**SESUG 2004**

# *Building Efficient State Transition Diagrams from Charges, Collection and Call Detail Transactional Feeds*

Brett C Peppe, Consultant, Redmond, WA

## ABSTRACT

This paper describes three efficient (constant memory, linear time) techniques to collect and process state information from transactional feeds of arbitrary size and complexity. These techniques are:

- Singular-State Vector Collector
- State-Transition (S,T) Collector
- Synthetic Key Collection and Lookup

Transactional systems are designed to track the state of individual customer interaction events as they occur. However, Marketing, Demand Analysis and Risk Portfolio groups often need to determine the impact or effect of these customer events over time. This implies that they are interested in tracking change between transactions, i.e. the set of transitions for each customer represents the opportunity or risk. *State Transition Diagrams are a powerful analysis paradigm for tracking the paths of change in or between transactional systems.*

## THE WILD WIDGET COMPANY HAS A PROBLEM

They are growing quickly and beginning to be mired in an ocean of operational data. They need to be able to estimate certain strategic trends in their customer care data, but the old techniques (spreadsheets, monthly reports) that they have used in the past are not working very well. They need a new a new approach to the problem. The Wild Widget Company (WWC) markets 5 varieties of widgets, which are mobile data driven devices with a wireless feed. The WWC is very successful and has grown to have a base of +8M customers. Among the many IT systems that they have built or purchased are Charges, Call Center and Collection systems. Customer billing by the WWC includes NRC, MRC and usage charges. Bills are sent out monthly. Customers can choose from three different monthly plans. 7x24 Call Centers field questions or problem reports from customers. A Collection system tracks if customer accounts are past-due and drives treatment, an inbound call router and an outbound auto-dialer to remind customers to bring their accounts current. Each of these systems has one or more transactional feeds.

Since the quality of Customer Care has become a strategic issue of great importance, the Senior VP of Marketing wants some research done to determine:

- How often are customers on an effective Price Plan?
- Loading on the Call center for accounts entering Treatment
- Time from Out-Call by the Auto-Dialer to In-Call by the Customer

## OUR SVP WANTS AN ANSWER ASAP- HOUSTON WE HAVE A PROBLEM

You point out that these are not simple questions to answer. He smiles and points out that you're the SAS wizard so go back to your office and cook up some magic *(wink)*. You leave to start planning your plan of analytic attack on this set of problems.

By reading the CHARGES table and aggregating monthly usage and comparing this to the break point on the currently selected Price Plan you can answer item #1. However, the CHARGES table is huge. Therefore, an in-memory approach like the following will fail because of insufficient memory to hold the BAN*Month structure:

```
Proc summary  data=BILLING.CHARGES( where=( chg_type = 'USG' ) ) nway;
    Class  BAN Month;
    Var    usg_min;
    Out   out=chg_0( drop= _type_ keep=BAN month sum_min) sum(usg_min)=sum_min;
```

Perhaps using a BY GROUP to process one Billing Account Number (BAN) at a time like this will help:

```
Proc summary  data=BILLING.CHARGES( where=( chg_type = 'USG' ) ) nway;
    By     BAN;
    Class  Month;
    Var    usg_min;
    Out   out=chg_0( drop= _type_ keep=BAN month sum_min) sum(usg_min)=sum_min;
```

However, the BILLING.CHARGES table does not have a Month field and the primary index is on Account (BAN) and entry sequence number. Usually the transactions are in chronological order, but not always. So, some months may appear out of order. Furthermore, due to the huge size of the table (+300M rows), sorting is difficult. Even if the BILLING.CHARGES table had appropriate indexes and fields in place, we still would need to collect this aggregation relative to price plan in the order that it occurred. *Sigh….*

You push the BILLING.CHARGES analysis aside and go on to the Collections/Call Center problem. *Oh no*, the Collection system operates *completely independently* of the Call Centers. The reason is that the WWC purchased these systems from different vendors after going through a Buy versus Build decision. Therefore, that the *only information* that these two system have in common is the Billing Account Number of the customer.

A brief assessment of the Collection system turns up the following possibly useful transactional datasets:

- Short Message Service (SMS) account reminder list
- Out-Call messages list for the Auto-Dialer
- Changes in account Treatment Status (TREATLOG)

After some thought and exploratory data analysis, you reason that the SMS reminders are sent during the grace period; therefore these accounts are not yet in treatment. Likewise, Out-Call message lists simply drive the auto-dialer. Therefore, scanning the Collection transactional table (TREATLOG) for accounts Entering Treatment (EV) and Exiting Treatment (ET) is the way to go.

The WWC has two separate Call Centers, one for Financial Care operations (past due accounts) and one for Customer Care operations (current accounts). Both accept feeds from the centralized Collection system. The only synchronous key between them is the BAN. When, or even if an account enters Treatment, is completely independent of when they contact the Call Center. This is because they may call for any number of reasons, only one of which is responding to treatment out-calls. So, no explicit merge key exists between the two systems. *Therefore, a mapped or synthetic lookup key will have to be manufactured*.

Each Call Center has several transactional data sources, but the best for our purposes is the Call Detail Record (CDR) which records one row per customer call. We will recover the Date-Time and Billing Account number from the call that is most recent after the change in Treatment.

Now we have been stopped on both fronts by problems associated with the efficient collection, analysis and lookup of state from large transactional tables. What we need to accomplish these tasks are SAS techniques that operate on constant memory in linear time. ***What we need is some SAS Judo that uses an inherent characteristic of these transactional systems to help us solve (throw to the mat) the problem space*.**

## MOST BUSINESS DATA IS TRANSACTIONAL

Transactional systems are a fact of life today in any business of significant size. As the name implies, the basic unit of information in these operational systems is a transaction. In general, most customer interaction events drive new records into the transactional database, but some events may update existing rows. The information captured on the transaction record reflects the design requirements of the particular business system. For example, customer x was sent a bill on date y or customer x was called on date z. In general, each transaction only captures the information (or state) of a single customer interaction event. In smaller companies, these tables are manageable. As a business grows larger, these tables can grow to hundreds of millions of rows so analytical efficiency becomes a must-have.

## SAS HAS MANY METHODS TO ANALYZE DATA ON SPECIFIC TRANSACTIONS

PROC FREQ is the classic technique to determine the observed distribution of values or classes within a table. In a similar fashion, PROC MEANS or SUMMARY are often used to aggregate transactions within classes and report statistics such as the mean and variance for numeric variables. However, these procedures are designed to inspect, analyze and report *on values in specific rows*.

## WE NEED TO KNOW DATA BETWEEN TRANSACTIONS

What happens when the analysis requires determining the *value transitions between transactions in the order that they occurred?* This type of longitudinal analysis is often required by Marketing, Demand Analysis or Risk Portfolio groups who need to know how the account arrived at a particular state. However, SAS does not provide any convenient procedures to efficiently collect between row statistics.

## TRANSACTIONAL GOTCHA'S TO WATCH FOR

There are several issues to keep in mind when attempting to analyze state between transactions including the following:

- Indexes
- Lack of explicit analysis keys
- Different field definitions in separate schemas
- Artificial or artifact Classes
- Discontinuous state
- Work-in-Progress (WIP) items
- Invalid Items
- Back-out items
- Censored state series
- Effective Date range on coded classes
- Complex logic is difficult to understand, code and test

*Transactional databases are usually organized around transactions, not accounts*. So as a rule of thumb, any primary indexes on these feeds have been created to support the application itself, not an analysis of its account history. Indexes facilitate finding individual records efficiently; not tracking state between them.

