ABSTRACT

SAS/OR software provides several facilities for managing projects and resources. This tutorial describes how to design a menu-driven system for managing multiple tasks that are to be completed by several different individuals. Using the CPM procedure, the system tracks utilization of manpower, monitors project progress, and estimates feasibility of starting new projects within a given time frame. The system uses Release 6.06 Screen Control Language with SAS/FSP and SAS/AF software to make entry of the projects simple.

INTRODUCTION

Programming projects present a unique set of scheduling problems. When developing a large software system, the project manager must break the design specifications into functionally discrete subtasks and identify the best programmer for each subtask. In addition the project manager must estimate the time required to complete each subtask and identify interdependencies among assignments.

Once this is completed, the project manager is faced with shuffling resources (programmers) to complete the entire system in the most efficient manner. After the projects have started, the project manager must also deal with record keeping and missed deadlines. A good scheduling system, therefore, should enable the project manager to update the system interactively and continuously.

This discussion presents one approach to managing these problems using Version 6 SAS/OR software in conjunction with base SAS, SAS/FSP and SAS/AF software. The CPM procedure is used to schedule the projects, therefore its features are briefly explained. The following example demonstrates a method that eases the data entry burden, tracks on-going projects, assigns specific programmers to specific jobs, and accounts for absences. In addition, the example uses PROC CPM options to stagger project start times, align start dates, and update the schedule to reflect actual completion times.

THE CPM PROCEDURE

The SAS/OR product offers a procedure called CPM. The CPM procedure uses the critical path method to find the shortest completion time for a project, given a set of priorities and constraints. PROC CPM enables you to define your project with an activity-on-node network or with an activity-on-arc network. This discussion designs a project scheduling system based on the activity-on-node method. For further explanation of the critical path method and for more references, see the SAS/OR User's Guide, Version 6, First Edition.

The general syntax of the CPM procedure is simple. Many options are available to enable you to control the processing. In order to have complete control and provide the most flexibility, the following program uses most of the available options. The complexity is hidden from the user of the system by a SAS/AF menu system.

```
proc cpm
   /* Specify input and output data set names */
   data=projects
   resin=resin
   holidata=cpm.holidays
   workdata=cpm.workdata
   caledata=cpm.caledata
   out=schedule
   /compress
   /* Schedule unit is the datetime hour */
   interval=dthour;
   /* Specify what variables contain which info */
   activity project; successor post;
   duration duration;
   aligndate algndate;
   aligntype algntype;
   pid person detail remdur;
   /* Specify actual vars and options. A_START */
   /* and A_FINISH indicate the times that */
   /* projects actually started or finished. */
   /* REMDUR indicates the hours remaining. */
   actual /
   a_finish = a_finish
   a_start = a_start
timenow split
   timenow=%sysdate:09:00
   remdur = remdur;
   /* Specify holiday vars, start and finish. */
   holiday holista /
   holifin=holifin;
   /* Specify resource variables and options */
   /* that determine how resources are used. */
   resource &namelist /
   period=effedate
   activityprty=actprty
   schedule=actprty
   obstype=obstype
   splitfag; /* project splits permitted */
r
```
PROC CPM - INPUT DATA SETS

PROC CPM expects to find certain input data sets depending on the options specified. The five possible types are listed below, but refer to the SAS/OR User’s Guide for an explanation of their requirements and proper usage:

* Activity or project data set
* Resource data set
* Holidays data set
* Work day pattern data set
* Calendar data set

The first two data sets as they relate to this application are described below.

THE PROJECTS DATA SET - PROJECTS

The first data set is the project data set, whose name is passed to PROC CPM via the DATA= option. The project data set contains the list of subtasks that must be scheduled, variables that provide information about those subtasks, and a list of any subtasks that depend on another subtask.

