Modgen Version 12.1.1

Developer's Guide

By Statistics Canada

Table of Contents

Preface

Modgen is a microsimulation development framework that is used to create, maintain, and document microsimulation models with either interacting or non-interacting populations. Modgen hides underlying mechanisms like event queueing and creates a stand-alone model executable with a complete visual interface and detailed model documentation. As a result, you, the model developer, can concentrate on the code that is specific to your model—the declaration of simulated actors with their events, parameters and output tables.

The aim of this guide is to present the concepts, background, and examples that you need in order to create or modify a microsimulation model using Modgen.

Contents of this guide

The guide is structured as follows:

An overview of microsimulation with Modgen

- ➢ provides a brief overview of microsimulation models and an introduction to key Modgen concepts.

Creating a model

- ➢ outlines the system and software requirements for creating a model, describes the relevant Visual Studio environment, and introduces ideas for organizing the code and data in your model. This chapter also explains how your Modgen code turns into a model executable file, and it describes a Modgen wizard for creating the foundation of a brand new model.

Running a model simulation

- ➢ summarizes the commands available to initiate a model simulation run, describes the steps that take place sequentially during a simulation run, and itemizes the different output files available at the conclusion of the simulation run.

Elements of the Modgen language

- ➢ identifies the main elements of the Modgen language required to build a model and how the elements are declared and used. These elements include actors, states (characteristics of the actors), events, parameters, tables. This chapter also explains how to document the Modgen elements using labels and notes and introduces techniques through which Modgen enables multiple languages.

The simulation engine

- ➢ describes the fundamentals of the simulation engine that powers a simulation run for either an interacting population or non-interacting population model.

Derived state specifications

- ➢ provides details on the many diverse functions that are available to create derived states for Modgen models. (A derived state is a state whose value is obtained from an expression that is usually based on other states; its values are automatically maintained by Modgen throughout a simulation run.) The functions are broken down and listed, according to their general purposes (e.g. event counts, durations, transformations).

Parameters

- ➢ focuses on parameters which are exogenous to a model and which give the model user a degree of control over the simulations that are run because the ability to change parameter values allows different scenarios to be explored. This chapter examines several aspects of parameters (how to initialize them, access them in model code, extend them, hide them, specify decimals, etc.)

The table facility

- ➢ describes the structure of the two main types of tables in a Modgen model (regular tables populated on-the-fly during the course of a simulation run, and “user tables” which get their values following the conclusion of the run.) The chapter explores several aspects of tables (defining them, hiding them, scaling them, specifying decimals, etc.) and includes several different examples.

Interactions amongst actors—links and actor sets

- ➢ reviews the two major methods through which actors can interact in Modgen models—links (so that an actor is in relationship with other actors) and actor sets (to dynamically maintain a collection of actors).

Run-time functions and macros

- ➢ provides details on several run-time functions that are available in Modgen as you build your model. The functions are broken down by general purpose (e.g. those that modify the progress window during a simulation run, those associated with the random number generator, those used for case weighting and population scaling).

Technical topics in Modgen

- describes miscellaneous technical features of Modgen that can assist you as you develop or refine your model (e.g. using function hooks, controlling precision, cloning observations)

Examining the individual lifetimes of actors

- introduces the concept of tracking in a model, including the commands required to implement tracking and some cautions on using tracking wisely.

Debugging and optimizing tools and techniques

- discusses concepts and techniques for debugging Modgen models, based on three general forms of errors (compile-time bugs, run-time errors, unexpected results). The chapter also incorporates techniques to help you optimize your model code (e.g. running a memory usage report).

Documentation of a Modgen model

- categories the different levels of documentation that can exist for a model (primary, encyclopedic, scenario) and outlines how each level is implemented.

Distribution of a Modgen model

- provides a high-level overview of some of the issues that need to be considered before you can release your model for others to use.

Using this guide

If you are new to Modgen, you should start with the first four chapters of this guide (An overview of microsimulation with Modgen, Creating a model, Running a model simulation, and Elements of the Modgen language). You can also examine the section of Modgen's website dedicated to those who are relatively new to either Modgen or microsimulation itself: http://www.statcan.gc.ca/microsimulation/modgen/new.

Before designing your model, it will be necessary to understand the various elements of the Modgen language, so that chapter should be next. Efficient programming in Modgen will require knowledge of the powerful derived state specifications also, so it would also be helpful to become familiar with the different categories of derived state specifications available for Modgen models.

The chapters on interactions amongst actors, run-time functions and miscellaneous technical topics outline other utilities or features of Modgen that are available when building models. The tracking and the debugging and optimization chapters describe tools that are useful in diagnosing problems or refining performance in your model, whereas the documentation and distribution chapters include many useful considerations when preparing your models to be used by others.

Status of the documentation

The version of this guide corresponds to Version 12.1.1 of Modgen. However, Modgen is continuously being improved with new enhancements or functionality. Any newer features or changes since Version 12.1.1 of Modgen are outlined in the release notes for subsequent Modgen releases, available in the start menu under Modgen 12. If you wish to learn about features proposed or scheduled for future releases of Modgen, please contact microsimulation@statcan.gc.ca. You can also contact this address to be added to an email list that will keep you informed of new public releases of Modgen, as well as upcoming training opportunities.

The interface of Modgen models is described in a separate guide, the "*Guide to the Modgen Visual Interface*". The Modgen framework also includes tools to use with models, such as the BioBrowser, which is described in the "*BioBrowser User's Guide*".

Background knowledge

Modgen was designed to enable analysts to create microsimulation models without requiring the services of professional programmers. That implies that you do not have to be an expert programmer in C++ to use Modgen but you do have to have some basic familiarity with programming concepts.

Conventions used in this guide

The following typographic or formatting conventions are used throughout this guide:

Description	Example
Function names are all bold with accompanying parentheses, be they actor functions, Modgen run-time functions, derived state specifications, or any other supporting C++ functions	**GetRandomUniform()**
Function arguments are italicized within the function	**duration(*observed_state, value)***
Optional portion of a function argument	<…>
Names of states and parameters are bold and italicized	***sex***
State values are non-bold and italicized	*female*
Publication names are non-bold and in both double quotes and italics	*"BioBrowser User's Guide"*
Filenames are in single quotes	'model_name.mpp'
Menu items are in bold; multiple menu selections are separated with a '>' symbol	**File** > **New**
Website URL's & email addresses are in italics	*micrsosimulation@statcan.gc.ca*
Names of Modgen commands or keywords are bold	**time_type**

An overview of microsimulation with Modgen

Overview of longitudinal microsimulation models

In their most general terms, microsimulation models attempt to represent individuals, or micro elements, in a system in order to obtain information about the population or system they constitute. Examples of microsimulation models in the social sciences can be broken down into static cross-sectional models or dynamic longitudinal models.

The first cross-sectional models in the field of economic policy were income tax simulators, which provided estimates of the expected revenue and distributional impacts of tax reform. Such models were designed to study the cross-sectional effect of policy change, e.g. by identifying immediate winners and losers of policy reform. These models, while technically complex, are conceptually rather straightforward. They are based on data for a representative sample of the population at a point in time (typically with annual income data). The rules and provisions of the income tax and/or transfer systeme are carefully codified and parameterized as algorithms. A simulation consists of computing, for each individual or family in the sample, the tax liability and transfer benefits according to either the status quo rules or the rules under some hypothesized reform. The software then allowed comparisons of these ‘base’ and ‘variant’ simulations – for example with respect to aggregate costs and revenues – to determine gainers and losers by such characteristics as income group and family size.

These kinds of cross-sectional models, however, are completely inappropriate for issues in a domain such as pension policy. The reason is that both public and private pension systems typically pay benefits for many years after retirement (e.g. from age 65), where the amount of benefits depends in a complex way on earnings over the span of decades prior to retirement. Moreover, pension benefits can also depend significantly on marital and fertility histories, on survivor and orphan benefits, and on health histories (e.g. for disability pensions). Because of these factors, a longitudinal microsimulation model becomes a much more useful simulation approach since, by design, longitudinal model frameworks incorporate the rules of the pension systems to be analyzed and have the means to generate a dynamic view of a representative population to whom these pension rules (both base and variant) can be applied.

Demographic projections are another domain in which longitudinal microsimulation modeling has a large potential role. Certainly one key set of inputs to a pension analysis can be one or more demographic projections, the latter in turn based on various fertility, mortality, migration and other scenarios or assumptions. Relatively simple modeling approaches have been used in the past for demographic projections but longitudinal microsimulation models add a richness to the analysis because they allow more compelx simulations. Such models also link the underlying multiple elementary processes, which can usually only be studied in isolation without microsimulation, to more truly obtain the influence of each process on the overall population.

In general, dynamic longitudinal microsimulation models represent an appropriate approach to simulation when:

- the differences in heterogeneous populations are important and there are too many possible combinations of characteristics under examination to split the population into a manageable number of groups
- behaviours are complex at the macro-level but better understood at the micro-level
- the processes of interest have a memory, i.e. events that have occurred in the past can have a direct influence on what happens in the future.

Overview of Modgen models

The Modgen development framework includes the Modgen language which is, essentially, a superset of the C++ programming language in which you, in the role of model developer, simulate the lifetimes of a set of actors (e.g. a core individual and related family members). All models developed in Modgen require that you specify those actors and the relationships between them.

Modgen supports the development of two categories of longitudinal microsimulation models. The first is a case-based or non-interacting population model in which a specified number of cases are simulated sequentially and independently of one another. Examples of these models are the LifePaths, Demosim and the Pohem models developed by Statistics Canada. The other category is a time-based or interacting population model, involving several interacting actors or agents over a specified period of time. An example of such a model is the IDMM (Infectious Disease Microsimulation Model), also developed by Statistics Canada. This is a model in which the spread of disease is directly linked to the potential interactions of any of the actors in the population over a pre-determined length of time.

Models can also be classified as continuous time models or discrete time models. Microsimulation models in which hazards compete to determine the time of the next event in a person’s life are referred to as continuous time models since the time interval between events is not fixed. A more rigid type of model which fixes the interval between events is referred to as a discrete time model. Either modeling approach can be accommodated with Modgen.

Model actors are defined by their respective sets of characteristics. These characteristics are called states and contain the descriptive information of the actor. Actors also have events which transform the values of the states. For example, if you were modeling a pension system, the model actors would likely be persons. The set of states for the person actor would include age, labour force status and level of earned income. The values of these states would change for the person actor over time as events impacted on the individual. For example, the event which dictated when people retire would change the value of the labour force status state.

There are two main kinds of states in any Modgen model—simple states and derived states. A simple state is a state whose value is always set or modified explicitly in event code. A derived state is a state whose value is always maintained automatically by Modgen while a simulation run is taking place.

For any model, a simulation takes place through the execution of events. Each event consists of two parts: a part to determine the time of the next occurrence of the event, and a part to determine the consequences when the event takes place. The values of simple states are modified within the latter. Modgen has a built-in simulation engine to process the event queues for each individual case in a case-based model or over all actors simultaneously in a time-based model.

The relationships between model actors are necessary if the information set of one actor influences the events of another actor. For example, a person could have relationships with other persons who form the person’s family. Assume for the purpose of this example that the amount of pension income entitled by a person is a function of the number of that person’s dependents, then the events of other members in the family, such as leaving the home for children, will influence this person’s simulated pension.

Modgen ensures that the actor states and the links between actors are continuously updated through the course of the simulation exercise. This happens amongst all actors within a specific case for a case-based model or within the entire population for a time-based model. In this way, the causality of states across actors is always up-to-date.

The simulation of a case-based model is performed on a case-by-case basis. A case can be either one individual or a small set of individuals that are related, such as a family. Usually, the simulation of the case is concluded when the last model actor ceases to be active. Modgen then begins the simulation of the next case. The number of cases to be simulated is specified through a simulation setting in Modgen’s visual interface.

All of the actors in a time-based model can potentially interact with one another. Rather than waiting until an entire population stops being active, a total amount of time controls when such a simulation run concludes. This amount of time is also specified via a scenario setting in Modgen’s visual interface before the simulation run begins.

Glossary of Important Terms

In order to successfully develop models in the Modgen language, it is essential to understand how the basic model elements interact. The following is a list of these key elements:

cases

Modgen simulates one case at a time in a case-based model. A case contains either one actor or a set of actors with established relationships (links). Modgen maintains these links between actors at all times during the simulation. The number of cases to be simulated is specified through a simulation setting in Modgen's visual interface.

actors

Actors are at the core of a Modgen model and represent the entities being simulated. The main actor in most microsimulation models is the Person actor, but other actors can also be simulated, such as a household, a pension plan, an animal, etc. These actors are created at the beginning of the simulation process and undergo changes to their characteristics as they proceed through the simulation. Actors are defined by their characteristics (states) and by the events which transform their states. (Actors are similar to class objects in C++; both states and events can be thought of as class members.)

states

These elements define the characteristics of the actors over the span of their lifetimes. There are two major kinds of states in Modgen—simple states and derived states. A simple state is a state whose value can be initialized and changed by the code that a model developer creates to implement events. A derived state, on the other hand, is a state whose value is given as an expression which is normally derived from or based on other states. A derived state's value is automatically maintained by Modgen throughout a simulation run. States can be thought of as properties or variables of the actor class and are declared using either standard C++ types, special types available to any Modgen model (such as TIME) or types that are defined for a specific model.

events

In Modgen, simulation takes place through the execution of events. Each event consists of two functions: a time function to determine the time of the next occurrence of the event, and an implementation function to determine the consequences when the event takes place. The values of simple states are modified within event implementation functions. A built-in simulation engine processes the event queues during a simulation run.

actor links

These elements allow the access of states between model actors. (They are analogous to symmetric C++ pointers between classes.). The links can be one-to-one, one-to-many, or many-to-many.

parameters

These elements are exogenous to the model and are found in .dat files which the model's .exe file reads during the execution phase. Model parameters generally specify the components of the simulation exercise. These may include, for example, the eligibility rules for a pension scheme or the estimated participation rate in a new social program. Model parameters can be scalars or arrays and are cast using either standard C++ types or special types that are available to Modgen models. The ability to change parameter values allows different scenarios to be explored and thus gives model users a degree of control over the simulations they run.

tables

Analysis of results of a Modgen models is done internally, because Modgen has a powerful cross-tabulation facility built in to report aggregated results in the form of tables. There are two central elements of a table declaration—its captured dimensions (defining when an actor enters and leaves a cell) and its analysis dimension (recording what happens while an actor is in that cell). When running a simulation, the data is added to the table on-the-fly, thus removing the need to create and write to large temporary interim files during the run for subsequent reporting.

scenario settings

The settings control the nature of a model simulation run rather than the core elements (such as actors or states) of the simulation. Examples include the number of cases to be simulated (case-based model), the total time for a simulation (time-based model), the starting random number seed, or a weighting factor for the cases to be simulated. These settings are specified through the common visual interface available for all Modgen models, as discussed in the separate publication, "*Guide to the Modgen Visual Interface*."

Creating a Model

System and software requirements

Although some Modgen model developers have successfully run Modgen on Mac and Unix, the Modgen development framework was designed for a a Windows environment. More specifically, Modgen has been tested on Windows 7, 8.1 and 10 (both 32-bit and 64-bit).

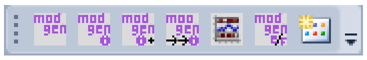
To develop or modify a Modgen model, you need a copy of the Modgen software. Modgen now includes the components included in Modgen Prerequisites. Modgen Prerequisites can be downloaded: http://www.statcan.gc.ca/microsimulation/modgen/download.

In addition, before attempting to install Modgen, you must have Microsoft Visual Studio 2015 (Community, Professional, Premium or Ultimate) installed on your machine so that you can subsequently compile models and generate executables; the Express Edition of the Visual Studio product is not sufficient. If Visual Studio is not already installed, you will receive an error message if you try to install Modgen.

If you only need to run a Modgen model, you only need to have the appropriate version of the Modgen Prerequisites software installed on your machine. (For Modgen 12.1.1, the corresponding version of Modgen Prerequisites is 12.1). Modgen Prerequisites can be downloaded from: http://www.statcan.gc.ca/microsimulation/modgen/download

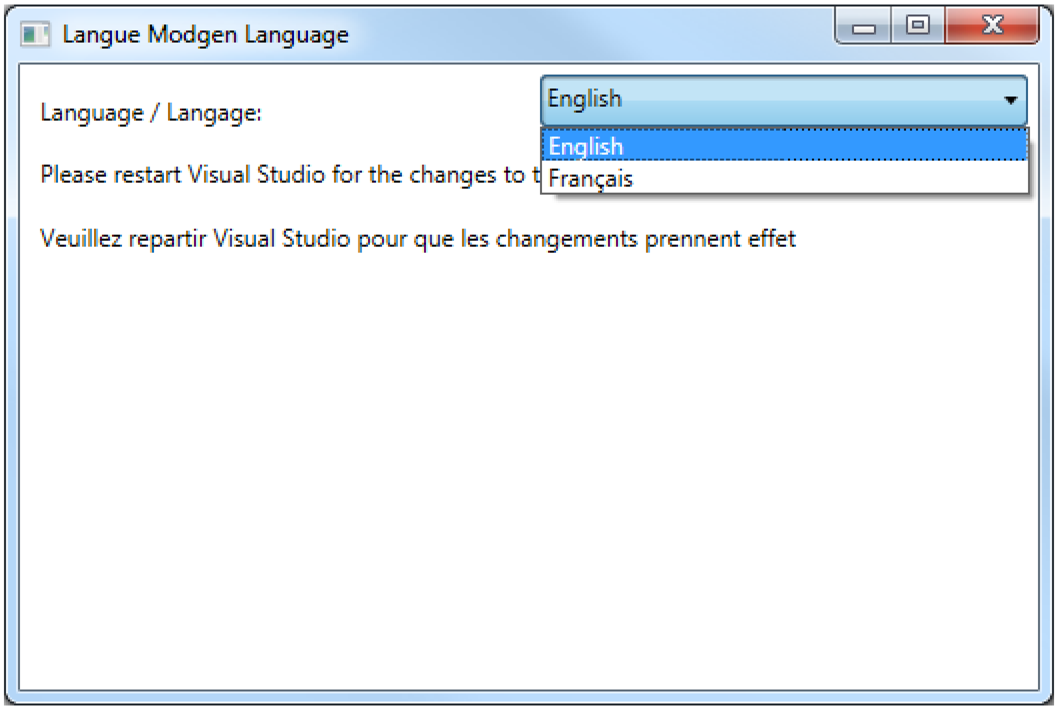
Modgen 12 integration into Visual Studio 2015

Previous versions of Modgen included a toolbar that was added to Visual Studio in order to expose Modgen functionality:



This has changed in Modgen 12. Below are listed the functions, in left to right order, that were provided by the toolbar, and the location where these items are located in Modgen 12.

- Run Modgen: this button ran the Modgen pre-compiler on the code in the open solution. This is now done automatically by Visual Studio when building the model.
- Modgen Developer's Guide: the Developer's Guide can be opened via the Windows Start menu, under Modgen 12.
- Modgen Release Notes: the Release Notes are also found in the Windows Start menu, under Modgen 12.
- Model Help: you can generate the help file for the model by running (F5 or Ctrl+F5 in Visual Studio), and selecting Help -> Help on [ModelName].
- Open BioBrowser: you can run BioBrowser via the Start Menu.
- Change Modgen language to French: this option now is available through a utility installed with Modgen. It can be found in the Start Menu under Modgen 12, and is called **Langue Modgen Language**:



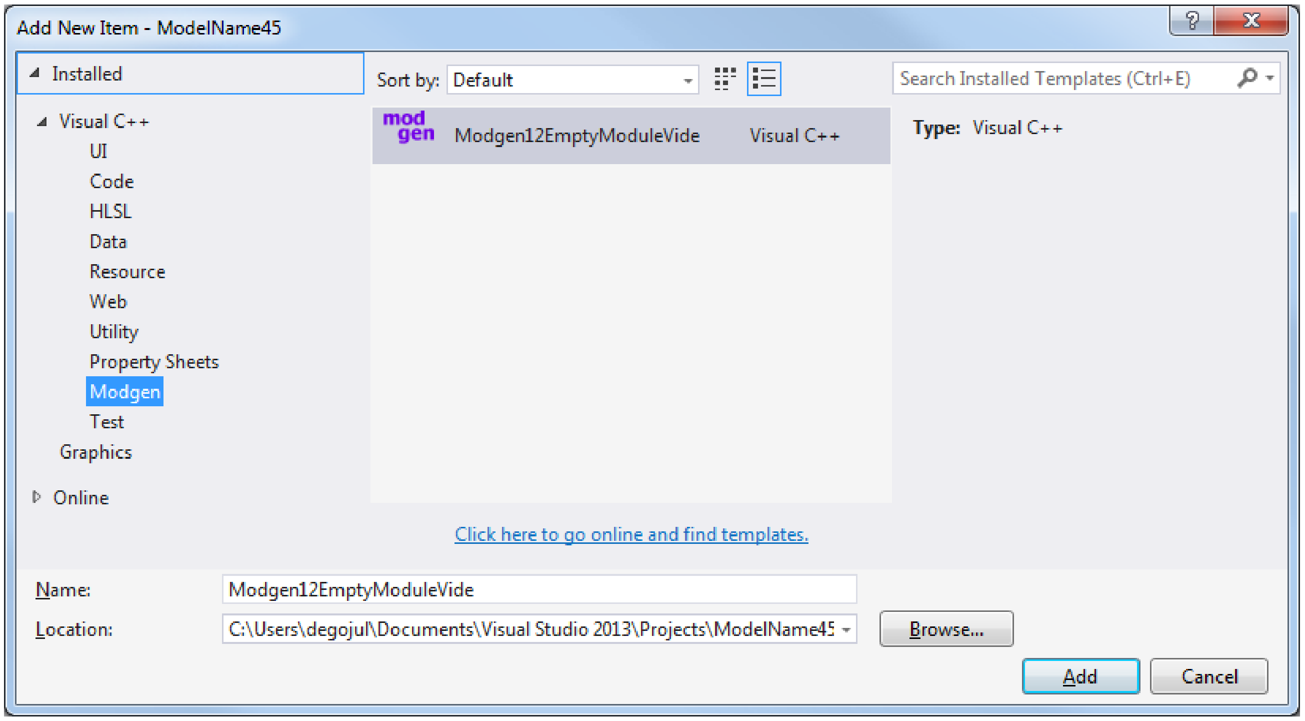
Language / Langage:

Please restart Visual Studio for the changes to t

Veuillez repartir Visual Studio pour que les changements prennent effet

A change in the selection takes effect after re-opening Visual Studio.

- Add New Item: This is equivalent to choosing **Add new item** from the **Project** menu, or to right-clicking the project name in the “Solution Explorer” window and choosing **Add** > **New Item**. After the wizard begins, the following dialog box appears:



Name: Modgen12EmptyModuleVide

Location: C:\Users\degojul\Documents\Visual Studio 2013\Projects\ModelName45

In this dialog box after selecting “Modgen12EmptyModuleVide”, you specify the name of the new module. Note that module names cannot contain spaces or special characters; the wizard will refuse to create a module if the name is not a valid Modgen name. You should also not change the location. The output of the wizard will depend on the selection.

- **‘Modgen12EmptyModuleVide’ module**: The wizard creates an empty mpp file and adds it to your project in the correct folder.

Modgen models as viewed within the Visual Studio environment

At the top level in the Visual Studio environment, there is a solution (.sln) file. The role of a solution is to act as a container for all related projects and to allow for the specification of project properties through configurations. (For example, a solution always includes a configuration to create either a debug version of an executable or a release version of an executable.) The solution (.sln) file specific to a Modgen model should always have the same name as the model and reside in the same folder as the corresponding project files. A solution file is generated automatically by Modgen’s New Model Wizard and as a developer, you rarely need to modify it. When you are building or modifying your model code with Visual Studio, you always do so by first opening the model’s .sln file.

Within a solution for a Modgen model, there is also at least one C++ project (.vcxproj) file that must have the same name as the model. In fact, the name of the model's executable is taken from the name of the Modgen project. The role of a project is to act as a container for all files related to the project, and to allow for the specification of properties used in building the eventual model executable. The project needs to contain all of the files that are required in order to compile the model. The project file is also created automatically by Modgen's New Model Wizard and again, as a developer, you rarely have any need to modify it.

Within a Visual Studio project, the subfolders are called filters. A Modgen project does not need to have filters but even in a small model, filters can be useful, for example, to organize and separate the source code or modules for the model (.mpp files) from the corresponding C++ (.cpp) files and from the scenario (.dat and .scex) files that contain the parameters to run a simulation and the characteristics to define a particular simulation run. In fact, Modgen's New Model Wizard generates these three filters (Modules, C++ files, Scenario files) by default for a new model, but you can add additional filters by right-clicking on the project's parent folder name and choosing **Add** > **New Filter** from the context menu.

Note that every file from your project does not necessarily have to be grouped under one of the filters. Any file that does not correspond to a specific filter, such as a 'readme.txt' file or a log file, can still be seen at the main level of the project. Furthermore, files can be moved from one location within the project to another, as required, by conventional drag-and-drop techniques.

Organizing the model code, parameters and scenarios

The organization of the model code, parameters and scenarios is crucial to any successful model.

.mpp files

The extension ".mpp" denotes a Modgen file. Modgen files are text files with a specific extension that contain source code. The source code of a model is located in one or many .mpp files, all located in the folder of the Visual Studio project. The Modgen pre-compiler will read those files to create the ".cpp" code for the model.

An .mpp file, 'module_name.mpp', contains all the necessary definitions of model parameters and data types, as well as the actor definition and event functions, associated with a particular module. For example, the module could be demography, mortality, loan calculations, education progression, etc.

Most models have a separate module that contains basic definitions used throughout the model, such as the model version number, model type (case-based or time-based) and languages supported by the model. (The explicit definition of at least one language is mandatory in Modgen). This module also usually contains the simulation engine code to run each case of a case-based model or an entire time-based model. Modgen's new model wizard generates a 'model_name.mpp' file to fulfill this role for brand new models.

Modgen makes no restrictions on the number of .mpp files defined in the model. Nor are there restrictions governing how the elements of the model are organized in the .mpp files. There is one limitation, however--the name of an .mpp file cannot be the same as the name of an actor, parameter, state, event, etc. For example, a model with an actor named “Person” cannot also include a file named ‘Person.mpp’.

Overall, each module of the model must exist in the same folder and have an .mpp file extension. Modgen in turn parses all files that have the extension .mpp from within that working folder. It is good practice to group all of these files together within an “.mpp files” filter in your project.

.dat files

Modgen has no limit to the number of parameter (.dat) files that it can read during its execution phase. Therefore it is good practice to group related model parameters together in .dat files. Generally, it is also good practice to create a .dat file containing all the model parameters associated with a particular module. For example, if there were model parameters which pertained to education, then a ‘scenario_name(Education).dat’ file would be an appropriate file in which to store these parameters, whereas ‘scenario_name(Earnings).dat’ would be a natural choice of file to contain earnings-related parameters.

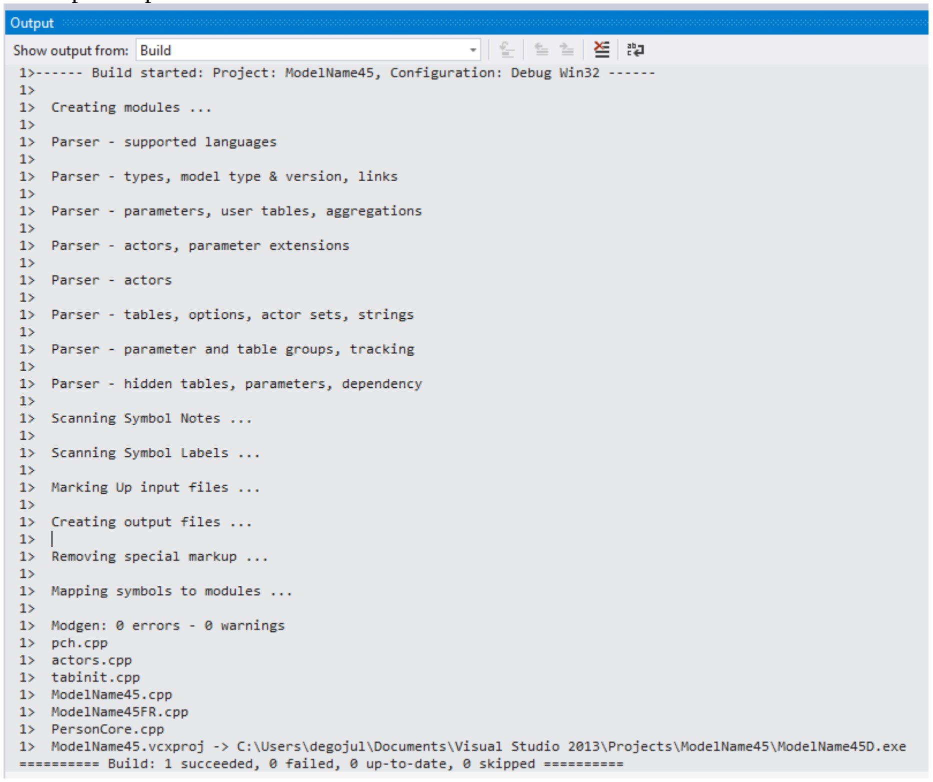
scenario files

It is critical to work from a copy of the model during any analysis which requires modification to the .mpp files. This is prudent since mistakes will not then force you to re-install the model from the setup program. This also enables the results of the modified model version to be compared with the results of the shipped version. Modgen manages specific scenario inputs and outputs using scenario files. All inputs (.dat files) and outputs (.mdb, .xls, .txt files) are prefixed with the scenario file name.

Creating a model executable

It is necessary to create a new model executable file when modifications to the underlying model algorithms or table specifications are made. An executable file is created in one step in Modgen 12, unlike two steps in previous versions. The executable is built by choosing **Build > Build Solution** from the menu. If you have any compilation errors in your model code, you can double click the error in the output window to navigate to the problem piece of code.