*Transactions in separate tables may not share an explicit key.* At one time, this problem was less common because many businesses created and maintained their own infrastructure software. However, in today's competitive environment, businesses often mix and match their own custom software with that from external vendors. This may be good for fixed expense reduction in the short term, but it sometimes costs dearly in later integration woes. The missing cross system key is a very common problem in the modern business environment, when a mix of internally created and external purchased software is present. One way deal with this issue is to create a synthetic merge or lookup key that maps one system onto another.

*Transactions in separate schemas may have different field definitions.* This is a far more common problem then you may think. Just because two fields have the same name does not guarantee that they are declared and used the same. In a similar fashion, these fields may not share the same level of referential or contextual integrity.

*Transactions often record measurements at a level different then what is needed in the analysis*. When it is necessary to use a classifier to create sub and super classes, one must take care not to create artificial or artifact classes.

*Transactions may exhibit discontinuous state due to types*. One should not write code that depends on a particular measurement state being present on every row. Often, transactional tables are used for several purposes within a given subsystem by assigning types to transactions. For example, transactional records of type A may not reflect state on those of type B.

*Transactions may reflect partially complete or Work-in-Progress items*. These transactions are by definition incomplete and great care should be taken to carefully decide whether or not they belong in the tracked state.

*Transactions may contain Invalid values*. Errors do happen and it is a good idea to run a few scans to determine the real distribution of the states in the transactional feed. Act like you are from Missouri. Hopefully any errors are not problematic.

*Transactions may include Back-out items.* Some transactions exist solely to back-out or negate a previous transaction, usually via an aggregation operation like a sum(). For example, in Payments and Adjustments tables this is standard practice. Back-outs usually share the same Entry Sequence numbers of the master transaction, but different Activity Sequence numbers. Back-outs can be thought as nested transactions.

*Transactions in a series may be censored.* When state is being harvested across many transactions it is prudent to check if the entire series (path) is present. For example, a sequence of transactional states may be missing a starting state (left censoring) or a ending state (right censoring). Intermediate missing states are more difficult to detect.

*Transactions often contain coded values that are only valid within certain effective date ranges.* Make sure that the coded values for you analysis are within their effective date ranges.

*Transactional state recovery may require complex many valued logic*. Often SAS programmers start writing mazes of IF-THEN-ELSE or SELECT blocks without completely mapping the rule space. This is always a mistake.

## SO, EXACTLY WHAT IS A TRANSACTION?

A transaction is a database record that has the properties of *Serialization, Association and Completeness*.

*Serialization* means that the application order of the transactions is known for any customer. Usually this is accomplished via a unique entry sequence number or date-time (possibly both) that is stamped on every record. Businesses use the serialization property for controlling processing. Customers use this serialization property as an inquiry reference to a particular event.

*Association* means that each transaction describes the interaction with one and only one customer. Sometimes this customer association can be part of a larger hierarchy. As long as the customer level remains constant, the analysis is unaffected. Aggregate analysis performed on such a hierarchy is known as a rollup analysis.

*Completeness* means that each transaction represents an entire interaction event. A completed transaction is further described by the acronym ACID, where the letters stand for the following:

- Atomicity Each transaction describes a single or atomic unit of work
- Consistency All Transactions of a particular type always execute the same
- Isolation One transaction has no effect on other transactions
- Durability During a system failure, transactions are not damaged

***Since the purpose of a transactional system is to record the state of a customer interaction at a particular moment in time, then the set of transactions for a particular customer represents the interaction history of the business with that customer.***

If the number of transactions is relatively small (thousands of rows), then simple file viewers, spreadsheets and close attention to detail are often adequate to determine the effects between transactions. However, for businesses of any significant size, the transactional feeds can become quite large. If we choose methods that only require constant memory and run in linear time, then our transactional scans will easily scale from medium sized feeds (~1M rows) to very large feeds (hundreds of millions of rows).

## STATE TRANSITION DIAGRAMS - AN EASY WAY TO VISUALIZE PATHS OF CHANGE

If a customer account is active, then the tracking state will change over time. A customer account can also be thought of as an object whose state changes. An effective way to represent these sequential changes of state is to report them as graphs, in either visual or tabular form. Graph theory has evolved to answer a variety of state-path problems including connectedness, minimum cost, maximum flow, path prediction and optimization. Depending on the discipline, business and continent that you happen to be in, this technology is described by a variety of names such as:

- State Transition Diagrams (STD)
- Labeled Transition Systems (LTS)
- Petri Nets
- State Charts (UML)
- Activity Diagrams (UML)
- Markov Chains
- Finite State Machines (Mealy FSM, Moore FSM)

A state is the set of values that describe an object at some particular moment in time. For example, a customer account at time x is either current or past due. When an event occurs that causes an object to change its state, the change is known as a path or transition. It is common to assume that the time required to change state of an account is relatively insignificant relative to the total path time itself.

Transactional feeds reflect the needs of their designs, but our analysis may require the determination of a super-state or sub-state not explicitly stored. When this occurs, classifiers can be used to define new valid states of the object solely for the purposes of an analysis. ***Therefore, the state of an object is related to what you are attempting to measure and the classifier that is applied.***

So, it is all about objects, states, paths and events. Transactional business systems track account state *after* the application of events. For example bill production, payment, etc. Some of the values represent continuous

behavior, others discrete behavior. ***The key concept is that any customer account has a set of serialized state-paths that can be analyzed.***

In this paper we shall use State Transition Diagrams (STD) and State Vectors, which are very convenient abstractions for any business process. A State Vector is a specialized type of STD that merely contains the linear history of a singular state.

## GRAPH THEORY IS AN OLD FRIEND IN NEW CLOTHES

The idea of analysis via graph inspection is an old one in statistics. For example, the use of scatter diagrams in correlation studies or Pareto diagrams in cause-effect studies. A State Transition Diagram (STD) is a graph, where each vertex (node) represents a state and each edge (arc) a transition caused by an action or event. It is traditional to label the initial state as zero (0) and subsequent states as 1,2,3,…,n as needed. Each of the transitions is often labeled as well. To build a graph model of an object system, one needs to know the non-empty set of pairs **(S,T)** where:

**S** is the set of *valid states* of the object
**T** is the set of *valid transitions* between these states

The state of whatever we are interested in tracking is known as **S.** Each state may contain any number of values (numeric, character, date, etc.) as is required. The **T** in each pair is the definition of the transition itself, which can be a defined value (discrete behavior) or a transformation (classifier operating on a set of values). Sometimes this notation is written as $\mathbf{(S_f,S_t)}$ where $\mathbf{S_f}$ is the State (from#) and $\mathbf{S_t}$ is the State (To#) for the Transition. Both of these notations are equivalent for the application discussed in this paper if the set of transitions is unique.

In simple systems, STD diagrams are convenient to draw as real graphs, but most systems are better suited to a tabular equivalent. Therefore, for most STD that we should expect to encounter, the tabular form is preferred and the one that we shall use in this paper. Tabular STD representations have several key properties, including that they are:

- Searchable
- Self-Documenting
- Self-Verifying
- Self-Validating
- Extremely Flexible
- Expandable
- Additive
- Map-able

A tabular STD is *searchable*. As the process acquires more rules, the ability to determine if a particular state-transformation exists becomes more cumbersome and error prone. SAS has a variety of convenient methods to search tables.

