Application of Some Advanced PROC SQL Features in Clinical Trial Programming
Changhong Shi, Merck & Co., Inc.
Sylvianne B. Roberge, Merck & Co., Inc.

ABSTRACT
The DATA step, as the foundation of SAS, provides many programming options. However, SAS programs using DATA steps sometimes get long and the logic gets complicated. PROC SQL can be used as an alternative. For example, PROC SQL can provide very elegant code when a data set needs to be appended to another data set, excluding duplicate records. In this paper, we will review some useful, usually overlooked, features of PROC SQL and their applications in clinical trials. Five examples are discussed in this paper. They include: creating macro variable(s) with compact code, building a macro variable with a list of values dynamically, appending one data set to another one and excluding duplicate records, joining tables, especially many to many joins and self join, and filling in missing data. All of the examples contrast the use of PROC SQL with other SAS techniques, especially DATA step coding.

DISCUSSION
In the following section, five typical clinical trial programming tasks are discussed. The PROC SQL code that performs each task is contrasted with the code for other methods.

1. Create macro variable(s) in compact code

Case A:
Sometimes during data checking and cleaning, we need to obtain the aggregate value of a field and store it in a macro variable for use in later processing. For example, we want to calculate the maximum number of records where contact number is not missing for all patients in several data sets. This can be accomplished using a DATA step or PROC SQL. The input data set is:

```plaintext
data offtrt;
  input pat_num 1-7 contact 9-10 value 12-13;
cards;
1111135 20 13  
1111135 40 80  
1111166 20 20  
1111166 .  13  
1111166 .  15  
run;
```

The code for the DATA Step is:

```plaintext
%macro getmaxcnt(inset=,varmax=);
%global &varmax;
proc sort data=&inset;
  by pat_num;
run;

data _null_;  
  set &inset end=eof;  
  by pat_num;  
  retain count maxcnt;  
if (first.pat_num) then 
  count=0;  
if (contact ne .) then 
  count=count+1;  
if (last.pat_num) then 
  maxcnt=max(maxcnt, count);  
if eof
  call symput("&varmax",trim(left(put(maxcnt,3.))));
run;
%mend getmaxcnt;
```

We can call the macros using:

```plaintext
%getmaxcnt(inset=offtrt, varmax=offmax);
```

The code for PROC SQL is:

```plaintext
%macro getmaxcnt(inset=,varmax=);
%global &varmax;
proc sql noprint;
  create view reccount as
  select pat_num, count(contact) as reccount
  from &inset
  group by pat_num;
  select max(reccount) into :&varmax
  from reccount;
quit;
%mend getmaxcnt;
```

We can call the macros using:

```plaintext
%getmaxcnt(inset=offtrt, varmax=offmax);
```

We can see that the advantages of using PROC SQL to get the maximum number of records (or other aggregate values) in comparison with the other two methods are:

1. We do not have to presort the input data as is necessary with the DATA step method.
2. We do not create any intermediate data set as in the PROC FREQ and the DATA step methods.

Case B:
Not only can PROC SQL be used for the creation of a single macro variable, it can also create several macro variables in one statement. For example, suppose we have a data set called quantile, and every row contains q1, median and q3, for 25th percentile, median, and 75th percentile values for several groups of patients. The input data set is:

```plaintext
data quantile;
  input q1 1-4 median 6-10 q3 12-16;
cards;
9.11 20.02 76.89  
8.32 40.32 80.23  
6.16 60.91 20.15  
7.34 40.45 60.03  
9.57 60.65 56.78  
run;
```

The code for the DATA Step is:

```plaintext
%macro getmaxcnt(inset=,varmax=);
%global &varmax;
proc sort data=&inset;
  by pat_num;
run;

data _null_;  
  set &inset end=eof;  
  by pat_num;  
  retain count maxcnt;  
if (first.pat_num) then 
  count=0;  
if (contact ne .) then 
  count=count+1;  
if (last.pat_num) then 
  maxcnt=max(maxcnt, count);  
if eof
  call symput("&varmax",trim(left(put(maxcnt,3.))));
run;
%mend getmaxcnt;
```