A sample output window is shown below:



```
1>------ Build started: Project: ModelName45, Configuration: Debug Win32 ------
1>
1>  Creating modules ...
1>
1>  Parser - supported languages
1>
1>  Parser - types, model type & version, links
1>
1>  Parser - parameters, user tables, aggregations
1>
1>  Parser - actors, parameter extensions
1>
1>  Parser - actors
1>
1>  Parser - tables, options, actor sets, strings
1>
1>  Parser - parameter and table groups, tracking
1>
1>  Parser - hidden tables, parameters, dependency
1>
1>  Scanning Symbol Notes ...
1>
1>  Scanning Symbol Labels ...
1>
1>  Marking Up input files ...
1>
1>  Creating output files ...
1>  |
1>  Removing special markup ...
1>
1>  Mapping symbols to modules ...
1>
1>  Modgen: 0 errors - 0 warnings
1>  pch.cpp
1>  actors.cpp
1>  tabinit.cpp
1>  ModelName45.cpp
1>  ModelName45FR.cpp
1>  PersonCore.cpp
1>  ModelName45.vcxproj -> C:\Users\degojul\Documents\Visual Studio 2013\Projects\ModelName45\ModelName45D.exe
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

If the compilation was performed in release mode, the executable has the name ‘model_name.exe’ (e.g. ‘LifePaths.exe’); if the compilation took place in debug mode instead, the executable’s name has an additional letter ‘d’ on its end (e.g. ‘LifePathsd.exe’)

How Modgen code becomes a model executable file

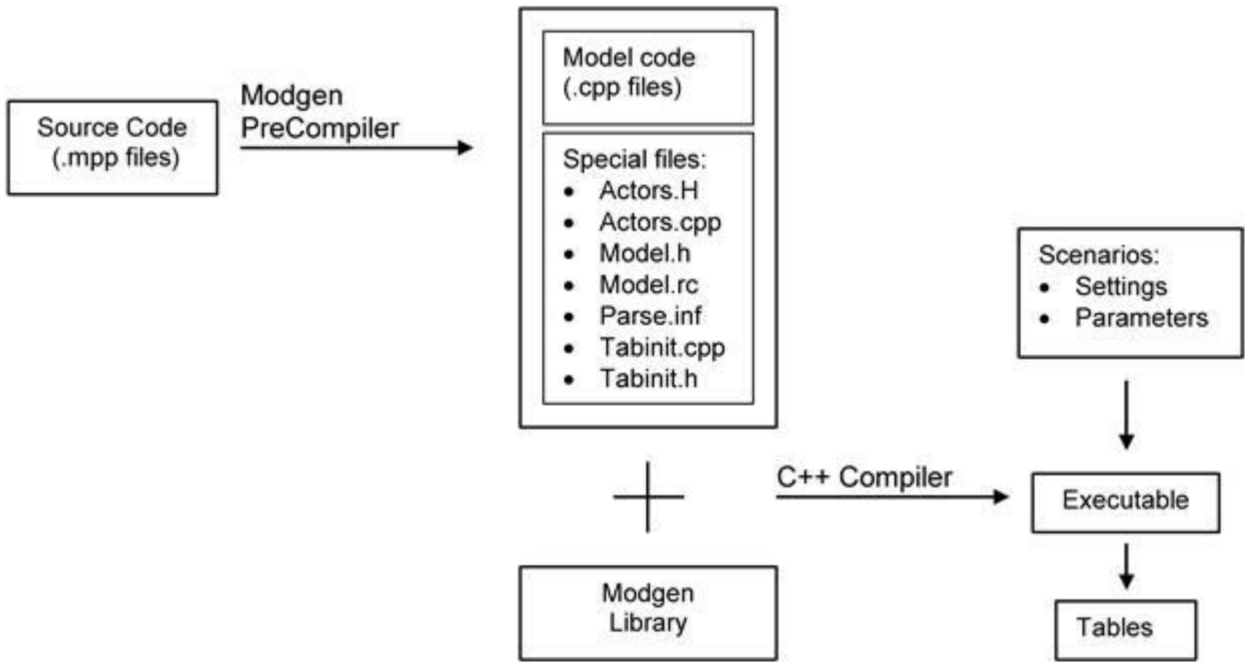
Modgen requires that a new executable file be built for each modification to a model specification. In this way, the speed in which cases are simulated can be maximized. In comparison with similar models written in interpretive languages, Modgen can simulate 10 to 100 times more cases per second. In order to achieve these speeds, Modgen uses a pre-compile approach to convert Modgen commands into instructions understood by the C++ compiler.

In every specification, the Modgen elements and C++ code which define the specifics of the model are contained in a set of .mpp files which are input to the Modgen pre-compiler. In this pre-compilation phase, the ‘Modgen.exe’ file converts the Modgen elements into standard C++ function calls and instructions. All symbols or elements that you declared in these .mpp files are protected via a C++ “namespace” called mm (originally, modgen_model), thus eliminating any conflicts between the names of entities used in a model and the names used internally by Modgen or by Microsoft. (However, any declarations within the ‘custom.h’ file are global, i.e. they are not by default contained within this namespace. To include a declaration within the mm namespace, you need to place it within a “namespace mm()” statement.)

The output of the pre-compilation process is a set of .cpp files and a set of special files: 'ACTORS.H', 'ACTORS.CPP', 'TABINIT.H', 'TABINIT.CPP', 'model.h', 'MODEL.RC' and 'PARSE.INF'. A default empty version of file 'custom.h' and 'custom_early.h' can be optionally created manually. (The files 'custom.h' and 'custom_early.h' are useful for models that contain C++ modules that you coded directly, or in models that contain global functions that you coded directly and that need to be accessible in all modules of the model. In particular, custom header files that declare global names defined in custom C++ modules should be included in the file 'custom.h', rather than directly in .mpp files.) All of these files, plus a set of C++ and special Modgen libraries, are then passed to the C++ Compiler to generate the application's executable file.

These special files are automatically included in the project during the build process. Parameter files (.dat) can also be inserted into the project, although they are not essential for compilation.

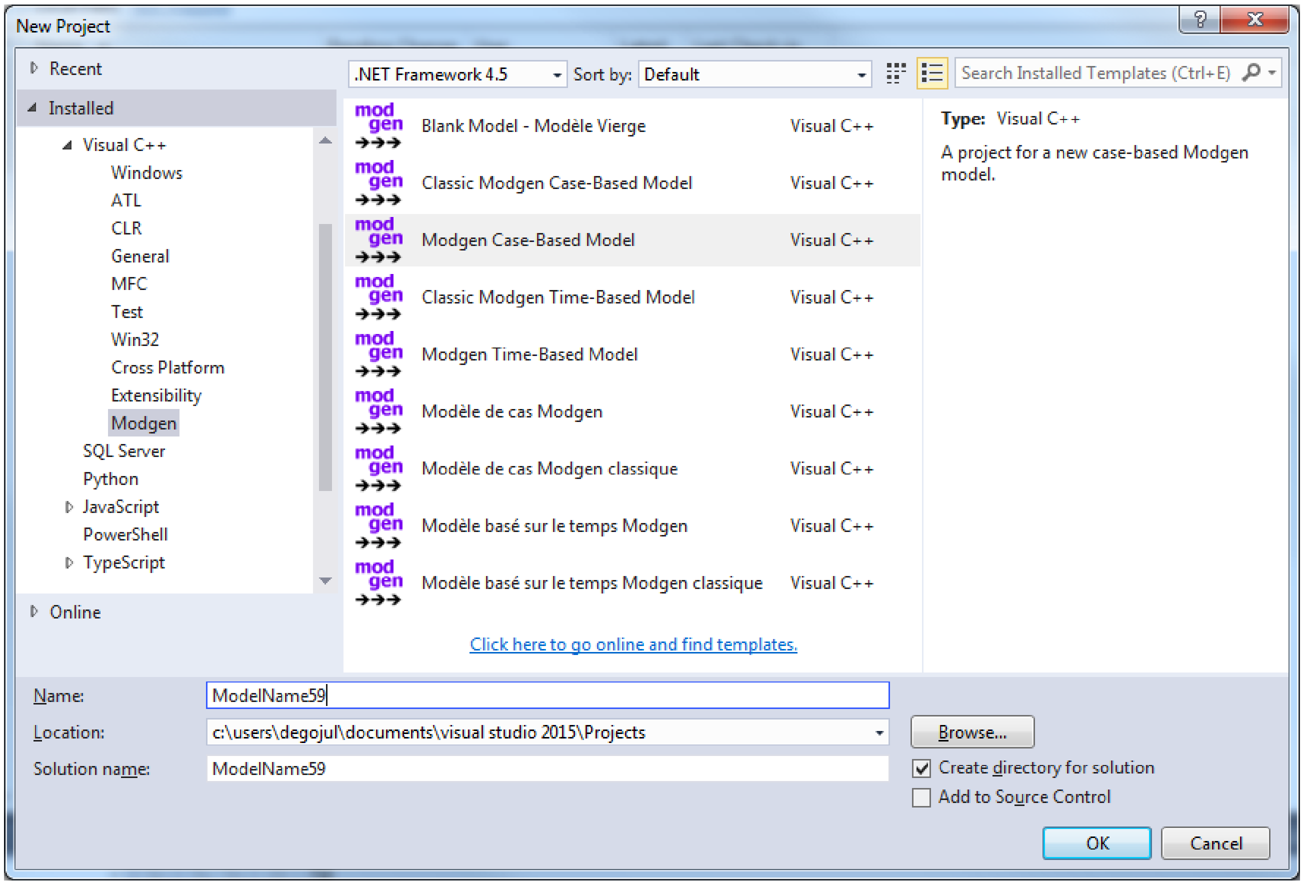
The following diagram shows the relationships between the various files needed to generate a simulation model executable file for the application.



New model wizard

To facilitate the creation of a new model, Modgen includes a wizard that helps to establish the underlying foundation or structure for either a new case-based model or a new time-based model.   To use the wizard, first open Visual Studio, choose **New** on the **File** menu, and then choose **Project**.

The following "New Project" dialog box appears:


New Project

Recent
Installed
Visual C++
Windows
ATL
CLR
General
MFC
Test
Win32
Cross Platform
Extensibility
Modgen
SQL Server
Python
JavaScript
PowerShell
TypeScript
Online

.NET Framework 4.5   Sort by: Default   Search Installed Templates (Ctrl+E)

Blank Model - Modèle Vierge	Visual C++
Classic Modgen Case-Based Model	Visual C++
Modgen Case-Based Model	Visual C++
Classic Modgen Time-Based Model	Visual C++
Modgen Time-Based Model	Visual C++
Modèle de cas Modgen	Visual C++
Modèle de cas Modgen classique	Visual C++
Modèle basé sur le temps Modgen	Visual C++
Modèle basé sur le temps Modgen classique	Visual C++

Click here to go online and find templates.

Type: Visual C++

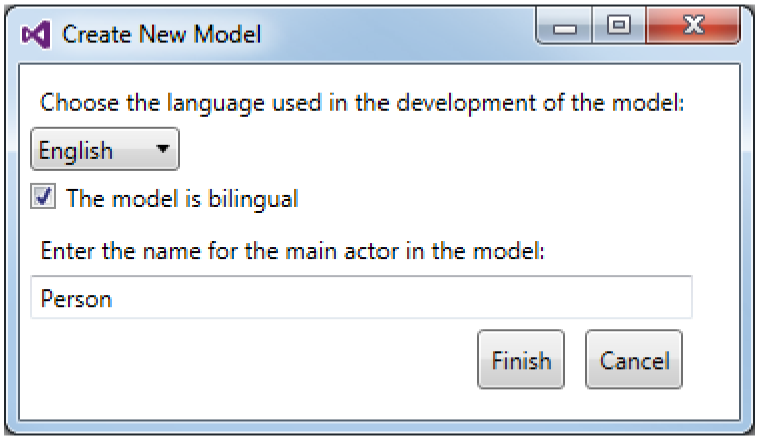
A project for a new case-based Modgen model.

Name: ModelName59
Location: c:\users\degojul\documents\visual studio 2015\Projects
Solution name: ModelName59

Browse...
[✓] Create directory for solution
[ ] Add to Source Control
OK   Cancel

To start the wizard for a new Modgen project, you should ensure that 'Modgen' is selected as the project type (under the Visual C++ category) and that either 'Modgen Case-Based Model' or 'Modgen Time-Based Model' is selected as the corresponding Visual Studio installed template. Next, enter a Name for your model or project, which by default also becomes its Solution Name, and click **"**OK"**.**

This in turn leads you to a Create New Model screen dialog on which you can specify the name of the model's main actor, the model's working language (English or French), and an indication of whether or not the model is bilingual, as shown in the following example.   (While this example shows the screen for a new case-based model, the entities that can be adjusted are identical to those which can be changed for a new time-based model.) By clicking "Finish" on the Welcome screen, the underlying structure of the new model is generated as a .sln file. The .sln file includes another file, 'readme.txt', which outlines the exact contents of the .sln file.



Choose the language used in the development of the model:

English

[✓] The model is bilingual

Enter the name for the main actor in the model:

Person

Running a model simulation

Running the model executable

When invoked, a compiled model's executable provides the user of that model with a generic visual interface for editing parameters, running the simulation and browsing the simulation outputs. This interface is fully described in a separate publication, "*Guide to the Modgen Visual Interface*"; however, a brief description is provided here. Note that you, as the developer of the model, must provide an initial default set of parameter values (in .dat files) that are necessary for the model to run. This initial set is often referred to as the base case values.

A compiled model's executable will not launch a simulation without a saved scenario file (.scex extension). A Modgen scenario file is an xml file containing the settings necessary to run a simulation. There are two methods to run a simulation for an existing model:

- by double-clicking on the model's executable file in Windows Explorer (or by starting it, either without any arguments or with a previously saved scenario file, via **Start > Run**). This opens Modgen's visual interface; at that point you can use **Scenario** > **Settings** to verify or modify your simulation options, and then **Scenario** > **Run** to start the actual simulation run.
- in batch mode, directly on the command line by supplying the scenario file with the –sc command line argument.

Method 1: Invoking the model's visual interface (Scenario > Run)

model_name <scenario_filename>

- No other arguments are permitted.
- If no argument is provided, the visual interface for the model is started but with no open scenario file

- Use **Scenario** > **Run** to launch the actual simulation run.

Method 2: Running the model in batch mode

model_name –sc scenario_filename <-computational_priority>

- This runs the model with scenario settings taken from scenario_filename, and provides an optional override for computational priority. In batch mode, the visual interface is still opened but its screen is minimized.
- The computational priority is optional and can be either normal or low, with the default value being low.
- No other arguments are permitted

Running the model in documentation-only or help-only mode

A model can also be run in either documentation-only mode or help-only mode. The former generates an updated documentation database which is useful, for example, when using the Translation Assistant tool available with Modgen. The latter creates help files for all of the languages known to the model. No actual simulation run takes place in either of these modes.
Syntax:

model_name -doc <filename> <-s>

- This creates a documentation database for your model. There is no visual interface when the model runs in documentation-only mode, and no simulation run takes place.
- If no filename is specified, the documentation database is named 'model_name(doc).mdb'
- If a filename is given but it has no extension, .mdb is automatically added to the filename.
- Upon completion, a message box appears to indicate either a successful completion or a failure. This message box is disabled if the optional "-s" argument (silent mode) is specified.

model_name -help

- This creates help files for all the languages known to the model. There is no visual interface when the model runs in help-only mode, and no simulation run takes place. The files are compiled help files called <model_name>2_<language_code>.chm , as long as a .chm compiler is installed on the machine. (Otherwise, the help files are created as HTML files and stored in folders called Gen<language_code>; within each such folder, the main or starting HTML file is named '2.htm'.). It is also specified to avoid the compilation into .chm format and preserve the html files by adding the argument "-html".

Running the model in a distributed mode

Either the subsamples from case-based models or the replicates from time-based models can be distributed across machines on the network. This capability is achieved through the functionality of Executor, a Statistics Canada application that pools available workstation resources together to optimize job execution opportunities. In any given scenario of a distributed execution run, Modgen models automatically use the pool from the instance of Executor running on the machine.

The roles of master and slave jobs in a distributed run are as follows. When a distributed job is started, the master job creates the slave jobs. Once those jobs are created, the master job itself becomes a regular slave job. Like all other slave jobs, it will run then exit. Before exiting, each slave job checks to see if it is the last job to be completed. If this is true—that is, if all the other slaves have been successfully completed—then the last-running slave becomes the new master. The new master then reassembles the data and removes the slave jobs from the request database.

As a result, a slave can be resubmitted independently. Should all but one of the slave jobs have been successful, you can resubmit that particular slave job. If the resubmitted run is successful, the corresponding slave will become the new master and reassemble the data.

Phases of running a model

Generally, when a model simulation run takes place, the run can be thought of as sequentially progressing through six stages or phases, although only three of the six stages are mandatory:

- Parameter reading (mandatory)
- Parameter validation (optional)
- Presimulation (optional)
- Simulation (mandatory)
- Postsimulation (optional)
- Tabulation (mandatory)

Parameter reading

The first step is the reading of the parameter values into the model.   This capability is provided automatically by Modgen and is discussed further in “Initializing model parameter values in .dat files”.

Parameter validation

You can specify many different kinds of validation functions in order to help validate or confirm that the values being input for the parameters make sense within the model’s context.   These validation functions are described in more detail in “Parameter validation routines and parameter normalization”

PreSimulation

After all of the parameters have been read (and validated if you have incorporated validation), the presimulation phase begins, as defined in one or more **PreSimulation()** functions. (Each module of a model can have its own separate **PreSimulation()** function). Often, a presimulation function is used to calculate the values of model-generated parameters. A presimulation function, however, cannot be used to change the states of actors because, at this phase, the actors have not yet been created.

Simulation

The simulation phase is the major phase of a model run, and every model MUST contain a **Simulation()** function. Modgen's new model wizard creates such a function for both new case-based and new time-based models. Once created, the **Simulation()** function rarely needs to be modified.

In a case-based model, the **Simulation()** function essentially creates a case simulation loop to move through each case, one at a time, calling the **StartCase()**, the **CaseSimulation()**, and **SignalCase()** functions respectively within that loop. (The **StartCase()** and **SignalCase()** functions are internal to Modgen, return no results, and are not found in the Modgen model code. **StartCase**() notifies Modgen that the next case is about to begin, whereas **SignalCase**() tells Modgen that the case has completed and updates the simulation progress bar in the visual interface.) The **CaseSimulation()** function actually controls the simulation of the case. It is a standard C++ function and has no special meaning to Modgen. It creates the next actor by calling the C++ **New** operator and a **Start()** function, processes an event queue loop to simulate the case, and concludes by removing all current actors in the case.

In a time-based model, the **Simulation()** function is more complicated because it has to include both the creation of the entire starting population and the event loop that controls the simulation of this entire interacting population.

PostSimulation

This phase, defined within a **PostSimulatiou()** function, contains the code to be run after the simulation phase but before data is tabulated. In practice, this phase is not commonly used, although one possible role can be to clean up or reinitialize any global variables that are used within the model.

Tabulation

This last phase of a model run produces the tables. In one sense, this is a misleading description since the tabulation actually occurs on-the-fly as the model simulation run takes place. In this phase, however, the tables used in that simulation run are actually saved into an output database. You can also optionally establish additional tables called "user tables" that are constructed from values in the regular tables that were built on-the-fly. These user tables need to be defined upfront via a separate **UserTables()** function within your model code, but it is at this final phase of the simulation run that the user table values are actually calculated and saved within those user tables.

Modgen model output files

Modgen manages model run output by using the scenario name followed by a bracketed identifier indicating the type of Modgen output and an extension indicating the type of file. The files can be grouped into three categories:

Table outputs for aggregate model results

scenario_name(tbl).mdb – Microsoft Access output. This file is always produced by a model simulation run and used by the visual interface for table browsing. Note that this browsing capability does NOT require that Access itself be installed on the machine.

scenario_name(tbl).xls – Microsoft Excel table outputs. This is the default name used by the Excel table export feature of Modgen. This file is created by first selecting **Scenario** > **Export** from the visual interface menu and then choosing "MS Excel format"

scenario_name(tbl).txt – Text table outputs. This is the default name used by the text table export feature of Modgen. This file is created by first selecting **Scenario** > **Export** from the visual interface menu and then choosing "Text format"

Tracking outputs for examining individual life histories

scenario_name(trk).mdb – Microsoft Access tracking output used by the BioBrowser application. This file is only produced if it is requested via the **Scenario** > **Settings** dialog box of the visual interface. Note that BioBrowser does NOT require that Access also be installed on the machine.

scenario_name(trk).txt - Text tracking output. This file is only produced if it is requested via the **Scenario** > **Settings** dialog box of the visual interface.

Log and error outputs

scenario_name(log).txt – Modgen log file (always produced). This file contains the time at which the simulation started, as well as selected information on workstation characteristics such as computer name, user name, number of processors, and processor model. The version of Modgen and the name and version of the model are also included.

If the model should crash or hang, the log file contains additional diagnostic information that may be useful to the Modgen development team in identifying the cause of the problem.

The final total performance statistics are written at the end of the log file. However, interim performance statistics are also written periodically to the log file during a simulation run. As a result, this file can be used to monitor the progress of a simulation run while the run is still ongoing.

In a distributed simulation run, the log files of each slave job are merged into one master log file (with the same final name, 'scenario_name(log).txt') after which the individual slave log files are deleted.

scenario_name(err).txt – Modgen error file (produced only if errors are encountered when parsing the parameter input files).

scenario_name(trc).txt – Modgen file containing detailed information about errors in case a model crashes unexpectedly; this file is deleted if no errors are encountered.

scenario_name(debug).txt – additional Modgen log file containing the results of event tracing and/or checksums if either of those options were activated with the **options** command.

Elements of the Modgen language

This section describes the elements of the Modgen language which are not part of the standard C++ language. These elements greatly facilitate the representation of a microsimulation model in a structured programming environment. Almost all Modgen elements must be defined before they appear in simulation code. Therefore, the definition of these elements not only allows for the successful compilation of the model, but also serves as a source of documentation of how the elements work together.

Defining the type of microsimulation model

Modgen allows the creation of a case-based or a time-based model. Therefore, it is necessary to specify type of model being constructed. To do this, use one of the following statements (near the top of your main.mpp file):

Syntax of model_type statement:

```
model_type case_based <,just-in-time>;
model_type time_based <,just-in-time>;
```

The **just_in_time** keyword is optional and affects the advancement of time when an event occurs. It is usually only found in time-based models but is allowed for either category of model; it is discussed further in “The simulation engine”.

The appearance of the scenario settings screen in Modgen’s visual interface will also vary, based on the contents of the model_type statement.

It is also necessary to specify if the model is using discrete or continuous time. This is done through the time_type statement. Although several types are allowed, the most common type, and default type is the “double” for continuous time models. The type “float” is also used, in particular when memory is limited. For discrete time models, the time type would generally be “int”, although other types allowed are char, short, int, long, uchar, ushort, uint, or ulong. (The types uchar, ushort, uint or ulong are synonyms used in Modgen for unsigned char, unsigned short, unsigned int, or unsigned long, respectively). If the time type is omitted in the Modgen code, Modgen uses double as a default.

Modgen also defines the type TIME through the time_type definition. The TIME type is often used to define states which are closely linked to the model's concept of time.

Modgen automatically declares the states ***age*** and ***time*** through the time_type definition. Thus, you do not need to specify an age or time state in any of your model's actor declarations, even though no dynamic microsimulation model would be complete without them.
Examples:

```
// This is an example for a discrete time model

time_type int;

// For a continuous time model, use one of the following two choices:

time_type double;
time_type float;
```

Declaring the actors

The actor declaration or actor block contains the list of all states, events, and actor functions associated with the actor. You do not need to specify an ***age*** state in this declaration because Modgen automatically supplies it, based on the time_type definition. Although the actor declaration lists all events and actor functions, the source code for the events and actor functions, which can make use of all elements of C++, is outside of the actor declaration.

<u>Syntax of an actor declaration:</u>

```
actor name //language_code  label
{

        type  simple_state;                 // simple state without initialization
        type  simple_state = {value};       // simple state with initial value contained within curly braces
        type  derived_state = expression;   // creates a derived state which does not have to be assigned to
                                            // an actual state name
        event event_time_function, event_implementation_function;    //event functions  associated with the actor
        actor_function_prototype;           // C++ function which is called by an event function
                                            // or the simulation engine

};
```

A label, or short description, can be added for each entity declared in the actor block by using double slashes immediately followed by a language code (e.g. EN or FR). An example of such a specification (for a simple state called

earnings) is:

```
float earnings={0};  //EN Lifetime earnings to date
```

Example of actor declaration or actor block:

```
// Declaration of a “person” actor.

// In this simple example, there is only one event for the person

// actor which occurs once a year to calculate the person's disability status

// and corresponding annual income conditioned on the person being alive.

// This event is called AgeOneYearEvent.

// To estimate the waiting times between the execution of this event,

// the timeAgeOneYearEvent function is used.

// In this example, SEX, DFLE_STATE and LIFE_STATE are classification types;

// sex, earnings, dfle and life_status are simple states;

// duration, value_at_changes, and value_at_entrances are

// derived state specifications that create derived states.

// The model elements ‘classification type’, ‘simple state’ and

// ‘derived state’ are all defined elsewhere in this section.


actor Person            //EN Individual actor definition

{

   // States required for this specific simulation

   // Note: age is also required but Modgen automatically generates an age state

   SEX   sex;                         //EN Sex

   int   earnings = {0} ;             //EN Lifetime earnings to date

   DFLE_STATE   dfle = {NO_DIS};      //EN DFLE State

   LIFE_STATE   life_status = {ALIVE}; //EN Life Status


   // states required for tabular reports
   int   person_year = duration (life_status,ALIVE);   //EN Person year

   int   age_at_death = value_at_changes ( life_status, age ); //EN Age at death
(method 1)

   int   death = value_at_entrances ( life_status, NOT_ALIVE, age ); //EN Age at
death (method 2)

   // Functions to define the one single event in this example

   event timeAgeOneYearEvent, AgeOneYearEvent;

   // Functions required by the simulation engine

   void Start();

   void Finish();

};
```

Default argument values for actor function prototypes

Modgen supports default argument values for actor functions. Functions declared with default argument values can be called with fewer arguments than required by the function prototype.
Example

In this example, the specified defaults are used for each of birth, immig, eSex, and eProv in the call to the **Start()** function.

```
actor Person

{

   void   Start( logical bTentative, logical dom,  TIME birth = TIME_INFINITE,

      TIME immig = TIME_INFINITE, SEX eSex = MALE, PROVINCE eProv = ONT );

};

prDominant->Start( FALSE, TRUE ); // defaults used for other four arguments
```

Actor sets

Actors can also belong to actor sets. An actor set is a collection of actors dynamically maintained by Modgen, i.e. Modgen internally handles membership in the actor set based on criteria that you provide.

Actor sets are useful in time-based models with many interacting actors.   Actor sets are intended to facilitate the development of such models by eliminating the need for you to write code to maintain and manipulate the sets. For example, Modgen provides functions that can be used to select an actor randomly from the set, and to iterate over all members of the set.

Actor sets are discussed in much more detail in “Interactions amongst actors—links and actor sets”.

Defining the types

While all C++ types are available for use by Modgen models, there are additional special types available with Modgen that can, in turn, define the states (and sometimes the model parameters as well) used by the model. Some special types are available to any Modgen model, namely TIME, logical, real, counter, and integer. Other types are created for each specific model with the assistance of the keywords **range**, **partition** and/or **classification**.

Modgen types available to any Modgen model

TIME

The Modgen type TIME, has special meaning in Modgen. Modgen uses it to distinguish between discrete and continuous time models. TIME is used to declare states that store time values and to define the return type of several functions. It takes its value from the definition of the time_type statement.

logical

Boolean states or parameters are defined using the logical type. The values supported for logical states are FALSE (zero) and TRUE (1).

Example

```
// The state, in_union, is TRUE when the marital_status state
// becomes either MS_MARRIED or MS_COMMON_LAW
logical in_union = marital_status==MS_MARRIED || marital_status==MS_COMMON_LAW;
```

real, counter, and integer

These three Modgen types can be used to declare actor states that contain floating point (real type), non-negative integral (counter type), and integral (integer type) values. The actual C++ types that get used in the model are controlled by the real_type, counter_type, and integer_type statements, respectively. These types and type statements are described further in “Debugging and optimizing tools and techniques”. Note however that these types are not allowed for parameter declarations.

Special types available to a specific model only

range

A range is defined by a set or interval of integer values between a specified minimum and maximum value (which may be equal). It is used for defining the size of model parameters or data members. (A data member is a multidimensional state that can be modified in a model's code but that, unlike simple states, cannot be used in expressions.) It is also used to declare states.

Syntax of a range definition

range name { min_value, max_value }; //language_code label

The label is preceded by a language code (e.g. EN or FR) followed by a space; the language code immediately follows the double slashes.

Examples:

```
// The range, LIFE, is defined by the integers 0 through 99
range LIFE      { 0, 99 };      //EN duration of life

// The range, DFLE_AGE, is defined by the integers 15 through 99
range DFLE_AGE   { 15, 99 };   //EN age at dfle changes

// In these examples, LIFE and DFLE_AGE become types that are defined using the
// keyword "range" and that are available only to the current model.
```

If your model code tries to assign a value to a state that exceeds the maximum value of the range, then the maximum value of the range is assigned to that state instead. Similarly, if the code tries to assign a value lower than the minimum value of the range to a state, the minimum value of the range is assigned to the state instead. (This default behaviour can be overridden, however, by the "bounds_error" debugging option.)

The following macros are supplied with Modgen to help you work with ranges:

COERCE(*range_name, state_name*)
Forces the value of the state to fall within the lower and upper limits of a range.
Example:

```
range REP_YEAR70 { 1970,1979 };

COERCE(REP_YEAR70, year);
```

Returns the actual year if the state, year, is between 1970 and 1979 inclusive; returns 1970 if year is less than 1970; returns 1979 if year is greater than 1979. The COERCE statement in this example is equivalent to:

```
(year < MIN(REP_YEAR70)) ? MIN(REP_YEAR70) : (year > MAX(REP_YEAR70)) ?
```

```
MAX(REP_YEAR70) : year;
```

MAX(*range_name*)

Returns the maximum value of the range.

MIN(*range_name*)

Returns the minimum value of the range.

RANGE_POS(*range_name***,** *state_name*)

Converts range values to 0-origin indices. This is particularly useful for indexing the array of parameters where range is one of the dimensions. **RANGE_POS()** also handles out-of-range values, forcing them to fall within the range.

Example

```
range CALENDAR_YEAR

{

      1899, 1999

};

nYearInd = int( current_time );

fte_earnings = 52 * weekly_hours * earn_rate*

                  AvgIndWage[RANGE_POS( CALENDAR_YEAR, nYearInd )];
```

SIZE(*range_name*)

Returns the size of the range

WITHIN(*range_name***,** *state_name*)

Returns a logical value indicating whether a given value of the state falls within the upper and lower bounds (inclusive) of the given range.

Example

```
range REP_YEAR70 { 1970,1979 };

WITHIN(REP_YEAR70, year);
```

The code in the WITHIN statement is equivalent to

```
year >= MIN(REP_YEAR70) && year <= MAX(REP_YEAR70);
```

partition

A partition defines a strictly increasing set of breakpoints for a continuous numeric quantity. It is typically used in the specification of model parameter dimensions or in tables as an argument to the **split()** or **self_scheduling_split()** derived state specifications or to the **SPLIT()** function.. When used with **split()** or **self_scheduling_split(),** Modgen will partition or separate the continuous state into the subgroups defined by the partition name.

Syntax of a partition definition

partition partition_name //language_code label

{

double *breakpointvalue*, ... , double *breakpointvalue*

};

Example

The last line of this example aggregates the values of XYZstate into age groups 'under 15', '15 to under 20', '20 to under 25', '25 to under 30', etc., finishing with '50 and over'

```
// In this example, AGE_GROUP1 becomes a type that is defined using the
// keyword "partition" and that is available only to the current model

partition AGE_GROUP1

{

      15, 20, 25, 30, 35, 40, 45, 50

};

parameters
{
    float    AvgIndWage[AGE_GROUP1];  //EN Average industrial wage
};

split(XYZstate, AGE_GROUP1)
```

The following run-time macros are supplied with Modgen to help you work with partitions:

POINTS(*partition_name*)

Returns the values defining a partition as an array of floating point (type "real") numbers. Therefore, POINTS(partition_name)[0] gives the first point in the partition, POINTS(partition_name)[1] the second point, etc.

Example:

Assume that the following partition is defined:

```
partition AGE_PARTITION //EN Age group
{
      15, 45, 65
};
```

Then `POINTS(AGE_PARTITION)[0]` has a value of `15`, `POINTS(AGE_PARTITION)[1]` has a value of `45` and `POINTS(AGE_PARTITION)[2]` has a value of `65`.

Example:

The following code sample computes the lower and upper bounds of the age group nAgeGroup.

```
int nAgeGroup = 0;
double dLower = 0;
double dUpper = 0;

// lower limit of age group
if ( nAgeGroup == 0 )
{
    // set lower to be minimum age (zero)
    dLower = 0.0;
}
else
{
    dLower = POINTS(AGE_PARTITION)[nAgeGroup - 1];
}

// upper limit of age group
if ( nAgeGroup == SIZE(AGE_PARTITION) - 1 )
{
    // set upper to be maximum age in model
    dUpper = MAX(LIFE_SPAN);
}
else
{
     dUpper = POINTS(AGE_PARTITION)[nAgeGroup];
}
```

Note that the macro **SIZE**(*partition_name*) gives the number of intervals in the partition, and not the number of points that define the partition. This is consistent with the use of **SIZE()** for ranges and classifications: **SIZE()** gives the number of distinct values in the range, classification, or partition. The number of intervals in a partition is always one more than the number of points defining the partition. In other words, the number of elements in the array **POINTS**(*partition_name*) is always equal to **SIZE(***partition_name*) – 1. Note that because index values of arrays in C++ always start at 0, the maximum allowed index for the array **POINTS**(partition_name) is **SIZE**(partition_name) – 2

SIZE(*partition_name*)

Returns the size of the partition, i.e. the number of intervals defined by the partition (which is 1 + the number of breakpoints defined by the partition)

SPLIT*(dValue, partition_name*)
Partitions *dValue* by the classification groups defined in *partition_name*. This function is typically used in the **PreSimulation()** function to initialize model-generated parameters which use a partition as one of the dimensions.

Note the **SPLIT()** uses the same arguments as **split()** which is used to define a derived state. Thus, **SPLIT()** can only be used in simulation code while **split()** can only be used in actor or table definitions.

classification

A classification type is defined as a collection of levels together with their labels. The classification type may be used to define the dimensionality of a state or model parameter.

Syntax of a classification definition

```
classification classification_ name  // language_code1 label1 <//language_code2 label2…>
{
        level,    //language_code1 label1 <//language_code2 label2…>
        level,    //language_code1 label1 <//language_code2 label2…>
        level     //language_code1 label1 <//language_code2 label2…>
};
```

Examples

```
// In these examples, SEX and DFLE_STATE become types that are defined using the
// keyword “classification” and that are available only to the current model


// A classification type, SEX, is defined by its two gender levels
classification SEX
{
      //EN Gender
      S_FEMALE,    //EN Female
      S_MALE       //EN Male
};
```

```
// A classification type, DFLE_STATE, is defined by four disability status
// levels or categories
classification DFLE_STATE     //EN Disability State
{    DS_NO_DIS,           //EN No Disability
    DS_MILD_DIS,         //EN Mild Disability
    DS_MODERATE_DIS,     //EN Moderate Disability
    DS_SEVERE_DIS        //EN Severe Disability
};
```

Modgen also provides a facility to convert the values from one classification into those of a second classification, as long as the second classification is a strict aggregation of the first. The procedure to accomplish this, which uses both the **aggregation** keyword and the derived state specification **aggregate()**, is discussed in detail in "Technical topics in Modgen".

The following run-time macro is available in Modgen to help you work with classifications.

SIZE(*classification_name*)

Returns the size of the classification

Declaring the states

States describe all of an actor's characteristics. (In C++, they are properties of actor objects). States are declared in the actor declaration or actor block for each actor, and there is no limit on the number of states that can be declared. There are two general categories of states: simple states and derived states. (Note as well that there are other types, such as multi-dimensional arrays, that can be declared in the actor declaration but that will not be treated by Modgen as states; such entities are called data members. These declarations become data members of the object class, in C++ terminology).

Simple states

Simple states are always scalars whose types are one of the regular choices available to C++ programmers, such as int, long, float and double, or one of Modgen's special types, such as TIME, logical, classification or range. (The types char and short can also be used for small signed integers. In addition, the types uchar, ushort, uint and ulong are available for unsigned integer values, where uchar is a synonym for the C++ type "unsigned char", ushort is a synonym for the C++ type "unsigned short", uint is a synonym for the C++ type "unsigned int", and ulong is a synonym for the C++ type “unsigned long”.)

Simple states can be initialized either with their declaration (using curly brackets), or in the **Start()** function of the actor. During the simulation, actor states are modified within event implementation functions. If a simple state is assigned an initial value, that value must be enclosed with curly braces.

Example

```
double earnings = {0};
```

If curly braces are not used, the implication is that the state is a derived state whose value is maintained by Modgen at all times. Any statement in model code that attempts to modify a derived state generates an error when the model is compiled.

Identity States

An identity state is declared using a C++ arithmetical/logical expression which involves other states. Its value is maintained as an identity automatically by Modgen as the dependent states change. An identity state is conceptually similar to a cell in an Excel worksheet which is specified through a formula which involves other worksheet cells.

Example:

```
actor Person {
   double total_earnings = wages + self_employed_earnings;
};
```

This example creates an identity state total_earnings whose value will be maintained automatically by Modgen whenever the value of wages or self_employed_earnings changes. The identity state total_earnings can be used freely in model code with the assurance that it will always have the correct value given by the formula which declared it.

Derived states

A derived state is a state maintained internally by Modgen which has a fixed simple relationship to the current and previous values of other states. Unlike an identity state, this relationship cannot be specified using just the current value of other states. For example the derived state **duration()** is based on the built-in state **time**. The value of **duration()** is the total amount of time the actor has been in the simulation, up to the current event in the simulation. It could be computed internally by Modgen as the difference in the current value of **time** and the value **time** had when the actor first entered the simulation.

A derived state requires no declaration as such, it just needs to be used in a table expression or an identity state declaration to spring into existence. Derived states are created via derived state specifications which can be broken down into different categories according to their functionality:

- Event counts
- Event occurrences

- Event conditions
- Event triggers
- Durations
- Spells
- Transformations
- Links
- Self-scheduling states

The derived state specifications in the first four categories all create derived states associated with changes to the states of actors.   The derived state specifications in the durations and spells categories create derived states associated with durations of the states of actors. The derived state specifications in the transformation category create states by directly transforming other states, while those in the links grouping provide information about other actors who are linked. Finally, the derived state specifications in the self-scheduling state category introduce the notion of a time interval to all other states but they differ from the other states in that they create additional events that cause the time of the simulation to actually stop at designated points of time.

Overall, the collection of derived state specifications available through Modgen is extensive in number and diverse in functionality. They are discussed in much more detail in the chapter "Derived state specifications".

Built-in states

Modgen creates the following special states for each actor:

actor_id

The ***actor_id*** state is a unique identifier for an actor that is generated by Modgen for each actor in the model. It can be read by model code but not modified. Note that while ***actor_id*** uniquely identifies each actor in a simulation, it should not be used as an identifier to compare actors across different simulation runs. Its value is guaranteed to be unique only within a specific simulation.

actor_subsample_weight

The ***actor_subsample_weight*** state is equal to ***actor_weight*** multiplied by the number of sub-samples. This state indicates the weight applied to sub-sample quantities, whereas ***actor_weight*** is applied to total quantities. The ***actor_subsample_weight*** state may be tracked and displayed with the BioBrowser application.

actor_weight

A special state called ***actor_weight*** is available for every actor. A call to the **SetCaseWeight()** function sets the value of the ***actor_weight*** state for every actor in the case. The ***actor_weight*** state may be tracked and displayed with the

BioBrowser application.

age

The ***age*** state indicates the age of the actor and is defined by the model type TIME (which in turn is dependent on the model's **time_type** definition. This state can be set only in the actor's **Start()** function, if not set there it will be initialized to zero. Thereafter, it can only be referenced in model code.

case_id

The ***case_id*** state contains a unique identifier for each case and is of type long. For case-based models, this state identifies the current case number, based on origin 1. (The state is not used in time-based models.) The ***case_id*** state is always tracked in all case-based models. This state can be referenced in model code but never set or modified by the code.

case_seed

The ***case_seed*** state indicates the starting seed of the current case and is of type double. (The state is not used in time-based models.) Note that two random number generators are used for each subsample, one to generate the starting seed for each case, the other to generate the stream of random numbers for the simulation run. This results in more consistent model runs, as an extra call to the random number generator in one case will not affect the starting seed for the next case. As is the situation with ***case_id***, ***case_seed*** can be referenced in model code but never set or modified by that code.

time

The ***time*** state indicates the current time and is typed by the model type TIME. This state is also used for the state tracking which is part of the model debugging facility. This state can be set only in the actor's **Start()** function; if not set there it will be initialized to the time of the simulation at the actor's creation (which can be but does not have to be zero). Thereafter, it can only be referenced in model code.

Modgen also generates labels internally for these states in both English and French, as per the following:

	English label	French label
actor_id	Actor identifier	Identificateur d'acteur
case_id	Case identifier	Identificateur de cas
case_seed	Case random seed	Valeur aléatoire d'amorçage de cas
time	Time	Temps
age	Age	Âge
actor_weight	Weight	Poids
actor_subsample_weight	Sub-weight	Poids du sous-échantillon

Declaring and defining the model's events

Actors evolve through time through events. Each event in Modgen is declared via two functions. The first function returns the expected time for the given event to occur, conditional on no other event occurring first. The second event function consists of code that implements the given event (i.e. that specifies what happens when the event occurs) by changing the values of states. An example of an event is the loss of a job. In this example, a hazard of job loss as a function of an individual's characteristics (such as duration of current job) could be used in the waiting time portion of the event time function. The event implementation function could in turn consist of code to set state values that describe the individual's new employment status and earnings rate after job loss.

Modgen's simulation engine moves the actors through time in a simulation run, using the events defined in the model; the manner through which this takes place is described in "The simulation engine".

As stated above, you must code two C++ functions to correspond to each model event. All elements of the C++ language can be used in these functions. Both of these functions are declared in the **event** definition which is a component of the actor definition.

Syntax of an event declaration

event time_function, implementation_function <, priority_code>**;**

Note that time_function, implementation_function and priority_code (if used) are separated by commas.

- time_function returns the absolute time of the next occurrence of the event. It usually has no arguments but it can have an optional argument that is a pointer to an int variable, as described in "Events with Memory". There are three rules which must be followed when coding a time function (or else the Modgen compiler generates errors). First, you cannot change the values of a state in an event time function--you can only refer to such a state. Second, you cannot use pointers, actor functions or continuously updated states within an event time function. Finally, you cannot use continuously updated states, i.e. age, time or a duration state.
- implementation_function does not return a result and usually has no arguments but it too can have an optional argument that is an int variable, as described in "Events with Memory".
- priority_code is optional. If absent, the priority code is deemed to be 0; when present, the code must either be an integer in the range of 0 to 250, or else the keyword **time_keeping**. The **time_keeping** keyword, however, should only be used in the declaration of an event whose purpose is to keep track of time, such as the "Clock" event in Statistics Canada's LifePaths model and the "Ticker" event in some time-based models. Events with this priority will automatically be assigned a higher priority than the events for which you explicitly declared the priority; however, events with the time_keeping priority will have a slightly lower priority than that assigned to the event created for updating self-scheduling states. The **time_keeping** keyword can only be used once in your model.

  Note that priority codes are checked when there is a tie between two or more events. In such a situation, the event with the higher priority code is implemented first. If no priority code is specified, Modgen assigns a priority code of 0 (i.e. the lowest priority) to the event. In case of a tie between priorities, the events are executed in alphabetical order of event name. In case of a tie between identically-named events of different actors, the order of the creation of the actors is used, which means that the event for the first actor who was created will be executed first. Priorities should be assigned to events when there is a possibility of having simultaneous events. In particular, a "Clock" or "Ticker" event should always have the highest priority, which will automatically be the case if they are declared with the time_keeping priority.

Generally speaking, it is possible to categorize events as one of:

- hazard events
- clock type events

In a hazard event, the timing of the event is determined using hazards. The hazards are normally calculated using event history analysis and are fed directly into the microsimulation model.

Clock type events are events that occur at predictable times. An obvious example of this is a "Year" event which occurs on the first of January of every year and changes the year, which could be useful if, for instance, hazards of migration changed depending on the year. Another example is a "Birthday" event, which is shown in Example 2 that follows. Events that occur after an actor has been in a given state for a specific period of time are also considered to be clock-type events. As an example, a disease progression event that changes the state of the disease from latent to infectious and later immune would be a duration event and therefore a clock event.

Example 1 – a hazard event for mortality

Event declaration

```
actor Person
{
      //EN The person's death
      event timeMortalityEvent, MortalityEvent;

};
```

Event time function definition

```
TIME Person::timeMortalityEvent()

{

      TIME tEventTime = {TIME_INFINITE};
      // Draw a random waiting time to death from an exponential
      // distribution based on the hazard MortalityHazard.
      tEventTime = WAIT( - TIME(  log(  RandUniform(1)  )

            / MortalityHazard [modeled_age] )  );

      return tEventTime;

}
```

Event implementation function definition

```
void Person::MortalityEvent()

{
      alive = FALSE;
      // Remove the actor from the simulation.
      Finish();

}
```

In this example, ***MortalityHazard*** is a parameter which contains piecewise hazards of dying that change with each year of age, and ***modeled_age*** is a state of type range. The equation used in the time function assumes an exponential or uniform distribution, but other distributions can be used if necessary. **WAIT()** is a Modgen macro that takes a time increment and converts it to absolute time. Modgen automatically handles the recalculation of the event time each time the modeled age changes via the Modgen simulation engine, as described in "The simulation engine".

Example 2 – a Birthday event

Event declaration

```
actor Person            //EN Core Individual
{
      TIME  next_birthday;    //EN Next Birthday

      // Priority code 249 is used for this event

      event timeBirthdayEvent, BirthdayEvent, 249; //EN Birthday event

};
```

Event time function definition

```
TIME Person::timeBirthdayEvent()
{
      return next_birthday;
}
```

Event implementation function definition

```
void Person::BirthdayEvent()
{
      ...
      integer_age++;
      next_birthday += (TIME) 1;
}
```

In this example, the event time function simply returns the value of an actor's state, i.e. ***next_birthday***. This state is initialized, at the beginning of the simulation of an actor, to the time of the actor's next birthday. After that, it is maintained by the event implementation function.

Events with memory

Modgen includes the ability to record auxiliary information in the event time function that can later be retrieved and used by the associated event implementation function. This capability can be useful in situations where the waiting time is determined together with numerable particularities about the eventual event implementation. An example is the migration event of Statistics Canada’s LifePaths model in which a potential migration event has a waiting time as well as an associated destination that is inextricably associated with the waiting time.

It is possible for the time function of the migration event to record the associated destination, so that the destination can be set by the migration event’s implementation function if and when the migration occurs. This ability to save and retrieve event-specific auxiliary information can be activated as desired on an event-by-event basis.

To activate this capability for a particular event, the time and implementation functions of the event need to be coded to take an argument. Specifically, the time function needs to take as argument a pointer to an int variable, and the implementation function needs to have an argument of type int. The following example illustrates the process.
Example:

```
actor Person
{
      event timeClockEvent, ClockEvent, 250;
};

TIME Person::timeClockEvent(int *pnEventInfo)
{
      // Get year portion of next gong.
      int nGongYear = (int) next_gong_time;
      // The following records the calendar year of the next ClockEvent.
      *pnEventInfo = nGongYear;
      return next_gong_time;
}

void Person::ClockEvent(int nEventInfo)
{
      // The following retrieves the previously stored nGongYear.
      int nGongYear = nEventInfo;
```

```
        // Now you would add code that uses nGongYear
        ...
}
```

Declaring the model parameters

At least one model parameter must exist in order for a simulation run of the model to occur. The model parameters must be declared in Modgen model code (in an .mpp file) and defined (with values) in a parameter data file (a .dat file) in order for the model executable to make use of them successfully. If a parameter is missing from the .dat input files, Modgen indicates an error. If there is an unknown parameter in a .dat input file, Modgen gives an error message when the parameters are read but the simulation nevertheless continues. If a simulation was running, the error message is also written to the log file.

Modgen has the capability of generating a model parameter from two regular parameters. This facility can speed up execution if the parameter that you generate is used repeatedly in the simulation. The sizes of all model parameters are defined by classifications, ranges or partitions.

Syntax for Model Parameter Definition in .mpp files

```
parameters
{
        type    parameter_name[size]...[size];
};
```

Syntax for Model Parameter Definition in .dat files

```
parameters
{
        type    parameter_name[size]...[size] = <( repeater)>{ value, value, .... , value }
};
```

Parameter descriptions are entered in the .mpp files and can be placed on a separate line before the parameter definition or else follow on the same line as the definition. (The descriptions can also be entered using Modgen's LABEL command.) Each label is preceded by a language code, e.g. EN or FR, followed by a space.

Syntax of the parameter label specification

//label1 <decimals=n> <//label2 …>

- Single-line comments cannot be more than 4096 characters in length.
- When provided, the parameter decimal setting is used for both the formatting of the displayed value and the

rounding of a stored value. The default value for parameter decimals is -1 which means there is no decimal limit and no formatting on display. (Any negative decimal setting has the same interpretation.)

A variety of topics related to using parameters can be found in " Parameters".

Model generated parameters and the PreSimulation() function

Model generated parameters are declared just like regular parameters. The only difference in the definition of the parameter is the keyword **model_generated** before the parameter type. Model generated parameters need no entries in the .dat (data*)* files. All such parameters are initialized to zero at the moment of their creation.

Syntax for Model Generated Parameter Definition in .mpp files

```
model_generated  type  parameter_name[size]...[size];
```

Each model can define a special function, called **PreSimulation()**, which takes no argument and returns no result. When you have model generated parameters, such a function is required to assign the initial values of the model generated parameters, based on input parameters. The function supports all C++ language types and is executed after the input parameters have been read but before the actual simulation starts.

Parameters of all types (except of type "file") can be used in the definition of model generated parameters. Like input parameters, their values, which are assigned in **PreSimulation(),** should be their original values. Modgen performs all required transformations (e.g. cumrate) after **PreSimulation()** has been called. The parameters are also validated at that point.

Example (from an .mpp file)

```
parameters
{
...
    int DiseasesModeled[DISEASES];
    double MoralitytHazards[DISEASES];
    model_generated cumrate NonModeledCauseOfDeath[DISEASES];
...
}
...
void PreSimulation()
{
    int nIndex;

    for ( nIndex = 0; nIndex < SIZE( DISEASES ); nIndex++ )
{
        NonModeledCauseOfDeath[nIndex] = DiseasesModeled[nIndex]
                                          ? 0 : MortalityHazards[nIndex];
    }
}
```

Declaring the model's tables

Modgen has two styles of tabulation built in to generate aggregate results. The first or regular style, which is also the predominantly used style, involves the declaration of a table in the model code. Each such table is then populated on-the-fly during a simulation run without the need for any interim temporary storage files. At the end of the simulation, each table is transferred to an output database.

The other style of table is called a user table. User tables are used if tables are not sufficient, but require the model developer to create code to populatet the user tables. Modgen will not populate the user tables automatically, but rather execute the model code to populate the user tables at the end of the simulation. Model developers can create one or many **UserTables()** functions to add code that populates user tables. One advantage of user tables is that standard errors and coefficients of variation can also be easily generated for them.

Irrespective of the style (regular table or user table), a table has classification dimensions which determine the nature of the rows and columns of the tables, and an analysis dimension which can contain one or more expressions that determine the contents of each cell. Tables can also have optional decimal settings or scaling factors, as well as dimension totals for numeric dimensions.

Tables are a powerful feature of Modgen and are discussed further in “The table facility”.

Multilingualism, labels and notes in Modgen

Declaring the languages of the model

All Modgen models must have at least one language explicitly declared. You can choose one or more languages by entering the following **languages** declaration statement into an .mpp file (preferably within the .mpp file that contains your **model_type** declaration statement):

Syntax of the languages definition:

```
languages
{
        language_code, // language_name
        …
        language_code // language_name
};
```

Note that each language code is followed by a comma if there are more languages to be specified. However, there is no comma after the final language code.

Example:

```
languages
{
    EN, // English
    FR // Français
};
```

Defining symbol labels and notes

Most of the different entities or elements that make up a Modgen model are called symbols. The complete list of possible symbols (in alphabetical order) is:

- actor
- actor set
- aggregation
- event
- function (actor function or supporting C++ function)
- group (for parameters or for tables)
- link (single or multiple)
- module
- parameter
- state (simple or derived)
- table
- type (classification, range or partition)
- values or levels of classifications

For all Modgen symbols, you can and should provide both a label and a descriptive note for each symbol used in your model. These labels and notes in turn enable Modgen to generate encyclopedic documentation for your model, as is discussed further in "Documentation of a Modgen model".

Symbol labels

Symbols that are declared in .mpp files can be labeled using the LABEL statement. This syntax is useful for multilingual models which may label the symbols in just one (working) language in their main set of .mpp files and provide labels for additional languages in separate .mpp files. The labels can also be added along with the symbol definition; this style of label entry is used for the English labels in Examples 1, 2 and 3 that follow.

Syntax of a symbol label statement:

```
//LABEL ( symbol_name, language_code ) text
```

The maximum size per label (i.e. the maximum size of "text" in the above syntax statement) is 255 characters. In addition, the symbol_name can be the keyword **model** in order to provide a brief label for the model itself. The language_code is the code used to define the label's language in the **languages** statement (e.g. EN or FR) and is mandatory.

In the following examples, the working language of the model is defined as English so that the first block of code in each example would appear in a file 'Module_name.mpp', whereas the corresponding translations would appear in the file 'Model_nameFR.mpp'.

Example 1:

```
//LABEL(model, EN) Label for model
```

Example 2:

```
classification SEX

{

      //EN Gender
      S_FEMALE,      //EN Female
      S_MALE         //EN Male
};

//LABEL ( SEX, FR ) Sexe
//LABEL ( S_FEMALE, FR ) Féminin
//LABEL ( S_MALE, FR ) Masculin
```

Example 3:

```
actor Person            //EN Individual
{
…
    int           earnings = {0};         //EN Lifetime earnings to date
    DFLE_STATE    dfle = {NO_DIS};        //EN Disability State
    LIFE_STATE    life_status = {ALIVE};  //EN Life Status
    …
};

//LABEL ( Person, FR ) Individu
//LABEL ( Person.earnings, FR ) Gains totaux à ce jour
//LABEL ( Person.dfle, FR ) Situation-handicapé
//LABEL ( Person.life_status, FR ) Situation-vie
```

Example 4:

This example shows the definition of labels for a table. Tables are used by Modgen to report results on the simulated population. Tables are discussed in much more detail in “The table facility”

```
table Person X08_AvgEarnByAge //EN Cases and their average earnings by age group
{
    split( age, AGE_GROUPS )+  //EN Age Group
    * {
        unit,                  //EN Persons
        earnings / duration()  //EN Avg earnings
    }
};

//LABEL ( X08_AvgEarnByAge, FR ) Cas et gains moyens par groupe d'âge
//LABEL ( X08_AvgEarnByAge.Dim0, FR ) Groupe d'âges
//LABEL ( X08_AvgEarnByAge.Expr0, FR ) Personnes
//LABEL ( X08_AvgEarnByAge.Expr1, FR ) Gains moyens
```

To label modules, classifications, ranges, partitions, actors or tables, use their names as the symbol name in the LABEL statement, as in Example 1.

To label actor members (e.g. states, events, actor functions), use their names prefixed by the actor name and a period (e.g. Person.earnings) in the LABEL statement, as in Example 2.

To label a classification dimension of a table, use a dimension name (Dim0, Dim1, etc.) prefixed by the Table name and a period (e.g. X08_AvgEarnByAge.Dim0) in the LABEL statement, as in Example 3.

To label a table expression, use the expression name (Expr0, Expr1, etc.) prefixed by the Table name and a period (e.g. X08_AvgEarnByAge.Expr0) in the LABEL statement, also as in Example 3.

Symbol notes

Symbol notes allow you to encapsulate the Modgen model documentation into the compiled model for use by model users. Any notes entered in .mpp and .dat files are pre-formatted into paragraphs and lists before being displayed in the model's visual interface.

Symbol notes can be attached to any Modgen symbol used in a model, including modules, as well as to the model itself. There are two types of symbol notes:

- notes in .mpp files which should describe the usage of a particular symbol;
- notes in .dat files which should describe particular parameter values and which are labeled as value notes in parameter property windows in the visual interface.

Syntax of a symbol note:

```
/* NOTE (symbol_name, language_code ) text
*/
```

The language_code is the code used to define the label's language in the **languages** statement (e.g. EN or FR) and is mandatory.

To provide a note for the model itself, symbol_name is replaced by the keyword **model**.

When writing notes for actor members (states, events etc.) the symbol_name consists of the actor name followed by a period and the object name (e.g. Person.mar_status). The symbol names for classification dimensions of tables, as well as for table expressions, are prefixed by the table name and a period, followed by the dimension name (Dim0, Dim1, etc.) or expression name (Expr0, Expr1, etc.)

Example

```
/* NOTE ( WagePoints, EN )

The wage rates are relative earning rates that will be scaled to have mean 1.0.

These rates are applied to Firms' overall costs and profits

(divided by the number of individuals) to calculate the Consumer's income.
```

```
*/
```

Other languages

Modgen was designed so that multilingual models could be implemented. Any Modgen label or note can be documented in any number of languages, provided the language had previously been declared in the model. The information is then compiled into the model executable and optionally sent to the Modgen documentation database. Tracking database files have also been designed to hold multiple language descriptors enabling dynamic language switching within the BioBrowser application.

In enabling other languages for your model, several different documentation levels should be considered. The first level is the interface to Modgen itself. English and French language support are both already fully built in to this interface; all of the necessary character strings of the interface are contained within the files 'ModgenEN.mpp' and 'ModgenFR.mpp', both of which are parts of the Modgen distribution and which can be found in the Modgen installation directory. However, support for other languages can also be built, using either of these two files as a starting point. For example, you could translate the contents of 'ModgenFR.mpp' from French to Spanish, store the results as file 'ModgenES.mpp', and include this file (along with an appropriate **languages** statement including 'ES Espanol' as one of the languages of the model) in your model project, in order to provide a Spanish language interface. The additional file ('ModgenES.mpp' in this example) would have to be in the same directory as the other .mpp files in your model. The only drawback of this approach is that the files 'ModgenEN.mpp' and 'ModgenFR.mpp' are always kept up-to-date by Modgen as new versions of Modgen are released, but it would be your responsibility to adjust any additional language module file accordingly after each major Modgen update.

The other major level of documentation to consider, in terms of its scope, is the encyclopedic documentation generated by Modgen. Some of the character strings from the Modgen interface files ('ModgenEN.mpp' or 'ModgenFR.mpp' or any others that you have created) are used in the encyclopedic documentation but the vast majority of the contents are based on the symbol labels and symbol notes that are defined in your .mpp files. It is possible to insert the labels and notes in all languages in your .mpp files whenever the corresponding symbols are defined but the more usual convention or practice is to only specify these notes and labels in the working language of the model and then have a separate mpp file, 'model_name<language_code>.mpp', with translations for all of the symbol notes and labels. You would have one such file for each additional language defined in your model.

While you are responsible for obtaining the actual translations, there is an add-on tool developed by Statistics Canada, called the Modgen Translation Assistant, which facilitates the translation of these symbol notes and labels, both initially and during transitions as your model evolves. The tool can identify each string that requires translation, and after an initial translation, the tool can be used to isolate what strings were missed in or changed since that initial translation. For more information on this tool, contact *microsimulation@statcan.gc.ca*.

Other documentation levels to consider for translation are the primary documentation file and the documentation of various entities associated with a model’s scenarios. A model does not have to have a primary documentation file, but if it does, you will have created its contents as a compiled help file called ‘<model_name>1_<language_code>.chm’. Essentially, your job is to repeat this process for each language of the model, storing all such files in the same directory as your other .mpp files. You should also ensure that each primary documentation file has a link to the encyclopedic documentation file generated with the same language, i.e. ‘<model_name>2_<language_code>.chm’.

Documentation associated with a model’s scenarios is all essentially input through the generic visual interface to Modgen models. When you start the interface, you have a choice of the language in which you will work; the list of languages is based on those specified via the **languages** command in your model. Documentation for the scenarios has to be entered separately for each language via the visual interface.

In addition, you can create your own multilingual output strings with Modgen’s **string** command. This command is first followed by the name of the specific string, and then the actual label in the desired language.

The following example establishes an English and French label for the string called S_GENERATING_SPOUSES. When this string (i.e. S_GENERATING_SPOUSES) is later used as an argument to the run-time function **ModelString(),** the corresponding text is displayed and added to the log file in the language being used during the simulation run.)

Example:

```
string S_GENERATING_SPOUSES; //EN Generating spouses

string S_GENERATING_SPOUSES; //FR Génération des époux

...

sprintf( szMsg, "%s: %ld / %ld", ModelString( "S_GENERATING_SPOUSES" ),

lSpouseInd, SpouseMarketSize );

ProgressMessage( szMsg );
```

The simulation engine

The concept of time in a Modgen model

This section explains how Modgen's simulation engine advances time during a simulation run. As a developer, you will rarely, if ever, need to adjust the simulation engine code; however, the simulation engine is a fundamental component of every Modgen model and so it is important to understand the concepts underlying how it operates.

Essentially, the time during a simulation run corresponds to the time of the actors in the simulation. The time of a simulation run is actually a characteristic of the actors. This implies that time cannot and does not exist in a model until the actors exist. A state called ***time*** (of type TIME, which in turn is defined via a **time_type** statement that must exist in every model) is automatically generated and maintained by Modgen for all actors. You can initialize the ***time*** state within a **Start()** function, but this is the only place where you can do so because its value is automatically maintained by Modgen thereafter.

Example:

```
range MODELED_PERIOD

{

	1990, 2061 //EN Modeled time period

};

void Person::Start()

{

	// time is a state automatically maintained by Modgen.

	// It can be set only in the Start() function

	time = MIN( MODELED_PERIOD );
};
```

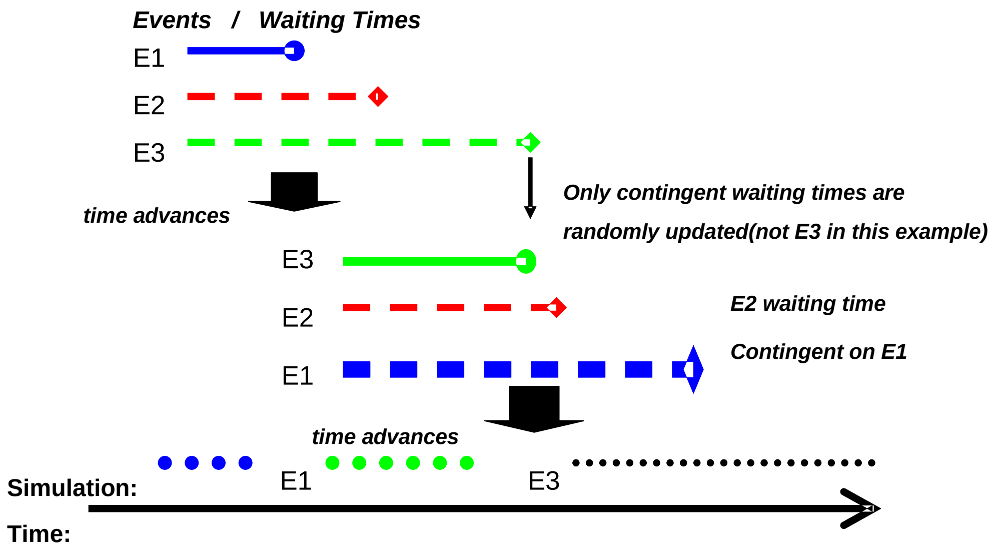
If ***time*** is not initialized within a **Start()** function, the time of a new actor is initialized to the current time of the simulation.

Once initialized, time advances in a simulation run as events occur and only as events occur. The implication is that time stops ONLY when events occur, which further implies that there is no fixed time increment between events. Events in Modgen are each defined with two functions—an event time function that calculates WHEN the event will occur next, and an event implementation function that determines WHAT will happen (e.g. what states will change) if and when the event occurs.

The event time functions cannot actually change state values, nor call any functions that can change state values; this is because nothing happens in the simulation run until a specific event takes place. When that event occurs, its corresponding event implementation function is the function that is executed and actually used to change state values.

The event times themselves are often generated from statistical distributions and based on hazards. In essence, the waiting time to each possible event is determined and the event that actually occurs next is the one with the shortest waiting time.

The following diagram illustrates the evolution of a simulated life course in a continuous time model. At the beginning, there are three events (E1, E2, E3), each of which has a randomly generated duration. In the example, E1 occurs first so it becomes the event that is executed; after that, durations for the three events are calculated. However, because E3 is not defined to be contingent on E1 in the example, its timing remains unchanged, whereas new times are recalculated for E1 and E2. After this happens, E3 ends up having the next smallest duration so it is executed next. The cycle then continues as times are again re-calculated for the events.



Modgen supports the development of two types of longitudinal microsimulation models—a case-based or non-interacting population model in which a specified number of cases are simulated sequentially and independently of one another, and a time-based or interacting population model that involves several interacting actors or agents over a specified period of time. While there are similarities in the underlying concepts of the simulation engine for both types of model, there are differences as well that correspond with the nuances of each model type; thus, each type is discussed separately in the remainder of this section.

Simulation engine for case-based models

A **Simulation()** function is mandatory in any case-based model. Most models have a separate module or .mpp file that contain basic definitions used throughout the model, such as the model version number, model type, and languages supported by the model. It is usually this same module that contains the **Simulation()** function code as well. In many case-based models, the **Simulation()** function makes a call to a **CaseSimulation()** function to process each case in the model, one case at a time.

Simulation() function

Standard example of **Simulation()** function for a case-based model

```
void Simulation()
{
      long lCase = 0;   // counter for cases simulated

      for ( lCase = 0; lCase < CASES() && !gbInterrupted && !gbCancelled &&
            !gbErrors; lCase++ )
      {
            // Tell the Modgen run-time to prepare to simulate a new case.
            StartCase();
            // Call CaseSimulation function, defined elsewhere in this module.
            CaseSimulation();
            // Tell the Modgen run-time that the case has been completed.
            SignalCase();
      }
}
```

In the example, “lCase” is a counter that loops through each simulated case. The *for* statement contains the actual loop through which each case is simulated. The number of cases to simulate for the current thread of executation is given by the Modgen run-time macro **CASES()**. The value of **CASES()**, summed over all threads of execution, will equal the number of cases specified in Modgen's visual interface before the simulation run begins. The **StartCase()** and **SignalCase()** functions are internal to Modgen and not actually found in the model code—**StartCase()** tells Modgen to prepare to simulate a new case, while **SignalCase()** tells Modgen that the case has been completed. The **CaseSimulation()** function controls the simulation of a case and is discussed in further detail below.

In the *for* statement, gbInterrupted indicates that the individual running the simulation has cancelled it but had also requested (via the visual interface) to receive partial reports in case of such a cancellation. (The current case will finish, however, before the partial reports are generated, since the current case is simulated via the **CaseSimulation()** function that was called within the loop.) Also in the *for* statement, gbCancelled indicates that the individual running the simulation has cancelled it and had not requested partial reports, while gbErrors indicates the presence of unexpected run-time errors.

CaseSimulation() function

In the following example of a **CaseSimulation()** function, the model's main actor is called "Person".
Standard example of CaseSimulation() function for a case-based model

```
void CaseSimulation()

{

        // Initialization section

        // Initialize the person actor

        Person *prPerson = NULL;

        prPerson = new Person();

        prPerson->Start();


        // Event queue section

        // Continue processing events until there are no more of them.

        while (!gpoEventQueue->Empty())

        {

                if (gbCancelled || gbErrors)

                {

                        gpoEventQueue->FinishAllActors();

                }
```

```
          else
          {
                // Age all actors to the time of the next event.
                gpoEventQueue->WaitUntil(gpoEventQueue->NextEvent());

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

CaseSimulation() – Initialization section

The first three lines of code (after the comments) initialize a pointer to the Person actor, and then initialize the main actor for the new case (i.e. Person, in this model). More specifically, *new* is a C++ operator that creates new objects, and the third line of code (i.e. prPerson->Start();) initializes all of the states, including ***time***, for the new Person actor.

The **Start()** function is mandatory and must be called once for each new actor. By default, it has no arguments, although you may choose to have as many arguments as you need for your model. If you do not explicitly provide a **Start()** function, Modgen includes an empty one with no arguments by default. The purpose of the **Start()** function is to initialize the actor's states when the starting values can vary from case to case and the initialization is too complex to be specified in a straightforward expression or statement. (If the starting values are always 0 or equivalent, Modgen does such initialization by default and so you do not have to explicitly initialize these states; however, a **Start()** function is still required.)

CaseSimulation() – Event queue section

The *while* statement is the event loop for the simulation run. In that statement, gpoEventQueue represents the event queue that was implemented by Modgen. The call to **WaitUntil()** advances the time to the next event and ages all actors to that time. The time of the next event to occur is returned by **NextEvent()**. The call to **Implement()** implements the next event by using the event implementation function associated with that next event. The call to **Implement()** also causes the time for future events to be recalculated as necessary and repositions those events in the queue. The events whose times are recalculated certainly include the event which just occurred but also include any event that depends on a state that was modified by the implementation function of the event which just occurred.

The cause of the end time in the simulation of a case will vary from model to model. It can happen, for example, when the main actor for a case either dies or reaches a certain age. If there are no future events for any actors, the event queue will be empty, which in turn will cause the simulation to end (as coded via the *while* statement in the example).

The condition that cause the simulation to finish can be modified. As an example, if a model simulates more than one actor, then the question of how the simulation ends must be resolved. In the **CaseSimulation()** example above, the simulation would end only if all the actors in the simulation are finished. In some case, we may want to specify that a simulation should stop if the first actor finishes, for instance. In such a situation, or in the situation where the simulation run has been cancelled or encountered unexpected errors (gbCancelled or gbErrors, respectively), the **FinishAllActors()** function is called—this function empties the event queue, thus effectively destroying all actors. It is necessary to have such a function because even if the main actor dies, there could be secondary actors associated with the case who are still alive, and thus they need to be terminated as well.

While the **FinishAllActors()** function removes any secondary actors who are still active in the case, the termination of the main actor is handled by the Modgen function **Finish()** which removes the actor from the simulation and table calculations and recuperates the memory that the actor was using. If no **Finish()** function is defined for an actor within the model code, Modgen automatically creates an empty version of it. Note that an actor should never be referenced by model code after **Finish()** has been called; otherwise, an error message is generated.

Simulation engine for time-based models

Each time-based model must have a **Simulation()** function, just as each case-based model requires such a function. However, there is no function equivalent to **CaseSimulation()** because in a time-based model, all of the actors can potentially interact with one another during the simulation run--so the concept of dealing with just one case at a time does not exist. Most models have a separate module or .mpp file that contain basic definitions used throughout the model, such as the model version number, model type, and languages supported by the model; it is usually this same module that contains the **Simulation()** function code as well.

Simulation() function

Standard example of Simulation() function code for a time-based model

```
void Simulation( )
{
        …
        //Initialization section
        for ( int nJ = 0; nJ < StartingPersonActors; nJ++ )
        {
                Person *prPerson = new Person();
```

```
            …

            prPerson->Start( … );

      }

      TimeReport(MIN(Modeled_Time));

      …


      //Event queue section

      while ( !gpoEventQueue->Empty() )

      {


            // get the time of next event, verify against the simulation end

            dCurrentTime = gpoEventQueue->NextEvent();


            if ( dCurrentTime > SIMULATION_END() || gbInterrupted || gbCancelled
|| gbErrors )

            {

                  // age all actors to the simulation end time

                  gpoEventQueue->WaitUntilAllActors( SIMULATION_END() );

                  gpoEventQueue->FinishAllActors()

            }

            else

            {

                  // age all actors to the time of the next event

                  gpoEventQueue->WaitUntil( dCurrentTime );


                  // implement the next event

                  gpoEventQueue->Implement();
```

```
                }…

        }…

}
```

Simulation() – Initialization section

Because all actors can potentially interact with one another in a time-based model, an entire starting population has to be created first, which is the role of the **for** loop in the sample **Simulation()** function. In that example, **StartingPersonActors** is a parameter that would be declared elsewhere in the model code and that would contain the number of actors in the simulation run. The line of code,

```
Person *prPerson = new Person();
```

creates one actor in that starting population, while the line of code,

```
prPerson->Start( … );
```

initializes all of the states, including time, for that actor. Note that the **Start()** function is mandatory for each actor. By default, it has no arguments, although you may choose to have as many arguments as you need for your model. If you do not explicitly provide a **Start()** function, Modgen will include an empty one with no arguments by default. The purpose of the **Start()** function is to initialize the actor's states when the starting values can vary from case to case and the initialization is too complex to be specified in a straightforward expression or statement. (If the starting values are always 0 or equivalent, Modgen does such initialization by default and so you do not have to explicitly initialize these states; however, a **Start()** function is still required.)