A tabular STD is *self-documenting*. The ability to clearly document logical or process definitions in a table is often critical to coding complex logic and testing it.

A tabular STD can be made *self-verifying* given the appropriate design. That is, a specialized program known as a Finite State Machine (FSM) can walk the table and determine between any two states:

- Connecticity How are they connected?
- Minimum Path What is the minimum path between them?
- Flow What are the available transition path characteristics?

A tabular STD can be made *self-validating*. That is, a program can use the tabular STD as a validation reference as it walks the transactional data to determine if all transactions observed are permitted.

A tabular STD is *extremely flexible*. The state can easily store any descriptive, summary or materialized data for the $\mathbf{(S_f,S_t)}$ transition table entry. Therefore it can handle arbitrary complexity (amount of detail) by adding new columns (fields).

A tabular STD is *expandable*. Because the tabular form accommodates additional (S,T) pairs by merely adding rows, it can handle an STD of arbitrary sophistication (#Rules).

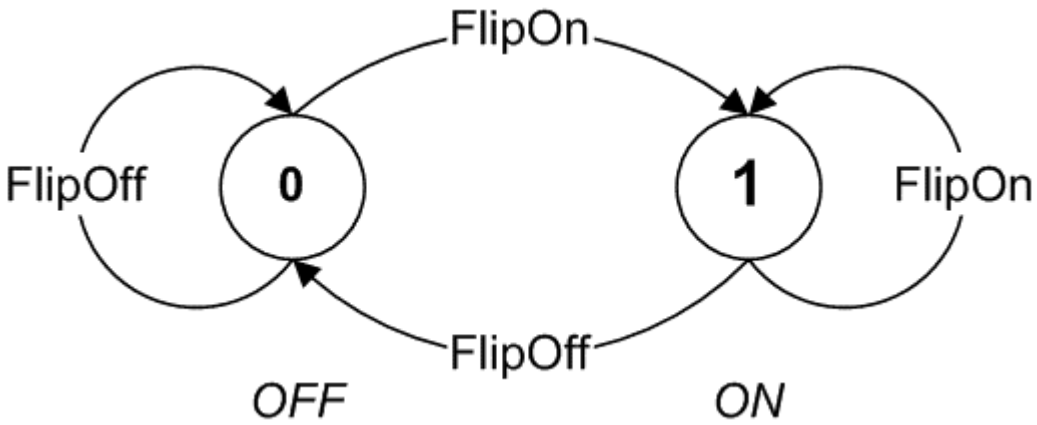
A tabular STD can be made *additive*. With care, an STD can be designed such that entries in separate tables can be combined if necessary. A specialized class of STD supports hierarchal state operations which are enumerated by Hierarchal State Machines (HSM). The properties of HSM on additive maps are beyond the scope of this paper.

A tabular STD is *map-able*. Tabular rows and columns conveniently map 1:1 to SAS observations and fields. There is one entry or row for every State-Transition. Arcs (or edges) on a STD graph correspond to entries in the equivalent STD table. Arcs have a defined domain set of values, the names of the transitions. Nodes have a defined domain set of values, the names of the states. Each of these can have further validation rules as required.

***All the techniques presented in this paper run in linear time and use constant memory***. Computer scientists call this O(n) behavior. As the size of the transactional feeds increases, this fact becomes more important. Relational databases provide a number of SQL based techniques to accomplish the equivalent state processing, but they do not always run in linear time or in constant memory. The exception to this blanket statement occurs when the underlying relational table can be partitioned and the hardware platform supports multiple processors, but this requires the support of your local DBA. I have successfully used production hardened forms of the techniques reported in this paper on transactional feeds containing more than a billion rows and can report that *yes, they do run in linear time and more or less in constant memory*.

## STD EXAMPLE 1: A TWO POSITION LIGHT SWITCH

A classic example of a STD is a two position light switch. The behavior of this simple machine is finite, but the series of actions on it could be infinite (On → Off → On → Off → On …). For a two position light switch initially in the OFF (0) position, the STD graph looks like this:



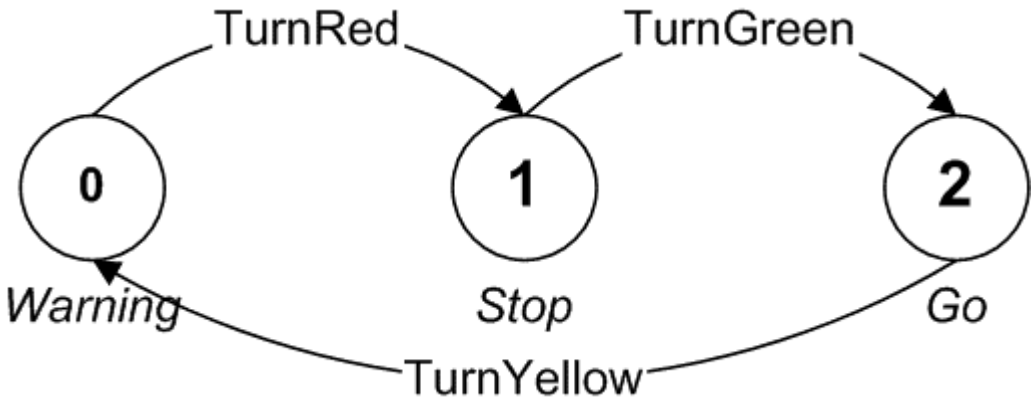

In this example, the two position light switch has two states, OFF (0) and ON (1), two actions (FlipOff, FlipOn) and four possible transitions as follows in the STD table:

| From (n) | S | T | To (n) |
|---|---|---|---|
| 0 | OFF | FlipOff | 0 |
| 0 | OFF | FlipOn | 1 |
| 1 | ON | FlipOn | 1 |
| 1 | ON | FlipOff | 0 |

Note that in this example, only two of the paths result in a change in state. The event (or action) that drives the change in state is someone playing with the light switch. If the From (n) = To (n) in a tabular STD, then this is equivalent to cycles in the graph where the state of the object did not change. Sometimes, cycles like these are called NULL transitions. The graph and the tabular form are equivalent representations of system behavior and represent all valid states, actions and transitions of the two position light switch system.

## STD EXAMPLE 2: A TRAFFIC LIGHT

Here is a STD graph of a Traffic Light, that has three states (Warning, Stop, Go) and whose initial state is Warning (0):




In this example, the Traffic Light has three states, Warning (0-Yellow), Stop (1-Red), and Go (2-Green) and three valid transitions as follows in the STD table:

| From (n) | S | T | To (n) |
|---|---|---|---|
| 0 | Warning | TurnRed | 1 |
| 1 | Stop | TurnGreen | 2 |
| 2 | Go | TurnYellow | 0 |

Note that in this example, all three transitions result in a change in state. The event that drives the change in state is timing, pedestrian cross-walk requests or car detection in an intersection lane.