Successors can be specified by listing all successor tasks in a set of variables in one observation (method 1). Alternatively, successors can be listed by using only one successor variable but repeating the observation for each additional successor (method 2). The two methods are compared below:

<table>
<thead>
<tr>
<th>METHOD 1</th>
<th>METHOD 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROJECT</td>
<td>PROJECT</td>
</tr>
<tr>
<td>SUCC1</td>
<td>POST</td>
</tr>
<tr>
<td>SUCC2</td>
<td></td>
</tr>
<tr>
<td>SUCC3</td>
<td></td>
</tr>
</tbody>
</table>

where B, C and D cannot begin until A is completed, and E and F must wait for project B to be completed. Method 1 has the advantage of simplicity but is limited to a specific number of possible successor tasks. Method 1 is nice from the input standpoint because all successors can be seen at once. In Version 5, the disadvantage to method 1 was that any unused successor fields were still occupying valuable disk space. With the new Version 6 data set architecture, this is no longer true. Unused or blank fields do not add to your disk storage requirements. The CPM statements used in method 1 are:

activity project;
successor succ1 succ2 succ3;

Method 2 wastes space through data redundancy. The information about PROJECT must be repeated for each successor, which is a waste of space in any version of SAS software. In addition, if you use PROC FSEDIT to enter your projects, you will find it difficult to remember which successor you have already entered and which you have not because you will only see one at a time. The advantage to method 2 is that there is no preset limit to the number of successors that can be hooked to a project. In automated systems this is certainly desirable. The CPM statements used in method 2 are:

activity project;
successor post;

where POST is a variable containing the name of the successor. Although PROC CPM still expects to receive the data in one of these two forms in Release 6.06, there is no reason why you cannot enter the data in a form of your preference and rearrange the layout as you pass the data to PROC CPM. In the example presented in this discussion, the person entering the projects into the system can view and edit the current subtask, as well as the successors and predecessors.

Instead of using one data set, the project specific data are stored in one data set and the relationship between projects is stored in another. For example, the permanent projects data set includes variables for project name, project priority, project description, duration in hours, alignment datetime, alignment type, hours used, percent complete, hours remaining, programmer identification, actual start datetime or actual finish datetime, and a calendar selection.

The relationship data set contains only two variables, PRE and POST, where PRE contains the name of a project that must be completed before the project named in POST. To determine successors for any project, find all occurrences of the project name in PRE and the predecessors will be in the variable POST. Likewise to find all of the predecessors of a given project, find all occurrences of that project name in POST and the value of PRE will name the predecessors.

Using SAS/AF software with Screen Control Language, you can present these separate data sets as one.
PROC FSEDIT is used to edit the projects data set, but the Screen Control Language (SCL) associated with the FSEDIT screen can open the relationship data set using a subsetting WHERE clause. Refer to SAS/FSF Software: Usage and Reference, Version 6, First Edition for a discussion of using SCL with PROC FSEDIT and an explanation of the executable sections of the program. Also refer to SAS Screen Control Language: Reference, Version 6, First Edition.

The WHERE function in SCL is used to subset a data set efficiently and dynamically. After applying a WHERE clause to a data set by using the SCL WHERE function, the data set appears to contain only those observations that meet your WHERE criteria. At that point, most other SCL functions, such as FETCH, only see the observations that have passed through the WHERE filter. The WHERE function uses indexes if they exist, so indexing is worthwhile if the data set is large.

The WHERE function expects the WHERE clause to be passed in the form: variable='string', for example, PRE='AAA/121'. For this reason, the WHERE clause must be constructed for each execution of the WHERE function in order to pass the current string value. For example, the SCL might look like this in the INIT section (which is executed when the user moves to a new observation in PROC FSEDIT):

```sas
/* Define WHERE clause to select observations */
/* WHERE PRE or POST matches the project name */
/* in the FSEDIT data set. Turn on WHERE. */
/***********************************************************/
wci=trim(project)';
wcl2='0r post=' || trim(project)';
rc=where(dsid,wcl1,wcl2);
%rcheck;
rc=rewind(dsid); /* return to top of data set*/
%rcheck;
/***********************************************************/
/* Read record from CPM.RELATE and increment */
/* obs counter. If PRE=PROJECT, obs is a */
/* successor so store in array SUCCS. */
```

The macro `%RCHECK` is a programmer-defined macro used to test return codes throughout this discussion but will not be discussed in detail. It is used here primarily as a space saving device and indicates places where return code checking should be implemented.