The **TimeReport()** function is required in a time-based model before the event queue loop begins. Its purpose is to set the initial time for the simulation run.

Simulation() – Event queue section

The **while** statement is the event loop for the simulation run; in that loop, the function **NextEvent()** provides the time of the next event that will occur, and assigns that time to the variable dCurrentTime. A check is then performed to determine if it is time for the simulation run to stop (as will be discussed shortly). Assuming that the simulation is still continuing, the call to **WaitUntil(** *dCurrentTime***)** advances to the time of the next event (i.e. *dCurrentTime*) and ages all actors to that time; the call to **Implement()** implements the next event by using the event implementation function associated with that next event. The statement also causes the time for future events to be recalculated as necessary and repositions those events in the queue. The events whose times are recalculated certainly include the event which just occurred but also include any event that depends on a state that was modified by the implementation function of the event which just occurred.

With respect to ending the simulation run, recall that a time-based model comes to an end once a pre-specified end time is reached. That end time is actually specified in Modgen's visual interface before the simulation begins, and its value is returned by the Modgen run-time macro **SIMULATION_END()**.   If the next event time (stored by the code example in the variable *dCurrentTime*) is later than the simulation end time, that next event will not occur and the simulation run will come to an end.   In such a situation, the **WaitUntilAllActors()** function advances the time of all actors to the simulation end time for tabulation and tracking purposes. The **FinishAllActors()** function empties the event queue, thus effectively destroying all actors; it is necessary to have such a function because some of the actors would otherwise not be finished at the simulation end time. Note also that if you use the just_in_time algorithm in your model (as described in the next section), the **FinishAllActors()** function performs both roles, i.e. you do not need to include **WaitUntilAllActors()** for models using the just_in_time algorithm.

The simulation run will also end if all actors are finished, i.e. there are no more events in the event queue. (This is the condition that is explicitly tested in the **while** statement.) This situation is less likely to occur in a time-based model than a case-based model because there is an entire population of actors involved in a time-based model simulation run, but it is nevertheless still possible.

Another way that the simulation can end is if the individual running the simulation cancels it or if unexpected run-time errors arise; these conditions are checked via gbInterrupted, gbCancelled and gbErrors, respectively. The difference between gbInterrupted and gbCancelled is that with gbInterrupted. The individual running the simulation had also requested before the simulation run began (via Modgen's visual interface) to receive partial reports in case of a cancellation, whereas with gbCancelled, no prior request was made to provide partial reports in case of a cancellation.

The just_in_time algorithm

The **just_in_time** keyword enables a different algorithm for advancing time that can result in dramatic improvements in computational efficiency for time-based models with many actors (although the algorithm can be used in a case-based model as well).

In all models, time advances when an event occurs. By default, the time for all actors is always updated when any event occurs. The optional **just-in-time** keyword, however, causes time to only be updated for the actor for whom the event occurs, as well as for any other actor affected by that event, i.e. whose states will change as a result of the event.   This capability is enabled by including the **just_in_time** keyword in the model type declaration, as follows:

Syntax of model_type declaraitons that include just_in_time:

```
model_type      case_based, just_in_time;
model_type      time_based, just_in_time;
```

The use of **just_in_time**, however, does impose rules on the use of continuously updated states in the model. For clarification, a continuously updated state is a state that falls into any one of the following three categories:

- The state itself is continuously updated, that is, the state does not change at fixed points in time

- The state is derived from a continuously updated state or another state which is derived from a continuously updated state.
- The state has the same name (member name) as another state for another actor which is in one of the previous two categories.

When **just_in_time** is used, the rules for continuously updated states are that:

- they cannot be used in any function that is not an actor function.
- a state cannot be derived from a continuously updated state from another actor.

Any violation of these rules is detected by Modgen during compilation and leads to an appropriate error message. The easiest way to be sure that your model respects these rules is to avoid using continuously updated states, except as a possible analysis dimension of a table or in an event implementation function.
Examples:

```
// the following is not allowed with just_in_time

double test = lSpouse->age ;

// the following IS allowed with just_in_time

double test = age;
```

Note that when **just_in_time** is used, the **WaitUntilAllActors()** function is not needed at the end of the simulation run to advance the time of all actors to the simulation end time; the **FinishAllActors()** function performs this role, along with its usual role of destroying all actors.

Derived state specifications

A derived state hasan associated specification which uses the names of other states and constants. Derived state specifications fall naturally into the following categories:

- Event counts
- Event occurrences
- Event conditions (both single occurrence and multiple occurrence of the events)
- Event triggers
- Durations
- Spells
- Transformations
- Links
- Self-scheduling states

The derived state specifications in the first four groups create derived states associated with changes to the states of actors. The derived state specifications in the durations and spells groups create derived states associated with durations of the states of actors. The derived state specifications in the transformation group create states by directly transforming other states, while those in the links group provide information about other actors who are linked. Finally, the derived state specifications in the self-scheduling group introduce the notion of a time interval for all other states; however, they differ from the other states in that they create additional events which cause the time of the simulation to actually stop at designated points in time.

Event counts

The derived state specifications in this category create derived states that act as counters, indicating the number of times a certain event has occurred. The corresponding specifications are:

- **entrances**
- **exits**
- **transitions**
- **changes**

In each of the following descriptions, the state *observed_state* must be one of the actor's states; however, there are no restrictions on *observed_state*'s type.

entrances(*observed_state, value*)
The overall number of times the value of *observed_state* has become *value* so far in the actor's simulated life. The result is returned as type counter.
Example:

```
entrances(marital_status,MARRIED)
```

provides the number of marriages an actor has had so far in the simulation run

exits(*observed_state, value*)
The overall number of times the value of *observed_state* has stopped being *value* so far in the actor's simulated life. The result is returned as type counter.
Example:

```
exits(marital_status,MARRIED)
```

provides the number of marriages that have been dissolved so far in the simulation run (irrespective of whether they were followed immediately by separation or by divorce.

transitions(*observed_state, value1, value2*)
The overall number of times the value of *observed_state* has changed from *value1* to *value2* so far in the actor's simulated life. The result is returned as type counter.
Example:

```
transitions(marital_status,COMMON_LAW,MARRIED)
```

provides the number of marriages the actor has had so far in the simulation run when the immediately previous marital status was COMMON_LAW. Thus, if the model allowed a possible transition from SINGLE to MARRIED, such marriages would not be counted in this example.

changes(*observed_state*)
The overall number of times the value of *observed_state* has changed so far in the actor's simulated life. The result is returned as type counter.

Example:

```
changes(job)
```

provides the number of job changes the actor has experienced so far in the simulation run (assuming that **job** is a state that reflects the actor's current job)

Event occurrences

The derived state specifications in this category create logical derived states that indicate whether or not a certain event has occurred. The corresponding specifications are:

undergone_entrance

undergone_exit

undergone_transition

undergone_change

In each of the following descriptions, the state *observed_state* must be one of the actor's states; however, there are no restrictions on *observed_state*'s type.

undergone_entrance(*observed_state, value*)
The result is FALSE until the value of *observed_state* becomes *value* in the actor's simulated life*;* the result is TRUE thereafter for the remainder of the actor's life. The result is returned as type logical.

Example:

```
undergone_entrance(marital_status,MARRIED)
```

indicates whether the actor has ever been married so far in the simulation run

undergone_exit(*observed_state, value*)
The result is FALSE until the value of *observed_state* stops being *value* in the actor's simulated life*;* the result is TRUE thereafter for the remainder of the actor's life. The result is returned as type logical.

<u>Example:</u>

```
undergone_exit(marital_status,MARRIED)
```

indicates whether the actor has ever stopped being married so far in the simulation run

undergone_transition(*observed_state*, *value,1 value2*)
The result is FALSE until the value of *observed_state* changes from *value1* to *value2* in the actor's simulated life*;* the result is TRUE thereafter for the remainder of the actor's life. The result is returned as type logical.
<u>Example:</u>

```
undergone_transition(marital_status,MARRIED,DIVORCED)
```

indicates whether the actor has ever become divorced directly after having been married; the result would still be FALSE if the actor transitioned instead from being married to being separated and then later divorced.

undergone_change(*observed_state*)
The result is FALSE until the value of *observed_state* changes for the first time in the actor's simulated life*;* the result is TRUE thereafter for the remainder of the actor's life. The result is returned as type logical.
<u>Example:</u>

```
undergone_change(marital_status)
```

indicates whether any change in marital status has taken place for the actor so far in the simulation run; thus, even a change from SINGLE to COMMON_LAW would yield a result of TRUE.

Event conditions—single occurrence

The derived state specifications in this category create derived states that report on the value of a given returned state when a certain event takes place. They only look at a single occurrence of an event, although this can be either the first time the event occurred, or the latest or most recent time that the event occurred in the simulation run. The corresponding functions are:

value_at_first_entrance

value_at_latest_entrance

value_at_first_exit

value_at_latest_exit

value_at_first_transition

value_at_latest_transition

value_at_first_change

value_at_latest_change

In each of the following descriptions, the states *observed_state* and *returned_state* must be amongst the actor's states; however, there are no restrictions on the types for either of these states.

value_at_first_entrance(*observed_state, value, returned_state*)
The result is the value of *returned_state* the first time that *observed_state* became *value* in the actor's simulated life. If *observed_state* has not yet ever become *value*, the result is zero (or equivalent). The result is returned with the same type as that of *returned_state*.

Example:

```
value_at_first_entrance(marital_status,MARRIED,integer_age)
```

returns the integer age of the actor at the actor's first marriage.

value_at_latest_entrance(*observed_state, value, returned_state*)
The result is the value of *returned_state* the latest or most recent time that *observed_state* became *value* in the actor's simulated life. The result is zero (or equivalent) up until the first time in the simulation that *observed_state* became *value*. The result is returned with the same type as that of *returned_state*.

Example:

```
value_at_latest_entrance (full_time_student,FALSE,school_type)
```

returns the type of school last attended on a full-time basis.

value_at_first_exit(*observed_state, value, returned_state*)
The result is the value of *returned_state* the first time that *observed_state* stopped being *value* in the actor's simulated life. The result is zero (or equivalent) until *observed_state* stopped being *value* for the first time. The result is returned with the same type as that of *returned_state*.

Example:

```
value_at_first_exit (marital_status,MARRIED,integer_age)
```

returns the integer age of the actor at the time of the actor's first marriage dissolution.

value_at_latest_exit(*observed_state, value, returned_state*)

The result is the value of *returned_state* the latest or most recent time that *observed_state* stopped becoming *value* in the actor's simulated life. The result is zero (or equivalent) up until the first time in the simulation that *observed_state* stopped becoming *value*. The result is returned with the same type as that of *returned_state*.

Example:

```
value_at_latest_exit(full_time_student,TRUE,school_type)
```

returns the type of school last attended on a full-time basis.

value_at_first_transition(*observed_state, value1,value2, returned_state*)

The result is the value of *returned_state* the first time that *observed_state* changed from *value1* to *value2* in the actor's simulated life. The result is zero (or equivalent) until *observed_state* changed from *value1* to *value2* for the first time. The result is returned with the same type as that of *returned_state*.

Example:

```
value_at_first_transition(marital_status,SINGLE,MARRIED,integer_age)
```

returns the integer age of the actor at the time of the actor's first marriage, assuming the actor went directly from being single to being married without an interim common-law period.

value_at_latest_transition(*observed_state, value1,value2, returned_state*)

The result is the value of *returned_state* the latest or most recent time that *observed_state* changed from *value1* to *value2* in the actor's life. The result is zero (or equivalent) up until the first time in the simulation that *observed_state* changed from *value1* to *value2*. The result is returned with the same type as that of *returned_state*.

Example:

```
value_at_latest_transition(marital_status,MARRIED,DIVORCED,integer_age)
```

returns the integer age of the actor the most recent time that the actor became immediately divorced after marriage, so far in the simulation.

value_at_first_change(*observed_state, returned_state*)

The result is the value of *returned_state* at the first time that *observed_state*'s value changed; until that first change, the result is 0 (or equivalent). The result is returned with the same type as that of *returned_state*.

Example:

```
value_at_first_change(ever_vaccinated,integer_age)
```

provides the integer age that the actor received the actor's first vaccination

value_at_latest_change(*observed_state, returned_state*)
The result is the value of *returned_state* the latest or most recent time that *observed_state* changed its value. The result is 0 (or equivalent) until the first change, and stays that way if *observed_state* never changes. The result is returned with the same type as that of *returned_state*.

Example:.

```
value_at_latest_change(year,integer_age)
```

returns the actor's age at the start of the year.

Event conditions—multiple occurrences

The derived state specifications in this category create derived states that sum the value of a given state over all periods of time that a certain event takes place. This capability is mostly used if you want to calculate averages in your model's output tables. The corresponding functions are:

value_at_entrances

value_at_exits

value_at_transitions

value_at_changes

In each of the following descriptions, the states *observed_state* and *summed_state* must be amongst the actor's states and *summed_state* must be of a type that is capable of being added or summed. (For example, *summed_state* could not be a state with a classification type.) There is no restriction on the type for *observed_state*.

value_at_entrances (*observed_state, value, summed_state*)
The sum of the values of the summed state over each time that the value of *observed_state* became *value* in the actor's simulated life. This function is used in conjunction with the derived state specification **entrances()** to calculate the average value of the summed state over all occasions when the observed state became *value* over the simulated lifetime. The result is returned with the same type as that of *summed_state*, i.e. either real or integer.

Examples:

```
value_at_entrances(disease_phase,DP_LATENT,age)
```

adds up each age at which the actor was infected with a disease (i.e. the actor's disease_phase state took on the value DP_LATENT

```
value_at_entrances(disease_phase,DP_LATENT,age)/entrances(disease_phase,
DP_LATENT)
```

provides the average age of infection by the disease when used within a table definition,

value_at_exits (*observed_state, value, summed_state*)
The sum of the values of the summed state over each time that the value of *observed_state* stopped being *value* in the actor's simulated life. This function is used in conjunction with the derived state specification **exits()** to calculate the average value of the summed state over all occasions when the observed state stopped being *value* over the simulated lifetime. The result is returned with the same type as that of *summed_state*, i.e. either real or integer.

Examples:

```
value_at_exits(disease_phase,DP_SICK,age)
```

adds up each age at which the actor stopped being sick (i.e. each time the actor's disease_phase state changed from DP_SICK to something else)

```
value_at_exits(disease_phase,DP_SICK,age)/exits(disease_phase,DP_SICK)
```

provides the average age at which people stopped being sick when used within a table definition.

value_at_transitions (*observed_state, value1, value2, summed_state*)
The sum of the values of the summed state over each time that the value of *observed_state* changed from *value1* to *value2* in the actor's simulated life. This function is used in conjunction with the derived state specification **transitions()** to calculate the average value of the summed state over all occasions when the observed state changed from *value1* to *value2* over the simulated lifetime. The result is returned with the same type as that of *summed_state*, i.e. either real or integer.

Examples:

```
value_at_transitions(disease_phase,DP_SICK,DP_REMISSION,age)
```

adds up each age at which the actor remitted from disease (i.e. each time the actor's disease_phase state changed from DP_SICK to DP_REMISSION).

```
value_at_transitions(disease_phase,DP_SICK,DP_REMISSION,age)/transitions(disease
_phase,DP_SICK,DP_REMISSION)
```

provides the average age at which people remitted from disease when used within a table definition.

value_at_changes (*observed_state, summed_state*)
The sum of the values of the summed state over each time that the value of *observed_state* has changed its value in the actor's simulated life. This function is used in conjunction with the derived state specification **changes()** to calculate the average value of the summed state over all occasions when the observed state has changed its value over the simulated lifetime. The result is returned with the same type as that of *summed_state*, i.e. either real or integer.

<u>Examples:</u>

```
value_at_changes(education_level,age)
```

adds up each age at which the education level changed for the actor.

```
value_at_changes(education_level,age)/changes(education_level)
```

provides the average age at which the education level changed when used within a table definition.

Event triggers

The derived state specifications in this category create logical derived states that indicate that a change in value has occurred in the current event. The result remains TRUE during the current event but becomes FALSE immediately after the event has completed. . Event triggers are useful in table filters for creating 'snapshot' style tables. The corresponding specifications are:

trigger_entrances

trigger_exits

trigger_transitions

trigger_changes

In each of the following descriptions, the state *observed_state* must be one of the actor's states; however, there are no restrictions on *observed_state*'s type.

trigger_entrances(*observed_state, value*)
The result is TRUE whenever the value of *observed_state* becomes *value* in the actor's simulated life*;* the result is FALSE immediately after the time changes again. The result is returned as type logical.

Example:

```
trigger_entrances(marital_status,MARRIED)
```

returns TRUE whenever the actor gets married and stays TRUE until the next time change thereafter

trigger_exits(*observed_state, value*)
The result is TRUE whenever the value of *observed_state* stops being *value* in the actor's simulated life*;* the result is FALSE immediately after the time changes again. The result is returned as type logical.

Example:

```
trigger_exits(marital_status,SINGLE)
```

returns TRUE whenever the actor stops being single (irrespective of whether the new status is married or common-law) and stays TRUE until the next time change thereafter

trigger_transitions(*observed_state, value,1 value2*)
The result is TRUE whenever the value of *observed_state* changes from *value1* to *value2* in the actor's simulated life*;* the result is FALSE immediately after the next time change. The result is returned as type logical.

Example:

```
trigger_transitions(marital_status,COMMON_LAW,MARRIED)
```

returns TRUE whenever the actor gets married after being in a common-law union immediately beforehand, and stays TRUE until the next time change thereafter

trigger_changes(*observed_state*)
The result is TRUE whenever the value of *observed_state* changes in the actor's simulated life*;* the result is FALSE immediately after the time changes again. The result is returned as type logical.

Example:

```
trigger_changes(education_level)
```

returns TRUE whenever the actor graduates (as it is only a graduation that can cause a change in education_level), and stays TRUE until the next time change thereafter

Durations

The following derived state specifications accumulate the values of a state over the range of time an actor was in a specified state:

duration

weighted_duration

weighted_cumulation

In each of the following descriptions, every state that is listed as a possible argument must be defined as an actor state; however, there are no restrictions on *observed_state*'s type.

duration (*observed_state, value*)

or

duration()

This function creates a state of type TIME which provides lengths of time. When *observed_state* and *value* are specified, it returns the overall length of time that *observed_state* has been *value* so far in the actor's simulated life, summed over all such time spells. If *observed_state* and *value* are not specified, it returns the total length of time the actor has existed so far in the simulation, which is thus equivalent to the result of **duration**(life_status, ALIVE).
Example:

```
duration(employed,TRUE)
```

returns the total length of time the actor has been employed so far in the simulation, including all jobs. Its value is 0 until the actor's first job begins.

weighted_duration (*observed_state, value, weighting_state*)

or

weighted_duration (*weighting_state*)

This function creates a state of type real that is the sum of *weighting state* over periods of time, where *weighting state* represents a rate (such as earnings rate or wage rates) and must be of a type that is capable of being summed. When *observed_state*, *value*, and *weighting_state* are all specified, this function returns the sum of *weighting_state* over the periods of time that the value of *observed_state* has been *value* so far in the actor's life, summed over all such time periods. Essentially, then, the rate is being accumulated over the specified time periods. If *observed_state* and *value* are not specified—that is, if only *weighting_state* is specified—then this function returns the sum of *weighting_state* over the entire life of the actor so far in the simulation.

Example:

```
weighted_duration(employed,TRUE,earnings_rate)
```

provides the actor’s total lifetime earnings from employment so far in the simulation

weighted_cumulation(*cumulated_state*, *weighting_state*)

This function creates a state of type real that contains the scaled sum of *weighting_ state* over periods of time, where *weighting state* represents a rate (such as earnings rate or wage rate) and must be of a type capable of being summed . Furthermore, *cumulated_state* must also be of a type capable of being summed. This function returns the sum of the product of *cumulated_state* and *weighting_state* over the entire life of the actor; it is essentially used when you need to adjust for inflation by creating a constant dollar version of a state cumulated in current dollars cumulated in current dollars.

Example:

```
weighted_cumulation(deflation_factor,earnings_rate)
```

provides the actor’s total constant dollar lifetime earnings from employment so far in the simulation, assuming that earnings_rate is 0 when employed is FALSE, and assuming that deflation_factor has been defined in the model as the base year’s Consumer Price Index divided by a given year’s Consumer Price Index.

Spells

The derived state specifications in this category operate on states over "spells" (i.e. periods of time over which a state assumes a specific value). These spell specifications fall into two sub-categories--those that operate on a "current” or “active" spell, and those that operate on a "completed” or “previous" spell. These two sub-categories of specifications behave very differently. In particular, the "active" specifications are saw-tooth in form, returning to zero at the end of each spell, whereas the "completed" specifications are step-like in form. The corresponding specifications are:

- **active_spell_duration**
- **completed_spell_duration**
- **active_spell_weighted_duration**
- **completed_spell_weighted_duration**
- **active_spell_delta**
- **completed_spell_delta**

In each of the following descriptions, every state that is listed as a possible argument must be defined as an actor state; however, there are no restrictions on *observed_state*'s type.

active_spell_duration (*observed_state, value*)

This function creates a state of type TIME which contains the length of time that has elapsed since the value of *observed_state* became *value*, for the current such spell only. The result is 0 when the value of *observed_state* is not *value*.

Example:

```
active_spell_duration(employed,TRUE)
```

provides the length of time an actor has been employed since the last period of unemployment.

completed_spell_duration*(observed_state, value*)

This function creates a state of type TIME which contains the length of time of the last completed spell or period (and only the last completed spell) when *observed_state* was *value.* The result is 0 until the value of *observed_state* both changed to *value* AND then later changed again to something else.

Example:

```
completed_spell_duration(employed,TRUE)
```

provides the length of time that the actor was last employed (assuming that the actor is not currently employed).

active_spell_weighted_duration (*observed_state*, *value*, *weighting state*)

This function creates a state of type real which contains the sum of the values of *weighting_state* since the value of *observed_state* became *value*, for the current such spell only. The result is 0 when the value of *observed_state* is not *value*. The state *weighting state* represents a rate (such as an earnings rate or wage rate) and must be of a type capable of being summed.

Example:

```
365*active_spell_weighted_duration(month_spell,TRUE,employed)
```

provides the number of days of employment in the current month (for a model in which the unit of time is a year and for which the state **month_spell** is reset at the start of each month)

completed_spell_weighted_duration (*observed_state*, *value*, *weighting state*)

This function creates a state of type real which contains the sum of the values of *weighting_state* over the last completed spell or period (and only the last completed spell) when *observed_state* was *value* . The result is 0 until the value of *observed_state* both changed to *value* AND then later changed again to something else. The state *weighting state* represents a rate (such as an earnings rate or wage rate) and must be of a type capable of being summed.
Example:

```
365*completed_spell_weighted_duration(month_spell,TRUE,employed)
```

provides the number of days of employment in the previous month (for a model in which the unit of time is a year and for which the state **month_spell** is reset at the start of each month)

active_spell_delta (*observed_state*, *value*, *differenced_state*)
This function creates a state of type real or integer (matching the type of *differenced_state*) which contains the change or difference in the value of *differenced_state* since the value of *observed_state* became *value*, for the current such spell only. The result is 0 when the value of *observed_state* is not *value*. The state *differenced state* must be of a type capable of being summed.
Example:

```
active_spell_delta(year_spell,TRUE,earnings)
```

provides the earnings so far in the current year (for a model in which the state **year_spell** is reset at the start of each year and earnings is accumulated lifetime earnings)

completed_spell_delta (*observed_state*, *value*, *differenced_state*)
This function creates a state of type real or integer (matching the type of *differenced_state*) which contains the change or difference in the value of *differenced_state* over the last completed spell or period (and only the last completed spell) when *observed_state* was *value* . The result is 0 until the value of *observed_state* both changed to *value* AND then later changed again to something else.. The state *differenced state* must be of a type capable of being summed.
Example:

```
completed_spell_delta(year_spell,TRUE,earnings)
```

provides the total earnings for the previous year (for a model in which the state **year_spell** is reset at the start of each year and earnings is accumulated lifetime earnings)

Transformations

The derived state specifications in this category create a new derived state by directly transforming another state. The corresponding specifications are:

- **split**
- **aggregate**

split (observed_state, partition_name)

This function creates a derived state that contains the partitioned values of another state. More specifically, it creates groups by partitioning *observed_state* according to the values defined in *partition_name*. The state *observed_state*, while not having any restrictions as to its type, must nevertheless be defined as one of the actor's states, whereas *partition_name* has to be a partition already defined in the model. When **split** is used within an actor definition to declare a derived state, the resulting type of the new derived state is integer (corresponding to the index of the value in the partition); however, when **split** is used in a table definition, the resulting type is instead a partition of type *partition_name*.

Note that **split** uses similar arguments and has a similar purpose (putting values into different subgroups with the subgroup boundaries based on a partition) as the Modgen derived state specification **self_scheduling_split** and the Modgen macro **SPLIT**. The split should never be used with continuously updated states. If a split is needed for a continuously updated state, such as age or time, then the **self_scheduling_split** should be used. The difference with the macro SPLIT is that SPLIT can only be used in simulation code (such as the **PreSimulation()** function or an event implementation function) to create the subgroups, whereas split is used and can only be used in an actor or table definition to create a derived state containing the subgroups.
Example:

```
partition AGE_GROUPS
{
     5, 10, 15, 20, 25
};
split(age,AGE_GROUPS);
```

If used in a table definition, this derived state would stratify the age of each actor into the appropriate five-year increment or subgroup (where the first subgroup was for age under 5, the second was for age greater than or equal to 5 but less than 10, etc, to the last subgroup of age being 25 or older). If used to create a derived state in an actor definition, the result would be 0 for an actor less than 5, 1 for an actor 5 and older but less than 10, etc., up finally to 5 for an actor 25 and older.

aggregate (*observed_state*, *classification_name*)

This function creates a classification state of type *classification_name* which is an aggregate of *observed_state*.

Aggregates are based on a process that involves going from one defined classification with many levels or values to a different defined classification state with fewer possible levels or values, all according to predefined rules. These predefined rules which establish the correspondence between the levels of the two different classifications are first identified via the Modgen keyword **aggregation.** Its syntax is:

```
aggregation aggregated_classification, detailed_classification
        {
    aggregated_level, detailed_level,
    aggregated_level, detailed_level,
    …
    aggregated_level, detailed_level
   };
```

In this definition, *aggregated_classification* and *detailed_classification* are previously defined classifications in the model, with constituent levels aggregated_level and detailed_level, respectively. (Note that there is no comma after the final detailed level immediately preceding the closing '}'.

For example, if a country was divided into provinces where each province belonged to a region and where each region could contain more than one province, then *detailed_classification* would refer to a PROVINCE classification and detailed_level would refer to each actual province name. Similarly, *aggregated_classification* would refer to a REGION classification and aggregated_level to the actual corresponding region name associated with each province. The call to **aggregation** would then map each province name to the name of its corresponding region, one at a time.

Links

The derived state specifications in this category create states that provide information about other linked actors. The functions that are available depend on the nature of the link, i.e. whether it is a single link (one-to-one or one-to-many) or a multiple link (many-to-one or many-to-many). For a single link, there is just one function, whereas for a multiple link, the following four functions are available:

- **count**
- **sum_over**
- **min_over**
- **max_over**

Actor links are discussed in more detail in the section "Interactions amongst actors - links and actor sets".

Single links

link_name->*observed_state*

This creates a state which represents exactly one state of the linked actor. The new state's value is the value of the linked actor's *observed_state*; if there is no linked actor, its value is instead 0 (or equivalent). The new state has the same type as *observed_state* although there are no restrictions on the actual type. The state *observed_state* does, however, have to be a state of the linked actor.
Example:

```
lSpouse->employed
```

returns TRUE if the actor's spouse is employed, and FALSE if the spouse is not employed or if the actor does not have a spouse

Multiple links

These derived state specifications create states to provide information either about the number of linked actors or about one of those linked actor's states. In each of the following descriptions, *link_name* refers either to the "many" side of a link.

count(*link_name)*

This returns, as type counter, the number of actors currently linked to the given actor by *link_name* , .
Example:

```
count(mlChildren)
```

returns the number of children that the actor has, at that point in time in the actor's life

sum_over*(link_name, summed_state)*

The sum of the values of *summed_state* for each actor currently linked to the given actor by *link_name*, returned as type real or integer. If *summed_state* is float, or real, the type is real, otherwise it is integer. The state *summed_state* must be a state of the linked actors and must be of a type capable of being summed.
Example:

```
sum_over(mlChildren,age0);
```

returns, as an integer, the number of children under 1 year of age that the actor has at that point in the actor's life, assuming that age0 is defined as a logical state that is TRUE when the corresponding actor is under one year of age.

min_over*(link_name, minned_state)*

The minimum value of *minned_state* amongst the actors currently linked to the given actor by *link_name*, returned with the same type as *minned_state*. The state *minned_state* must be a state of the linked actors and must be of a type capable of being summed. If no actors are currently linked, the value is 0.
Example:

```
min_over(mlChildren,integer_age);
```

returns, as an integer, the age of the youngest child that the actor has at that point in the actor's life

max_over*(link_name, maxed_state)*

The maximum value of *maxed_state* amongst the actors currently linked to the given actor by *link_name*, returned with the same type as *maxed_state*. The state *maxed_state* must be a state of the linked actors and must be of a type capable of being summed.
Example:

```
min_over(mlChildren,integer_age);
```

returns, as an integer, the age of the oldest child that the actor has at that point in the actor's life

Self-scheduling states

Self-scheduling states introduce the notion of a time interval into several of the other derived states available with Modgen. However, they are different from all other derived states because they actually cause the simulation to stop at explicitly defined points in time. They can thus be used to trigger or replace various events.

Self-scheduling states can be categorized into three broad groups:

- duration counters (which count the number of times a spell has lasted a given time interval)
- integer durations (which create states containing the precise integer part of a duration)
- split durations (which create states containing the precise partitioned value of a duration)

Duration counters

This group of self-scheduling derived state specifications create states that count the number of times a spell has lasted a given time interval. The derived state specifications in this category are:

- **duration_counter**
- **duration_trigger**

In each of the following descriptions, *observed_state* must be one of the actor's states.

duration_counter(*observed_state, value, time_interval*);

or

duration_counter (*observed_state, value, time_interval, max_time_interval*);

This function returns, as type counter, the number of time intervals of the specified size (i.e. the size specified as *time_interval)* that have elapsed, in the current spell only, since *observed_state* became *value.* The state *observed_state* must be one of the actor's states. The result is 0 whenever *observed_state* is not equal to *value*. When *max_time_interval* is also specified as an argument, the result is still the number of time intervals of the specified size, but only up until *max_time_interval* is reached, after which the result remains as *max_time_interval* (until *observed_state* is no longer equal to *value*)

Examples:

```
duration_counter(employed,TRUE,1)
```

returns the number of years of employment since the last period of unemployment

```
duration_counter(employed,TRUE,1,35)
```

returns the number of years of employment since the last period of unemployment, up to a maximum of 35

In both examples, implicit events are created at each anniversary year of employment, except that in the second example, no additional events are generated once the maximum of 35 years is reached.

duration_trigger(*observed_state*, *value*, *time_interval*)
This function creates a logical state indicating whether or not the duration of *observed_state* at the value *value* lasted as long as *time_interval*   The result is FALSE when *observed_state* is not equal to *value*, and it is FALSE up until the amount of time specified as *time_interval* has elapsed since *observed_state* became *value*; the result is thereafter TRUE, up until (and if) *observed_state* changes again.
Example:

```
logical worked6months = duration_trigger(employed,TRUE,0.5 )
```

returns TRUE if the actor has been working for at least 6 months (0.5 years) and is still working, and FALSE at all other times.

Integer durations

This group of self-scheduling derived state specifications creates states that return the precise or exact integer part of a duration. The durations can be various duration states as well as age and time. There is only one derived state specification in this group, self_scheduling_int, but it has six possible syntaxes, as follows:

- **self_scheduling_int**(**duration**(…))
- **self_scheduling_int**(**active_spell_duration**(*observed_state*,*value*))
- **self_scheduling_int**(**weighted_duration**(…))
- **self_scheduling_int**((**active_spell_weighted_duration**(*observed_state*,*value*,*weighting_state*))
- **self_scheduling_int**(*age*)
- **self_scheduling_int**(*time*)

self_scheduling_int(duration(*observed_state*,*value***));**

or

self_scheduling_int(duration());

This use of **self_scheduling_int** returns the exact integer part of the corresponding duration. When *observed_state* and *value* are specified, all time spells in the actor's life for which *observed_state* equals *value* are included. The result is 0 until *observed_state* becomes *value* for the first time.

Examples:

```
self_scheduling_int(duration())
```

returns the integer age of the actor (assuming the actor was created at age 0 in the model)

```
self_scheduling_int(duration(employed,TRUE))
```

returns the integer number of years that the actor has been employed, including all spells of employment

self_scheduling_int(active_spell_duration(*observed_state*,*value***));**

This use of **self_scheduling_int** returns the exact integer part of the length of time that has elapsed since *observed_state* became *value*, for the current such spell only. The result is 0 whenever *observed_state* is not equal to *value*.

Example:

```
self_scheduling_int(active_spell_duration(employed,TRUE))
```

returns the integer number of years of employment in the current spell of employment, and returns 0 when the actor is not employed

self_scheduling_int(weighted_duration(*observed_state,value, weighting_state***));**

or

self_scheduling_int(weighted_duration(*weighting_state***));**

This use of **self_scheduling_int** returns the exact integer part of the weighted duration. When *observed_state* and *value* are specified, all time spells in the actor's life for which *observed_state* equals *value* are included. The result is 0 until *observed_state* becomes *value* for the first time.