## HOLD THE COMPUTER SCIENCE, I NEED A PRACTICAL METHOD

*Hey, no pain – no gain*. The previous two examples nicely illustrate the ability of the FSM/STD abstraction to represent complex event driven systems within a very small amount of memory. However, we need practical methods to process large transactional feeds, discover the state-transitions and their characteristic distributions. ***Standby Odyssey, the procedure is coming…***

Building a STD has four basic phases, which are the following:

- *Read* the transactional feeds in key, serialized order
- *Assemble* the STD data structure by collecting state
- *Reduce* the STD working set to its minimum representation
- *Perform* the required analysis

Remember that an STD is simply an abstraction, albeit a very powerful one. In its simplest form, it can help you organize your research and design efforts. ***In its sophisticated forms, it can serve as the procedural road map for a programmatic search of transactional state space which is a phenomenal thing to observe.*** *My program (or data) does what? – Absolutely, uh , oh no, ooops….*

Because SAS does not support calculation of between row statistics other then the LAGn() and DIFFn() functions, we must write some custom data step code. Once we have collected state, we usually need one or more of the following results:

- What are the Observed Transition Paths (Maximum/Average/Least Common)?
- What are the Characteristic Distributions (Time, #Steps)?
- What is the state of Customer x from feed A in feed B at time y?

## TECHNIQUE (1): SINGULAR STATE VECTOR COLLECTOR

SAS provides a slick way to stash values of the same type across data step cycles with very little fuss called a _TEMPORARY_ array. Elements of a _TEMPORARY_ array have many useful features including:

- They retain their values across a data step cycle, without a RETAIN statement
- They do not need explicit variable names
- They are not written to the output dataset
- They are fast and efficient to initialize, assign and reference
- You can have as many _TEMPORARY_ arrays as you have available pool memory

Let us assume that we are scanning a dataset that is sorted on some <Customer ID> in ascending <Sequence #> order. If this is the case, then we can use the FIRST (dot) and LAST (dot) data step directives to easily identify the initial and final transactions for a particular customer. The game plan is to:

- Initialize data structures on encountering a new account
- Store state as it is discovered in successive elements of a _TEMPORARY_ array
- Output the collected state map after processing the last row of an account

Here is a basic data step template for harvesting state from a transactional feed using a _TEMPORARY_ array given the following definitions:

- MyRecoveredState is the output dataset containing a key and a simple STD map (xst:) as fields
- The recovered state vector (xst:) has at most &MAXSTLEN transformations
- <Customer ID> is the index key of the scanned table
- <State Changes> is the applied test to identify a change in state
- state_i_max variable is the #elements in the array state{*}
- Assume that CHARGES is accessible under libname=BILLING

```
%let  MAXSTLEN  =  %str(20);

data  work.MyRecoveredState( keep=<CustomerID> xst: );

    /*---- D/S DCL -------------------------------------------*/
    array  xstate{&MAXSTLEN}   xst1-xst&MAXSTLEN; /* Persistent */
    array   state{&MAXSTLEN}   _temporary_;       /* Temporary  */
    retain  state_i_max   0 ;

    /*---- D/S Input Processing ------------------------------*/
    set  BILLING.CHARGES;
    by  <Customer ID>;

    /*---- Initialize on New Account -------------------------*/
    if ( first.<Customer ID> ) then do;
          do i = 1 to dim( state );          /* Reset to NULL */
                state{i} = .;
          end;
          state_i_max = 0;                   /* Reset to Zero */
    end;

    /*----- Capture State on Change --------------------------*/
    if ( <State Changes> ) then do;
          if ( state_i_max < dim(state) ) then state_i_max +1;
          else do;
                put "*** &MAXSTLEN is TOO SMALL";
                abort abend;
          end;
          state{state_i_max} = <new state>;
    end;

    /*----- On Last Row, Transfer Temporary to Persistent, Write--*/
    if ( last.<Customer ID> ) then do;
          do i = 1 to dim( state );
                xstate{i} = state{i};
          end;
          output;
    end;
return;
```

***Now this code snippet harvests transactional state across many records and collapses it into a singular state vector (XSTATEn) on a single observation.***

The ***advantages*** of the _TEMPORARY_ array method are speed and flexibility on large transactional feeds:

- The speed of this code derives from the fact that it has a *single pass* design.

- The flexibility derives from the ability of the programmer to *track anything* across transactions.

The ***disadvantages*** of the TEMPORARY_ array method are design issues:

- The programmer *must correctly estimate the maximum size* of the states (&MAXSTLEN).
- *Specialized logic is required* to encode, recover and analyze collected state.

Now the coding template demonstrated above only collects a singular state vector for each account during the data pass (XSTATE), but there could any number of these singular vectors on the observation. This is a convenient technique when:

- The number of states is known before hand
- The Order of states must be preserved
- The State Vector is the analysis work product

## TECHNIQUE (2): STATE-TRANSITION (S,T) COLLECTOR

Often we need to build an STD from the observed values in a large, complex transactional feed in such a way to facilitate analysis with standard SAS procedures. The best way to do this is to create an output dataset as changes in state are indentifed. There are a number of advantages to storing State-Transitions in a SAS datasets because the output dataset is:

- Convenient for later analysis using standard SAS tools like FREQ and MEANS.
- Requires no specialized logic in the data step
- Eliminates the needs for guessing the maximum number of State-Transition pairs
- Easy to MERGE with other SAS datasets
- Can easily hold any number of state values, summaries, etc.

Before we see an application of this method to the BILLING.CHARGES problem, let us review a simpler demonstration of this technique. First, let us build a simulation that creates a randomly generated list of Charge Dates (CHG_DT) and Charge Minutes (CHG_MIN) within a given calendar year (2003) to practice on.

```
data  d_0( keep = chg_dt chg_min );
  attrib  chg_dt   format=YYMMDDD10. ;
  attrib  chg_min  format=8.2        ;
  do chg_dt = '01JAN2003'D to '31DEC2003'D by 1;
    if ( ranuni(0) < 0.70 ) then do;
      chg_min = 24*ranuni(0);
      output;
    end;
  end;
stop;
```

```
NOTE: The data set WORK.D_0 has 250 observations and 2 variables.
NOTE: DATA statement used:
      real time          0.03 seconds
      cpu time           0.01 seconds
```

Next, let us use some SAS judo to read this transactional simulation and in a single data pass, aggregate minutes within a calendar month. The problem that we have here is the same one that our analysis of the BILLING.CHARGES has, no month variable on the transactional feed. The difference is that this dataset is small enough to sort, but we will process it in entry order by using the relative array offset capability. Stated simply, SAS gives us the ability to defined arrays using absolute indices (1,2,…,11,12) or relative ranges (MLOW:MHIGH). This is an interesting feature of arrays that can be used a classifier or sorter in many problems. Our objective in this example is to read the entire list, count the number of transactions, sum the monthly minutes and print them out.

```
%let  MLOW  =  %str(1) ;   /*   January, first month of Year  */
%let  MHIGH =  %str(12);   /*  December, last month  of Year  */

Data  _null_;
  array  msum{&MLOW:&MHIGH} _temporary_ ;  /*  Monthly Sums    */
  array  mcnt{&MLOW:&MHIGH} _temporary_ ;  /*  Monthly Counts  */
  set  d_0  end=lastobs;
  if ( _n_ = 1 ) then do;
    do i = 1 to dim(msum);
```

```
        msum{i} = 0;
        mcnt{i} = 0;
      end;
    end;
    msum{ month(chg_dt) } + chg_min;     /* Accumulate Minutes  */
    mcnt{ month(chg_dt) } + 1;           /* Accumulate Count    */
    if ( not lastobs ) then return;
    do i = 1 to dim(msum);
      put mcnt{i}=3.  msum{i}=8.2 ;
    end;
  stop;
```

```
mcnt[1]=22  msum[1]=191.33
mcnt[2]=18  msum[2]=199.99
mcnt[3]=20  msum[3]=229.44
mcnt[4]=22  msum[4]=271.39
mcnt[5]=17  msum[5]=199.30
mcnt[6]=21  msum[6]=300.89
mcnt[7]=21  msum[7]=212.22
mcnt[8]=25  msum[8]=359.67
mcnt[9]=25  msum[9]=275.24
mcnt[10]=20 msum[10]=249.41
mcnt[11]=18 msum[11]=210.61
mcnt[12]=21 msum[12]=217.65
NOTE: There were 250 observations read from the data set WORK.D_0.
NOTE: DATA statement used:
      real time          0.02 seconds
      cpu time           0.02 second
```

