Introduction to Modgen

by

Claude Charette

Claude.Charette@statcan.ca

Statistics Canada

Workshop for Modgen users

May 27th 2008

(Content revised by Chantal Hicks in May 2017)

www.statcan.gc.ca/spsd/Modgen.htm

Statistics Canada Statistique Canada

Canada

Table of contents

Introduction

Modgen was designed to facilitate microsimulation model programming. Its purpose is to remove as many obstacles to microsimulation model creation as possible. Some of these obstacles are:

- Interface programming
- Documentation
- Simulation engine programming
- Bilingualism

Modgen eliminates all these obstacles as it provides the interface and the simulation engine, is bilingual and facilitates documentation. Obviously, this list is not exhaustive and is only given as an example.

One benefit of using Modgen to create microsimulation models is that there is no need to hire programmers to program them. Modgen takes care of most of the programming. What remains is usually simple enough that a person with no advanced knowledge of programming can do it. Having in-depth knowledge of programming can even be detrimental since the developer then has to limit himself to a programming style that will be understood by the analyst responsible for the model.

Also, it is not necessary to understand in detail what Modgen does in order to be able to use it. However, it might be useful to have a general idea of what it does, and providing such information is the purpose of this document.

Modgen installation

There are two products, and thus two installation programs, associated with Modgen. These products are:

- Modgen Prerequisites 12
- Modgen 12

Here is what each of those products contains and a description of what they do.

Modgen Prerequisites 12

In general, Modgen Prerequisites 12 contains everything necessary for models to run on a machine. If a developer wants to distribute a model, model users must install Modgen Prerequisites 12 before installing the model. This product contains:

- Modgen model user licence
- Modgen model user documentation

  This documentation is of concern primarily to users of the model's interfaces. It does not give any model creation assistance. This documentation includes:

  - The Modgen 12 Guide to the Visual Interface
  - The Modgen Prerequisites 12 Release Notes

- Cleaner application

  This application cleans up the temporary files left on a machine when a model simulation fails. If a simulation ends successfully, no temporary files will remain. It is installed with Modgen Prerequisites 12 and is run in each new Windows session. It is also possible to run the application manually but you must avoid doing this when a simulation is in progress.

- Modgen automated server

  A component that allows other programs to read scenario parameter file records.

- Microsoft Prerequisites

  Some files provided by Microsoft that are needed to run Modgen models.

Modgen 12

In general, Modgen 12 contains everything needed for model development, although Visual Studio 2015 or 2017 is also required (but not included in Modgen 12!) Specifically, this product contains:

- Modgen prerequisites

- Modgen model creation licence
- Everything necessary to create new models:

  This includes the Modgen precompiler, the Modgen library and header files.
- Documentation for model developers:
  This documentation is of interest to model developers. It provides concrete assistance on model creation, syntax to be used, etc. It includes:

  - Modgen 12 Developer's Guide
  - Modgen 12 Release Notes
- Integration with Visual Studio capabilities

  ~~These capabilities make it easier to use Modgen 12 inside Microsoft Visual Studio 2008. They include:~~

  - ~~Automatic configuration of Visual Studio 2008 for Modgen~~
  - ~~Modgen toolbars~~
  - ~~New module creation wizards~~
  - ~~New model creation.~~