Example:

```
self_scheduling_int(weighted_duration(disability_factor))
```

returns the integer number of years lived in perfect health, given that disability_factor is a state with values between 0 and 1 for each year, where 1 represents perfect health

self_scheduling_int(active_spell_weighted_duration(*observed_state,value, weighting_state***));**

This use of **self_scheduling_int** returns the exact integer part of the weighted length of time that has elapsed since the value of *observed_state* became *value*, for the current such spell only. The result is 0 when the value of *observed_state* is not *value*. The state *weighting state* represents a rate (such as an earnings rate or wage rate) and must be of a type capable of being summed

self_scheduling_int(*age*)

This use of **self_scheduling_int** returns the exact integer age of the actor, irrespective of the initial age specified by the model for the actor.

self_scheduling_int(*time*)

This use of **self_scheduling_int** returns the exact integer time (e.g. the current year) at any point in time during a simulation run.

Split durations

This group of self-scheduling derived state specifications creates states that contain the precise or exact partitioned value of a duration. The durations can be various duration states as well as age and time. There is only one derived state specification in this group, self_scheduling_split, but it has six possible syntaxes, as follows:

- **self_scheduling_split(duration(…), partition_name)**
- **self_scheduling_split(active_spell_duration(observed_state,value), partition_name)**
- **self_scheduling_split(weighted_duration(…),partition_name)**
- **self_scheduling_split((active_spell_weighted_duration(observed_state,value,weighting_state), partition_name)**
- **self_scheduling_split(age, partition_name)**
- **self_scheduling_split(time, partition_name)**

The difference between **split** and **self_scheduling split** is that **self_scheduling_split** will cause the time of the simulation to stop exactly at the breakpoints defined in the partition.

self_scheduling_split(duration(*observed_state,value*)**,** *partition_name***);**

or

self_scheduling_split(duration()**,** *partition_name***);**

This use of **self_scheduling_split** partitions the duration according to the values defined in *partition_name*. More specifically, it creates groups by partitioning *observed_state* according to the values defined in *partition_name.* The state *observed_state*, while not having any restrictions as to its type, must nevertheless be defined as one of the actor's states, whereas *partition_name* has to be a partition already defined in the model. When **self_scheduling_split** is used within an actor definition to declare a derived state, the resulting type of the new derived state is integer (corresponding to the index of the value in the partition); however, when **self_scheduling_split** is used in a table definition, the resulting type is instead a partition of type *partition_name*. When *observed_state* and *value* are specified, all time spells in the actor's life for which *observed_state* equals *value* are included. The result is 0 until *observed_state* becomes *value* for the first time.

Example:

```
self_scheduling_split(duration(employed,TRUE),VACATION_ENTITLEMENT)
```

can be used to calculate the leave entitlement of an actor. In a table, it allows for the classification of actors according to their leave entitlement. In this example, if an employee's vacation entitlement changed first after 5 years of service, then after 12 years of service, and finally after 20 years of service, VACATION_ENTITLEMENT would be a partition defined with breakpoints 5, 12, and 20

self_scheduling_split(active_spell_duration(*observed_state,value*) **,** *partition_name***);**

This use of **self_scheduling_split** partitions the duration of the current spell, that is, the time since *observed_state* became *value*, according to the values defined in *partition_name*. The state *observed_state*, while not having any restrictions as to its type, must nevertheless be defined as one of the actor's states, whereas *partition_name* has to be a partition already defined in the model. When **self_scheduling_split** is used within an actor definition to declare a derived state, the resulting type of the new derived state is integer (corresponding to the index of the value in the partition); however, when **self_scheduling_split** is used in a table definition, the resulting type is instead a partition of type *partition_name*.

e.g. self_scheduling_split(active_spell_duration(disease_phase, DP_REMISSION), REMISSION_YEARS) splits the duration of the remission period into categories as designated by the partition REMISSION_YEARS.

self_scheduling_split(weighted_duration(*observed_state,value, weighting_state*) **,** *partition_name***);**

or

self_scheduling_split(weighted_duration(*weighting_state*) **,** *partition_name***);**

This use of **self_scheduling_split** partitions the weighted duration according to the values defined in *partition_name*. When *observed_state* and *value* are specified, all time spells in the actor's life for which *observed_state* equals *value* are included. The result is 0 until *observed_state* becomes *value* for the first time. The state *observed_state*, while not having any restrictions as to its type, must nevertheless be defined as one of the actor's states; *weighting_state* must be an actor state and represents a rate (such as an earnings rate or wage rate) capable of being summed; and *partition_name* has to be a partition already defined in the model. When **self_scheduling_split** is used within an actor definition to declare a derived state, the resulting type of the new derived state is integer (corresponding to the index of the value in the partition); however, when **self_scheduling_split** is used in a table definition, the resulting type is instead a partition of type *partition_name*.
Example :

```
self_scheduling_split(weighted_duration(disability_factor),YEARS_GROUPS)
```

returns the integer number of years lived in perfect health (adjusted by disability) within age groups whose breakpoints are specified in YEARS_GROUPS, given that disability_factor is a state with values between 0 and 1 for each year, where 1 represents perfect health

self_scheduling_split(active_spell_weighted_duration(*observed_state,value, weighting_state*) **,** *partition_name***);**

This use of **self_scheduling_split** partitions the weighted duration or length of time that has elapsed since the value of *observed_state* became *value*, for the current such spell only, according to the values defined in *partition_name*. The result is 0 when the value of *observed_state* is not *value*. The state *weighting state* represents a rate (such as an earnings rate or wage rate) that is capable of being summed. Both of *observed_state* and *weighting_state* must be actor states, and *partition_name* has to be a partition already defined in the model. When **self_scheduling_split** is used within an actor definition to declare a derived state, the resulting type of the new derived state is integer (corresponding to the index of the value in the partition); however, when **self_scheduling_split** is used in a table definition, the resulting type is instead a partition of type *partition_name*.

self_scheduling_split(*age***,** *partition_name*)

This use of **self_scheduling_split** partitions the age of the actor according to the values defined in *partition_name,* where p*artition_name* is a partition already defined in the model. When **self_scheduling_split** is used within an actor definition to declare a derived state, the resulting type of the new derived state is integer (corresponding to the index of the value in the partition); however, when **self_scheduling_split** is used in a table definition, the resulting type is instead a partition of type *partition_name*

Example :

```
self_scheduling_split(age,AGE_GROUPS)
```

creates age groups

self_scheduling_split(*time***,** *partition_name*)

This use of **self_scheduling_split** partitions the time according to the values defined in *partition_name,* where *partition_name* is a partition already defined in the model. When **self_scheduling_split** is used within an actor definition to declare a derived state, the resulting type of the new derived state is integer (corresponding to the index of the value in the partition); however, when **self_scheduling_split** is used in a table definition, the resulting type is instead a partition of type *partition_name*

Example :

```
self_scheduling_split(time,YEAR_GROUPS)
```

creates groups of years.

Parameters

Defining the model parameters

Parameters are used to input data to a model and generally to give more control to the individual performing the simulation runs. The parameters must be defined in both the Modgen model code (in an .mpp file) and a parameter data file (in a .dat file) in order for the model executable to use them successfully. Every model requires at least one .dat file in order for a simulation run to take place.

<u>Syntax for Model Parameter Definition in .mpp files</u>

```
parameters  <group_name>
{
        type        parameter_name[size]...[size];
};
```

<u>Syntax for Model Parameter Definition in .dat files</u>

```
parameters  <group_name>
{
        type        parameter_name[size]...[size] = <( repeater)>{ value, value, .... , value }
};
```

The size of each model parameter is usually determined by a classification or a range. If a parameter is missing from the .dat input files, Modgen indicates an error. If there is an unknown parameter in a .dat input file, Modgen gives an error message (which is also written to the log file) when the parameters are read but the simulation still continues to run.

Modgen's visual interface names the .dat files with the scenario name as follows: "scenario_name(file_name).dat". The initialization of parameter values in the .dat files is discussed later in this chapter.

Parameter descriptions are entered in the .mpp files and can either be placed on a separate line before the parameter definition or else follow on the same line as the parameter definition. (The descriptions can also be entered using Modgen's LABEL command as described in "Documentation of a Modgen model".)

<u>Syntax of a LABEL command</u>

//language_code1 label1 <decimals=n> <//language_code2 label2 …>

- Single-line comments cannot be more than 4096 characters in length..
- Parameter decimals are used for both the formatting of a displayed value and the rounding of a stored value. The default value for parameter decimals is -1 which means there is no decimal limit and no formatting on display.(In general, any negative decimal setting can be interpreted in the same way)

Model generated parameters and the PreSimulation() function

Modgen has the capability of generating a model parameter from two regularly defined parameters. This facility can speed up execution if the parameter that you generate is used repeatedly in the simulation

Model generated parameters are declared just like regular parameters. The only difference in the definition of the parameter is that the keyword "model_generated" is added before the parameter type in the .mpp file. Model generated parameters do not need any entries in the .dat files because their values are always calculated during the PreSimulation phase of a model simulation run.

<u>Syntax for Model Generated Parameter Definition in .mpp files</u>

    model_generated type name[size]...[size];

Wben model generated parameters are used, your model requires a function, **PreSimulation()** to assign the initial values of those model generated parameters, based on input parameters. When **PreSimulation()** is used, it is executed after the input parameters have been read but before the simulation itself starts.

Parameters of all types (except type "file") can be used in the definition of model generated parameters. Like input parameters, their values assigned in **PreSimulation()** should be their original values. Modgen will perform all required transformations (e.g. cumrate) after **PreSimulation()** has been called. The parameters will also be validated at that point. A model can include many PreSimulation functions if necessary.
Examples

```
parameters
{
...
   int DiseasesModeled[DISEASES];
```

```
    double MortalityHazards[DISEASES];
    model_generated cumrate NonModeledCauseOfDeath[DISEASES];
...
}
...
void PreSimulation()
{
      int nIndex;
      for ( nIndex = 0; nIndex < SIZE( DISEASES ); nIndex++ )
      {
            NonModeledCauseOfDeath[nIndex] = DiseasesModeled[nIndex]
            ? 0 : MortHazards[nIndex];
      }
}
```

Parameter groups

Parameter groups allow you to group parameters for visual selection in the Group View tab of Modgen’s visual interface. Parameter groups can only be declared in .mpp files. A specific parameter can belong to any number of groups. A parameter group may consist of parameters and/or other parameter groups. However, a group cannot be a member of itself, directly or indirectly; if this happens, Modgen will generate a "circular dependency" error.

Syntax of the parameter group definition:

```
      parameter_group groupname
      {
            parameter_name|groupname, ..., parameter_name|groupname
      };
```

The name of each member parameter or group is separated with a comma, but there is no comma after the final parameter or group. The parameters are displayed by the visual interface in the same order in which they are entered in the **parameter_group** command.

Example

```
parameter_group GDP_DEFLATOR
{ //EN GDP deflator parameters
      BasketUpdateTime, DeflatorComputeTime, DeflatorCommodThresh,
      DeflatorStoreThresh, PaascheFactor
};
```

Model generated parameters & parameter groups

The keyword **model_generated_parameter_group** operates just like **parameter_group**, but applies only to model generated parameters. Model generated parameters are not allowed as members of a parameter group; you must use a model generated parameter group for such parameters, instead. Furthermore, the only parameters allowed as members of a model generated parameter group are model generated parameters.

Initializing model parameter values in .dat files

Values for the model parameters are kept in .dat files. The visual interface names the .dat files according to the scenario name as follows: 'scenario_name(file_name).dat'. The rank and shape of the parameter and the meaning of each of its dimensions must be defined in both the .mpp file and the .dat file. The values of the parameters are expressed in the following ways:

Syntax for model parameter initialization in the .dat files

```
parameters
{
        type    parameter_name[size_dim1]...[size_dimn] = { value, value, .... , value }
                                        OR
        type    parameter_name[size_dim1]...[size_dimn] = ( repeater) { value, value, .... , value }
};
```

Repeaters can be nested within blocks of data values. Regardless of the number of parameter dimensions, the values for parameters are always entered as vectors in row-major order (i.e with the lowest dimension values, the columns, changing fastest and the higher-order dimensions changing the slowest). Each value is entered by cycling through the rightmost dimension values first. When these values have been used up, the next rightmost values are entered. This continues until all the values have been entered.

Example:

Suppose your parameter has two dimensions (rank two), has shape two by three, and contains the following values:

```
3    4    5

7    8    9
```

The statement in the .dat file to populate this parameter would then look like:

```
param[dim1_2values][dim2_3values] = { 3, 4, 5, 7, 8, 9};
```

A final trailing comma is allowed in the parameter initialization, so the following statement would be equally valid:

```
param[dim1_2values][dim2_3values] = { 3, 4, 5, 7, 8, 9, };
```

Parameter initializers may contain repeated blocks of values. The blocks can be nested.
Example:

```
param[dim] = { 0, (2) { 1, 2, (3) { 3, 4 }, 5 }, }
```

is equivalent to:

```
param[dim] = { 0, 1, 2, 3, 4, 3, 4, 3, 4, 5, 1, 2, 3, 4, 3, 4, 3, 4, 5 };
```

Modgen also recognizes numbers stored in scientific notation (with mantissa and exponent) in the .dat files. For example, 1E-10, 2.5e100 or 0.3E+101 are all valid.

The simplest method to edit parameter values within an existing model is to use Modgen's visual interface. However, if the parameter consists of one number (scalar) or a short vector of numbers, you can open the .dat file with your preferred text editor and change it directly. However, you should never modify the parameter definition portion of a .dat file when changing a parameter value.
Example

In this example, the value of the scalar parameter, FemaleProp, is changed from 0.5 to 1.0 to ensure that only females are simulated. Thus, the following original code excerpt within a .dat file:

```
//EN parameter specification
parameters

{
      double    FemaleProp = 0.5;
};
```

would be changed to

```
//EN parameter specification
parameters
{
      double      FemaleProp = 1.0;
};
```

It is not recommended that you use a text editor directly on a .dat file to change a range of model parameter values for a higher-dimensional array; with the dimensionality removed, it is difficult to ensure exactly which value of the parameter you are changing.

Parameter data types

Parameters can have any of the following data types, some of which are general C++ types, and some of which are specific to Modgen.

range

Modgen supports the definition of range parameters. The values of range parameters in .dat files are restricted to integers between the minimum and maximum limits of the range (inclusive).

Example in the .dat file

```
YEAR YearToSimulate = {2010};
```

classification

Modgen supports the definition of classification parameters. The values of classification parameters in .dat files are specified using the names for the levels in the classification (see example). Parameters of type classification display in the user interface as a drop-down list.

Example in the .dat file

```
PROVINCE ProvinceToSimulate = {SASK};
```

numeric

Numeric model parameters are defined as any C++ numeric type, e.g. int, long, float or double.

Example definition in the .mpp file

```
//EN annual earnings scale by disability status
```

```
double      annual_earnings_scale[DFLE_STATE];
```

Example definition in the .dat file (where DFLE_STATE is a classification state with four different levels)

```
//EN annual earnings scale by disability status

double annual_earnings_scale[DFLE_STATE] =

{ 10000.0, 7500.0, 2500.0, 0.0 };
```

logical

Modgen supports the definition of logical parameters. The values of logical parameters in .dat files can only be TRUE or FALSE. (However, if a parameter is a multi-dimensional array, none of its dimensions is permitted to have a logical type.)

Example in the .dat file

```
logical Alive = {TRUE};
```

cumrate [i]

Parameters of type cumate[i] give you the means to draw or sample from discrete probability distributions. They can have one or more dimensions (say that there are N dimensions overall), and the choice of outcome, determined from a cell position in the associated N-dimensional parameter array, can either be made by allowing all dimensions to be chosen randomly, or by allowing just the last i dimensions (where i is less than N) to be chosen randomly (in which case the values of the first N-i dimensions have to be pre-specified). When the last i dimensions are to be chosen randomly, the corresponding parameter type is specified as cumrate [i].   (The type 'cumrate' by itself is equivalent to cumrate [1].)

The underlying probabilities will sum to one over the last i dimensions of a parameter of type cumrate[i]. Such probabilities are said to be conditional on all previous dimensions in the array. For example, if there was a set of health status (sample state values "good", "fair", "poor", etc.) transition probabilities that differed by age and sex, a corresponding cumrate parameter could have three dimensions: age, sex, and health status. If it was a cumrate or cumrate [1] parameter, then each set of health care status probabilities would be conditional on a given age and sex.

One advantage of cumrate parameters is that the data entered for the parameters does not have to contain the actual probabilities--it can instead consist of frequency counts for each cell, as long as those frequency counts mirror the underlying probabilities. However, the input data in a cell always represents the individual value for that cell and NOT the cumulative values for all cells up to and including that cell. (Modgen performs all necessary cumulations via Lookup_ functions that are described in the next section.)

Examples in .mpp files:

Example 1:

```
// The model parameter, CumImmig, is a three-dimensional array whose
// size is determined by the classification SEX, the range YEAR,

// and the range IMM_AGE.

// The values of CumImmig express the probabilities of a person’s age

// at immigration conditional on their sex and simulation YEAR.

cumrate         CumImmig[SEX][YEAR][IMM_AGE];
```

Example 2:

```
// The model parameter, CumBirth, is a three-dimensional array whose

// size is determined by the classification SEX, the range YEAR_OF_BIRTH,

// and the classification PROVINCE.

// The values of CumBirth express the probability of being born in a given

// year and province conditional on the sex of the person

cumrate [2]      CumBirth[SEX][YEAR][PROVINCE];
```

piece_linear

A model parameter of type piece_linear stores a set of N points (their X and Y values) in a two-dimensional space: x[1], y[1], x[2], y[2], ..., x[N], y[N]. The X values must be given in increasing order (i.e. x[1] < x[2] < ... < x[N]). This set of points defines a piece-linear function y(x) in the following way:

- for x <= x[1] : y(x) = y[1]
- for x[K] < x <= x[K+1]: y(x) = y[K] + (y[K+1] - y[K]) / (x[K+1] - x[K]) * (x - x[K])
- for x > x[N]: y(x) = y[N]

A parameter of this type must be one-dimensional, defined by a range starting at 0, and it must contain an even number of values.

Example in the .dat file

```
range WAGE_POINT_VALUES    { 0, 7 };

...

piece_linear   WagePoints[WAGE_POINT_VALUES] =

      { 0, 40, 0.3, 85, 0.7, 115, 1, 205 };
```

file

A parameter of type file is used when your parameter data is in a file that you want to vary from scenario to scenario. For example, you could choose from different starting microdata files in a simulation via a parameter of type file.
Example in the .mpp file:

```
parameters

{

        file MyFile;        //EN File for testing

};
```

The value for a file parameter is provided in the parameter input file (.dat file) as the actual filename, enclosed in double quotation marks.
Example in the .dat file:

```
parameters

{

        file  MyFile = "C:\StatCan\tmp\thing.txt";

};
```

A Modgen model sets the current working directory to the directory of the scenario; thus, a file name specified using a relative path is interpreted relative to the scenario directory. For example, “thing.txt” (with no leading path) refers to the file “thing.txt” located in the scenario directory, while “..\thing.txt” refers to the file “thing.txt” located in the parent directory of the scenario directory.

Modgen does not permit arrays of parameters of type file, nor does it allow model-generated parameters of type file.

Accessing model parameters in C++ functions

There are several ways of accessing model parameters in Modgen; each way is specific to the parameter’s type.

numeric, logical, range, classification type parameters

The value of parameters of these types is obtained in C++ code by using the name of the parameter. For multi-dimensional parameters, the name is followed by indices to a specific cell, as in the example below. Parameters of type logical represent true as 1 and false as 0. Parameters of type range represent values as integers, with the first value being the lower limit of the range. Parameters of type classification represent values as integers, with the first value being 0.

.
Examples:

```
// The state, tfem, is assigned the value of the scalar parameter FemaleProp.

tfem = FemaleProp;

// The state, eSex, is assigned the value of zero if the random number,

// RandUniform(), is less than or equal to the value of FemaleProp.

// The state, eSex, is assigned the value of one, otherwise

eSex = ( RandUniform() <= FemaleProp );

// The state, z, is assigned a value from the model parameter matrix,

// ProbMort. The value corresponds to the location in the array

// defined by the current values of sex and of age-1

z = ProbMort[sex][age - 1];
```

cumrate [i] type parameters

Access to cumrate model parameters is gained through the use of the special function **Lookup_**<parameter_name>(); such a function is automatically generated by Modgen for every parameter of type cumrate. The Lookup functions draw values from discrete probability distributions specified through the parameter identified by parameter_name. The syntax of a Lookup function is as follows:

bool **Lookup_**<parameter_name>(RandUniform(), int *index1*, ,,,, int *indexK*, int **dest1*, ..., int **destN*)

The first argument is a call to obtain a random number, using the Modgen runtime function **RandUniform()**. The index variables in the argument contain values of indexes in the non-cumulated dimensions (they must be of type int or classification) whereas the dest variables are destination variables that receive the values of the cumulated dimensions (they must be of type int).

In the following example, a cumrate [2] parameter, CumBirth, is declared first, followed by the definition of its corresponding Lookup function.
Example:

```
// The model parameter, CumBirth, is a three-dimensional array whose

// size is determined by the classification SEX, the range YEAR_OF_BIRTH,

// and the classification PROVINCE.

// The values of CumBirth express the probability of being born in a given

// year and province conditional on the sex of the person
```

```
cumrate [2]      CumBirth[SEX][YEAR][PROVINCE];


// In the Lookup function, the states YEAR and PROVINCE are defined,
// conditional on the comparison of the random number, RandUniform(),
// with values in the CumBirth parameter. The lookup is conditional
// on the sex of the person.
Lookup_CumBirth( RandUniform(), eSex, &nYear, &nProv );
```

(In this example, note that the value of nYear that is returned is a dimension value of YEAR and not the actual year itself. Thus, if you did want to obtain the actual birth year, assuming that YEAR is a range state, you would have to use the **MIN()** macro and add MIN(YEAR) to the value of nYear returned by the **Lookup_CumBirth()** function, to obtain that year of birth.) That is, the expression (to return an integer birth year result) would be:

```
birthyear = MIN(YEAR) + nYear
```

It is not unusual for cumrate parameters to contain empty distributions (all zeroes), either for structural or sampling reasons. In order to allow you to detect and handle empty distributions in cumrate parameters, the **Lookup__**<parameter_name>**()** function returns a value of TRUE if the underlying distribution is non-empty, and a value of FALSE otherwise. This enables you to take appropriate fall-back action, such as terminating the simulation or writing a warning message using **WriteLogEntry(),** if an attempt is made to draw from an empty distribution during the simulation. Note as well that **Lookup__**<parameter_name>**()** returns the first index of the distribution if the distribution is empty; this result is returned through the dest arguments specified in the call to the Lookup function..

While the **Lookup_**<parameter_name>**()** function returns a TRUE or FALSE to indicate the existence of empty distributions, Modgen provides an additional run time function, **EmptyDistributions_**<parameter_name>**()**, for each parameter of type cumrate. This function returns, as an integer, the number of empty distributions contained within the cumrate parameter, i.e. within parameter_name. (However, the function does not work if it is called in **PreSimulation()** for a model-generated cumrate parameter; if called in such a situation, the function returns a result of 0.)

piece_linear type parameters

Access to piece_linear model parameters takes place through the use of the special function **Lookup_**<parameter_name>**()**, which is automatically generated by Modgen for every parameter of type piece_linear.

Example

In this example, the four X values (i.e. 0, 0.3, 0.7, and 1.0) contain the cumulative probabilities for the corresponding wage interval and the Y values (i.e. 40, 85, 115, 205) contain the wage breakpoints. Inside each interval, wages are uniformly distributed. For example, the probability of the wage being in the [40, 85] interval is 0.3; for the [85, 115] interval, it's 0.4 (0.7 - 0.3) and for the [115, 205] interval, it's 0.3 (1.0 – 0.7). The tertile wage breakpoints (dLowerWage and dUpperWage) for the WagePoints parameter are obtained via the two calls to the **Lookup_WagePoints()** function.

```
range WAGE_POINT_VALUES    { 0, 7 };
...
parameters
{
...
      piece_linear   WagePoints[WAGE_POINT_VALUES] =
      { 0, 40, 0.3, 85, 0.7, 115, 1, 205 };
...
};
dLowerWage = Lookup_WagePoints( .33 );
dUpperWage = Lookup_WagePoints( .66 );
```

Generally speaking, the X values are always stated in increasing order (i.e. x[1] < x[2] < ... < x[N]), and the Y values returned by **Lookup_**<parameter_name>**()** have the following behaviors:

- for x <= x[1] : y(x) = y[1]
- for x[K] < x <= x[K+1]: y(x) = y[K] + (y[K+1] - y[K]) / (x[K+1] - x[K]) * (x - x[K])
- for x > x[N]: y(x) = y[N]

Modgen also supplies the following run-time function, available in two forms, to assist with linear interpolation:

double **PieceLinearLookup**(double *dNumber*, double **pdXY*, int *nSize*);

double **PieceLinearLookup**(double *dNumber*, double **pdX*, double **pdY*, int *nSize*);

In both forms, *dNumber* is the X value for which the corresponding Y value is returned as a result. In the first form, *pdXY* points to a vector of value pairs (x1, y1, x2, y2, ...) and *nSize* contains the length of that vector (i.e. 2 * number of pairs). In the second form, *pdX* points to the vector of X values (x1, x2, ...), *pdY* points to the vector of Y values (y1, y2, ...) and *nSize* contains the length of both vectors (i.e. number of pairs). The X and Y points can come from different parameters. If a source parameter has more than one dimension, the X and Y points are taken from the last dimension. Note that the preferred and more flexible method for linear interpolation is to use the second form of the **PieceLinearLookup()** function. With this method, you do not define a parameter as type piece_linear; you simply use parameters containing the X and Y points in a function call to **PieceLinearLookup()**

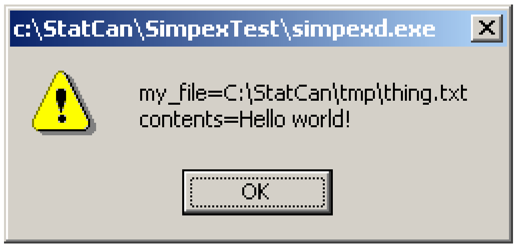
file type parameters

Modgen represents a parameter of type file as a C++ string (more specifically as a CString). A parameter of type file can be used directly as an argument to a function that takes a string argument, such as the C++ function **fopen()** in the following example. The value of the string ("Hello world!") can be accessed directly via the parameter name (***my_file***), also as shown in the example.

Example:

```
void PreSimulation()
{
      CString szWork;
      CString szFileContents;

      FILE *fp = fopen( my_file, "r" );

      if ( fp != NULL )
      {
            char szBuffer[100];
            fgets(szBuffer, 100, fp);
            szWork.Format(" my_file=%s\n contents=%s", my_file, szBuffer);
      }
      else
      {
            szWork.Format(" my_file=%s\n Unable to open file", my_file);
       }
       AfxMessageBox(szWork);
}
```

When the simulation is run, this example would produce a dialog box with contents similar to the following:



```
my_file=C:\StatCan\tmp\thing.txt
contents=Hello world!
```

Parameter validation routines and parameter normalization

Modgen provides support for parameter validation. You can write a parameter validation function called **ValidateParameters()** and Modgen will call it in the appropriate situations (corresponding to the specific activities that you want to validate). Each .mpp file can have a separate validation function for the parameters contained therein.

The prototype of **ValidateParameters()** is:

```
bool ValidateParameters( SCENARIO_EVENT eEvent );
```

where *eEvent* is any of the constants defined in the following table:

Value of eEvent	**Description**	**Visual interface menu event**
SCENARIO_SAVE_EVENT	called when the scenario is saved	**Scenario** > **Save**
CELL_MOVE_EVENT	called when the user of the visual interface moves or navigates from one cell of the parameter to another	
BLOCK_CHANGE_EVENT	called when one or more parameter cells are modified	
SCENARIO_RUN_EVENT	called just before the simulation is started from the visual interface (the event is not handled for simulations started in batch mode).	**Scenario** > **Run/resume**
SCENARIO_MENU_EVENT	called when **Scenario** > **Validate** is selected (Note that **Validate** is only a valid option under **Scenario** when there is at least one **ValidateParameters()** function in your model; otherwise, this option does not exist under **Scenario**.)	**Scenario** > **Validate** (as long as the model contains at least one **ValidateParameters()** function)

ValidateParameters() returns TRUE or FALSE. A result of TRUE indicates that the validation was performed successfully (possibly with some adjustments or with user approval to go ahead in case of problems) and Modgen proceeds with the event that called the function (**Scenario** > **Save**, Cell Movement, Data Change, **Scenario** > **Run/resume**, or **Scenario** > **Validate**, respectively). A result of FALSE causes Modgen to cancel the event. For example, a SCENARIO_SAVE_EVENT with a FALSE result would mean that the scenario would not be saved, thus returning the user of the visual interface to that visual interface but without a freshly saved scenario).

There are some additional Modgen-supplied utility functions that can be called by **ValidateParameters().** Many of these functions are used in the example of the next section.

bool **ParameterChanged**(const char **parameter_name*);
This function returns TRUE if parameter *parameter_name* was modified in the current edit session.

double **GetParameterValue**(const char **parameter_name* <, …>);
This function returns the value of the specified parameter cell. The number of optional arguments (indicated by …) equals the number of dimensions in the parameter. Each argument (after *parameter_name*) represents the cell's position (based on origin 0) in that dimension. Each argument is also separated with a comma, except that no comma follows the final argument. The function can only be used with parameters of simple numeric types (int, long, float, double).

Example:

In this example, assuming that AGEPROV is a two-dimensional parameter with eight rows and ten columns, the following retrieves the value of the third row and seventh column, and assigns that value to dValue.

```
dValue = GetParameterValue(AGEPROV,2,6);
```

void **SetParameterValue**(const char **parameter_name* , double *dValue* <, …>);
This function sets the value of the specified parameter cell. The number of optional arguments (indicated by …) equals the number of dimensions in the parameter. Each argument (following *dValue*) represents the cell's position (based on origin 0) in that dimension. Each argument is separated with a comma, except that no comma follows the final argument. The function can only be used with parameters of simple numeric types (int, long, float, double).

bool **CheckParameterTotals**(const char **parameter_name* , int *nDimension*, double *dDesiredTotal*, double *dToleranceFactor*);
This function verifies dimension totals for the dimension specified by the *nDimension* argument and returns TRUE if all dimension totals match *dDesiredTotal*. The function can only be used with parameters of simple numeric types (int, long, float, double). For parameters of type "int" or "long" the match has to be exact. For parameters of type "float" or "double" the comparison tolerance is determined by the *dToleranceFactor* argument; this tolerance factor is multiplied by the parameter storage precision which is indicated by the "decimals=" setting in the parameter's label. (If there is no decimals setting, the default storage precision is used--6 decimals for "float" and 15 decimals for "double"). The comparison tolerance is thus equal to *dToleranceFactor* * pow(10, - decimals). If a total along a specified dimension is zero, this in itself will not cause **CheckParameterTotals()** to return a result of FALSE..

int **ConfirmUndoParameterChanges**(const char **parameter_name*);
This function displays a dialog box that gives the user of the visual interface a chance to cancel all parameter changes if the validation routine failed. The return code indicates the button that was pressed in the dialog box (IDYES, IDNO or IDCANCEL). As the model developer, you can then program appropriate actions, depending on the return code.

bool **NormalizeParameter**(const char **parameter_name* , int *nDimension*, double *dDesiredTotal*);
This function re-scales the parameter by the ratio of the desired total and the real total along the indicated dimension. The function can only be used with parameters of simple numeric types (such as int, long, float or double). After this operation, the parameter totals along *nDimension* should be reasonably close to the desired total. The function returns TRUE if the parameter was modified during the normalization process. If a total along the specified dimension is zero, **NormalizeParameter()** will not try to normalize the dimension.

int **ConfirmCallToCorrectionRoutine**(const char **parameter_name*);
or

int **ConfirmCallToCorrectionRoutine**();
This function has two prototypes. It can be called when parameter validation failed for a specific parameter (which is the

version with the argument *parameter_name*) or for a group of parameters. It will display a dialog box informing the user of the visual interface that parameter validation failed and asking if the correction routine should be called. The return code indicates the button in the dialog box that was pressed (IDYES, IDNO or IDCANCEL). As the model developer, you can then program appropriate actions, depending on the return code.

void **MessageValidationFailed** (const char **parameter_name*);

or

void **MessageValidationFailed() ;**

This function simply displays an error message indicating that the parameter validation routine failed. It is similar to the **ConfirmCallToCorrectionRoutine()** function; however, its dialog box contains only the OK button and there is no follow-up question about whether the correction routine should be called. The function takes an optional argument (a parameter name) if the message is only related to one specific parameter.

Example of parameter validation code

In this example, validation is applied to a parameter called Coefficients, of type real (float or double), for which the elements of the first dimension should sum to 1. No validation takes place in this example when a parameter cell(s) changes or when a user of the visual interface navigates between different cells. However, if that user of the visual interface changes any of the parameter values and then enters **Scenario** > **Save** or **Scenario** > **Run/resume** or **Scenario** > **Validate,** a check is made to ensure that the values in the first dimension add up to 1.0, given the specified tolerance factor. If not, the user is given a chance to correct the values—and if the user replies yes, the parameter values are normalized via a call to **NormalizeParameter().**

```
bool ValidateParameters( SCENARIO_EVENT eEvent )

{

      bool bProceed = TRUE;

      int nResult;


      switch ( eEvent )

      {

            case SCENARIO_SAVE_EVENT:

            if ( ParameterChanged( "Coefficients" ) )

            {
```

```
                        if ( !CheckParameterTotals( "Coefficients", 1, 1.0, 100 ) )
                        {
                              nResult = 
ConfirmCallToCorrectionRoutine("Coefficients");
                              if (nResult == IDYES)
                              {
                              NormalizeParameter( "Coefficients", 1, 1.0 );
                              }
                              else if (nResult == IDCANCEL)
                              {
                                    bProceed = FALSE;
                              }
                        }
                  }
                  break;
            case CELL_MOVE_EVENT:
                  break;
            case BLOCK_CHANGE_EVENT:
                  break;
            case SCENARIO_RUN_EVENT:
                  if ( !CheckParameterTotals( "Coefficients", 1, 1.0, 100 ) )
                  {
                        nResult = ConfirmCallToCorrectionRoutine("Coefficients");
                        if (nResult == IDYES)
                        {
                              NormalizeParameter( "Coefficients", 1, 1.0 );
                        }
```

```
                else if (nResult == IDCANCEL)
                {
                      bProceed = FALSE;
                }
          }
          break;
    case SCENARIO_MENU_EVENT:
          if ( !CheckParameterTotals( "Coefficients", 1, 1.0, 100 ) )
          {
                nResult = ConfirmCallToCorrectionRoutine( "Coefficients" );
                if ( nResult == IDYES )
                {
                      NormalizeParameter( "Coefficients", 1, 1.0 );
                }
          }
          break;
    default:;
  }
  return bProceed;
}
```

Parameter extensions

Parameter extensions allow you to initialize a parameter with fewer data points than its definition requires. The parameter can then be extended along the first dimension using one of two extension methods supported by Modgen. (The choice of method is specified in the .mpp file, not the .dat file):

Syntax of the extend_parameter command (two possible methods):

```
extend_parameter   parameter_name ;
extend_parameter   parameter_name   related_parameter_name ;
```

The first method extends the data by repeating the last stored slice until the parameter is fully populated.