***This example demonstrates the essence of SAS judo:***

- Pass the data feed only once
- Use in-memory structures that are defined at compile time
- Use memory-to-memory operations whenever possible
- Allow the Access engine to optimize blocks (READBUFF=nnn) by reading in a linear fashion
- Restrict slower code to initialization and termination processing
- Only execute fast code in the majority of the data cycle steps
- Use the structure of the data to your advantage
- Use SAS functions to build classifiers
- Minimize the number of instructions executed during the data step cycle

Like in judo, in which competitors attempt to use leverage, momentum and gravity against their opponents,the appropriate use of classifier functions and a state harvester has turned an ugly problem space into one a simple solution. Instead of fighting the transactional series (gravity), we let it work for us. ***Result – problem #1 is headed for the mat, it just doesn't know it yet.***

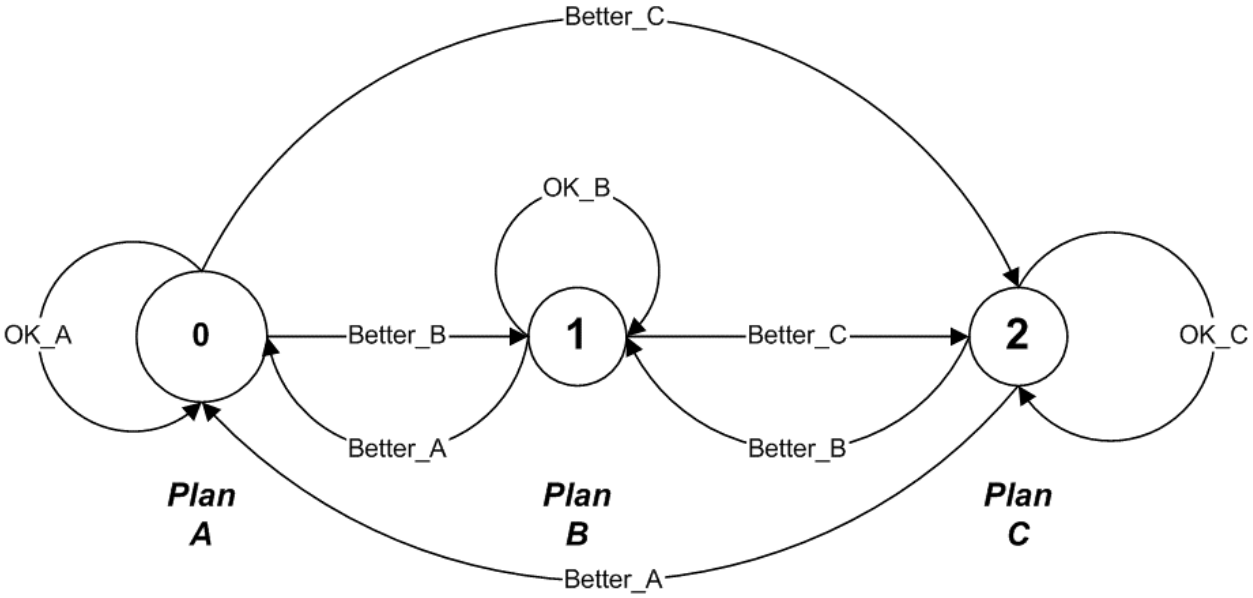
We need to collect monthly summaries for each account from BILLING.CHARGES for comparison with the billed price plan to determine if the customer was on a effective plan for their widget usage. Recall that this table is huge, does not have an explicit month field and is physically organized by account and entry sequence. Our task is to walk the transactions for each account, then write the price plan state map for their account history on the last entry. The previous example gives us an efficient technique for aggregating monthly usage; for the remainder we can use the State Vector Collector technique.

Lets us define a STD that illustrates the question asked by the Marketing SVP *"How often are customers on an effective Price Plan?"* I am going to make the table definitions simple in order to facilitate the demonstration. In business of this size, these tables might easily have hundreds of fields and rows each.

The PRICEPLAN table has only three rows as follows:

| PPlan | Low Usage(min) | High Usage(min) | Description |
|---|---|---|---|
| A | 0 | 100 | Budget |
| B | >100 | <400 | Monthly Value |
| C | >400 | (HIGH) | Unlimited Usage |

If one implemented the Price Plan classifier as a SAS format clause, then by supplying a summed monthly value, the format would return a Price Plan. This value could be compared to the chosen customer price plan and a decision made if the customer is on the most effective plan. Before we cut the code, let us take a look at the transition paths that a customer account may show:




The tabular STD for the Price Plan looks like this:

| From (n) | S | T | To (n) |
|---|---|---|---|
| 0 | Plan A | OK_A | 0 |
| 0 | Plan A | Better_B | 1 |
| 0 | Plan A | Better_C | 2 |
| 1 | Plan B | OK_B | 1 |
| 1 | Plan B | Better_A | 0 |
| 1 | Plan B | Better_C | 2 |
| 2 | Plan C | OK_C | 2 |
| 2 | Plan C | Better_A | 0 |
| 2 | Plan C | Better_B | 1 |

In order to help us, let us build a simulation that creates data for roughly 50K accounts and use this for our demonstration.

- Study starts in January 2001 and Ends in December 2003
- Roughly 50K accounts (bans 10K – 60K)
- No churn, i.e. accounts are in-service for the entire study period
- Accounts do not change their Price Plan during the study
- 65% Budget Plan (A)
- 20% Value Plan (B)
- 15% Unlimited Plan (C)
- Bills are produced for each month with widget usage
- The Budget Plan (A) has a mean of 78 min/month and a daily usage probability of 0.35
- The Value Plan (B) has a mean of 176 min/month and a daily usage probability of 0.58
- The Unlimited Plan (C) has a mean of 274 min/month and a daily usage probability of 0.64

```
%let  YLO      =  %str(2001);
%let  YHI      =  %str(2003);
%let  BAN_BEG  =  %str(10000);
%let  BAN_END  =  %str(60000);

data     bill_0( keep= ban ccyymm pplan min_sum index=( banmth=(ban ccyymm) ) )
      charges_0( keep = ban chg_dt chg_min );
```

```
  attrib  ban      format=8.         ;
  attrib  ccyymm   format=$6.        ;
  attrib  chg_dt   format=YYMMDDD10. ;
  attrib  min_sum  format=8.2        ;
  attrib  chg_min  format=8.2        ;
  do ban = &BAN_BEG to &BAN_END by 1;
    pplan_rdm = ranuni(0);
    select;
      when( pplan_rdm < 0.65 ) pplan = 'A'; /* 65% A */
      when( pplan_rdm < 0.85 ) pplan = 'B'; /* 20% B */
      otherwise                pplan = 'C'; /* 15% C */
    end;
    do yy = &YLO to &YHI by 1;
      do mm = 1 to 12 by 1;
        min_sum = 0;
        do dd = 1 to 28 by 1;
           use_rdm = ranuni(0);
           chg_dt  = mdy(mm,dd,yy);
           select;
             when( (pplan='A') & (use_rdm < 0.35)) grpmn =  78;
             when( (pplan='B') & (use_rdm < 0.58)) grpmn = 176;
             when( (pplan='C') & (use_rdm < 0.64)) grpmn = 274;
             otherwise                             grpmn = .  ;
           end;
           if ( grpmn ) then do;
             chg_min =  ( grpmn * ( 1 + 1.9*rannor(0) ) ) / 30;
             if ( chg_min > 1 ) then do;
               min_sum + chg_min;
               output charges_0;
             end;
           end;
        end;
        ccyymm = put( mdy(mm,1,yy), YYMMN6. );
        output  bill_0;
      end;
    end;
  end;
stop;

NOTE: The data set WORK.BILL_0 has 1800036 observations and 4 variables.
NOTE: The data set WORK.CHARGES_0 has 14391537 observations and 3 variables.
NOTE: DATA statement used:
      real time          2:48.99
      cpu time           1:53.16
```