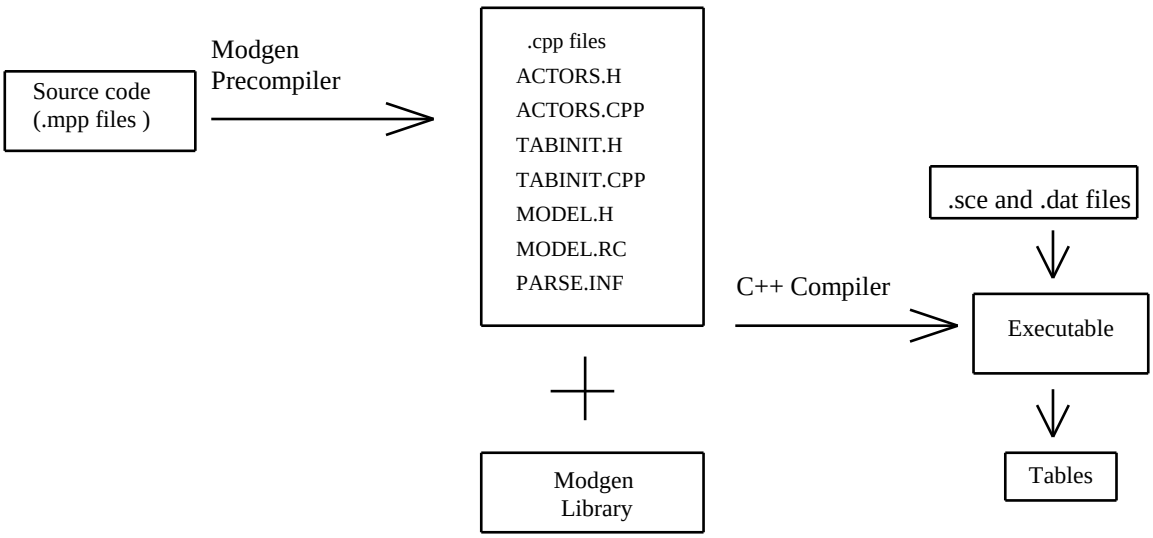
Modgen components

Three main components make up Modgen. These are:

- The Modgen precompiler
- The Modgen library
- The automated server

The first two components are used together to create models and are described in this document. The automated server, which makes it possible to access functions that read from and write to parameter files, will not be discussed here.

As a reminder, the chart below shows the links among the files needed to create executable simulation model files as part of an application. It also shows the inputs/outputs that executable model files need.



Note that Modgen is not independent from Visual Studio. Model developers must have the appropriate version of Visual Studio to be able to create models (Visual Studio 2015 or 2017 for Modgen 12).

Precompiler

The precompiler is the executable file "Modgen.exe". The Modgen language in which developers write models is an extension of the C++ language. Before being compiled, model code must be converted to C++ code. That is what the precompiler does. It reads the .mpp files containing the model code and creates equivalent .cpp files, along with special files needed for model creation. Here is a more detailed description of what Modgen does at the precompilation stage:

- Checks that the syntax is correct for every Modgen language element. The C++ syntax is then checked by the C++ compiler.
- Checks that essential functions such as "Simulation" are in the model code.
- Figures out the actors and creates the classes by bringing together all the actor declarations scattered throughout the .mpp files.
- Figures out the desired tables and creates classes for tables.
- Figures out the relationships among states and other symbols (derived states, events, tables) and creates the code necessary to maintain those relationships.
- Creates the code necessary to update the time for each actor.
- Creates an internal structure allowing the model to create a model help system independently as needed.

Note that this list is not exhaustive. The precompiler is called automatically when a model is built.~~by using the button on the Modgen toolbar in Visual Studio labelled "Modgen 10" or using the equivalent item in the "Tools" menu~~.

Library

The Modgen library contains the entities that all models have in common. At compilation time, links are made with the Modgen library to incorporate it into the model's executable file. In this way, each model is an independent executable file. Amongst other things, the library contains:

- The interface for all Modgen models
- The simulation engine
- The help generator

The C++ language requires certain header files in order to incorporate the library into models. These files are installed with Modgen. In addition to the Modgen library, there are other Microsoft libraries and header files supplied with Visual Studio and incorporated into models.

Bilingualism

Canadian law requires that all software applications created by a Government of Canada agency be bilingual – i.e. that English and French be treated equally. Since Modgen is a software application created at Statistics Canada, it must be bilingual. However, in Modgen's case, more than one level of bilingualism is required. In fact, Modgen is an application that allows other applications to be created. It is thus not only necessary for Modgen to be bilingual, but also that it allows bilingual applications to be created.

Precompiler

Modgen itself is bilingual. English and French are treated as equally as possible. The language chosen must be given in an argument on the precompiler's command line.

Example:

c:\Program Files\StatCan\Modgen 12\Modgen.exe –EN

In practice, the Modgen precompiler is called when building a model. The language used by Modgen can be changed using and utility installed with Modgen called Langue Modgen

Models

Modgen contains everything needed for model interfaces to be bilingual – i.e. in English and French. In addition, model developers can very easily use languages other than English and French. All that is necessary is to translate one of the files containing all the Modgen character strings (ModgenEN.mpp or ModgenFR.mpp) and include it as a module in the model.