The second method uses a one-dimensional related parameter and requires that the leading dimensions of both the original parameter and related parameter be of type range. The parameters themselves must be of type double. In addition, the range corresponding to the original parameter that is being extended must either match the range or be completely contained within the range corresponding to the related parameter. The original parameter is then extended using the ratios between the corresponding cells of the related parameter as multiplying factors on the stored values, as is illustrated in the examples that follow. This second extension method is most useful for timeseries parameters which include historical and projected years, with time being the leading dimension of the parameters.

Parameter extensions are often useful for parameters containing dollar-denominated values. The values beyond the stored historical years may, for example, be projected using an index based on the Average Industrial Wage, as in the example below that extends the MaxLoan parameter. Modgen's visual interface indicates such values by displaying them in light gray within the grid.

Examples:

```
extend_parameter UniTuitionFees ;
extend_parameter MaxLoan AvgIndWage ;
```

In the following example, the range (MODELED_YEAR) of the parameter being extended (CCTuitionFees) is completely contained within the range (CCTUT_MODELED_YEAR) of the related parameter (CCTuitionFeeIndex).

```
range MODELED_YEAR

{

      1985, 2099

};

range CCTUT_MODELED_YEAR

{

      1980, 2101

};

double      CCTuitionFees[MODELED_YEAR][PROV_OF_STUDY][CC_TUITION_FOS];

double      CCTuitionFeeIndex[CCTUT_MODELED_YEAR];

extend_parameter  CCTuitionFees CCTuitionFeeIndex ;
```

In this example, assume that 2009 is the last year with a true data value for CCTuitionFees in the .dat file. An extended value for 2010 would thus be determined by multiplying the 2009 value of CCTuitionFees by the ratio of

CCTuitionFeeIndex for 2010 over CCTuitionFeeIndex for 2009. Similarly, an extended 2011 value for CCTuitionFees would instead multiply the 2009 value by the ratio of CCTuitionFeeIndex for 2011 over CCTuitionFeedIndex for 2009—and so on.

Parameter decimals

You can specify the number of decimals stored with a numeric parameter. Unlike the decimals setting in table expressions, parameter decimals are used both for formatting a displayed value and rounding a stored value. (In table expressions, decimals are only used for formatting the displayed values). Parameter decimals prove particularly useful for extended parameters, especially those containing dollar-denominated values.

To specify the number of parameter decimals, you need only include a "decimals=" string in the parameter's label. (This is the same convention that is used to specify the number of decimal settings in table expressions.)
Example (from .mpp file):

```
//EN Income exemption decimals=0 //FR Exemption du revenu

double  IncomeExemption[MODELED_YEAR];
```

The decimals are specified in the .mpp files. The default value for parameter decimals is -1 which means that the values are stored as entered without any rounding and that there is no formatting when the values are displayed. In general, any negative decimals setting can be interpreted in the same way.

Hiding parameters

The function **hide()** is used to prevent parameters from appearing in the visual interface to a Modgen model.

Syntax of hide():

hide (parameter_name {parameter_name, parameter_name …});

The arguments are the names of the parameter(s) or parameter group(s) that are to be hidden, with each parameter name or parameter group name separated by a comma. (Note that the same function, **hide()**, can also be used to hide tables in the visual interface; however, parameters and tables cannot be combined as arguments.) When the name of a parameter group is used, the impact is equivalent to using the name of each parameter that is a member of the group. A parameter group is shown in the visual interface only if it contains at least one parameter that is not hidden. Hidden parameters are not included in the database exported to Microsoft Excel when such an export is requested through the visual interface (even if the user of the interface asked to include the model's parameters in the export request).

The table facility

This section describes the Modgen table facility in detail, and also provides examples of how to define output tables in Modgen.

Overview of the table facility

The standard means of reporting model results is through the cross-tabulation facility. This facility enables you to specify a wide variety of tables on the simulated population. Generally, the tabulations summarize information on the following (usually categorized by any number of classification states):

- values of continuous states such as earnings or costs.
- durations of actor states.
- counts of actors or events (state changes).

Unlike normal cross-tabulators, Modgen tabulation has the added richness of time to take into account. To maximize computational efficiency and to minimize overhead, Modgen tabulations are accumulated on-the-fly. To understand this, you can think of the actors moving through time, journeying through the cells of a table as states change. Modgen builds the table outputs with little knowledge of the actor's past and no knowledge of the future. As an actor enters a cell in the table, the opening values of various dependent states are put aside and a cell entry count (called **unit**) is incremented by one. When the actor exits a table cell and a request to move time forward is detected by Modgen, a table update is triggered using the exiting state values, the opening state values and the setting for a cumulation operator. The cumulation operator tells Modgen how to summarize the values within the cell. This operator can determine minimums or maximums but the default operator is called delta. The delta operator subtracts the opening values from the exiting values and adds the result to the running total already within the cell. The delta method of cumulation only makes sense for states which can be expressed as cumulative lifetime.

Continuous states such as earnings meet this requirement nicely with a minimum amount of overhead. Suppose a person actor has been defined with a state containing cumulative lifetime earnings and you wish to tabulate average earnings by age in one-year intervals. Each case has one such actor and the simulation has been launched with 100 cases. For the numerator, when the first actor, say, turns 30, his cumulative lifetime earnings are put aside, **unit** is incremented by one for that table cell, and time moves forward with any number of intervening events until age 31. At that time, his cumulative earnings at age 30 are subtracted from his cumulative earnings at age 31 in order to obtain the earnings for that interval. When the second case comes along, the same calculations take place on cell entry and exit, the result for the numerator being added to the running total from the previous case. This process continues for all 100 cases. Note that with this technique, if the underlying behavioural model updates the earnings state every two weeks, no computational overhead is added to the tabulation process, since the differences in cumulative earnings are still only evaluated once each year.

When the last case is completed, the division for the average takes place. You could use **unit** as the denominator for the average but it would be a poor choice; instead, the use of durations is essential for the tabulation process. Suppose you wanted to tabulate average annual earnings within 10-year age intervals rather than within the one-year interval originally described above. .You would then really want to divide by the interval duration (in this case, ten, but which just happened to be one in the original example above). The use of **unit** as the denominator would be incorrect because you would not want to use the number of actors entering the cell. To get the annual average for the ten-year interval, you would need to sum the earnings over the ten-year interval for all actors and then divide by the sum of the interval durations for all actors. (Modgen supplies the derived state specification **duration()** to determine the sum of all actors' interval durations.)

This works well for earnings but a different sort of question arises if you would like to count events within an age interval. The events can be viewed as changes to a specific value of a state. The tabulation is an interval delta of the cumulative lifetime changes to that state value. Again, Modgen provides derived state specifications which can do the lifetime accumulations for you. For example, suppose you wanted to tabulate the total number of marriages at age 30 for all 100 actors. In this case, you would tabulate with a derived state specification called **entrances()**, with arguments specifying the state ***marital_status*** and state value *MARRIED*. When combined with the delta cumulation operator, the cell will contain the number of marriages during that one-year interval. .

Table specifications are defined using the **table** command in the model's .mpp files. In general, classification, partition or logical states define the table's dimensionality while table expressions specify the cell values in the table. The former are called the classification dimensions of the table; the latter is called the analysis dimension. The general purpose of tabulation is to show the relationship between the states in the classification dimension and the table expressions in the analysis dimension. A table can only have one analysis dimension but that dimension can be made up of multiple table expressions.

All tables are associated with a model actor. An optional filter allows you to report on subsets of the simulated population. Labeling and naming each table is especially helpful in conjunction with writing output to Microsoft Excel spreadsheets and Microsoft Access output databases (both of which are done through Modgen's visual interface).

You can also define table groups for use in the Group View option of the model's visual interface. In addition, the user of the visual interface is allowed to select, at run time, a subset of tables to be accumulated and sent to output. The simulation time for a specific run can be sped up considerably by excluding unneeded tables. For more information on table output, refer to the "*Guide to the Modgen Visual Interface*".

Table and table group definitions

Table definitions

Syntax of the table command:

```
table <sparse> actor_name <table_name> <[filtering_criteria]>  <// language_code  table_label>
{  < classification dimension  <+> <// language_code  classification_dimension_label>
            *>
   < classification dimension <+> *>
            …
            {
                        analysis dimension table expression,  <// language_code  expression label > <expression options>
                        <analysis dimension table expression,  <//language_code   expression label > <expression options>>
                        …
                        <analysis dimension table expression  <// language_code  expression label > <expression options>>
            }
            <*
   classification dimension  <+> < // language_code  classification_dimension_label> >
            …
} ;
```

If the table is very large and very sparse (contains many zeroes), you may save space in your output database by adding the optional **sparse** keyword to the table statement.

There is no limit to the number of classification dimensions in a table. The "*" character is used to delimit the classification dimensions. The dimension label is optional. However, if no label has been explicitly provided, Modgen constructs a language-specific label by using the language-specific label of the underlying state.   Also, if the label is absent, the "*" character can be placed on the same line as the dimension. An optional "+" character is used to indicate that a dimension total be computed for the corresponding classification dimension.

Each table can contain only one analysis dimension but that dimension can have any number of table expressions, each separated by a comma. (No comma is used or permitted after the final expression, however.) The table expression options specify decimal and scaling factors for the data in the body of the table. However, if the analysis dimension appears in the columns of your table (i.e. no classification dimension statements are specified after the table expressions of the analysis dimension), you can optionally specify different decimal and scaling factors for each column of your table.

The table_name is optional; if a name is not specified, default names (i.e., Table X1, Table X2, etc…) are generated by Modgen. You should use a table_name however for the Microsoft Excel output option and/or the Microsoft Access output options, both of which are initiated through Modgen's visual interface.

Once the simulation run is complete for all cases in a case-based model or for the prescribed amount of time in a time-based model, the table expressions are evaluated with the aggregate values of the states therein. Therefore, expressions which are ratios of states report the aggregate value of the numerator over the aggregate value of the denominator and not the average of the actor-specific ratios from the simulation run.

Table group definitions

Table groups allow you to group tables for visual selection in the Group View tab of Modgen's visual interface. Table groups can only be defined in .mpp files. A given table can belong to any number of groups. A group may consist of tables and/or other table groups. However, a group must not be a member of itself, directly or indirectly; if this happens, Modgen will generate a "circular dependency" error.

Syntax of a table group definition:

```
table_group groupname
{
    //language_code label
    table_name|groupname, ..., table_name|groupname
};
```

The tables are displayed by the visual interface in the same order in which they are entered in the **table_group** command Example:

```
table_group LOAN_TABLES
{
      //EN Loan-related Tables
      LOAN, FORGIVEN, INTRELIEF, LOANNEEDS
};
```

Table expressions

The table expressions in the analysis dimension can include:

- the **unit** keyword
- states
- cumulation keywords
- tabulation operators
- derived state specifications
- numeric constants in conjunction with the arithmetic operators + - * /, the sqrt function, and parentheses ().

Each of these (except for numeric constants with arithmetic operators) are discussed further in this section.

Expression labeling and table display options (scaling and decimals)

The table expressions in the analysis dimension, which define the contents of the table cells, are each separated by a comma in the table specification. Each table expression can be followed by a label (starting with a // and followed by a language code such as EN or FR). If no label is provided, Modgen constructs a language-specific label by using the language-specific label of the underlying state. When provided, the label can also contain special formatting strings which control the presentation (scaling and/or number of decimals) of the tabular results:

decimals= n
indicates that the value of the expression is to be reported to *n* decimal places. The table decimals may also be changed dynamically by using the Decimals tab on the Table Properties dialog box of Modgen's visual interface. If the table has more than one expression in its analysis dimension, a different number of decimals may be specified for each expression.

scale= [-]m
indicates that the values of the expression are to be multiplied by 10***m.* A negative sign can proceed *m* to effectively divide the result by 10***m*.

The unit keyword in table expressions

The **unit** keyword creates a table-specific state that stores the number of actors that entered each cell of a table. Therefore, it is useful for reporting the number of actors who had a specific set of characteristics at some point in their simulated lifetime. It is also useful as a denominator when calculating average values with continuous states, as shown in Example 2.

The **unit** keyword can, however, lead to double counting so it should be used with caution. For example, if you were counting entrances into the *MARRIED* value of a ***marital_status*** state, people who were married twice during their lifetimes would each be counted twice and would thus inflate the result if you were only interested in counting individuals who had ever been married at least once in their lifetimes.

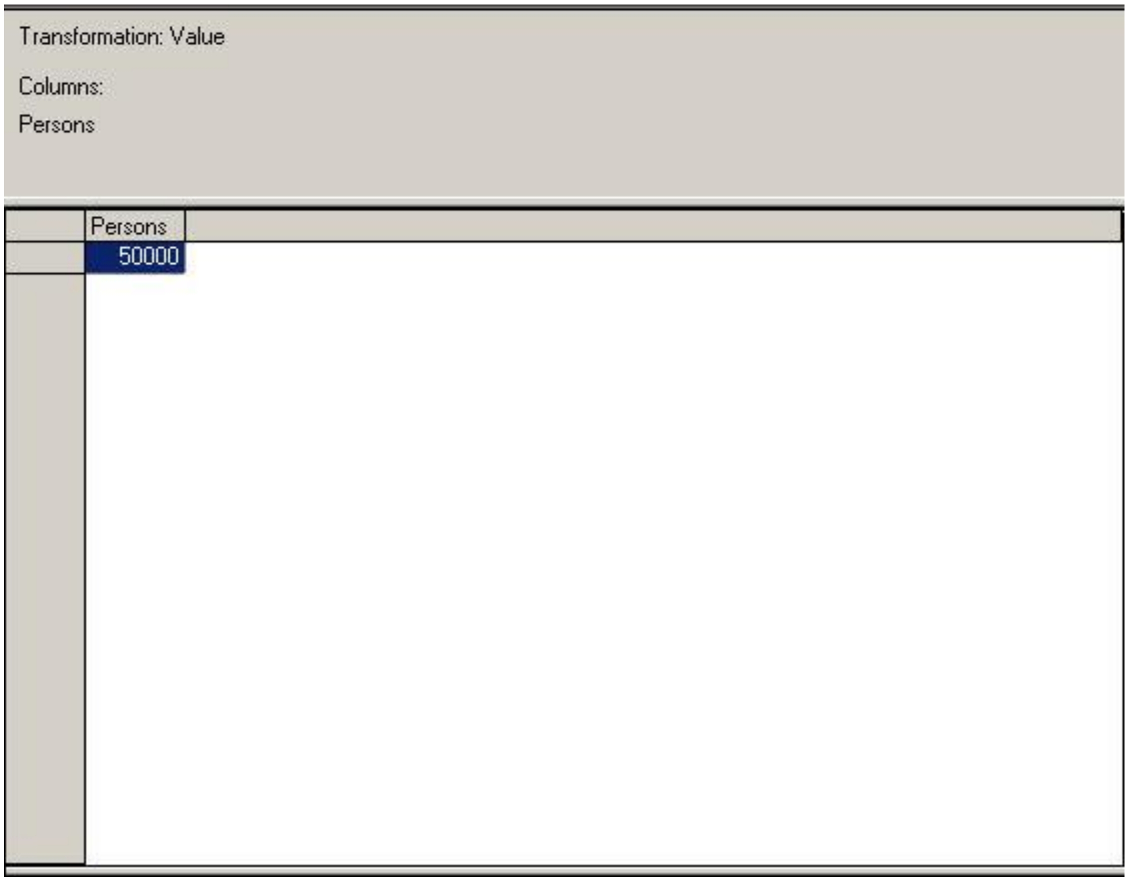
Example 1

In this table we report the total number of person actors in a simulation run. In this example, the special state **unit** is the only table expression specified, and there are no classification dimensions at all. Note how the label in the comment beside **unit** in the table specification (i.e. “Persons”) is carried through to the report. The table label (“Number of cases simulated”) is also displayed on both the title bar and the overall list of the model’s tables in Modgen’s visual interface. However, each example in this section shows just the resulting table itself, and this resulting table does not include the table label.

Table Specification

```
table Person //EN Number of cases simulated

{

      {

      unit //EN Persons

      }

};
```

Table Result


Transformation: Value

Columns:
Persons

	Persons
	50000

States in table expressions

The values of continuous states can be reported in the analysis dimension using the table facility. Generally, the best-suited states for such reporting are those which are accumulating over the lifetime of an actor and which have no fixed increments when their values change.

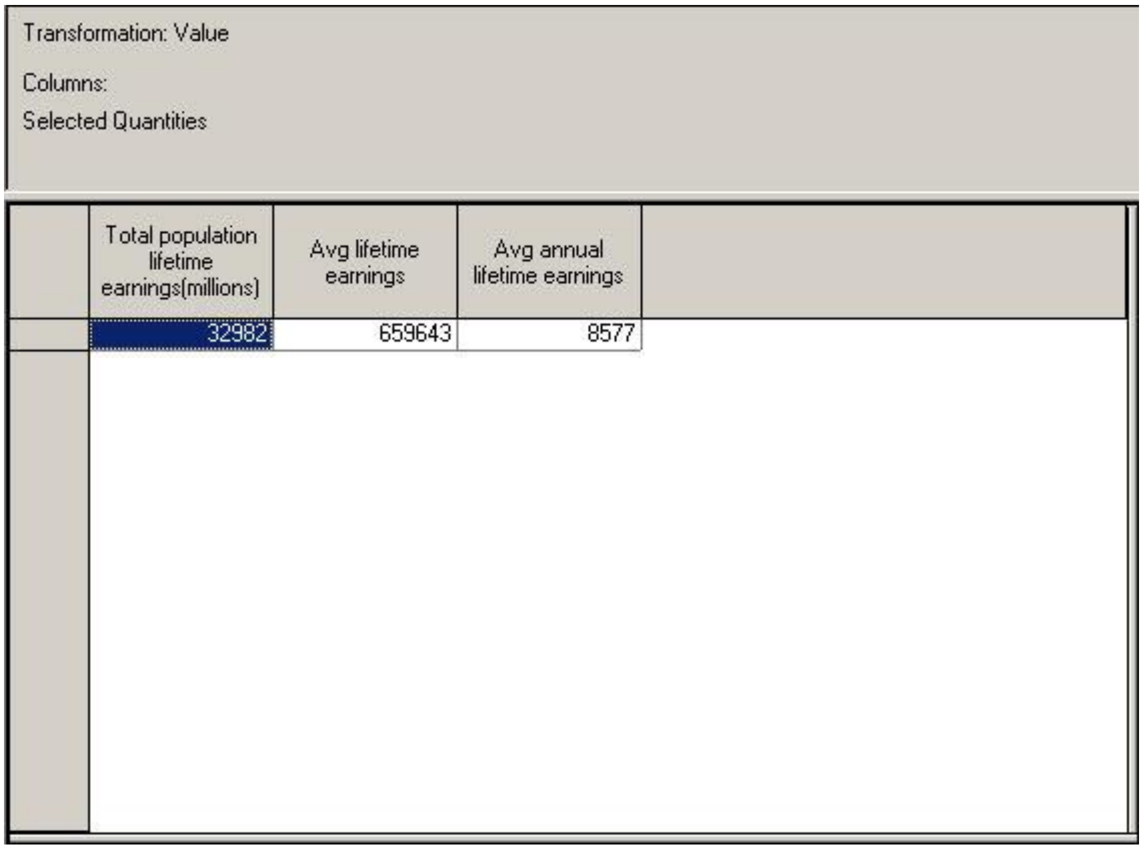
Example 2

The table in this example has three table expressions in its analysis dimension: the total population's accumulated lifetime earnings (scaled to millions of dollars), the average accumulated lifetime earnings of each person, and the average annual earnings of each person. (Note that when there is more than one expression in the analysis dimension, Modgen shows the label specified after the final expression in the output tables; if there is no such label, Modgen instead uses the heading "Selected Quantities" in the output tables. When there is only one expression, Modgen displays that expression's label in the output tables.)

Table Specification

```
table Person //EN Earnings summary

{

      ...

      {

            earnings, //EN Total population lifetime earnings(millions) scale=-6

            earnings / unit,        //EN Avg lifetime earnings

            earnings / duration ()  //EN Avg annual lifetime earnings

            ...

      }

};
```

Table Result



Transformation: Value

Columns:
Selected Quantities

	Total population lifetime earnings(millions)	Avg lifetime earnings	Avg annual lifetime earnings
	32982	659643	8577

Cumulation operators in table expressions

Cumulation operators control the way in which states for all actors are summarized within table cells. They can be interpreted as running totals or running minimums/maximums. In all cases, the state can be a simple state or a derived state.

delta(*state*)

This is the default cumulation operator. It allocates the difference between the value of the state when the actor enters a table cell and the value of the state when the actor leaves a table cell.

delta2(*state*)

This operator is similar to the **delta()** operator but reports the squared value, which is useful if you wish to calculate the standard deviation of the simulated quantity.

max_delta(*state*)

This keyword reports the maximum value of the state between when the actor first enters a table cell and when the actor leaves it.

min_delta(*state*)

This keyword reports the minimum value of the state between when the actor first enters a table cell and when the actor leaves it.

max_value_in(*state*)

This keyword returns the maximum value of the state over all cell entries when these actors enter the cell of the table.

max_value_out(*state*)

This keyword returns the maximum value of the state over all cell entries when these actors exit a cell of the table.

min_value_in(*state*)

This keyword returns the minimum value of the state over all cell entries when these actors enter the cell of the table.

min_value_out(*state*)

This keyword returns the minimum value of the state over all cell entries when these actors exit a cell of the table.

nz_delta(*state*)

This keyword returns a 1 in instances where the **delta()** operator would produce a non-zero value and 0 otherwise.

nz_value_in (*state*)

This keyword returns a 1 in instances where the **value_in()** operator would produce a non-zero value, and 0 otherwise.

nz_value_out (*state*)

This keyword returns a 1 in instances where the **value_out()** operator would produce a non-zero value, and 0 otherwise.

value_in(*state*)

This keyword returns the value of the state when the actor first enters the cell of the table.

value_in2(*state*)

This keyword returns the squared value of the state when the actor first enters the cell of the table.

value_out(*state*)

This keyword returns the value of the state when the actor exits a cell of the table.

value_out2(*state*)

This keyword returns the squared value of the state when the actor exits a cell of the table.

Tabulation operators in table expressions

The table facility distinguishes between two types of tabulation: interval tabulation and event tabulation. These operators control the tabular output differently when a state in the classification dimension and a table expression in the analysis dimension in a table specification change simultaneously within the same event. The new values of the table expressions are tabulated in the cell previous to the change under the **interval()** operator. The values of the table expressions are tabulated in the cell after the event has occurred under the **event()** operator. The default tabulation is the **interval()** operator, as it is the operator of choice for situations where duration-related variables are being analyzed. Sometimes however, you may need to count certain events by their type, where the count and the type could change at the same time. In such cases, event tabulation should be used.

Syntax of the tabulation operator:

tabulation_operator(*state*)

where **tabulation_operator** is either **event** or **interval**.

If both a tabulation operator and a cumulation operator are used, then the syntax is:

cumulation_operator (**tabulation_operator** (*state*))

The state argument can be either a simple state or a derived state expression.

Example:

In this example, both the analysis and classification dimensions of the table depend on the state ***education_level***, which means that they each change at the same time. Because the goal of the table in this example is to report the age when the person obtained the changed education_level, the age needs to be reported in the new education level that was obtained—hence, the **event** tabulation operator is needed.

```
table Person EducationLevels  //EN Age of obtaining education levels
{
      {
                  //EN Average age of obtainment
            event(value_at_changes(education_level,age)) /
```

```
                event(changes(education_level))     }
      * education_level       //EN Education level
};
```

Derived state specifications in table expressions

A rich set of derived state specifications is available for the purpose of table generation, as discussed in "Derived state specifications". Here are some examples:

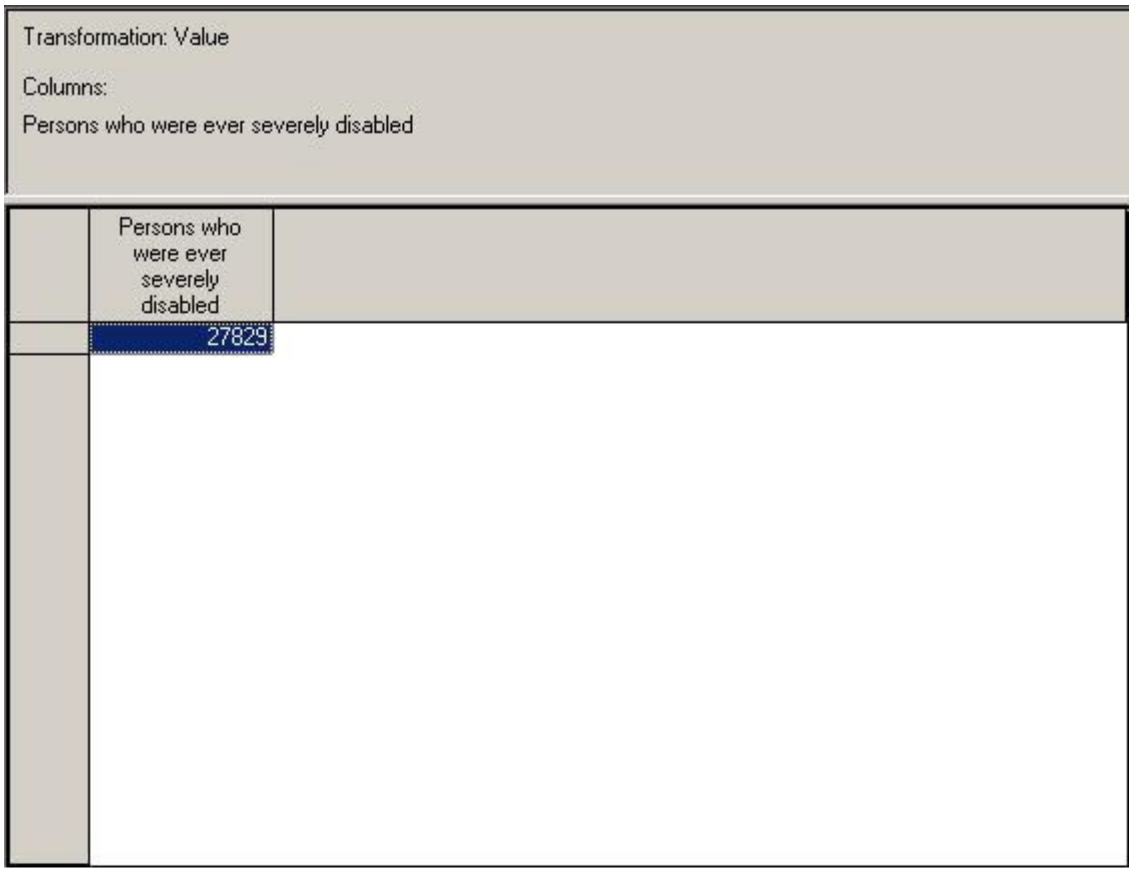
Example 3

This table consists of an expression which gives the number of persons who were ever classified as severely disabled. The results indicate that of the 50000 simulated persons, 27829 were severely disabled at least once in their life.

Table Specification

```
table Person //EN Selecting the severely disabled
{
      {
            //EN Persons who were ever severely disabled
            entrances(dfle, SEVERE_DIS )
      }
};
```

Table Result


Transformation: Value

Columns:
Persons who were ever severely disabled

	Persons who were ever severely disabled
	27829

Example 4

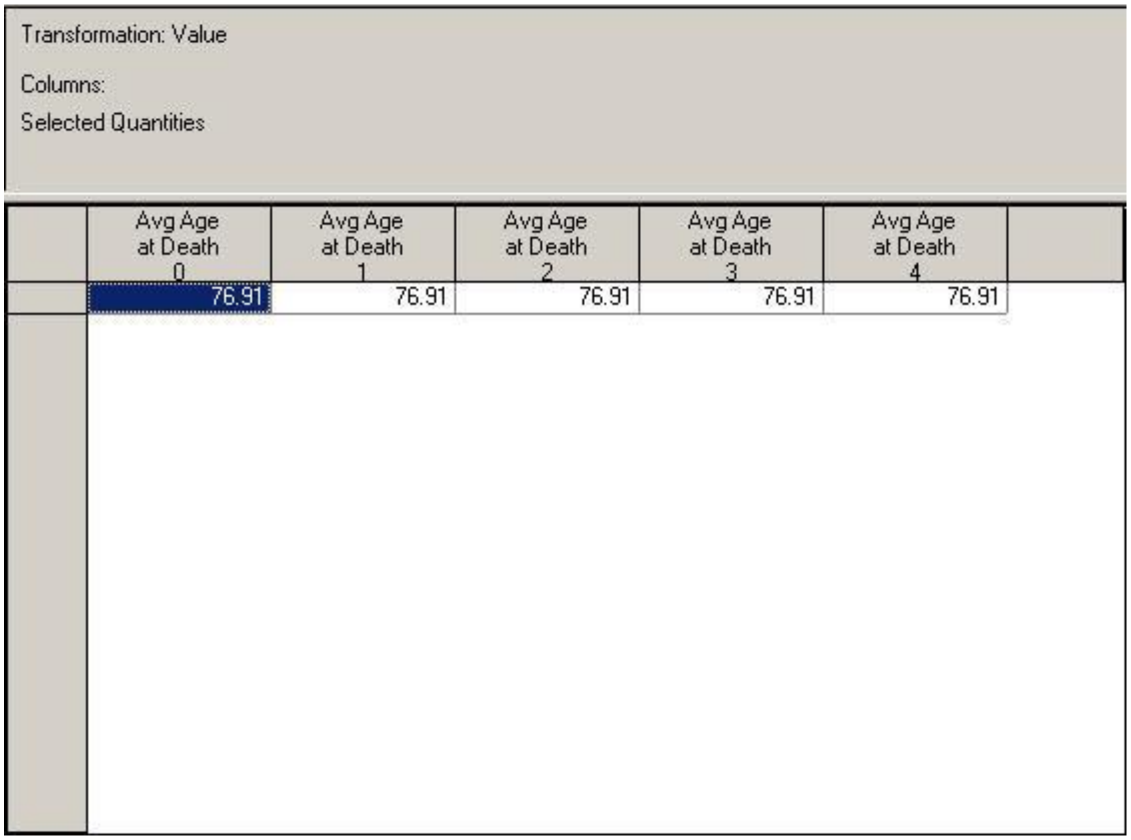
In this example, we calculate the average age at death, five separate ways for the population as a whole, using some of Modgen's derived state specifications, plus the special state created with the **unit** keyword.

Table Specification

```
table Person //EN different ways to calculate age at death

{

        {

                value_at_entrances (life_status, NOT_ALIVE, age)

                / unit, //EN Avg Age at Death 0 decimals=2

                value_at_exits ( life_status, ALIVE, age )            / unit, //EN Avg Age at
Death 1 decimals=2          value_at_transitions (life_status,ALIVE,NOT_ALIVE,age)

                / unit, //EN Avg Age at Death 2

                value_at_changes ( life_status, age )

                / unit ,      //EN Avg Age at Death 3 decimals=2
```

```
            duration ( life_status, ALIVE )

            / unit //EN Avg Age at Death 4 decimals=2

    }

};
```

Table Result



Transformation: Value

Columns:

Selected Quantities

	Avg Age at Death 0	Avg Age at Death 1	Avg Age at Death 2	Avg Age at Death 3	Avg Age at Death 4
	76.91	76.91	76.91	76.91	76.91

Empty cells in tables

It is possible to specify tables in which some cells remain empty (or zero) at the completion of the simulation run. In such situations, an undefined table expression is reported as a blank cell in the Modgen visual interface.

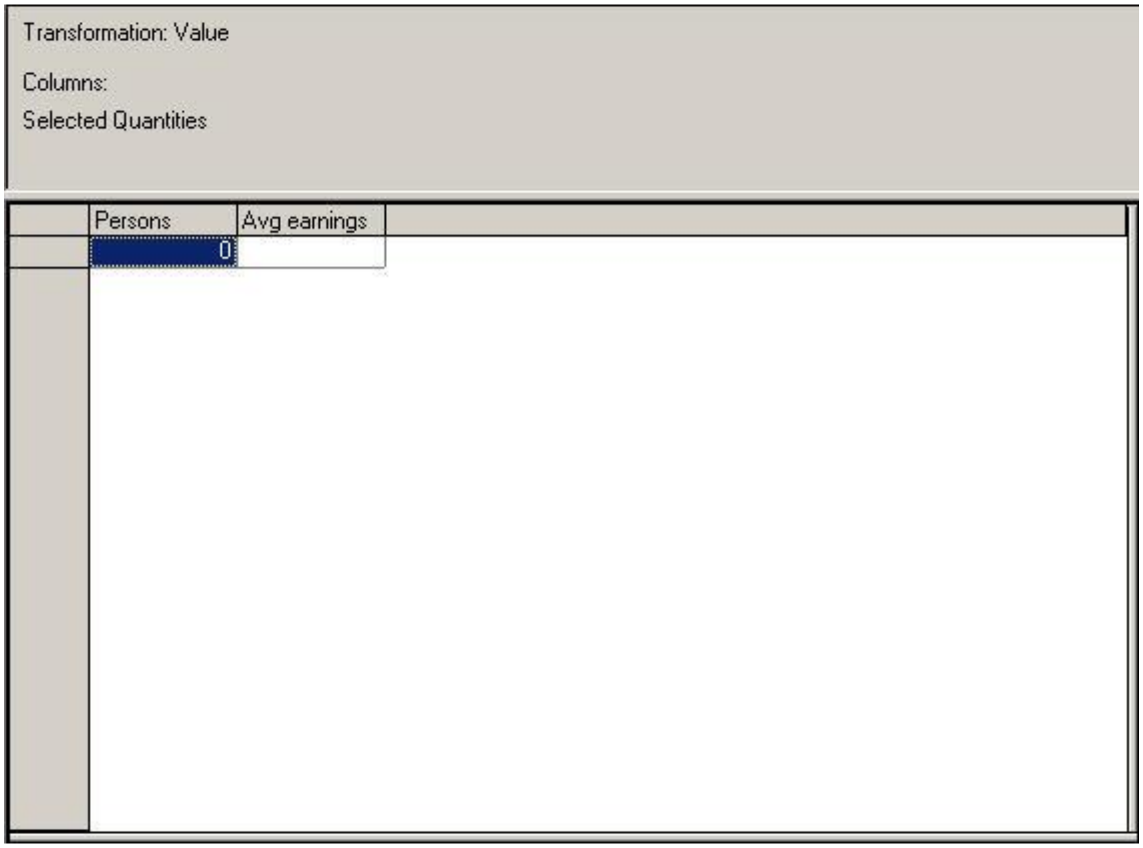
Example 5

In this example, the table reports results for the female population under the age of 0. The average earnings expression is undefined since the value for **unit** is zero.