We can call the macros using:

```plaintext
%getmaxcnt(inset=quantile, varmax=q1, q2, q3);
```

We can see that the advantages of using PROC SQL to get the maximum number of records (or other aggregate values) in comparison with the other two methods are:

1. We do not have to presort the input data as is necessary with the DATA step method.
2. We do not create any intermediate data set as in the PROC FREQ and the DATA step methods.

Case C:
Not only can PROC SQL be used for the creation of a single macro variable, it can also create several macro variables in one statement.

For example, suppose we have a data set called quantile, and every row contains q1, median and q3, for 25th percentile, median, and 75th percentile values for several groups of patients. The input data set is:

```plaintext
data quantile;
  input q1 1-4 median 6-10 q3 12-16;
cards;
9.11 20.02 76.89  
8.32 40.32 80.23  
6.16 60.91 20.15  
7.34 40.45 60.03  
9.57 60.65 56.78  
run;
```
If we want to store these three variables for only the first row, we can use DATA Step as follows:

``` SAS
data _null_; set quantile (obs=1); call symput('q1',trim(left(put(q1,6.2)))); call symput('q2',trim(left(put(median,6.2)))); call symput('q3',trim(left(put(q3,6.2)))); run;
```

The corresponding PROC SQL code is:

``` SQL
proc sql inobs=1; select trim(left(put(q1,6.2))), trim(left(put(median,6.2))), trim(left(put(q3,6.2))) into :q1,:q2,:q3 from quantile; quit;
```

The DATA step method requires a data _null_ step and three function calls to create the three macro variables that PROC SQL can do in one statement. Here, we begin to see PROC SQL’s power, which will be appreciated when we want to build a series of “grouped” macro variables. For example, we can modify the preceding example to illustrate this use. Assume there are 10 records in data set quantile. If we want to assign macro variables to all of the q1, median, and q3 values, we can use the following PROC SQL statements to create macro variables for each group.

``` SQL
proc sql; select count(*) into :n from quantile; select trim(left(put(q1,6.2))), trim(left(put(median,6.2))), trim(left(put(q3,6.2))) into :p1 - :p%left(&n), :m1 - :m%left(&n), :r1 - :r%left(&n) from quantile; quit;
```

The code can be even shorter if we take advantage of the fact that SAS will assign only as many macro variables as needed from the specified range of 1 to 999:

``` SQL
proc sql noprint; select '_'||compress(put(point,2.)) into :pt separated by ' ' from timept; quit;
```

PROC SQL is the easiest way:

``` SQL
proc sql noprint; select '_'||compress(put(point,2.)) into :pt separated by ' ' from timept; quit;
```

If we use a DATA Step, the code will be:

``` SAS
data _null_; set timept end=eof; retain pt ; length pt $ 100; if (_n_=1) then pt='_'||compress(put(point,2.)); else pt =left(trim(pt)) ||  ' _'||put(point,2.); if eof then call symput('pt', pt); run;
```

As we can see, a DATA _null_ step has to be used, the code is longer, and the concatenation code is trickier than the corresponding PROC SQL code.

3. Append one data set to another one, excluding duplicate records

Appending one data set to another one and eliminating duplicate records requires considerable code in a DATA step. PROC SQL, on the other hand, can do it conveniently in one step using the UNION function.