Even though Modgen is bilingual, an interface will only have the language(s) defined in the model. To make a model bilingual, it must be defined as bilingual. Here is how to define the languages used by the Government of Canada:

```
// These are the languages in which the model can be viewed.
languages
{
      EN,    // English
      FR     // Français
};
```

Remember that all models distributed by the Government of Canada must be bilingual. This might not be clear if a model is a software package or a data product. However, it makes no difference since in both cases it must be fully bilingual if it is produced by a Government of Canada agency.

Modgen model elements

Modgen can be used to create two different kinds of models:

- Case-based models

  A case-based model is a model where the simulation takes place case by case. A case typically consists of a main actor with, if required, secondary actors surrounding the main actor. However, in some models, there can be more than one main actor per case if, for example, a case simulates a family.

  The number of cases to be simulated is defined as a run control parameter. The “LifePaths”, “PopSim” and “Pohem” models are examples of case models.

- Time-based models

  A time-based model is a model in which a population, together, is simulated over a certain time. The duration of the simulation is then defined as a run control parameter. The “CVMM” and “HIVMM" models are examples of time-based models.

Model content

Case-based models and time-based models are made up of Modgen symbols. It is these symbols and the relationships among them that control the simulation. A short description of each symbol type is given here. For a more detailed description, please refer to the Modgen 12 Developer’s Guide.

Actors

First, Modgen models simulate the lives of actors. An actor may be any entity the model developer wishes to simulate – a person, a dwelling, a pension plan, etc.

Actors are defined by their:

- States

- Events
- Functions
- Hooks

States

States describe actors' characteristics. There are two major kinds of states in Modgen models:

- Simple states:

  Simple states are states whose values are not maintained automatically by Modgen. Rather, the values of simple states are changed in the code created by the model developer. Simple states should only be changed inside an event or a function called inside an event.

  Example:

```
int   age_int ;   //EN Integer age
```

  Simple states may have an initial value. To distinguish them from derived states, initial values are assigned to simple states using curly brackets.

  Example:

```
int   age_int = {0} ;   //EN Integer age
```

- Derived states:

  Derived states are states whose values are given as an expression in the declaration. Modgen maintains the value of these states throughout the simulation. Normally, these states are derived from other states, which explains their name.

  Example:

```
//EN Year
MODELED_TIME modeled_year = COERCE(MODELED_TIME, year);
```

  In addition to simple expressions like the one in the example above, there are also functions that model developers may choose to use to create derived states.

  Example:

```
//EN Age at year start
int   age_debut_annee = value_at_latest_change( annee, age_int);
```

Generally, states are declared to be of type:

- Integer (int)
- Real (float or double)

- Logical (logical)
- Time (TIME)
- With a type created in the model (classification, range, partition)

Events

Events play a key role in Modgen models since it is by executing events that simulation is done. Each event is made up of a pair of functions:

- a function to determine the time of the event
- a function to determine the consequences of the event

Event times are updated as needed. When a time must be recalculated, Modgen re-evaluates the value of the event time function. It assumes that event times must be recalculated when:

- one of the states used in an event time function changes its value
- the event occurs.

An event time function should not affect the actors. That implies that states can never be changed inside an event time function. The Modgen precompiler ensures that that does not happen.

The opposite is true for event functions. Functions that determine the consequences of events are in fact what are executed when events occur. It is inside event functions that states are changed. States should never be changed outside an event or a function called by an event function once initialization has been completed. Again, Modgen ensures that that does not happen.

Functions

Functions that are members of an actor may be used to generalize sections of code that will be used in one or more events. In general, a function should be created when a section of code will be used in several places. In that case, using a function simplifies maintenance since when the code has to be changed, it need only be changed in one place. Not using functions in those cases increases the risk of introducing bugs. Sometimes, we might want to use a function even if the code is not reused, either because it might be reused eventually or to increase modularity.

In addition to functions created by the model developer, there are two functions that actors must have. These are the “Start” and “Finish” functions.

“Start” function

There must be a “Start” function for each actor. It is called once for each actor when the actor is created. It contains the actor’s initialization, including the initialization done by Modgen. Note that a state initialized in the “Start” function neither affects tables nor derived states such as “entrances()”. For that reason, it is better to do those initializations in the “Start” function and not in the function that creates the actor. If the model developer has no initialization to do and does

not include the “Start” function, Modgen creates a function containing only the initialization of states to the default values set by Modgen.