So, the simulation above has created 1,800,036 records in a BILL table and 14,391,537 records in a CHARGES table. Remember, this is just a simulation to demonstrate a technique. As such, some issues (Plan Change, Churn) have been deliberately left out. Likewise, some of the statistical movement is deliberately exaggerated to make the point clear. The MIN_SUM field in the BILL table was added so that, should you decide to copy and run this code, you can see for yourself that the aggregator shown below does work as designed.

When we do distribution analysis on the simulated BILL table we get the following:

```
  proc freq  data=bill_0;
     tables  pplan  / list missing ;

NOTE: There were 1800036 observations read from the data set WORK.BILL_0.
NOTE: PROCEDURE FREQ used:
      real time          1.65 seconds
      cpu time           0.62 seconds

                                          Cumulative    Cumulative
       pplan    Frequency     Percent     Frequency      Percent
       ---------------------------------------------------------
```

```
        A         1169604       64.98       1169604        64.98
        B          362520       20.14       1532124        85.12
        C          267912       14.88       1800036       100.00
```

We will now process the BILL.CHARGES table in order to decide which price plan the customer was on and determine the Effective Price Plan (EPPLAN) according the cutoff values stated earlier.

```
   data std_0( keep= ban ccyymm pplan epplan min_sum amth_: );

     /*---- D/S DCL -------------------------------------*/
     array  msum{&YLO:&YHI,1:12} _temporary_ ;
     array  mcnt{&YLO:&YHI,1:12} _temporary_ ;
     array   mdt{&YLO:&YHI,1:12} $ 6 _temporary_ ;
     attrib  amth_min  format=8.2  ;  /* Actual Minutes       */
     attrib  amth_cnt  format=3.   ;  /* Actual Counts        */
     attrib  epplan    format=$1.  ;  /* Effective Price Plan */
     attrib  pplan     format=$1.  ;  /* Chosen Price Plan    */

     /*---- D/S Input Processing ------------------------*/
     set  work.charges_0;
     by   ban;

     /*---- Initialize on New Account -------------------*/
     if ( first.ban ) then  do;
       do i = &YLO to &YHI;
         do j = 1 to 12;
           msum{i,j} = 0;
           mcnt{i,j} = 0;
           mcnt{i,j} = .;
         end;
       end;
     end;

     /*---- Aggregate Usage, Count and Date --------------*/
     msum{ year(chg_dt), month(chg_dt) } + chg_min;
     mcnt{ year(chg_dt), month(chg_dt) } + 1;
      mdt{ year(chg_dt), month(chg_dt) } = put( chg_dt, YYMMN6. );

     /*---- On Last Row in Account, Process --------------*/
     if ( last.ban ) then do;
       do i = &YLO to &YHI;
         do j = 1 to 12;
           if ( mdt{i,j} ne . ) then do;
             ccyymm = mdt{i,j};
             set bill_0 key=banmth;
             amth_min = msum{i,j};
             amth_cnt = mcnt{i,j};
             select;
               when( amth_min <= 100.0 ) epplan = 'A';
               when( amth_min <  200.0 ) epplan = 'B';
               when( amth_min >= 200.0 ) epplan = 'C';
               otherwise ;
             end;
             output;
           end;
        end;
       end;
     end;

     /*---- D/S Recycle Processing ----------------------*/
   return;
```

```
NOTE: The data set WORK.BILL_0 has 1800036 observations and 4 variables.
```

```
NOTE: The data set WORK.CHARGES_0 has 14391537 observations and 3 variables.
NOTE: DATA statement used:
      real time          2:48.99
      cpu time           1:53.16
```

Note that this State Transition Collector technique processed 14,391,537 CHARGE detail simulation records, looked up the monthly Price Plan state for every single month (1,800,036) in the study period in under three minutes. This performance is ~85K CHARGE details records per second on my laptop. **Assuming everything else is reasonably constant; this procedure should run on 300 Million CHARGE details rows in an hour.**

When we run a basic distribution on the State Transition Diagram (STD_0) we see the following:

```
proc freq  data=std_0;
   tables  pplan*epplan  / list missing;
```

| pplan | epplan | Frequency | Percent | Cumulative Frequency | Cumulative Percent |
|---|---|---|---|---|---|
| A | A | 1169459 | 64.97 | 1169459 | 64.97 |
| A | B | 145 | 0.01 | 1169604 | 64.98 |
| B | A | 86744 | 4.82 | 1256348 | 69.80 |
| B | B | 259045 | 14.39 | 1515393 | 84.19 |
| B | C | 16731 | 0.93 | 1532124 | 85.12 |
| C | A | 4579 | 0.25 | 1536703 | 85.37 |
| C | B | 97008 | 5.39 | 1633711 | 90.76 |
| C | C | 166325 | 9.24 | 1800036 | 100.00 |

***This aggregate STD answers the SVP's questions regarding Price Plan effectiveness on a monthly basis during the study period.***

For customers choosing the WWC Budget Plan (A), the account usage is almost totally (100%) within their plan:

- (A-A) The Chosen Plan was the Effective Plan almost 100%
- (A-B) Only a tiny number (145/1,169,604) of months did these accounts exceed their price plan

For customers choosing the WWC Value Plan (B), the account usage is usually (71%) within their plan:

- (B-A) Chosen Plan was too High in 23.9% of the months (86,744/362,520)
- (B-B) Chosen-Plan was the Effective-Plan in 71.5% of the months (259,045/362,520)
- (B-C) Chosen Plan was too Low in 4.6% of the months (16,731/362,520)

For customers choosing the WWC Unlimited Usage Plan (C), the account usage is usually (72%) within their plan:.