Table Specification

```
table Person[sex==FEMALE && age < 0] //EN perverse example: females under the
age of zero

{
	{
		unit,    //EN Persons
		earnings / unit //EN Avg earnings
	}
};
```

Table Result



Transformation: Value

Columns:

Selected Quantities

	Persons	Avg earnings
	0	

Standard errors and coefficients of variation

All of the table output examples in this section have shown actual tabulated amounts or values, which is the default behaviour and which is conveyed by the heading “Transformation: Value” at the top of each table. However, Modgen also allows tables to be produced that show standard errors (reflecting the variability of simulation estimates) or coefficients of variation (reflecting the reliability of simulation estimates).

Such tables are requested through Modgen’s visual interface to your model and are available as long as there were **n** subsamples (for a case-based model) or **n** replicates (for a time-based model) specified in the visual interface (where **n**>1) before the most recent simulation run. These statistics are produced by creating **n** parallel sub-samples (case-based model) or **n** separate replicates (time-based model) for each simulation run and then calculating the values from those observations.

Table dimensions

The classification dimensions of a table are specified through states of type classification, derived state specifications which return discrete values, or nested derived state specifications which return discrete values. There is no limit to the number of classification dimensions in a table. These dimensions are linked together with the “*” character to determine the total rank of the table. To include the sum over a particular dimension, place a “+” character after the dimension name.

Table dimensions whose values are constant for each actor

These types of tables are the easiest to interpret as each actor is present in only one cell of the table over the actor’s entire lifetime. For example, persons are either male or female, and can have only one age at death.

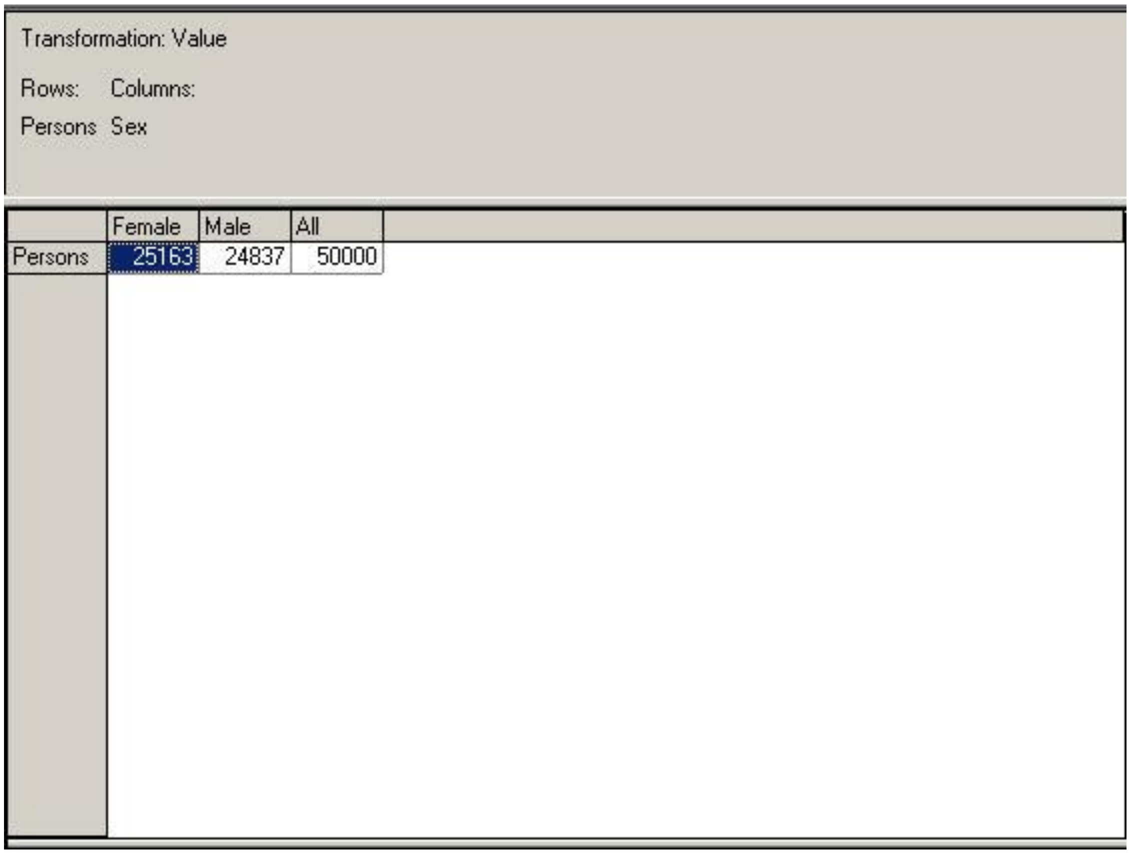
Example 6

In this table we stratify the total number of persons in the simulation run by their sex. The state created by **unit** is the only table expression specified in the analysis dimension. The table’s column dimension contains the single classification state **sex**. Note that a “+” character follows the classification state, indicating that the table includes a separate extra column for the sum of males and females.

Table Specification

```
table Person //EN stratify persons by gender (in columns)
{
      {
            unit //EN Persons
      }
      * sex+
};
```

Table Result



Transformation: Value

Rows: Columns:
Persons Sex

	Female	Male	All
Persons	25163	24837	50000

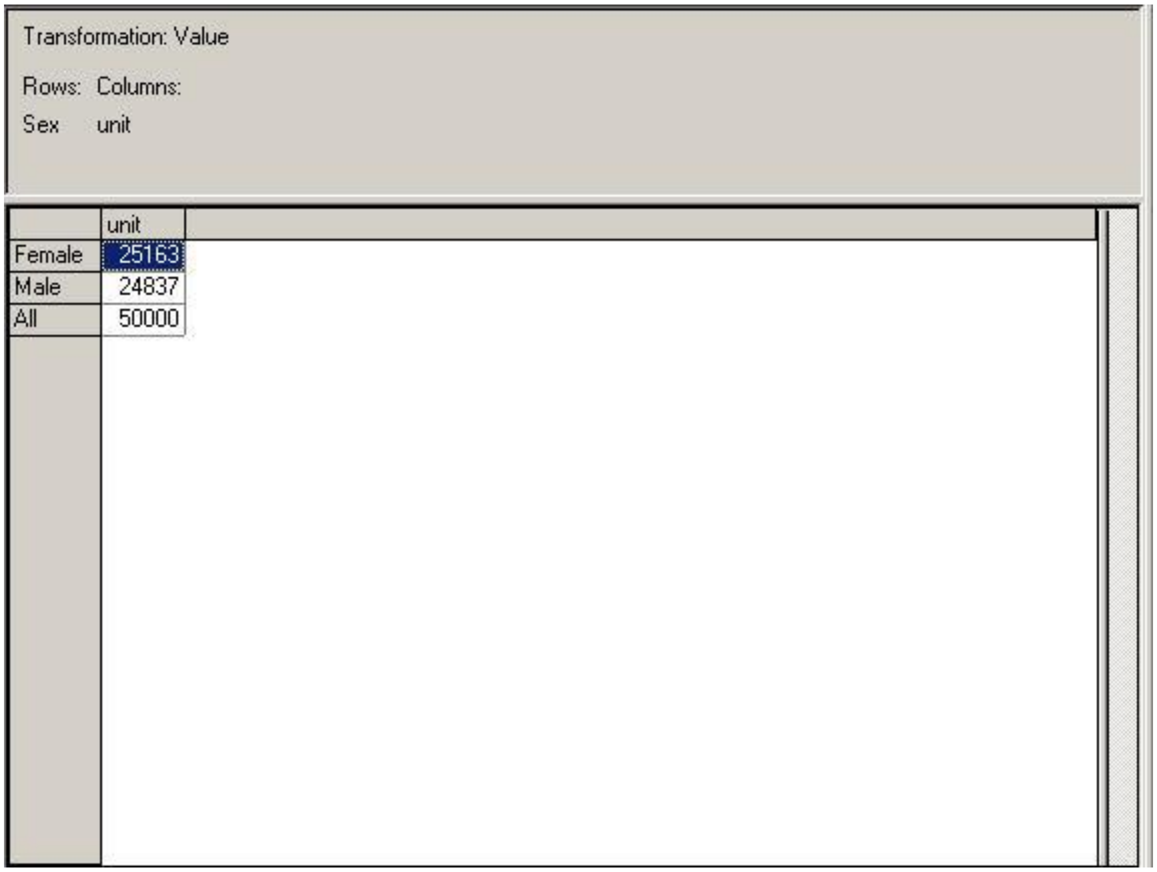
Example 7

In this table, we stratify the total number of persons in the simulation run by sex along the row dimension.

Table Specification

```
table Person //EN stratify persons by gender (in rows)

{

      sex+ *

      {

            unit //EN unit

      }

};
```

Table Result


Transformation: Value

Rows: Columns:
Sex unit

	unit
Female	25163
Male	24837
All	50000

Example 8

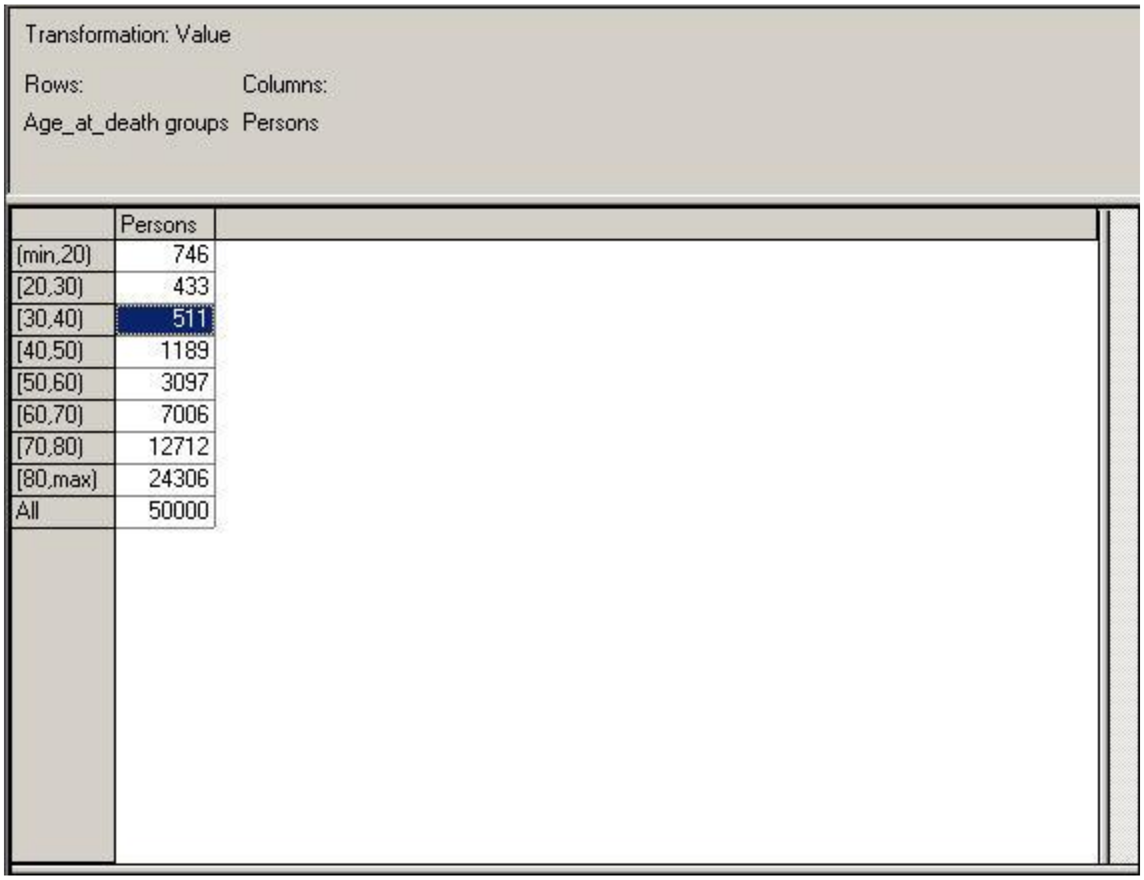
In this example, the age at death is reported by age class. The age breakdown is determined by the partition AGE_GROUPS, which would have been defined earlier in the model code via the statement:

```
partition AGE_GROUPS { 20, 30, 40, 50, 60, 70, 80 };
```

Table Specification

```
table Person //EN cases stratified by age at death

{

      split (age, AGE_GROUPS )+ //EN Age_at_death groups

      *

      {

            entrances (life_status,NOT_ALIVE) //EN Persons

      }

};
```

Table Result



Transformation: Value

Rows:	Columns:
Age_at_death groups	Persons

	Persons
(min,20)	746
[20,30)	433
[30,40)	511
[40,50)	1189
[50,60)	3097
[60,70)	7006
[70,80)	12712
[80,max]	24306
All	50000

Table dimensions whose values change for each actor

There are many states in classification dimensions whose values change over the course of an actor's simulated lifetime. Examples of such states are a person's age or disability status. If such a state is used to define a table's classification dimension, then an actor may be present in more than one table cell over the actor's lifetime. Therefore, these types of tables require careful thought in their formulation.

Example 9

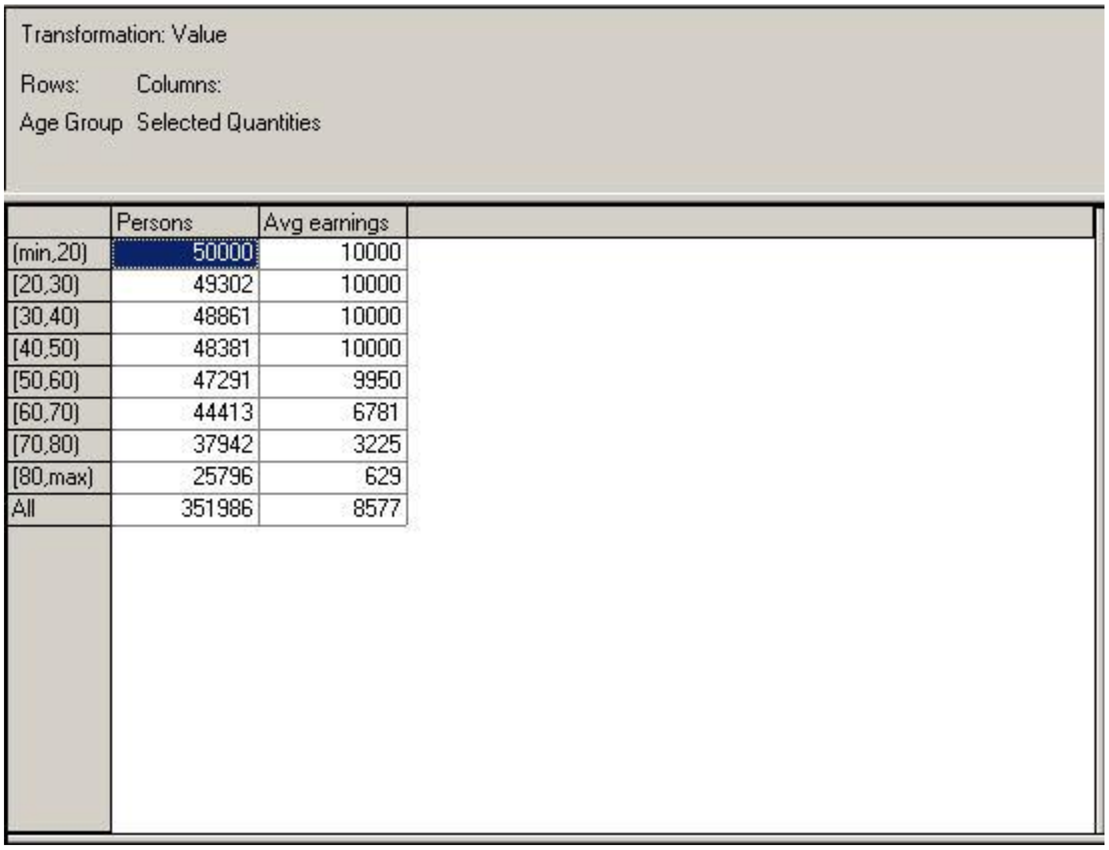
In this example, the first column reports the population (using the keyword **unit)**, classified into one of eight age groups using the derived state specification **split()**. The sum at the end of this column is greater than the number of persons overall (50000) since each individual in the simulation run increments each row of the table as that individual passes from one age class to the next. Therefore, this table shows the survivorship pattern of the simulated individuals. The results can be interpreted as follows: of the 50000 cases, 49302 of them survived to their 20th birthday, 48861 survived to their 30th birthday, 47291 survived to their 50th birthday, and 44413 survived to their 60th birthday. This column can also be interpreted as the number of persons who ever entered the respective age group.

The second column of the table reports the average annual earnings of the population. The derived state specification **duration()** is used as a denominator to correctly estimate the years of a person's life.

Table Specification

```
table Person //EN cases and their average annual earnings by age group

{

      split ( age, AGE_GROUPS )+  *

      {

            unit,        //EN Persons

            earnings / duration () //EN Avg earnings

      }

};
```

Table Result



Transformation: Value

Rows: Age Group

Columns: Selected Quantities

	Persons	Avg earnings
(min,20)	50000	10000
[20,30)	49302	10000
[30,40)	48861	10000
[40,50)	48381	10000
[50,60)	47291	9950
[60,70)	44413	6781
[70,80)	37942	3225
[80,max)	25796	629
All	351986	8577

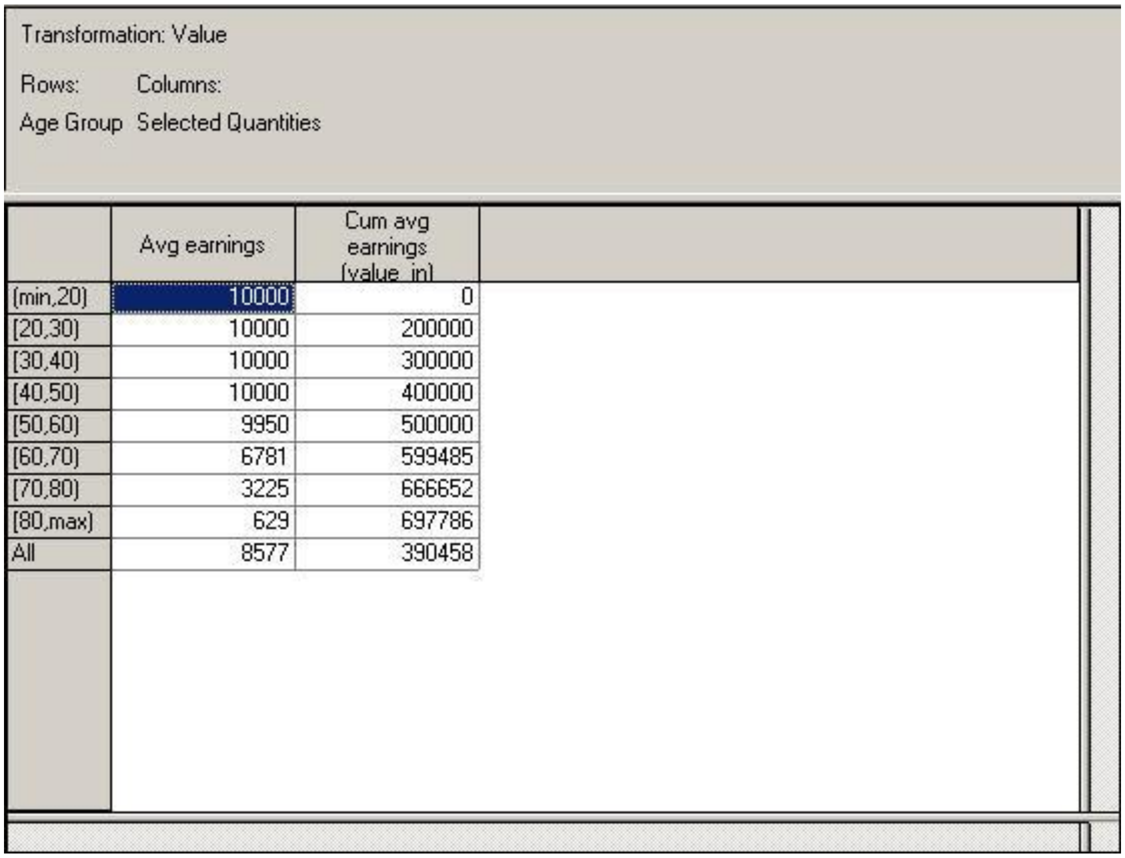
Example 10

In this example, the population is classified into one of eight age groups using the derived state specification **split()**. The first column of the table reports the average annual earnings of the population (which was the second column in Example 9). The second column of the table reports the accumulated earnings of the population over their lives to date, using the **value_in()** cumulation operator to get the accumulated earnings entering each ten-year age group.

Table Specification

```
table Person //EN average annual and cumulative earnings by age group

{

      split ( age, AGE_GROUPS )+  *

      {

            earnings / duration () , //EN Avg earnings

            value_in (earnings) / unit //EN Cum avg earnings (value_in)


      }
};
```

Table Result


Transformation: Value

Rows: Age Group
Columns: Selected Quantities

	Avg earnings	Cum avg earnings (value_in)
(min,20)	10000	0
[20,30)	10000	200000
[30,40)	10000	300000
[40,50)	10000	400000
[50,60)	9950	500000
[60,70)	6781	599485
[70,80)	3225	666652
[80,max)	629	697786
All	8577	390458

Table filtering

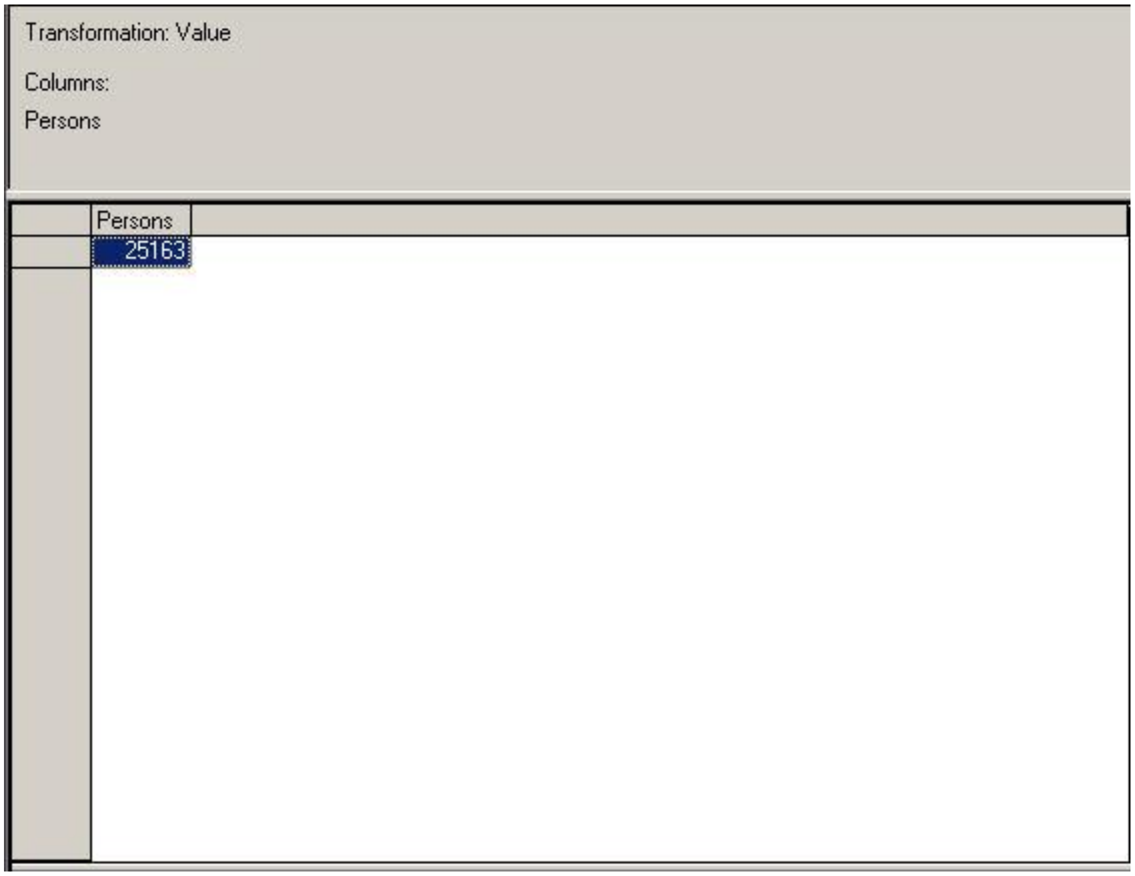
The **table** command allows you to select subsets of the simulated population through a filtering option. For example, if you filter on a disability state and specify a duration analysis state, you get durations which begin at the point the individual entered that disability state.

When creating your filter, you can use C++ boolean operators, such as ==, >, >=, <, <=, !=, &&, or ||.
Example 11

In this example, a filter is used to only report on females
Table Specification

```
table Person      //EN filtering on the females

[sex==FEMALE]

{

      {

            unit //EN persons

      }
};
```

Table Result


Transformation: Value

Columns:
Persons

	Persons
	25163

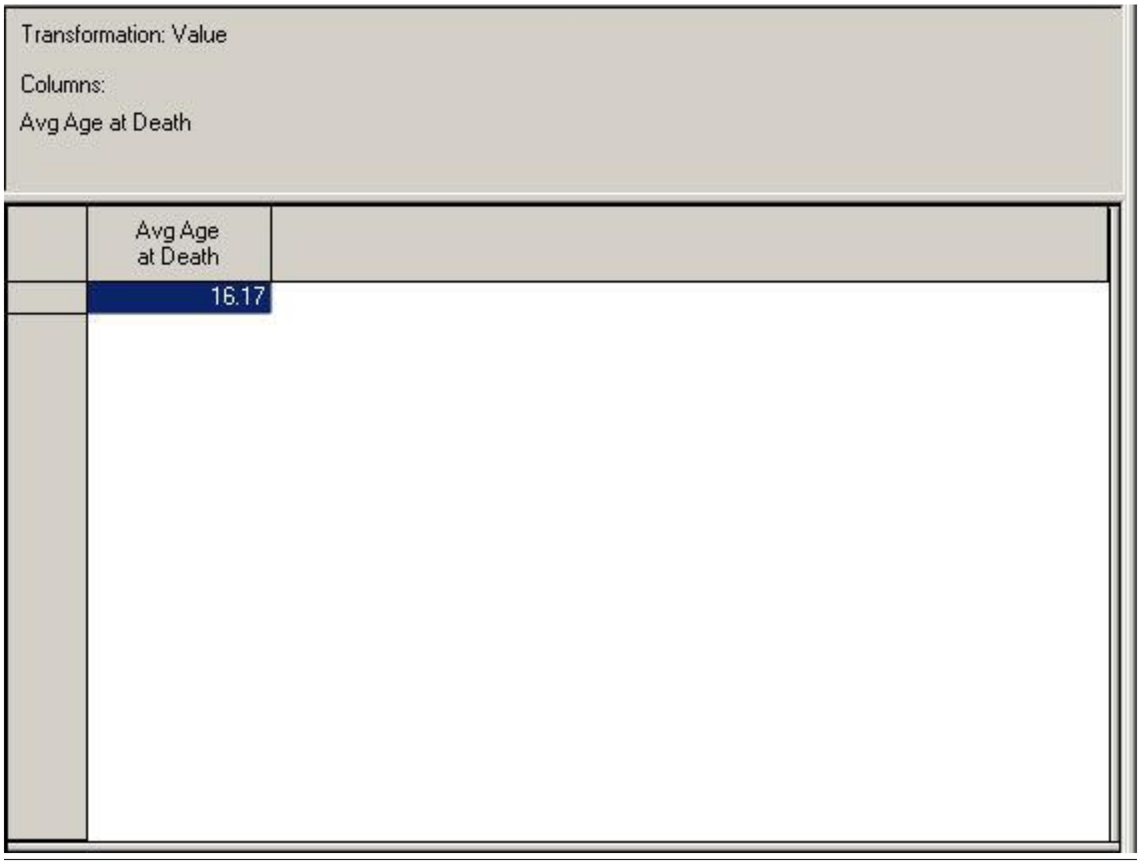
Example 12

In this example, the table attempts to report the average age at death of persons who were ever disabled over the course of their life but does so erroneously. The error was made to show how the results of a duration derived state specification are conditioned on the table filtering. What the results actually reveal are the average number of years an individual is in a disability state before death. (To get the age at death, the numerator in this example would change from duration(life_status, ALIVE) to value_out(duration(life_status, ALIVE))).

Table Specification

```
table Person      //EN Filtering on the disabled

[dfle!=NO_DIS]

{

      {

            duration ( life_status, ALIVE )

                  / unit //EN Avg Age at Death decimals=2

      }

};
```

Table Result



Transformation: Value

Columns:
Avg Age at Death

	Avg Age at Death
	16.17

Example 13

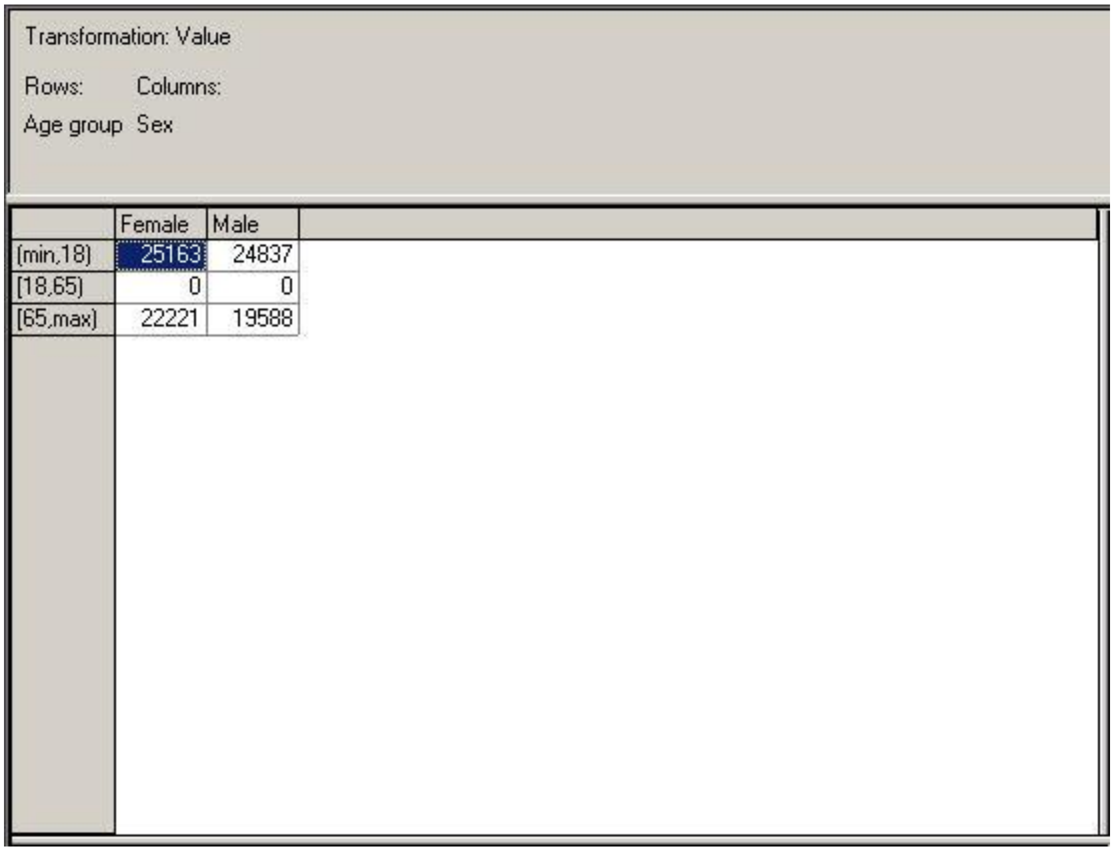
In this example, the table contains the results for persons who were ever under 18 and over 65. The age breakdown is determined by the partition AGE_GROUPXS, which would have been defined earlier in the model code via the statement:

```
partition AGE_GROUPXS
{
      18, 65
};
```

Table Specification

```
table Person      //EN persons younger than 18 or older than 65
[age<18 || age >=65]
{
      {
            unit //EN persons
      }
      * split (age,AGE_GROUPXS)*sex
};
```

Table Result


Transformation: Value

Rows: Age group

Columns: Sex

	Female	Male
(min,18)	25163	24837
[18,65)	0	0
[65,max)	22221	19588

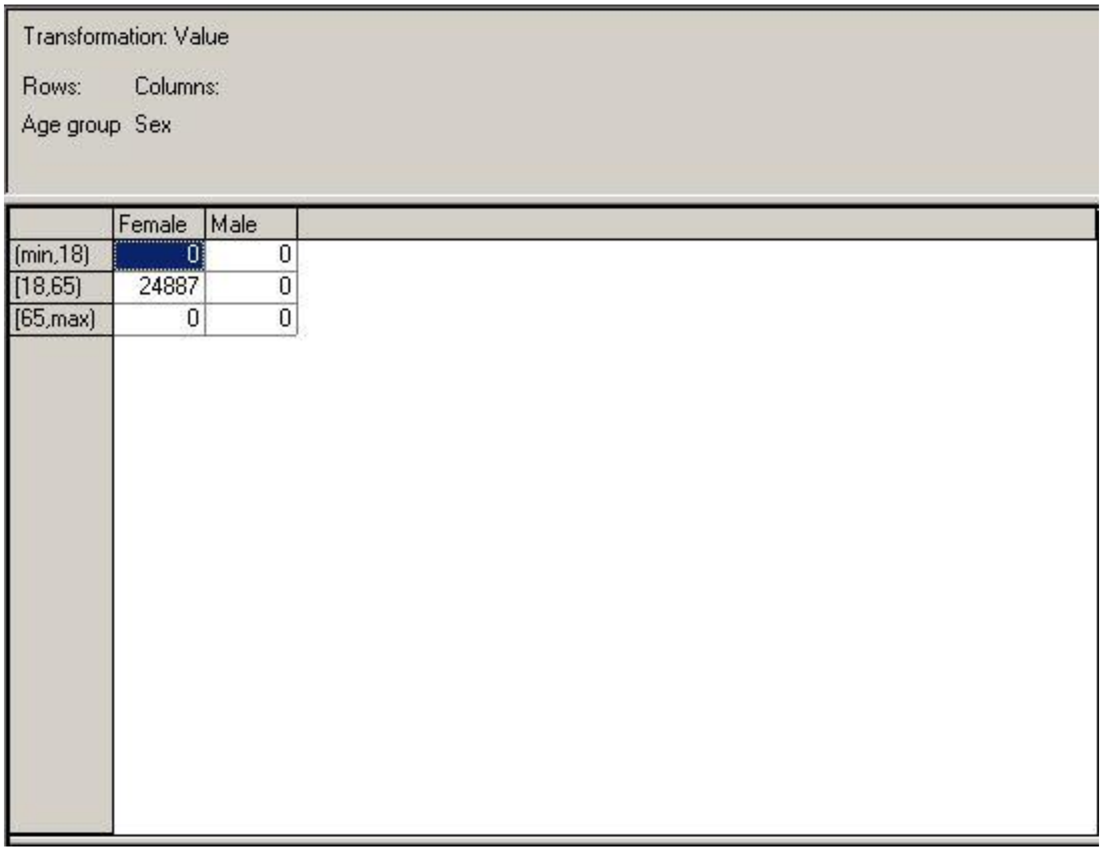
Example 14

In this example, the table is restricted to females who were ever older than 18 and younger than 65.