The WHERE clause is used to select all pairs that relate to the current project that the user is viewing with PROC FSEDIT. Each pair is evaluated to determine whether the pair represents a predecessor or a successor relation. Predecessors are loaded into window variables PRED1 through PRED12 and successors are loaded into the variables SUCC1 through SUCC12, which are referenced by array names SUCCS and PREDS. The WHERE clause is used only once for both predecessors and successors in order to keep the logic simple and to make updating easy.

Used for updating, a relative observation number is associated with each pair and is stored in arrays SOBS and POBS. When the user moves to another observation or ends the application, the program checks to see if a relative observation number exists. If one does not, the entry is assumed to be a new successor or predecessor and is appended to the data set. The code in the TERM section might look like this:

```sas
/* if POST = PROJECT, obs is a predecessor */
/* so store in array PREDS. */
/***********************************************************/
i=0; j=0; obnum=0;
rc=fetch(dsid);
do while (rc=0);
obnum i=1;
if pre=project then
  do;
i+1;
succs(i)=post;
sobs(i)=obnum;
  end;
else
  do;
j+1;
preds(j)=pre;
pobs(j)=obnum;
  end;
rc=fetch(dsid);
end;
if rc=1 then _msg_=' ';
else do;
  _msg_ =sysmsg();
  return;
end;
rc=rewind(dsid);
%rcheck;
```
TERM: /* Executes when you leave an observation*/

do j=1 to 12;
**************************************/
/* If name found, but no obs number, */
/* must be new, so add it. */
**************************************/
if sobs{j}=. and succs{j}='I' then do;
  pre=project;
  post=succs{j};
  rc=append(dsid);
%rcheck;
end;  
**************************************/
/* If obs number found, then we must */
/* update or delete it. */
**************************************/
else if sobs{j}='.' then do;
  obnum=sobs{j};
  rc=delete(dsid);
%rcheck;
/* Adjust relative obs nums */
do i=1 to 12;
  if sobs{i}>obnum then sobs{i}=sobs{i}-1;
  if pobs{i}>obnum then pobs{i}=pobs{i}-1;
end;
else do;
  post=succs{j};
  pre=project;
  rc=update(dsid);
%rcheck;
end;
end;
$sobs{j}=.';
succs{j}=I1;
/* repeat the same process */
/* for predecessors */
retum;

The beauty of this approach is that any updates to successors automatically show up in the appropriate predecessor list without the user making the additional edit. This is also true of updates to the predecessors. Editing is simply a matter of typing over the field, blanking out the field, or adding a field.

The only restriction with this approach is the limit of twelve successors and twelve predecessors. For most projects, twelve is probably more than sufficient. To accommodate more than twelve, you could use the DISPLAY function with an extended table and present an unlimited list of successors and predecessors. The drawback to this approach is that only one active window can be scrolled or updated at a time. This slows the process by requiring the user to get in and out of either the predecessor or successor windows.

Another option would be to use the twelve array slots as before, but provide a scroll button that would instruct the program to reload the array with a new set of values. This way you simulate the functionality of a scrolling dynamic table without the loss in flexibility.

Another field that must be provided in the PROJECTS data set is the number of hours used. This could be updated each day by simply adding each day's hours to the current number used, but with SCL you can build in an automatic hours accounting system.

With the addition of the <Add> button next to the hours worked field, the user can enter PROC FSVIEW or a specifically designed SCL data entry system. For example, the following SCL code opens an FSVIEW window when the user places the cursor on <Add> and presses ENTER. Within PROC FSVIEW, the user can enter each day's hours of work on the project that PROC FSEDIT is currently displaying. This provides an automatic log of hours which can be used for billing or tracking purposes. The code below is found in the MAIN section:

```
MAIN:
cmd=curword();
if cmd='<Add>' then do;
call symput('duration',
duration);
fname=
'cpm.hrsused(where=(project=''
|| trim(project)||''));
call fsview(fname,
'edit',
'cpmaf.fsview.projects.formula');
hrid=open('cpm.hrsused');
call set(hrid); hrs_used=O;
hrwhere=compress(
'project='''||project||''');
rc=where(hrid,hrwhere);
%rcheck;
rc=fetch(hrid);
do while(rc=O);
  hrs_used = hrs_used + hrsworkd;
  rc=fetch(hrid);
%rcheck;
end;
cmd='1';
edithrs='<Add>';
close(hrid);
%rcheck;
end;
```