For example, we have a data set that contains time points in our study such as 20, 40, 60, 80. It is saved in the timept data set:

``` SAS
data timept; input point 1-2; cards; 20 40 60 80 run;
```

We also have a lab data set that has a patient number, time point, and the test value:

``` SAS
data lab; input pat_num 1-7 point 9-10 value 12-13; cards; 1111135 20 13 1111135 40 80 1111135 80 10 1111166 20 20 1111166 40 13 1111166 60 15 run;
```

As we can see, there is a missing time point, 60, for patient 1111135, and a missing time point, 80, for patient 1111166. We need to add records for the missing time points so that every patient has all lab data time points. One way to accomplish this is to use PROC TRANSPOSE on the lab data set. After we transpose, we will get variable names like: _20, _40, _60, _80 in the output data set and we want to store this list of values in a macro variable for later processing.

PROC SQL is the easiest way:

``` SQL
proc sql noprint; select '_'||compress(put(point,2.)) into :pt separated by ' ' from timept; quit;
```

If we use a DATA Step, the code will be:

``` SAS
data _null_; set timept end=eof; retain pt ; length pt $ 100; if (_n_=1) then pt='_'||compress(put(point,2.)); else pt =left(trim(pt)) ||  ' _'||put(point,2.); if eof then call symput('pt', pt); run;
```

As we can see, a DATA _null_ step has to be used, the code is longer, and the concatenation code is trickier than the corresponding PROC SQL code.

2. Build a macro variable with a list of values dynamically

PROC SQL is also useful when we want to store a list of values in a macro variable.
In this example we have two data sets: the phone data set and the phonenew data set. We want to append the phonenew data set to the phone data set. However, we do not want to keep the duplicates. The easiest way we have found is to use PROC SQL. We can see the comparison of PROC SQL and the DATA step in the following example:

```
data phone;
input pat_num 1-7 contact 9-10 value 12-13;
cards;
1111135 20 13
1111135 40 80
1111135 60 10
1111166 20 20
1111166 40 13
run;

data phonenew;
input pat_num 1-7 contact 9-10 value 12-13;
cards;
1111135 60 10
1111166 80 80
1111166 60 15
run;

proc sql;
create table phonerec as
select *
from phone
union
select *
from phonenew;
quit;
```

The result table is:

<table>
<thead>
<tr>
<th>pat_num</th>
<th>contact</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111135</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>1111135</td>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>1111135</td>
<td>60</td>
<td>10</td>
</tr>
<tr>
<td>1111166</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>1111166</td>
<td>40</td>
<td>13</td>
</tr>
<tr>
<td>1111166</td>
<td>60</td>
<td>15</td>
</tr>
</tbody>
</table>

As we can see, the duplicate record for patient 1111135 at contact 60 is excluded from the final data set.

If we replace the PROC SQL code with the SET statement in DATA step:

```
data phonerec;
set phone phonenew;
run;
```

The result data set will be:

<table>
<thead>
<tr>
<th>pat_num</th>
<th>contact</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111135</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>1111135</td>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>1111135</td>
<td>60</td>
<td>10</td>
</tr>
<tr>
<td>1111166</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>1111166</td>
<td>40</td>
<td>13</td>
</tr>
<tr>
<td>1111166</td>
<td>60</td>
<td>15</td>
</tr>
</tbody>
</table>

As we can see, the duplicate record for patient 1111135 at contact 60 is not excluded from the final data set.

As illustrated in the above example, we see that the advantages of using PROC SQL with the UNION function to append data compared to using SET statements are:

a. Duplicate records are implicitly excluded.
b. The sorting order is maintained so no explicit sort is needed.

Of course, if we want to keep all records, including the duplicate records for some reason using PROC SQL, we can just replace UNION with UNION ALL. UNION ALL is equivalent to the SET statement in a DATA step.

4. PROC SQL and table joins

In clinical trials, we usually obtain our original input data from a relational data base like Oracle, Sybase or SQL Server. Since an objective of relational data base design is to minimize data duplication, we often have to join two or more tables in order to obtain all the necessary variables for a data set.

Both data step and PROC SQL can be used to join two or more tables. The following table compares these two methods.

