A Relational Approach to SSG

Guidelines for More Maintainable SSG Skeletons

By Steve J. Martin

The Symbolic Stream Generator (SSG) is a powerful and flexible macro-writing tool for Unisys 1100/2200 computers. If you work with other operating systems (such as MVS or MS-DOS), you probably know that they provide a macro or proc facility as an inherent part of their job control language. The OS 1100 Exec does not provide such a facility and, were it not for SSG, this would be a major deficiency. With SSG, however, we have a macro facility far richer than most operating system-supplied ones.

This article provides some tips on writing maintainable SSG skeletons and using SSG to implement Third Normal Form (3NF) relations. We'll discuss how to use SSG to construct a quasi-relational database. This doesn't mean you can use SSG instead of a real relational database (such as Relational Data Management System [RDMS] or ORACLE) for major applications. Instead, you'll learn how you can incorporate principles of the relational data model into SSG skeletons.

This is not an argument for the value of the relational model. The arguments for and against this model are well-known. The strongest defense I have found is in E.F. Codd's *The Relational Model for Database Management: Version 2* (Addison-Wesley, 1990).

Example of 3NF Relations

The number of applications you can write using SSG is limited only by your imagination. I use it as part of jobs that allocate Transaction Processing (TIP) files, save and recover databases, compile programs, make security changes and perform many other technical support tasks that require "smart" job control.

In the rest of this article, we'll consider a trivial set of Stream Generation Statements (SGSs) that keep track of a system's disk packs. These SGSs might be part of a larger application that, for example, generates Executive Control Language (ECL) to catalog and reserve TIP files. For clarity and brevity, we'll keep things as simple as possible. (For the uninitiated, SGSs are symbolic files or elements containing parameters that an SSG skeleton program uses to generate output, control execution flow, etc.)

Consider the SGSs in Figure 1. The EQUIPMENT SGS describes the two types of disk equipment used at the site. This SGS has two fields. The first field contains the model number. The site has model 9720 and model 9494 disk drives. The second field indicates storage capacity in software tracks. A model 9494 disk drive holds 95,000 tracks.

The DISKPACK SGS describes the disk drives installed. This SGS contains five fields. The first field gives the physical device name by which the hardware complex knows the disk. The second field describes the model number of the disk.

You can see that this refers to the first field of the EQUIPMENT SGS. The third field describes the current status of the disk drive (up, down or reserved). The fourth field is the logical pack-id by which programmers know the pack. The fifth field is a flag indicating whether the disk drive is fixed or removable. (We'll explain the purpose of the backslashes later.)

You can probably think of other attributes of disk drives such as the prep factor and the amount of space currently in use. We've omitted these for ease of presentation.

Now that we've seen the actual values of the SGSs describing the disk drives on the system, let's generalize. The SGSs in Figure 1 implement two relations. The EQUIPMENT SGS implements an EQUIPMENT relation of degree two, uniquely identified by model number. The DISKPACK SGS implements a DISKPACK relation of degree five, uniquely identified by device name and supporting attributes for model number, pack status, logical pack-id and removability.

Thus, each SGS label implements a different relational table. The label of the SGS represents the name of the relation. Notice that the model number attribute of the DISKPACK relation is a ***foreign key***. This means that it relates a disk pack to another relation — in this case, the EQUIPMENT relation. For example, if we want to know the storage capacity of disk D01, we use its model number to find the appropriate EQUIPMENT entry.