Example 1:

```
actor Personne  //EN Person
{
      void  Start();    //EN Starts the person actor
};


void Personne::Start()
{
}
```

Example 2:

```
actor Personne  //EN Person
{
      //EN Starts the person actor
      void  Start( int nObs, logical lImmigrant, double dTempsDebut ,
                  logical lEnfant, Personne *prMere );
};

/*NOTE(Personne.Start, EN)
      Function which starts the person actor.
*/
void Personne::Start( int nObs, bool bImmigrant, TIME tTempsDebut,
                        bool bEnfant)
{

      immigre_apres_recensement = bImmigrant;
      num_recense = nObs;

      ne_apres_recensement = bEnfant;

      time = CoarsenMantissa(tTempsDebut);
...

      vivant = TRUE;
      emigrant = FALSE;


      // Year
      annee = (int) time;
};
```

In the example above, the states “vivant” and “emigrant” should have been initialized when they were declared. In general, states initialized in the “Start” function are states whose values are not constant but may depend, for example, on values given as arguments to the function.

“Finish” function

There is also a “Finish” function to contain everything that must be done when an actor is deleted. For example, we might want all actors to which an actor is linked to be deleted also. If the model

developer has nothing in particular to do before an actor is deleted and omits the “Finish” function, Modgen automatically creates a basic function. Note that Modgen deletes all links to other actors when the “Finish” function is called but does not call those actors’ “Finish” functions.

Example:

```
actor Personne  //EN Person
{
      void  Finish();    //EN Ends Person actor
};

void Personne::Finish()
{
    // Empty for now
}
```

Hooks

Hooks are used to link functions to events or to other functions. This makes it possible to divide events or functions into different sections belonging to different models.

In particular, many hooks are used with the “Start” and “Finish” functions to do initialization related to a module inside this module itself. By default, all hooks are inserted at the end of functions. If we want all hooks to be inserted elsewhere, we can do this by defining “IMPLEMENT_HOOK” at the appropriate place in the function.

Here is an example where hooks are inserted at the start of the function:

```
/* NOTE(Person.Finish,EN)
      Function which cleans up the actor once finished.
*/
void Person::Finish()
{
      Person      *prChild = {NULL};
      int          nIndex = {0};

      IMPLEMENT_HOOK();

...

      if ( tentative && sex == FEMALE )
      {
            mlChildren->FinishAll();
            mlChildrenAtHome->FinishAll();
            mlBiologicalChildren->FinishAll();
      }
}
```

Note that in the case of the “Finish” function, it makes more sense to have the hooks at the start of the function since the actor’s properties are no longer valid at the end of it.

It can also happen that we want to insert hooks at a different place from the others. In that case, we create an intermediate function to which hooks are attached and call that function at the proper place.

Example:

```
actor Person      //EN Core individual
{
      void  Start( ...); //EN Function which initializes Person actor
      void  StartClockHere(); //EN Start Clock dummy function
};

/* NOTE(Person.StartClockHere,EN)
      Dummy function which allows the clock to be initialized prior to
      all other hooked functions and derived functions.
*/
void Person::StartClockHere()
{
}

void Person::Start( ... )
{

...
      time = birth;
      StartClockHere();
...
      year_of_birth = year;
      month_of_birth = month_of_year;
      day_of_birth = day_of_month;
...
}
```

In this example, the “StartClockHere” function will be executed immediately after the time is initialized. The other “Start” function hooks are only executed at the end of the function. This makes it possible for them to use clock states such as “year”, “month”, etc. to initialize other states. The “StartClockHere” function does not contain code but rather associated hooks:

```
actor Person      //EN Core individual
{
      //EN Initialize the clock at birth of actor
      void StartClock();

      hook StartClock, StartClockHere;

};
```

It is the “StartClock” function, defined in another module, which contains the code that initializes the clock and its states such as “year”, “month”, etc.

Actor sets

Actor sets are collections of actors maintained dynamically by Modgen. This means that membership in an actor set is handled by Modgen based on criteria provided by the model developer. Actor sets can be multidimensional. Dimensions of actor sets are specified using states of the actors. For each combination of values of the dimensions of an actor set, subsets of actors are created. Modgen automatically ensures that each actor can only belong to one such subset. To find out more about actor sets, you can refer to the Modgen Developer’s Guide.