The call to SYMPUT is used to pass the current value of DURATION to PROC FSVIEW. That lets users know how much time they have to work with. The variable FSTMT contains the data set specification for the FSVIEW call. Note that the WHERE clause is passed
as a data set option. The call to FSVIEW also specifies a formula catalog. Refer to the SAS/FSP User’s Guide for further explanation of using formulas.

Upon returning from FSVIEW, the WHERE function is used to resubset the data set and sum all of the hours used. In this example, the user cannot update the HRS_USED field without using this facility. The HRS_USED field is protected by the FSEDIT attributes, but the calculated value is stored permanently.

Along with these basic functions, you can add many time-saving features with SCL. For example, you can use the SHOWLIST function to provide a list of valid alignment types. If the user enters a question mark in the ALGNTYPE field, a window is displayed with a selection list. Also, you can provide shorthand features using SCL. For example, if the user enters a slash followed by numbers in the successor or predecessor list, the main project is assumed to match the current main project, and the successor or predecessor name is constructed using the current main project name.

THE RESOURCE DATA SET - RESIN

The resource data set, called RESIN in this example, determines what resources are available for completing a project. In this case the resources are individual programmers, and each programmer is associated with a single variable. In most applications of PROC CPM, the resources are supplies, such as a quantity of parts or gallons of fuel. In this case the resource variable has a value of one or zero; the programmer is working or not working.

For example, in PROC CPM you would specify the resources by using the RESOURCE statement as follows:

```sas
resource ecb ett dhm cjg gsk sg j/ options...
```

where each variable name is actually the initials of the programmer. The problem with this approach appears to be that the source code must be changed every time a new programmer joins the organization. To keep this process dynamic, you can automatically build a macro variable that contains the identifications of all programmers. The macro variable can be created by scanning the project data set once and extracting each name. For example, the program below reads the PROJECTS data set and the VACATION data set, and when the last line has been read, two macro variables are created. Macro variable NAMELIST contains the list of programmer IDs, and macro variable SIZE contains the number of programmers in the system.

```sas
data projects;
length namestr $ 200;
retain namestr ;
set cpm.projects
  cpm.vacation
end=lastline;
if "%index(namestr,trim(person)) then
  namestr=trim(namestr)" then
  ncnt+1;
end;
if lastline then /* If last record */
do;
call symput('namelist',namestr);
call symput('size',left(ncnt));
end;
```

To pass the resource list to PROC CPM, code the RESOURCE statement as follows:

```sas
resource &namelist;
```

In this case &NAMELIST is limited to about 50 programmers if you use one macro variable.

VACATION, ABSENCES AND BUSINESS TRIPS

In this application, the WORKDATA, CALEDATA and HOLIDAYS data sets are used as described in the SAS/OR User’s Guide. By designing specific FSEDIT screens with simple SCL behind each, you can make data entry of these data sets much easier and include valuable validation checks.

To deal with any sort of absence, this application uses an additional data set called VACATION. By using SCL behind the FSEDIT screen, you can reduce the amount of work necessary to schedule vacations, business trips, and other absences. Each absence is treated as a project with a mandatory-start alignment type. The person entering the information must only enter the person’s ID, the number of days out, the date and time the absence begins, the type of absence (vacation, sickness, etc.), and a brief description of the absence.

The project code is generated by concatenating the type of absence with a unique number, which in this case is the observation number in the VACATION data set. PROC CPM would not be able to distinguish between vacations without the unique number. This number may change for each execution of PROC CPM, but because the number is meaningless, it does not matter.