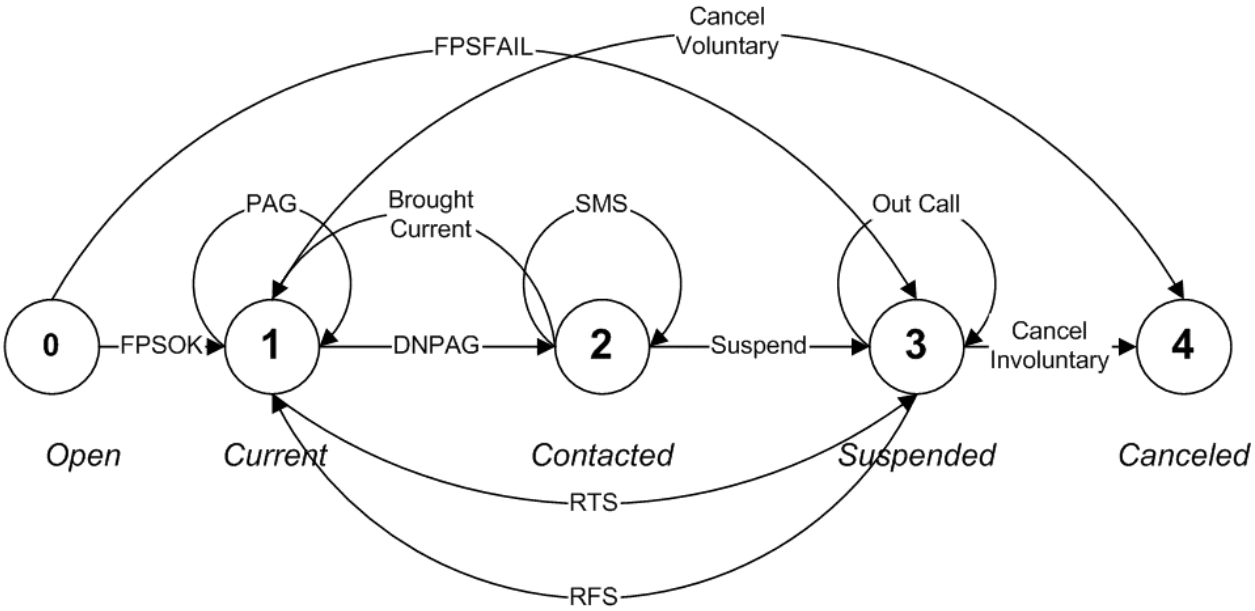
- (C-A) Chosen Plan was very High in 1.7% of the months (4,579/267,912)
- (C-B) Chosen Plan was too Low in 36.2% of the months (97,008/267,912)
- (C-C) Chosen-Plan was the Effective-Plan in 62.1% of the months (166,325/267,912)

Marketing knows that new customers are more expensive to acquire then retaining existing ones. From this study, it appears that:

- No mediation is required on behalf of the Budget Plan (A)
- 28.5% (23.9% + 4.6%) of the Value Plan (B) accounts are on a plan that does not reflect their needs
- 37.9% (1.7% + 36.2) of the Unlimited Plan (C) accounts are on a plan that does not reflect their needs

## TECHNIQUE (3): SYNTHETIC KEY COLLECTION AND LOOKUP

Here is a STD for a *simple* Collection system.




This example has only five states (Open, Current, Contacted, Suspended, Canceled) and twelve (12) transitions. Even though this STD has a relatively simple graph, it is becoming *visually crowded*. There is little room remaining for additional information. The tabular STD is just as concise as seen below.

| From (n) | S | T | To (n) |
|---|---|---|---|
| 0 | Open | FPSFAIL | 3 |
| 0 | Open | FPSOK | 1 |
| 1 | Current | PAG | 1 |
| 1 | Current | DNPAG | 2 |
| 1 | Current | RTS | 3 |
| 1 | Current | Cancel Voluntary | 4 |
| 2 | Contacted | SMS | 2 |
| 2 | Contacted | Brought Current | 1 |
| 2 | Contacted | Suspend | 3 |
| 3 | Suspended | Out Call | 3 |
| 3 | Suspended | RFS | 1 |
| 3 | Suspended | Cancel Involuntary | 4 |

A STD can easily represent a complex business process such as Collections. By mapping all valid transitions with an STD, such a diagram is invaluable because it can be used to create code that is:

- Self-Documenting
- Self-Verifying
- Self-Validating

Let us inspect an abbreviated set of the business rules that gave rise to this STD below:

When the customer first ***Opens (0)*** their account, there is always some risk that they will not pay their first bill. In collections processing, this concept is known as *First Payment Suspense*. The Collection system must react very quickly to suspend the account if the first payment is not received by the due date. So, the two transitions from the initial open state are FPSOK (customer paid by due date) and FPSFAIL (customer did not pay by due date). Therefore, there are two (2) transitions in the STD from state (0-Open).

If their account is ***Current (1)*** on the Due Date then it said to have been **Paid As Agreed (PAG)**. If a customer with has a current account and wants to ***Cancel*** their service, then they transition to Cancel (4) which is terminal state (i.e. no exiting arcs). If the account is past-due or **Did Not Pay as Agreed (DNPAG)** then it transitions to ***Contacted (2)***. Sometimes an account may have been suspended, called in a payment arrangement but subsequently defaulted. In such a case, they are **Return to Suspend (RTS)** status. Therefore, there are four (4) transitions in the STD from state (1-Current).

If the account is past-due, then the Collections system marks it to be ***Contacted (2)***. The form of this contact in this STD is to send an **SMS** message, wait a period of time (the grace period) and then mark the account as Entering Treatment. It is customary to allow a grace period during treatment, the details of which can become very complicated (another STD). This period of time usually depends on many factors including how the customer has paid their bill in the past. If during this grace period, the account ***brings their account current***, then they revert to state (1). However, if during this grace period they fail to bring their account current or default on an existing agreement, then they are **Suspended** and transition to state(3). Therefore, there are three (3) transitions in the STD from state (2-Contacted).

If the account has been suspended, then the Collection system begins generating **Out Call** records and letters to the customer. If while suspended, the account is brought current (**RTS or Return From Suspend**), then it reverts to state (1-Current). If the customer fails to make suitable arrangements, then eventually the account is **Canceled-Involuntary** and the terminal state (4) of ***Canceled*** is set. These accounts are referred to outside agencies.

Note that the STD describes this set of Collection business rules in a very concise fashion, so can serve as a part of the business process documentation.

The WWC auto-dialer uses different scripts based on the Treatment status of the account. Recall that the transactional feed from the COLLECTION system is *completely independent* of the Call Detail Record. So, to answer our SVP question we must link state in COLLECTION, determine the target Call Center to most recent calls in the CDR. The TREATLOG table that we have elected to use as our transactional baseline for Treatment status of an account has many fields, but we are only interested the Date that a Customer account Entered and Exited treatment. So, in order to accomplish this we only need the following fields from TREATLOG:

- BAN Billing Account Number
- COL_ACTV_DATE Collections Activity date-time of this transaction
- COL_ACTV_CODE Collections Activity Code of this transaction

For the Collections Activity code (COL_ACTV_CODE), the only state changes that we interested in are when (If) the account Enters Treatment (EV) or Exits Treatment (ET). Even though the COLLECTION system has many operational codes, we are only interested in this state map. Incoming customer calls are routed based on their treatment status, if they in Treatment then they are routed to Financial Care (FIN) who may want to talk with them regarding their account. Otherwise, the call is routed to Customer Care.

Note that if an account has never been Past-Due, then by definition they have never been in Treatment. So, this class of customer account will never appear in the TREATLOG table. Therefore, they will always be directed to Customer Care (CAR).