Links

Links define relationships among actors of the same or different types. There can be links between one actor and a single other actor, one actor and several actors, or several actors and several actors. In all cases, Modgen automatically creates and maintains reciprocal links. This means that a model's code need only assign one of the links and the reciprocal link will be altered to reflect the change.

Examples of links:

```
// Declare the one-to-many link for family membership
 link
      Person.lFamily      //EN Family
      Family.mlMembers[]  //EN Family members
;


// link between children and parents
link
      Person.mlChildren[]     //EN Children
      Person.mlParents[]      //EN Parents
;

// link between person and spouse
link Person.lSpouse;          //EN Spouse
```

For more information on links, you may refer to the Modgen 12 Developer's Guide.

Parameters

Parameters give model users a certain amount of control over simulations. They should be used everywhere the developer wishes to give control of a model to the user. Allowing model users to specify hazards to control various aspects of a simulation, for example, gives them an opportunity to explore different scenarios.

There are two main kinds of parameters:

- parameters

  This is the most frequent kind. Parameters are used when users are to enter values. Parameter values are found in .dat files. Users thus control the values of these parameters directly.

  Example:

  ```
  parameters
  {

        //EN total mortality hazards (m total)
        double            MortHazard[MODELED_AGE][SEX];

  };
  ```

- model-generated parameters

These are parameters that must be derived, often from other values given in parameters. For each model-generated parameter, there must be corresponding code in the "PreSimulation" function, as illustrated in the following example.

Example:

```
parameters
{

      //EN Residual mortality hazards
      model_generated double
            ResidualMortHazard[MODELED_AGE][SEX];

};
```

Tables

Modgen models produce output in the form of tables. There are two different table types:

- tables

  A model's cross-tabulated tables generate a simulation's aggregated results.

- user tables

  These tables are misnamed. It is not model users who are referred to but Modgen users - in other words, model developers!

  User tables allow model developers to create code to calculate the value of each cell. For each user table, code must be placed in a "UserTables" function to control the table's contents. Typically, this code reads and combines values from various tables to create other tables.

Table and parameter groups

Modgen allows some symbols to be grouped together. These groups are mainly used to group symbols in model interfaces but are also described in the model's help generated by Modgen. There are three kinds of symbols that can be grouped:

- model parameters
- model-generated parameters
- tables and user tables

Membership of a symbol in a group must be specified in the group's declaration. Note that a group itself can contain another group of the same kind, thus giving the model developer the ability to create a hierarchy.

Example of a parameter group:

```
parameter_group DEMOGRAPHY  //EN Demography parameters
{
      HasardMortalite, HasardNatalite, AgeMaximal, ProbabiliteGarcon,
      VariationFecondite
};
```

Example of a table group:

```
table_group DemographyTables { //EN Demography tables
    BirthRates,
    Population,
    Population5Year
};
```

Types

Type symbols are support symbols. They are not essential to simulations since they do not play any specific role in the running of a model. However, they are everywhere since they are used to define other symbols such as states and parameters. They are also used to give labels to tables. There are three kinds of type symbols:

- Classifications

  Set of levels or categories with their labels.

  Example:

  ```
  classification LANGUE2
  {
        //EN francophone
        L2_FRANCO,
        //EN non francophone
        L2_NON_FRANCO
  };
  ```

- Ranges

  Integer number ranges

  Example:

  ```
  range AGE_NATALITE {15, 49} ; //EN Age for birth rates
  ```

- Partitions

  Set of boundary points for continuous (or discrete) state variables.

  Example:

  ```
  //EN Age groups
  partition GROUPE_AGE
  {
        5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75,
  ```

```
        80, 85, 90, 95, 100
};
```

The descriptions presented here are taken from the Modgen 12 Developer's Guide. You can refer to it for a more complete description.

Course of a model simulation

In addition to the symbols contained in them, models are also made up of sets of general functions that control their execution. The process described here is for that of a case-based model. However, most of what is described here also applies to time-based models. Note that this section does not deal with the issue of parallel processing that is incorporated into Modgen.

Parameter reading

The first step of a case or time-based model simulation is to read the parameters. It is not necessary to create a global function in the model code to manage parameter reading or interactive modification. That is provided by Modgen.

Parameter validation

Modgen includes functions that allow developers to validate and change the values of the parameters provided by users. For more information on those functions, you can refer to the Modgen 10 Developer's Guide.