<table>
<thead>
<tr>
<th>DATA Step Technique</th>
<th>Proc SQL Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Simple Merge</td>
<td>1. Inner Join</td>
</tr>
<tr>
<td>data adher;</td>
<td>data adher;</td>
</tr>
<tr>
<td>input pat_num 1-7</td>
<td>input pat_num 1-7</td>
</tr>
<tr>
<td>contact 9-10</td>
<td>contact 9-10</td>
</tr>
<tr>
<td>adhdt mmdyy8.;</td>
<td>adhdt mmdyy8.;</td>
</tr>
<tr>
<td>cards;</td>
<td>cards;</td>
</tr>
<tr>
<td>1111135 20 03/16/95</td>
<td>1111135 20 03/16/95</td>
</tr>
<tr>
<td>1111135 40 09/10/95</td>
<td>1111135 40 09/10/95</td>
</tr>
<tr>
<td>1111166 20 04/07/96</td>
<td>1111166 20 04/07/96</td>
</tr>
<tr>
<td>1111166 40 10/02/96</td>
<td>1111166 40 10/02/96</td>
</tr>
<tr>
<td>1111166 60 05/01/97</td>
<td>1111166 60 05/01/97</td>
</tr>
<tr>
<td>run;</td>
<td>run;</td>
</tr>
<tr>
<td>data phone;</td>
<td>data phone;</td>
</tr>
<tr>
<td>input pat_num 1-7</td>
<td>input pat_num 1-7</td>
</tr>
<tr>
<td>contact 9-10</td>
<td>contact 9-10</td>
</tr>
<tr>
<td>visdt mmdyy8.;</td>
<td>visdt mmdyy8.;</td>
</tr>
<tr>
<td>cards;</td>
<td>cards;</td>
</tr>
<tr>
<td>1111135 20 03/26/95</td>
<td>1111135 20 03/26/95</td>
</tr>
<tr>
<td>1111135 40 02/10/96</td>
<td>1111135 40 02/10/96</td>
</tr>
<tr>
<td>1111166 20 04/07/96</td>
<td>1111166 20 04/07/96</td>
</tr>
<tr>
<td>1111166 40 10/20/96</td>
<td>1111166 40 10/20/96</td>
</tr>
<tr>
<td>run;</td>
<td>run;</td>
</tr>
<tr>
<td>data phonenew;</td>
<td>data phonenew;</td>
</tr>
<tr>
<td>input pat_num 1-7</td>
<td>input pat_num 1-7</td>
</tr>
<tr>
<td>contact 9-10</td>
<td>contact 9-10</td>
</tr>
<tr>
<td>from phonenew;</td>
<td>from phonenew;</td>
</tr>
<tr>
<td>quit;</td>
<td>quit;</td>
</tr>
</tbody>
</table>

1. Left or Right Join:

```
data dre;
input pat_num 1-7 month 9-10 region1 12 region2 14;
cards;
```

2. Selected Merge 1:

```
data dre;
input pat_num 1-7 month 9-10 region1 12 region2 14;
cards;
```
### DATA Step Technique

cards;
1111135 1 1 2
1111135 6 2 3
1111166 12 1 4
1111166 6 2 4
1111168 12 3 4
run;
data demo;
input pat_num 1-7
height 9-11 weight 13-15 smoker 17;
cards;
1111135 187 180 1
1111166 176 136 0
1111168 170 196 1
run;

data phone;
input pat_num 1-7
contact 9-10 phn13dsp
age 12-14;
cards;
1111135 20 100
1111135 40 150
1111135 60 134
1111166 20 160
1111166 40 171
1111167 70 165
run;
data phone2;
merge phone (in=p) violator (in=v);
by pat_num;
if p and not v;
run;

data demo;
input pat_num 1-7
height 9-11 weight 13-15 smoker 17;
cards;
1111135 187 180 1
1111166 176 136 0
1111168 170 196 1
run;
proc sql;
create table dph as
select
one.low,
one.pair,
(one.lwt-two.lwt) as lwt_d,
(one.smoke-two.smoke) as smoke_d,
(one.ptd-two.ptd) as ptd_d
from phone as d
left join demo as m
on
m.pat_num= d.pat_num;
quit;