Assume that the TREATLOG table is opened under the COLN libname by a SAS Access engine. So, here is the code to scan the TREATLOG table and build a State-Transition map for all accounts. The output datasets in this example only contain three fields (ban, from_dt, to_dt) to make the example more readable, but this technique can be used to bridge any number of fields to the CDR record for linkage and subsequent analysis.

```
/*===================================================================*
 *  [P01]    Build a List of Treatment Range records
 *-------------------------------------------------------------------*
 *
 * This procedure reads the TREATLOG table extract and builds two
 * datasets.
 *    work.LKPBASE
 *    work.LKPINDX
 *
 * The Lookup Base (LKPBASE) dataset contains one obs for each time
 * that the account was in treatment and a date range for the treatment
 * series.
 *    BAN        Billing Account Number
 *    FROM_DT    Date that Treatment was Entered (EV)
 *    TO_DT      Date that Treatment was Exited (ET)
 *
 * The Lookup Index (LKPINDX) dataset contains one obs for each account
 * that appeared in the TREATLOG table extract and a count of the
 * number of Treatment series (n_lkp).  This is an important point,
```

```
 * if a BAN appears in the CDR, but not in the TREATLOG table extract
 * then it will not return a row during a SET ... KEY=BAN lookup
 * operation.  When this happens, the _IORC_ will return a non-zero
 * value indicating a non-match condition.
 *    BAN        Billing Account Number
 *    N_LKP      Count of Treat series in TREATLOG table sample
 *
*=================================================================*
 */
 data  work.lkpbase( keep= ban from_dt to_dt  index=(ban) )
       work.lkpindx( keep= ban n_lkp          index=(ban) );

 /*---- DCL for output rcd ----------------------------------------*/
   attrib  ban       format=$9.       ;
   attrib  from_dt   format=YYMMDDN8. ;
   attrib  to_dt     format=YYMMDDN8. ;

 /*---- DCL for Counters ------------------------------------------*/
   retain  n_EV     0 ;           /*  #Entering Treatment (total)  */
   retain  n_ET     0 ;           /*  #Exiting Treatment (total)   */
   retain  IsTrt    0 ;           /*  IsTreated?( T | F )          */
   retain  n_lkp    0 ;           /*  Number of Lookups            */
   retain  from_dt  0 ;           /*  Treatment Begin Date         */

 /*---- D/S Input Processing --------------------------------------*/
   set COLN.TREATLOG( keep= ban col_actv_date col_actv_code );
   by  ban;

 /*---- Initialize on first row of new Account --------------------*/
   if ( first.ban ) then do;
     n_ET  = 0 ;                  /* RESET: Counters/Flags to NULL */
     n_EV  = 0 ;
     IsTrt = 0 ;
     n_lkp = 0 ;
     from_dt = input( substr(col_actv_date,1,10), MMDDYY10. );
   end;

 /*---- Track TREATLOG Event Counts -------------------------------*/
   select( col_actv_code );
     when( 'EV' ) n_EV + 1;
     when( 'ET' ) n_ET + 1;
     otherwise ;
   end;

 /*---- Track Entering Treatment ----------------------------------*/
   if ( col_actv_code = 'EV' ) then do;
     from_dt = input( substr(col_actv_date,1,10), MMDDYY10. );
     IsTrt = 1;
   end;

 /*---- Track Leaving Treatment -----------------------------------*/
   if ( ( col_actv_code = 'ET' ) or ( IsTrt and last.ban ) ) then do;
     to_dt = input( substr(col_actv_date,1,10), MMDDYY10. );
     output lkpbase;
     IsTrt = 0;
     n_lkp + 1;
   end;

 /*---- Track Leaving Treatment -----------------------------------*/
   if ( last.ban ) then  output lkpindx;

 /*---- D/S Recycle Processing ------------------------------------*/
return;
```

```
run;
```

OK, now let us process the CDR log transactions and assign the Treatment status to each account on the Call transaction. See the usefulness of Method (1) in recovering state captured and passed by Method (2).

```
/*=================================================================*
 *  [02]   Assign the Call Center to Each CDR Record
 *-----------------------------------------------------------------*
 *
 * This procedure walks the CDR list one obs at a time and assigns
 * the Call Center (FIN,CAR) depending on the status of the lookup
 * via the LKPBASE and LKPINDX table.  See step [01] for a description
 * of these tables.  In brief the cases are the following:
 *
 *   Assign   Description
 *   ______   ___________________________________________
 *    CAR     Not in the TREATLOG extract
 *    CAR     In the TREATLOG extract, NOT in Treatment
 *    FIN     In the TREATLOG extract, IN Treatment
 *
 *=================================================================*
 */

 data  work.cdrnew( keep= ban call_: );

  /*---- D/S PDV Definition ---------------------------------------*/
    attrib  ban          format=$9.       ;
    attrib  call_dt      format=yymmddn8. ;
    attrib  call_assign  format=$3.       ;

  /*---- D/S Misc DCL ---------------------------------*/
    retain n_lkp  0 ;
    array  beg{50}  _temporary_ ;
    array  end{50}  _temporary_ ;

  /*---- D/S Input Processing -------------------------------------*/
    set CALLCTR.CDRLOG;
    by  ban;

  /*---- Initialize on First Obs of New Account -------------------*/
    if ( first.ban ) then do;
      set  lkpindx  key=ban;
      if ( _IORC_ ne %sysrc(_SOK) ) then do;
        n_lkp   = 0;
        _error_ = 0;
      end;
      do  i = 1 to min( n_lkp, dim(beg) );
        set lkpbase  key=ban;
        beg{i} =  from_dt ;
        end{i} =  to_dt   ;
      end;
    end;

 /*---- Assign Call Center based on Lookup Results ---------------*/
   if ( n_lkp = 0 ) then call_assign = 'CAR';
   else do;
     do  i = 1 to n_lkp;
       if ( ( call_dt >= beg{i} ) & ( call_dt <= end{i}) ) then do;
         call_assign = 'FIN';
         leave;
       end;
       if ( call_assign = ' ' ) then  call_assign = 'CAR';
     end;
   end;
```

```
    /*---- D/S Recycle Processing ----------------------------------*/
    return;

  run;
```

This code-base reliably associates the desired information in the TREATLOG transactional feed to Call Center CDR by customer accounts. These code snippets could be expanded to provide enhanced functionality should the need arise. Now, since the information has been consolidated onto one row, the analysis requested by the Marketing SVP can be done fairly easily.

The time from Out-call to In-call is now just a simple subtraction:

```
Elap_days = from_dt – call_dt;
```

Calculation of any desired State-Transition (S,T) distribution by Call Center is just as easy:

```
Proc freq   data=<statedataset>  order=freq;
    By  call_assign;
    Tables   S*T / list missing;

Where S & T are whatever you choose…
```

## CONCLUSIONS

State Transition Diagrams (STD) are powerful abstractions for any programmer who has to deal with analyzing state in complex transactional feeds because they:

- Facilitate concise descriptions of business processes
- Aid in the development of complex many valued logic
- Are Self-Documenting
- Are Self-Verifying
- Are Self-Validating
- Are Efficient abstractions

From a practical point of view, the STD abstraction can be implemented via a variety of methods, several of which have been presented here. You need to decide which method is best for your particular situation. If you are only looking for a specialized set of states and you want to accomplish all the processing in a single data step, then choose Method (1). Otherwise, if the states or their transitions are complex, divided across multiple feeds or you want to use standard SAS in-record procedures on the collected state maps, then choose Method (2).

***I hope that this brief presentation on STD gives you new and useful tools for your programming toolkit.***

## CONTACT INFORMATION

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


Name: Brett C. Peppe
Email: brettpeppe@DASconsultants.com
Web: www.DASConsultants.com