FIGURE 1

SGSs to Describe Disk Drives

```
EQUIPMENT    9720    90000
EQUIPMENT    9494    95000

DISKPACK     D01     9720    UP    DRS001    FIX
DISKPACK     D02     9720    UP    DRS002    FIX
DISKPACK     D03     9720    UP    DRS003    FIX
DISKPACK     D04     9720    UP    REM005    REM
DISKPACK     D05     9494    UP    REM001    REM
DISKPACK     D06     9494    UP    REM002    REM
DISKPACK     D07     9494    UP    REM003    REM
DISKPACK     D08     9494    DN    \         \
```

FIGURE 2

MetaSGSs

```
RELATION EQUIPMENT    1    EMODEL     . model number
RELATION EQUIPMENT    2    TRACKS     . tracks
RELATION DISKPACK     1    DEVNAME    . device name
RELATION DISKPACK     2    DMODEL     . model number
RELATION DISKPACK     3    STATUS     . status
RELATION DISKPACK     4    PACKID     . logical pack-id
RELATION DISKPACK     5    FIXREM     . FIX or REM
```

FIGURE 3

Code to Process RELATION SGSs

```
*INCREMENT R TO [RELATION]
*SET [RELATION,R,3,1] = [RELATION,R,2,1]
*LOOP R
```

Guidelines for Using SSG

SSG is a special-purpose, high-level programming language. As such, we must use it in a consistent and disciplined manner if we are to write ***good*** skeletons (as opposed to those that merely work).

Here are three general guidelines for good SSG programming:

- Use SSG symbolically rather than positionally.
- Capture as much meaning as possible in the SGSs.
- Since SGSs act as the data files of SSG, they benefit from being in 3NF.

You can deny any of these guidelines while accepting the other two. Indeed, although I strongly support normalization, I occasionally violate it for performance reasons. In general, however, its advantages are great and I always start by normalizing my data. So, any departures from 3NF are carefully considered ones.

The following specific SSG tips provide ways you can apply the three general guidelines.

1) Do not make SGSs positional by row. It should not matter whether any particular row precedes or follows any other row in a set of SGSs. For example, we should not count on the first DISKPACK SGS to always be a fixed disk. Instead, we should include an attribute that describes whether or not a pack is fixed (as shown in Figure 1).

2) Do not use subfields. The basic SGS reference in an SSG skeleton is in the following form:

[label,occurrence,field,subfield]

On an SGS, you separate fields by spaces and separate subfields by commas. For example, in the SGSs in Figure 1, the value of [DISKPACK,3,4,1] is "DRS003."

There is a temptation to create SGSs that look something like this:

DISK 9720 UP FIX D01,DRS001

This SGS uses subfields in field 4 in order to group similar fields. The two subfields in field 4 represent pack names (physical and logical, respectively). This violates the fundamental relational tenet that relations must be two-dimensional (flat files). Rows and columns are all we need to represent two dimensions. The subfield provides a third dimension that is undesirable when implementing relational queries. In addition, always setting the subfield number to one simplifies all SGS references in our skeletons to:

[label,occurrence,field,1]

3) Use symbolic occurrence and field references. Our SGS references should look like "[DISKPACK,D,PACKID,1]" rather than "[DISKPACK,1,4,1]."

We should use symbolic references for the ***occurrence number*** and ***field number***. (We can use a numeric value for ***subfield number*** since it should always be one.) Making the occurrence number symbolic decreases the chance of using SGSs positionally. An SGS reference such as "[DISKPACK,1,4,1]" looks suspiciously like this particular occurrence has some special meaning not shown in the SGSs themselves. If certain entities do have special characteristics, we should add an attribute to the relation to capture this meaning, rather than rely on the position of occurrences within the relation.

. An even more serious problem with numeric field numbers is that they make the skeleton virtually unreadable and difficult to maintain. Would you rather encounter "[DISKPACK,D,4,1]" or "[DISKPACK,D,PACKID,1]?" The latter tells you the field's ***purpose***, while the former only tells you its ***location***. In addition, what would we do if we decide to drop a field no longer needed? With

numeric field numbers, we would have a lot of skeleton modifying and retesting to do, whereas with symbolic field references, it is a simple change.

4) Create a relational catalog to document SGSs. If we use symbolic field references, we must tell SSG their numeric value. We could do this by including a separate *SET command for each field of each SGS that a skeleton uses. For example:

```
*SET PACKID = 4
```

There is a serious drawback with this approach. We want to capture as much meaning as possible in the SGSs, but the *SET commands must occur in each skeleton that uses the SGSs. And these *SET commands must know what the SGSs look like.