### Proc SQL Technique

cards;
1111135 1 1 2
1111135 6 2 3
1111166 12 1 4
1111166 6 2 4
1111168 12 3 4
run;
data demo;
input pat_num 1-7
height 9-11 weight 13-15 smoker 17;
cards;
1111135 187 180 1
1111166 176 136 0
1111168 170 196 1
run;

data phone;
input pat_num 1-7
contact 9-10 phn13dsp
age 12-14;
cards;
1111135 20 100
1111135 40 150
1111135 60 134
1111166 20 160
1111166 40 171
1111167 70 165
run;
data phone2;
merge phone (in=p) violator (in=v);
by pat_num;
if p and not v;
run;

data demo;
input pat_num 1-7
height 9-11 weight 13-15 smoker 17;
cards;
1111135 187 180 1
1111166 176 136 0
1111168 170 196 1
run;
proc sql;
create table dph as
select
one.low,
one.pair,
(one.lwt-two.lwt) as lwt_d,
(one.smoke-two.smoke) as smoke_d,
(one.ptd-two.ptd) as ptd_d
from phone as d
left join demo as m
on
m.pat_num= d.pat_num;
quit;

### 3. Selected Merge 2:

data violator;
input pat_num 1-7;
cards;
1111135
1111166
run;
data phone;
input pat_num 1-7
contact 9-10 phn13dsp
age 12-14;
cards;
1111135 20 100
1111135 40 150
1111135 60 134
1111166 20 160
1111166 40 171
1111167 70 165
run;
data phone2;
merge phone (in=p) violator (in=v);
by pat_num;
if p and not v;
run;

### 4. Many-Many Merge *

Please Refer to Chart I below this table. We have two data sets. The first data set is an off-treatment data

### 4. Cartesian Product

data offtre;
input pat_num 1-7
contact 9-10 offtre
mmdy8.;
run;

### 5. Correlated queries, join to itself

In clinical trial programming, we often need to compute a difference between values in the case and control groups. In the following example for a case-control study, low is 0 for the controls and 1 for the cases. Other variables in each row contain related information.

### 5. Use some functions such as Lag to "remember" specific records

In order to do this, we need 3 sorts, 5 intermediate data sets, and 3 merge processes to get the final result. Please refer to a similar example in BASE SAS help manual.

Using a DATA step to do a many-many merge is cumbersome.

### 3. Subquery or correlated subquery:

data violator;
input pat_num 1-7;
cards;
1111135
1111166
run;
data phone;
input pat_num 1-7
contact 9-10 phn13dsp
age 12-14;
cards;
1111135 20 100
1111135 40 150
1111135 60 134
1111166 20 160
1111166 40 171
1111167 70 165
run;
data phone2;
merge phone (in=p) violator (in=v);
by pat_num;
if p and not v;
run;

### 4. Many-Many Merge *

Please Refer to Chart I below this table. We have two data sets. The first data set is an off-treatment data

### 4. Cartesian Product

data offtre;
input pat_num 1-7
contact 9-10 offtre
mmdy8.;
run;

### 5. Correlated queries, join to itself

In clinical trial programming, we often need to compute a difference between values in the case and control groups. In the following example for a case-control study, low is 0 for the controls and 1 for the cases. Other variables in each row contain related information.

### 5. Use some functions such as Lag to "remember" specific records

In order to do this, we need 3 sorts, 5 intermediate data sets, and 3 merge processes to get the final result. Please refer to a similar example in BASE SAS help manual.

Using a DATA step to do a many-many merge is cumbersome.

