Stat.Lifetest.Graphics.ProductLimitFailure2 / store=sasuser.templat;
quit;
```

### Writing Your Own Template

Assume that the data sets **fp**, **p**, and **anno** and the style **NoFrame** exist from previous steps. Then you can use the following steps to create the graph:

```
proc template;
  define statgraph Failure;
    BeginGraph;
      layout overlay / pad=(bottom=10% left=5%)
        xaxisopts=(offsetmin=0 offsetmax=0
          linearopts=(tickvaluelist=(0 3 6 9 12 15 18))
          label='Months since Randomization')
        yaxisopts=(offsetmin=0 offsetmax=0
          label="Patients with an Event (%)"
          linearopts=(viewmin=0 viewmax=100
            tickvaluelist=(0 20 40 60 80 100)));
      annotate / id='id';
      stepplot y=eval(100 - 100 * SURVIVAL) x=TIME /
        group=STRATUM index=STRATUMNUM;

      layout overlay / width=50% height=50% halign=0.95 valign=1
        xaxisopts=(offsetmin=0 offsetmax=0
          linearopts=(tickvaluelist=(0 3 6 9 12 15 18))
          display=(line ticks tickvalues))
        yaxisopts=(offsetmin=0 offsetmax=0
          linearopts=(viewmin=0 viewmax=30
            tickvaluelist=(0 10 20 30))
          display=(line ticks tickvalues));
      stepplot y=eval(100-100 * SURVIVAL) x=TIME /
        group=STRATUM index=STRATUMNUM;
    endlayout;
  EndGraph;
end;
quit;

data anno2;
  set anno;
  if y1space = 'DataValue' then do;
    x1 + -2;
    y1 + -0.8;
  end;
  if label =: 'P=' then x1 + 0.5;
run;

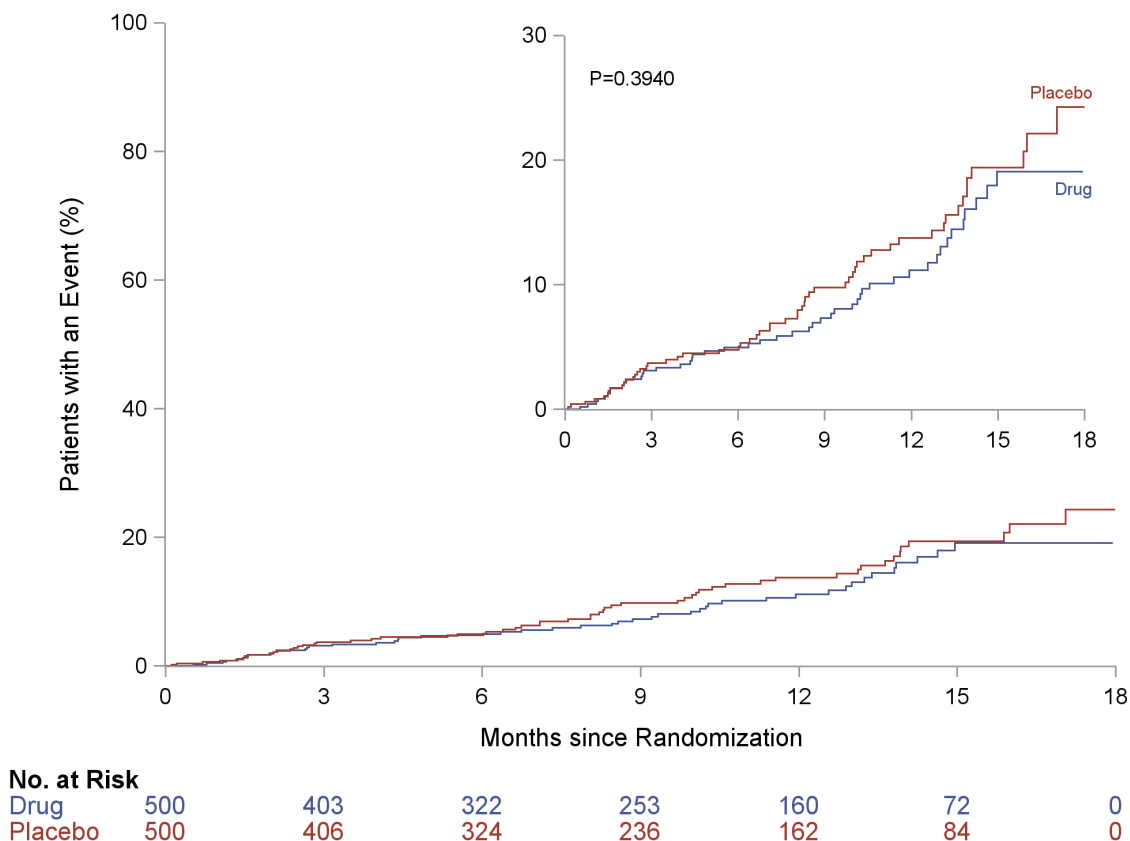
ods graphics on / reset=all border=off;