A better way is to create a set of "metaSGSs" — SGSs that describe the fields in other SGSs. The RELATION SGSs shown in Figure 2 accomplish this.

The RELATION SGSs provide a symbolic field name for each attribute of a relation. For example, the first RELATION SGS in Figure 2 tells us the name of the first field of the EQUIPMENT SGS is "EMODEL." Because the model number attribute occurs in both relations, I added a qualifying letter prefix (E or D) to each of them. This is analogous to Structured Query Language (SQL) syntax such as "EQUIPMENT.MODEL" and "DISKPACK.MODEL."

Rather than include relation-specific *SET statements for each field a skeleton references, we can use generic code that processes all RELATION SGSs and submits a *SET statement for each. This trivial logic, presented in Figure 3, would be *COPY'd into each skeleton as part of its initialization code. It creates global numeric variables with the names and values specified by the RELATION SGSs. The value of these variables indicates the field position of the corresponding attribute in the SGS. This code is completely general in that it will work for any SGSs defined by RELATION SGSs. (Note that Figure 3 violates the guideline about avoiding numeric field references. This single copy procedure is the only place I do this.)

You can think of the RELATION SGS as the relational catalog or data dictionary. You might consider expanding it to include other information (such as integrity constraints).

5) Give every relation a unique identifier. The identifier may be a single field on the SGS or a combination of fields. One of the most important reasons for unique identifiers is that the primary key of one relation is often used as a foreign key in another relation. If duplicates were allowed, the foreign key references would be ambiguous.

6) Provide an explicit null. Often, an attribute takes a null value in some rows of a relation. For example, if a disk is down, it is considered neither fixed nor removable. Some SSG programmers handle this by omitting the null field. Since SGS references are positional, this only works if the field is the rightmost field (or rightmost subfield within a field). But this approach is unwieldy and becomes unworkable if several attributes in the same row can be null.

A better approach is to use an explicit character string to represent a null value. The example in Figure 1 uses a single backslash for this. It is very important to pick a combination of characters that will never occur naturally in your data.

7) Normalize your SGSs. The advantages and disadvantages of normalization are well-documented, as are the various techniques. In short, 3NF involves breaking repeating groups into separate relations (SGSs) and ensuring that each attribute depends on the primary key, the whole primary key and nothing but the primary key.

As well as being valuable in their own right, the first six guidelines presented here let you take advantage of normalized SGSs. If you do attempt to implement 3NF data structures with SSG, you will quickly realize one major limitation: SSG lacks tools to allow easy manipulation of relations. You will have to code loops and case structures that traverse your relations and you may be tempted to throw up your hands in disgust.

Why navigate through 3NF relations? By implementing 3NF relations now, you will have them when and if you acquire a better tool for manipulating them. Someday, you may want to load them into a commercial relational database. Or you may choose to write your own query language.

Conclusion

The next logical step in the development of a quasi-relational SSG is to write a set of general-purpose SSG subroutines that provide a very powerful set of relational operators — a kind of "SQL for SSG." I have developed prototype SSG subroutines that implement important relational and set operators such as select, project, join, union and intersection. I am also using 3NF SGSs as a medium to support ***simulation*** of UNIX pipelines on the 2200.

By implementing these types of tools in conjunction with the SSG programming guidelines described in this article, you can quickly develop extremely powerful SSG-based batch jobs that support, for example, automated operations. Even if you don't want to go this far, following these guidelines can lead to more maintainable SSG skeletons. A relational approach to SSG considerably enhances the usefulness of what you may have guessed is my favorite Unisys 1100/2200 software product.

ABOUT THE AUTHOR

Steve J. Martin is a private contractor providing technical support to League Data Ltd., a Unisys 2200 site in Halifax, Nova Scotia. Mr. Martin has 10 years of experience in the computer industry, using both Unisys and IBM mainframes.