### 3. Subquery or correlated subquery:

data violator;
input pat_num 1-7;
cards;
1111135
1111166
run;
data phone;
input pat_num 1-7
contact 9-10 phn13dsp
age 12-14;
cards;
1111135 20 100
1111135 40 150
1111135 60 134
1111166 20 160
1111166 40 171
1111167 70 165
run;
data phone2;
merge phone (in=p) violator (in=v);
by pat_num;
if p and not v;
run;

### 4. Many-Many Merge *

Please Refer to Chart I below this table. We have two data sets. The first data set is an off-treatment data

### 4. Cartesian Product

data offtre;
input pat_num 1-7
contact 9-10 offtre
mmdy8.;
run;

### 5. Correlated queries, join to itself

In clinical trial programming, we often need to compute a difference between values in the case and control groups. In the following example for a case-control study, low is 0 for the controls and 1 for the cases. Other variables in each row contain related information.

### 5. Use some functions such as Lag to "remember" specific records

In order to do this, we need 3 sorts, 5 intermediate data sets, and 3 merge processes to get the final result. Please refer to a similar example in BASE SAS help manual.

Using a DATA step to do a many-many merge is cumbersome.
The result table for the difference between the case and control group is:

<table>
<thead>
<tr>
<th>pair</th>
<th>low</th>
<th>lwt_d</th>
<th>smoke_d</th>
<th>ptd_d</th>
<th>ht_d</th>
<th>UI_d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-34</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>27</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

We have an original lab data set like:

<table>
<thead>
<tr>
<th>pat_num</th>
<th>point</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111135</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>1111135</td>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>1111135</td>
<td>80</td>
<td>10</td>
</tr>
<tr>
<td>1111166</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>1111166</td>
<td>40</td>
<td>13</td>
</tr>
<tr>
<td>1111166</td>
<td>60</td>
<td>15</td>
</tr>
</tbody>
</table>

And we know that all patients should have four time points, 20, 40, 60, and 80 in our particular study, which is saved in a timept data set.

As we can see, in the original lab data: time point 60 is missing for patient 1111135, and time point 80 is missing for patient 1111166.

We want to build a data set (alllab data set) with all missing time points filled in. i.e.

<table>
<thead>
<tr>
<th>pat_num</th>
<th>point</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111135</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>1111135</td>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>1111135</td>
<td>60</td>
<td>10</td>
</tr>
<tr>
<td>1111135</td>
<td>80</td>
<td>10</td>
</tr>
<tr>
<td>1111166</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>1111166</td>
<td>40</td>
<td>13</td>
</tr>
<tr>
<td>1111166</td>
<td>60</td>
<td>15</td>
</tr>
<tr>
<td>1111166</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>

The two simple ways to do this task are: PROC SQL and PROC TRANSPOSE. We list the code for comparison between these two methods below:
Comparing the code, we can see that the PROC SQL code is more compact and the logic is easier to understand. In PROC TRANSPOSE, we first need to store the number of time points and the time point values in macro variables since we need them for the final output of alllab data set. The logic using PROC TRANSPOSE is more obscure and PROC TRANSPOSE is very memory demanding. Therefore, for large data sets, the PROC SQL method has more advantages when used to fill in the missing data points than the PROC TRANSPOSE method.

CONCLUSION

PROC SQL is a useful tool for SAS programmers to employ to minimize program maintenance and simplify programming. It complements the DATA step by reducing the need for presorting input data sets and for creating temporary data sets. As we have shown in this paper, it can make SAS programs more compact and the logic easier to understand.

REFERENCE


CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Changhong Shi
Merck & Co., Inc.
RY34-A320
126 East Lincoln Ave.
Rahway, NJ 07065
Work Phone: (732) 594-1383
Fax: (732) 594-6075
Email: changhong_shi@merck.com

Sylvianne B. Roberge
Merck & Co., Inc.
RY34-A320
126 East Lincoln Ave.
Rahway, NJ 07065
Work Phone: (732) 594-7853
Fax: (732) 594-6075
Email: sylvianne_roberge@merck.com

Special Thanks to Elaine B. Carnicki for getting us started and continuous support.