ods html image_dpi=300 style=noframe;

proc sgrender data=fp template=failure sganno=anno2; run;
```

The first step draws on the original template code and the modifications. The second step slightly changes the SG annotation data set to adjust for the slightly different sizes. The final step creates the graph displayed in [Figure 7](#). This method uses less code largely because huge parts of the template were discarded. In other examples, you might want fewer template customizations or you might need the dynamic variables. Knowing all the methods that are presented here gives you maximum flexibility.

**Figure 7** Graph Created by Using an Ad Hoc Template



## CONCLUSION

ODS Graphics gives you incredible power to customize graphs. These examples have many steps, yet each one is quite simple. Furthermore, you can program every step in SAS, ensuring reproducible results.

## ACKNOWLEDGMENTS

The authors are grateful to Sanjay Matange, Ying So, and Ed Huddleston for their help in preparing this paper.

## RECOMMENDED READING

For complete documentation about ODS and ODS Graphics, see *SAS Output Delivery System: User's Guide*, *SAS Graph Template Language: Reference*, *SAS ODS Graphics: Procedures Guide*, and *SAS Graph Template Language: User's Guide*. For more examples and more documentation, see Chapter 20, "Using the Output Delivery System"; Chapter 21, "Statistical Graphics Using ODS"; Chapter 22, "ODS Graphics Template Modification"; and Chapter 23, "Customizing the Kaplan-Meier Survival Plot" in the *SAS/STAT User's Guide*.

For a gentle and parallel introduction to PROC SGPLOT and the GTL, see the free web book *Basic ODS Graphics Examples* (<http://support.sas.com/documentation/prod-p/grstat/9.4/en/PDF/odsbasicg.pdf>).

For more advanced topics, including an introduction to SG annotation, axis tables, and template modification using the CALL EXECUTE statement, see the free web book *Advanced ODS Graphics Examples* (<http://support.sas.com/documentation/prod-p/grstat/9.4/en/PDF/odsadvvg.pdf>).

For tips, tricks, and the latest developments in ODS Graphics, see Sanjay Matange's blog *Graphically Speaking* (<http://blogs.sas.com/content/graphicallyspeaking/>) and his books (<http://support.sas.com/publishing/authors/matange.html>).

## EXAMPLE CODE

[Double-Click Here for All Example Code](#) (The link is supported in Adobe Reader but not in many browsers.) Each example is complete and includes data generation and storing the templates.

## CONTACT INFORMATION

Warren F. Kuhfeld  
SAS Institute Inc.  
SAS Campus Drive  
Cary, NC 27513  
(919) 531-7922  
Warren.Kuhfeld@sas.com

John King  
Ouachita Clinical Data Services, Inc.  
1769 Highway 240 W  
Caddo Gap, AR 71935  
(501) 351-0432  
iebupdte@gmail.com

## REFERENCES

- Holman, R. R., Bethel, M. A., Mentz, R. J., Thompson, V. P., Lokhnygina, Y., Buse, J. B., Chan, J. C., et al. (2017). "Effects of Once-Weekly Exenatide on Cardiovascular Outcomes in Type 2 Diabetes." *New England Journal of Medicine* 377:1228–1239.
- Kuhfeld, W. F. (2015). *Advanced ODS Graphics Examples*. Cary, NC: SAS Institute Inc. <http://support.sas.com/documentation/prod-p/grstat/9.4/en/PDF/odsadvg.pdf>.
- Marso, S. P., Bain, S. C., Consoli, A., Eliaschewitz, F. G., Jódar, E., Leiter, L. A., Lingvay, I., et al. (2016a). "Semaglutide and Cardiovascular Outcomes in Patients with Type 2 Diabetes." *New England Journal of Medicine* 375:1834–1844.
- Marso, S. P., Daniels, G. H., Brown-Frandsen, K., Kristensen, P., Mann, J. F. E., Nauck, M. A., Nissen, S. E., et al. (2016b). "Liraglutide and Cardiovascular Outcomes in Type 2 Diabetes." *New England Journal of Medicine* 375:311–322.
- Zinman, B., Wanner, C., Lachin, J. M., Fitchett, D., Bluhmki, E., Hantel, S., Mattheus, M., et al. (2015). "Empagliflozin, Cardiovascular Outcomes, and Mortality in Type 2 Diabetes." *New England Journal of Medicine* 373:2117–2128.