Presimulation

After the parameters are read, the presimulation phase begins. This phase will not do anything at all unless the model developer has defined one or more "PreSimulation" functions. Note that it is possible to create more than one "PreSimulation" function to increase modularity. In a future version of Modgen, it will even be possible to specify the execution order of a model's "PreSimulation" functions.

Normally, "PreSimulation" serves to calculate the model-generated parameters. At this stage, there is still just one simulation thread so the model-generated parameters can be changed without there being any risk of conflict between the different simulation threads. Moreover, a "PreSimulation" function cannot change actors since they have not yet been created.

Example:

```
parameters
{

        //EN total mortality hazards (m tot)
        double              MortHazard[MODELED_AGE][SEX];

        //EN cause-specific population death hazards for disease X (m(x))
        double              MortHazardX[MODELED_AGE][SEX];
```

```
      //EN cause-specific population death hazards for disease C (m(c))
      double            MortHazardC[MODELED_AGE][SEX];

      //EN Residual mortality hazards
      model_generated double
            ResidualMortHazard[MODELED_AGE][SEX];
};

void PreSimulation()
{     int         nAge = {0};
      int         nSex = {0};
      int         nTime = {0};

      for (nAge = 0; nAge < SIZE(MODELED_AGE); nAge++)
      {
            for (nSex = 0; nSex < SIZE(SEX); nSex++)
            {
                  // Calculate residual mortality hazard
                  ResidualMortHazard[nAge][nSex] =
                        MortHazard[nAge][nSex]
                        - MortHazardC[nAge][nSex]
                        - MortHazardX[nAge][nSex];
            }
      }
}
```

Simulation

Once the presimulation has ended, the simulation phase begins. The model must contain a "Simulation" function. Typically, for case-based models this will simply be a loop calling the "CaseSimulation" function for each model case.

Example:

```
void Simulation()
{
      long  lCase = 0;

      for ( lCase = 0; lCase < CASES() && !gbInterrupted &&
            !gbCancelled && !gbErrors; lCase++ )
      {
          StartCase();
          CaseSimulation();
          SignalCase();
      }
}
```

The "Simulation" function is more complicated for time-based models since it also has to create an initial population. It must also contain the events loop found in the "CaseSimulation" function described below for case-based models.

Note that this function is included in models created using a Modgen model creation wizard.

CaseSimulation

In case-based models, Modgen does not require a function called “CaseSimulation”. However, using such a function simplifies the code. This function is responsible for the simulation of one case and starts by creating the case’s starting actor. It is inside this function that the events loop controlling the simulation of the case should be found.

Example of an events loop:

```
// event loop for current case
while ( !gpoEventQueue->Empty() )
{

	if ( gbCancelled || gbErrors)
	{
		// in case of errors, close case and destroy all actors
		gpoEventQueue->FinishAllActors();
	}
	else
	{
		// advance time to next event
		gpoEventQueue->WaitUntil( gpoEventQueue->NextEvent() );

		// execute next event
		gpoEventQueue->Implement();
	}
}
```

This loop is at the heart of model simulation and should only be changed with great care. Note also that it is included in models created by one of Modgen’s model creation wizards.

In fact, this loop controls the Modgen simulation engine. Within the simulation engine, there is an events queue that maintains the wait time for each model event. This queue is ordered in such a way that events are executed in a specific order. The simulation engine within Modgen executes the events in the following order:

- The event with the smallest wait time is executed first
- If times are the same, the event with the highest priority is executed first
- If their priorities are the same, events are executed in alphabetical order
- If their names are also the same, the event of the actor first created in the simulation is executed first.

  Note that events with the same name have to be associated with different actors, since a given event can only be found once for each separate actor in the events queue.

PostSimulation

If needed, model developers may also define a “PostSimulation” function. If it is defined, this function is run after the simulation phase but before data is tabulated. This function is only rarely used. However, if global variables are used in a model, it might be useful to reinitialize them in

this function. Then, if the user restarts a simulation without shutting the model down, the global variables are initialized to the same values as in the first simulation.

Tabulation

The last phase produces the tables. This expression is however misleading since tabulation takes place in Modgen as the model runs. In this phase, the tables produced during that simulation run are simply saved in the output database.

However, user tables are calculated at this stage before being saved in the database. It is at this stage that the “UserTables” functions, created by the model developer to calculate the values in the user tables, are executed.