In addition the SCL code converts the number of days into hours in order to be compatible with the rest of the projects. The following SCL code can be used for this purpose:

```sas
FSEINIT: /* Executed once when FEDIT invoked */
  now=datetime();
  algntype='MS';
  return;
INIT: /* Executed as positioned on new obs */
MAIN: /* Executed when ENTER pressed */
  subproj=left(put(curobs(),4.));
  If mainproj='?' Then
    mainproj=showlist('SICK','VACATION','BUSINESS',
```
The CONTROL ENTER statement instructs SCL to execute the MAIN section when the user presses ENTER. Normally this only occurs if a field has been updated. The INIT section has no RETURN statement, so the MAIN section is executed as you move to a new observation. The SHOWLIST function automatically displays a window and lists the possible values of type of absence. When the user places the cursor on the desired option, the value is returned in the variable MAINPROJ.

EXECUTING PROC CPM

Once the data sets are ready, the user must select the option to run PROC CPM. It is more efficient to run PROC CPM once after all data sets are prepared.

This program begins with the data step described in the resource data set discussion. Next, the new PROJECTS data set is merged with the RELATE data set to bring in the successors, and the resource data set (RESIN) is created with one observation. At that point all data sets are ready, and PROC CPM can be executed. The results are stored in the SCHEDULE data set:

```sas
proc sort data=projects;
by project;
run;
proc sort data=cpm.relate;
by pre;
run;
data projects;
merge projects
  cpm.relate
  rename=(pre=project)
by project;
array people {&size} &namelist;
retain namestr n&namelist;
do k=1 to &size;
  if scan(namestr,k)=person then
    people(k)=1;
  else people(k)=0;
end;
run;
data resin;
length obstype $ 8
&namelist 3;
retain &namelist 1
obstypet 'reslevel';
effedate='01jan90:09:00'dt;
format effedate datetime13.;
informat effedate datetime13.;
rn;
/* next run PROC CPM */
```

RESULTS

Because successors were provided using multiple observations, use the COLLAPSE option to retain only one observation per project instead of one for every successor. Also, the output must be filtered to remove extra observations generated if you allow activity splitting, which this application does. If an activity is split, PROC CPM produces an observation containing the start and finish times for the entire subtask regardless of the splits. In addition, PROC CPM creates an observation for each split segment. The summary observation is helpful, but gets in the way when producing reports. An additional DATA step is run to delete this overall segment for split tasks.

At this point you can display the results in a variety of ways. One useful method is to create an FSEDIT screen similar to that of the PROJECTS data set but fill in the scheduled times. This way you can again see the associated predecessors and successors.

Many other useful facilities are described in the SAS/OR User's Guide. In particular, refer to PROC GANTT and PROC NETDRAW. In addition, the output can be displayed with PROC CALENDAR. The hardest part is getting the data into PROC CPM. Once CPM has executed, the data are in a form that is readily used by PROC GANTT and PROC CALENDAR. In the programming environment, it is also useful to print a daily schedule for each person. This is a simple DATA step application or even a PROC PRINT statement. In future releases of Version 6 of the SAS System, you will be able to save report forms using PROC REPORT.

A SUGGESTION

One feature that was not discussed is the LAG option of the SUCCESSOR statement. If you finish a project but must wait for approval from an outside source, you may find it helpful to use the LAG option to delay the time between the end of one job and the beginning of
another. You can associate a lag value with each successor, so you also need to add lag fields to the PROJECTS data entry window and process them just as you did the successor fields.

SUMMARY

The CPM procedure provides an extremely powerful tool for scheduling programming projects. By designing your own front-end menu and project entry system with SAS/AF and SAS/FSP software using SCL, you can meet your own needs easily without being limited by a finite set of designs. Easy and logical data entry is the key to making the CPM procedure work.

Although PROC CPM can work through tough scheduling problems, the user is still required to enter reasonable requirements. In any system, the data must be kept current, although PROC CPM warns you if the data are inconsistent.

SAS, SAS/OR, SAS/FSP, SAS/AF are registered trademarks of SAS Institute Inc., Cary, NC, USA.