Table Specification

```
table Person      //EN females older than 18 younger than 65

[sex==FEMALE && (age > 18 && age <=64) ]

{

      {

            unit //EN persons

      }

      * split (age,AGE_GROUPXS)*sex


};
```

Table Result


Transformation: Value

Rows: Age group

Columns: Sex

	Female	Male
(min,18)	0	0
[18,65)	24887	0
[65,max)	0	0

Hiding tables and establishing table dependency

The function **hide()** is used to prevent tables from appearing in the visual interface to a Modgen model, while the function **dependency()** allows a dependency relationship to be established between two tables or between one and several other tables.

<u>Syntax of the hide() and dependency() functions:</u>

```
hide(table_name, <table_name, table_name, …>);

dependency(table_name1, table_name2, <table_name3, ... >);
```

The arguments for **hide()** are the names of the table(s) or table group(s) that are to be hidden, with each table name or table group name separated by a comma. (Note that the same function, **hide(),** is also used to hide parameters in the visual interface; however, parameters and tables cannot be combined as arguments.) Using the name of a table group as an argument is equivalent to using the name of each member of the group. User table names (defined in the next section) are also valid choices for arguments.

The **dependency()** function establishes a dependency relationship of the first named table upon the remaining tables specified in the arguments; that is, the first named table depends on the remaining tables. The first named table can be either a regular table or a user table, but not a table group. The other arguments can be regular tables, user tables, or table groups. (The use of a table group is equivalent to indicating each of the table group's members.)

One common reason to establish a dependency relationship is if your dependent table (which is often a user table) depends on another table that is occasionally hidden during some simulation runs. Hidden tables are normally not calculated at all during a simulation run. However, when a dependency relationship is established, any hidden table from such a relationship will in fact be calculated so that each table depending on a hidden table can in turn also be calculated.

The following rules apply to both the **hide()** and **dependency()** functions.

- If a table is hidden and no other tables (included) depend on it, then it is not included in the simulation results.
- If a table (A) is hidden, but another table (B) is included and depends on A, then A will be included (that is, calculated) but will not be shown in the list of tables in the model's visual interface, nor in its output database; it will also be impossible to export A. However, if A is still hidden but you later remove B from the list of tables included in the simulation, then A will no longer be included.
- A table group is shown in the model's visual interface only if it contains at least one table that is not hidden.

Tabulation of simultaneous events

If two or more events are in fact simultaneous events, they are treated as distinct during tabulation, rather than being "amalgamated" into a single tabulation "super-event" in which tables would only be updated immediately before the time changed, rather than after each event. This means that tables are updated before the next event starts, regardless of the time. Thus, if two events occur simultaneously, the tables are updated twice.

Note that tables are insensitive to "intermediate" changes that occur in the value of a state during the implementation of an event. Only the values of states at the end of the implementation of an event have an effect on tables.

User tables

User tables are tables built from other tables defined in .mpp files. User tables are more powerful than spreadsheet calculations done on the regular output tables because they can also produce standard errors and coefficients of variation, which are not possible to obtain by simply working on the output tables themselves.

A user table definition is similar to a regular table definition, but there are a few differences. First, the command is called **user_table**, not **table**. The user table must also always have a name; it cannot contain filters; it is not related to any actor; it uses the classification, range or partition data types as its classification dimensions; and it uses names for its table expressions. The table expressions can still contain **decimals** and/or **scale** specifications, as is allowed within table expressions in regular tables.

Summation in user tables

Modgen also allows dimension totals in user tables. These are requested by adding a '+' character after the dimension name (just as with regular tables). Unlike regular tables, however, the request for dimension totals only results in the creation of storage space (the extra cells) for the total values. The totals are not automatically computed as a result of adding a '+' character to the dimension name. This is because various quantities can be stored in the user tables and the process of simply adding them up doesn't always make sense (e.g. for ratios). Thus, you will need to compute the totals manually if they are more complicated than just simple sums.

However, if the totals are simply the sums of cell values, Modgen supplies the function **TotalSum()** , with the following prototype, to compute all dimension totals for a specific user table:

Syntax of the TotalSum() function:

void TotalSum(const char **szTableName*);

The proper way to use this function is to fill the non-total cells of the user table first and then call the function to compute the sums. The dimension totals will contain UNDEF_VALUE if there were no values found in the entire dimension; otherwise the null values are treated as zeroes for the calculation of totals.

Nulls and zeroes in user tables

A special Modgen constant called UNDEF_VALUE is available to all models. It is meant for user tables in which you want to indicate that some cells contain a null (no data) value as opposed to a proper zero. All user tables cells are initialized to the null value UNDEF_VALUE; thus, you do not need to assign the UNDEF_VALUE to cells that contain no observations.

If the user table is very large and very sparse (contains many zeroes), you can save space in your output database by adding the optional **sparse** keyword to the **user_table** command as follows:

Syntax of the user_table command with the sparse option:

user_table sparse table_name ….

If the table is marked as sparse, then the cells containing zeroes are not saved in the output database.

Sample user table declaration

Here is an example of a user table declaration:

```
user_table ALLPS_Grads    //EN test table (combines ALLPS and Grads)

{

      REP_SCHOOL_YEAR    //EN School year //FR Année scolaire
      * PS_EDLEV     //EN Level of study //FR Niveau de scolarité
      *
      {
            ENROL_F,  //EN Enrolments - female //FR Inscriptions - féminin
            GRAD_F,   //EN Graduates - female //FR Diplômés - féminin
            ENROL_M,  //EN Enrolments - male //FR Inscriptions - masculin
            GRAD_M    //EN Graduates - male //FR Diplômés - masculin
      }
};
```

Manipulating contents of user tables

To fill the cells of the user table, the model must use a function called **UserTables ()**:

Syntax of the UserTables() function:

void UserTables();

Modgen supplies two functions, **GetTableValue()** and **SetTableValue()** to manipulate the contents of the tables:

Syntax of the GetTableValue() and SetTableValue() functions:

double GetTableValue(CString *szExprName*<, …>);
void SetTableValue(CString *szExprName*, double *dValue* <, …>);

The number of optional arguments (indicated by …) equals the number of classification dimensions in the user table. Each argument (after the name of the user table in **GetTableValue()** or the data value *dValue* in **SetTableValue()**) represents the indexes of the table's classification dimensions (based on origin 0). Each argument is also separated with a comma, except that no comma follows the final argument. *szExprName* identifies both the table and the table expression; it consists of the table name followed by a period (".") and the expression name; e.g. "ALLPS.Expr0". The expressions of a regular table are always named "Expr0", "Expr1", etc., whereas the expressions of a user table are named within the user table declaration.

GetTableValue() can read a cell of a regular table or a user table but **SetTableValue()** can only set a cell of a user table (using *dValue* as the new cell value). Modgen calls **UserTables()** for each subsample so that standard errors and coefficients of variation can be calculated.

Modgen also supplies two functions, **GetUserTableReplicate()** and **GetUserTableSubSample()**, to enable you to know the subsample or replicate from which the **UserTables()** function was called.

Example of UserTables() function

Here is an example of the **UserTables()** function, taken from Statistics Canada's LifePaths model:

```
void UserTables()

{

      double dValue;

      int    nLevel;

      int    nYear;


      for ( nYear = 0; nYear < SIZE( REP_SCHOOL_YEAR ); nYear++ )

      {

            for ( nLevel = 0; nLevel < SIZE( PS_EDLEV ); nLevel++ )

            {

                  dValue = GetTableValue("ALLPS.Expr0", nYear, nLevel, FEMALE );

                  SetTableValue( "ALLPS_Grads.ENROL_F", dValue, nYear, nLevel );

                  dValue = GetTableValue( "ALLPS.Expr0", nYear, nLevel, MALE );

                  SetTableValue( "ALLPS_Grads.ENROL_M", dValue, nYear, nLevel );

            }

      }

      for ( nYear = 0; nYear < SIZE( REP_SCHOOL_YEAR ); nYear++ )

      {

            dValue = GetTableValue( "Grads.Expr1", nYear, FEMALE );

            SetTableValue( "ALLPS_Grads.GRAD_F", dValue, nYear, PSE_TVOC );

            dValue = GetTableValue( "Grads.Expr1", nYear, MALE );

            SetTableValue( "ALLPS_Grads.GRAD_M", dValue, nYear, PSE_TVOC );


            dValue = GetTableValue( "Grads.Expr2", nYear, FEMALE );

            SetTableValue( "ALLPS_Grads.GRAD_F", dValue, nYear, PSE_PI );
            dValue = GetTableValue( "Grads.Expr2", nYear, MALE );
            SetTableValue( "ALLPS_Grads.GRAD_M", dValue, nYear, PSE_PI );

            dValue = GetTableValue( "Grads.Expr3", nYear, FEMALE );
            SetTableValue( "ALLPS_Grads.GRAD_F", dValue, nYear, PSE_CC );
            dValue = GetTableValue( "Grads.Expr3", nYear, MALE );
            SetTableValue( "ALLPS_Grads.GRAD_M", dValue, nYear, PSE_CC );

            dValue = GetTableValue( "Grads.Expr4", nYear, FEMALE );
            SetTableValue( "ALLPS_Grads.GRAD_F", dValue, nYear, PSE_BA );
            dValue = GetTableValue( "Grads.Expr4", nYear, MALE );
            SetTableValue( "ALLPS_Grads.GRAD_M", dValue, nYear, PSE_BA );

            dValue = GetTableValue( "Grads.Expr5", nYear, FEMALE );
            SetTableValue( "ALLPS_Grads.GRAD_F", dValue, nYear, PSE_MA );
            dValue = GetTableValue( "Grads.Expr5", nYear, MALE );
            SetTableValue( "ALLPS_Grads.GRAD_M", dValue, nYear, PSE_MA );

            dValue = GetTableValue( "Grads.Expr6", nYear, FEMALE );
            SetTableValue( "ALLPS_Grads.GRAD_F", dValue, nYear, PSE_PHD );
            dValue = GetTableValue( "Grads.Expr6", nYear, MALE );
            SetTableValue( "ALLPS_Grads.GRAD_M", dValue, nYear, PSE_PHD );
        }
}
```

Computed tables

Computed tables represent a third different category of table that can be generated by Modgen. Unlike regular tables or user tables, however, computed tables are created after a model has completely executed. Since they are created after the completion of a model run, and because they are generated, modified, or removed through the visual interface to a Modgen model, the techniques for using computed tables are described in the publication "*Guide to the Modgen Visual Interface"*-- but as background, they are described briefly here.

Computed tables can be produced to calculate either run differences or run ratios. When computed tables are defined, the following entities need to be specified:

- New table name (which must consist of alphanumeric characters or underscores only)
- New table description
- Original table name (existing table name from the current Modgen output database, implying that the output database must be generated first)
- Reference database name (output database generated by a different scenario)
- Reference table name (existing table name from the reference database)
- Computation algorithm ("Table Difference" or "Table Ratio")

The original and reference tables must also satisfy a set of criteria before they can be compared:

- They must be generated with the same number of sub-samples (or replicates for time-based models).
- The ranks must be identical.
- The positions of the analysis dimensions must be identical.
- The shapes of all dimensions must be identical.

Typically, computed tables are used to calculate differences or ratios of the same table from two different runs, which thus makes it very straightforward to satisfy these criteria. The dimension labels of the original table are used in the resulting computed table.

Each computed table definition is stored in a Microsoft Access file named 'scenario_name(ctbl).mdb' (e.g. 'base(ctbl).mdb')

Interactions amongst actors—links and actor sets

Actor links: How model objects access each other's states

There are many instances in the course of a model simulation run when one actor must access a state of another actor. Typically, in C++, you would associate pointers between the data structures (or classes) to perform this task. However, Modgen provides links for this purpose; links are essentially pointers between model actors. These links must be defined symmetrically between the model actors so that Modgen can continually update the dependencies between states from different actors.

Modgen also allows for linked states to be used in the classification dimension of a table definition. In this way, the table generation facility can also make use of links.

Since link types are derivations of C++ pointers, the definition, initialization and syntax for links follow many of the C++ conventions for pointers.

Link definition

The links in a model may connect one actor to another, one actor to many other actors, or many actors to many other actors. In all instances, the reciprocal links are automatically generated and maintained by Modgen. This means that you only need to assign one side of the link. The link name must be unique within the actor.

Syntax for a one-to-one link

```
link            actor_name.link_name ;
```

OR

```
link            actor_name1.link_name1  // language_code  link_label_1

                actor_name2.link_name2;  // language_code link_label_2
```

A one-to-one link is a relationship between actors where each actor can be linked to at most one other actor. (The relationship does not have to be filled.) In the first definition, a link is specified between two actors of the same type (actor_name). In the second definition, actor_name1 must be different from actor_name2, and labels can be defined for the links. Conventionally, link name starts with a lower case 'l', indicating that this is a single link.

Examples

```
// In the first example, a single link between the main person actor

// and his/her spouse is established.

// In the second example, two single links are established, one between

// the main person actor and a spouse, the other between

// an ancillary actor and his/her spouse

link Person.lSpouse;   //EN Spouse

link Person.lSecSpouse   //EN Second-Marriage Spouse

      Ancillary.lSpouse;   //EN Spouse
```

Syntax for a one-to-many link:

```
link   actor1_name.link_name1   actor_name2.link_name2[];
or
link   actor1_name.link_name1[ ]  actor_name2.link_name2;
```

A one-to-many link is a relationship between actors where an actor can be linked to many actors but they in return are only linked to the one actor. Conventionally, the link name for the 'one' side of a one-to-many link starts with a lower case 'l', whereas the link name for the 'many' side of the link starts with 'ml', indicating that the link holds a collection of actors.

Example

```
// A link is established between the spouse and the children

link Person.lSpouse   Person.mlChildren[ ];
```

Syntax for a many-to-many link

link   actor1_name.link_name1[ ]   actor_name2.link_name2[];

A many-to-many link is a relationship between actors where each actor can be linked to many other actors.

Example

```
// A link is established between the children and the parents

link Person.mlChildren[ ]   Child.mlParents[ ];
```

Link initialization

The initialization of a single link follows the C++ convention for initializing pointers. However, Modgen relies on special functions for the initialization of multiple links. Remember that both the actor and the pointer must be initialized before the link itself can be initialized.

Syntax for a single link initialization:

link_name = prPointer_name;

Example

```
// A link is initialized for the spouse, where

// prNonDominantPerson is a pointer to a valid person actor

lSpouse = prNonDominantPerson;
```

Syntax for a multiple link initialization:

link_name->**Add**(*prPointer_name*);

Example

```
// A link is initialized for the children, where

// prChild is a pointer to a valid child actor.

// The Add function is explained in the section "Functions for adding and

// removing actors, and for indexing through the link"

mlChildren->Add(prChild);
```

Derived state specifications acting on multiple links

These derived state specifications allow you to generate statistics over the "many" side of a link. Note that these spécifications can only be used within the actor definition. For the "many" side, the syntax is one of the following:

Syntax of derived state specifications operating on the “many” side of a link:

multilink_function (link_name)

multilink_state_function (link_name, state_name)

There is only one possibility for multilink_function: **count()**, which returns the number of linked actors. However, multilink_state_function can be one of:

sum_over()	- sum of the requested state over all linked actors
min_over()	- minimum of the requested state over all linked actors
max_over()	- maximum of the requested state over all linked actors

These functions are described in more detail in the Links section of the chapter “Derived state specifications”.

Example 1 – the number of children at Person’s home

```
actor Person

{

      //EN Number of Children Present
      int   children_at_home =   count( mlChildrenAtHome );

};
```

Example 2 – the age of the youngest child in the Household

```
actor Household

{

      children = count ( parChild );

      age_youngest_child = min_over ( parChild, age );

};
```

Functions for adding and removing actors, and for indexing through the link

For each requested multiple link Modgen defines a special class which includes member functions for adding actors, removing actors and iterating through the link. The following functions are defined:

Add(*actor_pointer*)

This function adds an actor of the right type to the link.

void **FinishAll**()

This function calls the Finish function for all connected objects.

***GetNext**(int *nInitPos*, int **pnPos*)

The ***GetNext**() function returns a pointer to the next object available through the multiple link, starting the search at the position specified by *nInitPos*. The position of the found element is also returned. Note that the object connected through a multiple link may not occupy consecutive positions.

Remove(int *nPosition*)

This function removes the actor from the link at the specified position.

RemoveAll()

This function removes all actors from the link.

Example:

```
Child *GetNext( int nInitPos, int *pnPos );
```

Example model code using GetNext()

```
// Connect the dominant person to the partner and partner's children

lSpouse = prNonDominant;

if ( bFemale )

{

      prChild = prNonDominant->mlChildren->GetNext( 0, &nIndex );

      while ( prChild != NULL )

      {

            mlChildren->Add( prChild );

            children++;

            prChild = prNonDominant->mlChildren->GetNext( nIndex + 1, &nIndex );

      }

}
```

Actor sets

An actor set is a collection of actors dynamically maintained by Modgen. The expression ’dynamically maintained‘ means that Modgen internally handles group membership based on criteria or characteristics that you, the developer, provide. Modgen provides functions that you can use to select an actor randomly from the set, to iterate over all members of the set, etc.

An example of an actor set is a set of actors currently susceptible to infection by a particular disease. As the simulation progresses, actors would enter the set at birth or when they lose maternal immunity. They would leave the set through death or by acquiring immunity as a result of infection.

Actor sets are useful in time-based models with many interacting actors because they facilitate the development of such models by eliminating the need to write code to maintain and manipulate such groups of actors. Modgen dynamically maintains the actor groups, using efficient internal algorithms, and provides a core set of functions to efficiently perform operations on these groups.

Actor set attributes

The following is a list of attributes—some required, some optional—that you specify for an actor set in your code:

actor name

Required. If a model has more than one type of actor, only one of the actor types can be used for a given actor set.

actor set name

Required. This is a unique name that identifies the actor set and allows operations to be performed on the actor set through function calls.

dimensions

Optional. These refer to one or more states**,** each having a fixed number of values (of type classification, range, or partition) that subdivide the actor set into distinct subsets, e.g. by city.

filter

Optional. This is a logical expression that defines whether a given actor is a member of the group, e.g. "marital_status == SINGLE".

order *state* or **order RandomStream(***nStream****)***

Optional. When used with ***state***, the state's values are used to order the members of the actor set. Modgen automatically maintains this order as the simulation progresses. When **RandomStream()** is used instead of ***state***, Modgen generates an internal state that is used to order the members of the actor set randomly. If the value of *nStream* is missing, Modgen generates a unique integer value (as Modgen also does with the functions **RandLogistic()**, **RandUniform()** and **RandNormal()**).

Examples:

```
actor Person

{

      SEX sex;            //EN Sex

      CITY city;          //EN City of residence

};
```

```
actor_set Person asEveryone; //EN All Persons

actor_set Person asEveryoneByCity[city]; //EN All Persons, in subsets by city

actor_set Person asSusceptible //EN Persons susceptible to infection
filter disease_status != IMMUNE;
order RandomStream();

actor_set Person asEligible[city][sex] //EN Persons eligible for marriage
filter marital_status != MARRIED && integer_age >= 16;

//EN Persons eligible for marriage, ordered
actor_set Person asEligibleOrdered[city][sex]
filter marital_status != MARRIED && integer_age >= 16
order income;
```

Additional functions that operate with actor sets

The following functions can be called on an actor set at run time. (They are implemented as member functions of the actor set.) If the actor set is multidimensional (i.e. it is split into subsets), a given function call applies to a specified subset.

int **count()**

Returns the current number of actors in the actor set.

actor * Item(*index* **)**

Retrieves the actor located at position *index* in the group. Valid values for *index* range from 0 to **Count( )** - 1. If a floating point value is supplied for *index*, the fractional part is discarded. If an invalid value for *index* is specified, **Item( )** returns the value NULL. The value of *index* associated with a given actor may change as actors enter and leave the actor set. Actors are retrieved in the order specified in the **order** clause. If no **order** clause is specified, actors are returned in an unspecified order. To retrieve actors in a random order, call the **Scramble()** function before iterating over the actor set using **Item()**.

actor * GetRandom(double *dRandom* **)**

Returns a random actor from the actor set. The argument *dRandom* is a uniform random number, as returned by the Modgen function **RandUniform()**. If the actor set contains no members, a NULL value is returned. This function is exactly equivalent to the expression

```
as->Item((int)(dRandom * as->Count())
```

Scramble( )

Results in the actors in the actor set being returned in a new random order when **Item()** is called. **Scramble()** is available only for actor sets that are ordered randomly. When called, it assigns a new random order to each actor in the actor set.

Examples:

```
// Select a random susceptible person
Person *prSelected = asSusceptible->GetRandom(RandUniform());

// Number of susceptible people in the population
int nSusceptibles = asSusceptible->Count();

// Iterate over all susceptible people
for (int nJ=0; nJ<nSusceptibles; nJ++)
{
      Person *prSusceptible = asSusceptibles->Item(nJ);
      // ...
      // Do stuff with prSusceptible
      // ...
}


// Select ten different random susceptible people
// Note that this code works only on randomly ordered actor sets
 asSusceptibles->Scramble();
 for (int nJ=0; nJ<10; nJ++)
{
      Person *prSusceptible = asSusceptibles->Item(nJ);
      // ...
      // Do stuff with prSusceptible
  // ...
}


// Number of males eligible for marriage in Ottawa, using the
// "Ottawa" + "Male" subset of the multi-dimensional actor set asEligible
int nEligible = asEligible[OTTAWA][MALE]->Count();
```

If an actor set is used in a case-based model, it is initialized to empty at the beginning of each case.

If you have code iterating through an actor set using **Item()** and you change a state affecting the actor set (e.g. a state referenced in the actor set filter or the order clause) inside the loop, the loop may miss some actors, or process the same actor twice. This is because actor sets are modified instantaneously when associated states change. If you still wish to change such states inside a loop, an auxiliary actor set can be created to "hold" the actors to be changed. Here is a sketch showing how this can be achieved:

```
actor Person
{
	logical in_holding_area = { FALSE };
};
actor_set Person asHoldingArea
filter in_holding_area;

//EN Move all susceptibles to holding area
int nSusceptibles = asSusceptible->Count();
for (int nJ=0; nJ<nSusceptibles; nJ++)
{
	Person *prSusceptible = asSusceptibles->Item(nJ);
	prSusceptible->in_holding_area = TRUE;
}

// Iterate over the holding area, and empty it at the same time.
Person *prTemp = asHoldingArea->Item(0);
while ( NULL != prTemp)
{
	// Do stuff to Person
	// ...
	// Remove Person from the holding area
	prTemp->in_holding_area = FALSE;
	prTemp = asHoldingArea->Item(0);
}
```

Additional code examples that use actor sets

Example 1

```
// Find the richest eligible male using an ordered array
int   nCount = 0;
Person      *prRichMan = NULL;

// Get the number of eligible males in the city
nCount = asEligibleOrdered[city][MALE]->Count();

// Find the richest
prRichMan = asEligibleOrdered[city][MALE]->Item(nCount - 1);
```

Example 2

```
// Find the richest eligible male using a non-ordered array
int         nCount = 0;
int         nIndex = 0;
Person      *prRichMan = NULL;
Person      *prTemp = NULL;

// Get the number of eligible males in the city
nCount = asEligibleOrdered[city][MALE]->Count();

// Loop through to find the richest male
for (nIndex = 0; nIndex < nCount; nIndex++)
{
      prTemp = asEligibleOrdered[city][MALE]->Item(nIndex);
      if (prRichMan == NULL || prTemp->income > prRichMan->income)
      {
            prRichMan = prTemp;
      }
}
```

Example 3

```
// Infect the whole susceptible population
```

```
Person *prTemp = NULL;
prTemp = asSusceptible->Item(0);
while (prTemp != NULL)
{
      // Change the disease status, which will also remove the
      // person from the actor set.
      prTemp->disease_status = INFECTED;
      prTemp = asSusceptible->Item(0);
}
```

Example 4

```
// List the 100 companies with the biggest profits

actor_set Company asEveryone; //EN All Companies
order profits;
...
int         nCount = 0;
int         nIndex = 0;
Company     *prTemp = 0;

// Get the number of companies
nCount = asEveryone->Count();

// Get the 100 companies with the biggest profits
for (nIndex = 1; nIndex <= 100; nIndex++)
{
      prTemp = asEveryone->Item(nCount - nIndex);
      // ...
      // do stuff with prTemp
      // ...
}
```

Note that whenever you are trying to find the ‘x first’ or ‘x last’ entities, or the median value, you need to have an ordered actor set.

Example 5

```
// Find the province with the biggest median income for individuals

classification PROVINCE       //EN Province

{
      //EN Newfoundland
      NFLD,
      //EN P. E. I.
      PEI,
      //EN Nova Scotia
      NS,
      //EN New Brunswick
      NB,
      //EN Quebec
      QUE,
      //EN Ontario
      ONT,
      //EN Manitoba
      MAN,
      //EN Saskatchewan
      SASK,
      //EN Alberta
      ALTA,
      //EN British Columbia
      BC,
      //EN Yukon
      YUK,
      //EN N. W. T.
      NWT,

      //EN Nunavut
      NUN

};


actor Person

{
      PROVINCE    prov_of_res;
};

// All Persons, in subsets by province
actor_set Person asEveryoneByProvince[prov_of_res];
order income;

...

int         nCount = 0;
int         nIndex = 0;
int         nMedianIndex= 0 ;
double      dPreviousMedian = -1;
Person            *prTemp  = NULL;
PROVINCE    eProvince = NFLD;
PROVINCE    eLastProvince;

for (nIndex = 0; nIndex < SIZE(PROVINCE); nIndex++ )
{
      eProvince = (PROVINCE) nIndex;
      nCount = asEveryoneByProvince[eProvince]->Count();
      nMedianIndex = nCount / 2;

      prTemp = asEveryoneByProvince[eProvince]->Item(nMedianIndex);
      if (prTemp->income > dPreviousMedian)
      {
            eLastProvince = eProvince;
            dPreviousMedian = prTemp->income;
      }

      // ...
      // do stuff with eProvince
      // ...
}
```

Run-time functions

There are many run-time C++ functions that can be used anywhere in the code of a Modgen microsimulation model. One example is the function **sgn()** which takes an argument, *dValue,* of type double and returns an int result of the sign of *dValue,* i.e. -1, 0, or 1. Many of the other available run-time functions can be grouped or categorized by their general purpose, as outlined below.

Functions which modify the progress window

void **ModelExit**(const char **szMessage*);
This function is used to exit the model cleanly when errors are detected during the run. The contents of the string argument are added to the "Errors and Warnings" box in the Model Run Progress window when errors are detected, after which the model exits cleanly (i.e. with no output generated).

CString **ModelString**(CString *szName*);
ModelString may be used to output multilingual strings from the model. It returns the label associated with string *szName* in the language of the simulation. It is particularly useful with the ProgressMessage function.

The labels are defined for each language using Modgen's **string** command. This command is first followed by the name of the specific string, and then the actual label in the desired language.   (The example below establishes an English and French label for the string called S_GENERATING_SPOUSES. When this string is later used as an argument to **ModelString(),** the appropriate text will be displayed and added to the log file in the language being used during the simulation run.)
Example

```
string S_GENERATING_SPOUSES; //EN Generating spouses

string S_GENERATING_SPOUSES; //FR Génération des époux
```

…

```
sprintf( szMsg, "%s: %ld / %ld", ModelString( "S_GENERATING_SPOUSES" ),
lSpouseInd, SpouseMarketSize );

ProgressMessage( szMsg );
```

void **ProgressMessage**(*szMsg*);

This function is used to display a message in the "Progress" window when running a model, especially if there is a time-consuming activity happening within a case or at the beginning of the simulation; for example, **ProgressMessage**("Generating spouses: 20 / 5000");

void **TimeReport**(double *Time*);

This function updates the simulation progress bar for time-based models. The first call to the function **TimeReport()** is used only to specify the time at the beginning of the simulation, but this call to **TimeReport()** is mandatory; this first call should take place in the function **Simulation()** before the actual simulation loop. The progress bar uses the time given in the argument on that first call to **TimeReport()** to calculate the percentage of the simulation run that is completed.

WarningMsg(const **char*);

This function displays the warning message, contained as its argument, in the progress window.

Functions associated with the random number generator

void **GetRandState**(CRandState *rRandState*);

Returns by reference a complete set of seeds and deviates. CRandState is a Modgen type.

double **RandLogistic()**;

Returns a logistic deviate using the inverse cumulative density function, F(x) = 1/(1 + exp(-x)). You may use this function either with no argument or with a unique constant integer argument; however, if no argument is provided, Modgen actually alters the .mpp file by assigning an available integer as the argument.

double **RandNormal()**;

Returns a normal random deviate. You may use this function either with no argument or with a unique constant integer argument; however, if no argument is provided, Modgen actually alters the .mpp file by assigning an available integer as the argument.

double **RandUniform()**;

Returns a Uniform floating point random number in the semi-open range 0-1 (i.e. 0 can be returned as a result but never 1). You may use this function either with no argument or with a unique constant integer argument; however, if no argument is provided, Modgen actually alters the .mpp file by assigning an available integer as the argument.

void **SelectRegularRandomGenerator**();

Switches back to the random number generator assigned to the current case, after using one of the special generators **SelectSpecialRandomGenerator1()** or **SelectSpecialRandomGenerator2()**.

void **SelectSpecialRandomGenerator1**();

void **SelectSpecialRandomGenerator2**()**;**

Allows the model to use different random number generators for different activities (e.g. generation of spouses).

void **SetRandAuxState**(long *lRandSeed*);

Sets the auxiliary random generator seed to the new value specified as *lRandSeed*.

void **SetRandState**(CRandState *rRandState*);

Sets a complete set of seeds and deviates. CRandState is a Modgen type.

Functions associated with subsamples, replicates and the size of a simulation run

long **GetAllCases**();

Returns the number of requested cases.

long **GetCaseID();**

Returns the unique identifier of the case currently being simulated. The value returned is between 0 and the number of requested cases, minus 1 (inclusive). The set of values returned depends on the current sub-sample being simulated. For example, if the number of subsamples is 2, the values returned for subsample 0 will be {0, 2, 4, … }, and the values returned for subsample 1 will be {1, 3, 5, 7, …}. For a given sub-sample, the values will always increase, as illustrated by the example.

int **GetCaseSample**();

Returns the sub-sample number associated with the current case.

int **GetDistributedSubSamples**();

Returns the number of subsamples that will be simulated on the current machine. If the simulation is not distributed across different machines, that number is the total number of subsamples specified in the scenario settings of the visual interface. If the simulation is distributed, that number is the number of subsamples per machine, again as specified in the scenario settings of the visual interface.

int **GetFirstSample**();

Returns the number of the first subsample simulated on the machine. Subsamples are numbered starting with 0. If the simulation is not distributed across different machines, the number of the first subsample is always 0. If the simulation is distributed, the number is equal to the slave number multiplied by the number of subsamples per machine. (For example, if there are 8 subsamples per machine, the number of the first subsample simulated by the first slave (slave 0) is 0, the number of the first subsample simulated by the second slave (slave 1) is 8, the number of the first subsample simulated by the third slave (slave 2) is 16, etc.)

int **GetReplicate**();

Returns the number of replicates for a time-based model.

int **GetSubSamples**();

Returns the number of subsamples for a case-based model.

long **GetSubSampleCases**(int *nSample*);

Returns the number of cases in the specified subsample.

int **GetThreadNumber**();

Returns the thread number of the current simulation thread, in origin 0, i.e. the first thread will return 0, the second thread will return 1 etc. (A thread is a series of instructions that are executed on a given machine. Different threads can run at the same time and the number of threads is specified in Modgen's visual interface before the simulation run, although the number of threads should not exceed the number of processors on the machine.)

int **GetThreads**();

Returns the number of simulation threads.

void **RestoreSubSampleSeeds**();

Restores the starting seeds in all subsamples

The following two associated macros are also available:

CASES()

Returns the total number of cases to be simulated by the current thread (when a case-based model is running).

SIMULATION_END()

Returns the time when the simulation of a time-based model ends. (The time starts at time 0)

The following two associated constants are also available:

MAX_SUBSAMPLES

This constant is the maximum number of subsamples in a case-based model or replicates in a time-based model that can be used by the model (currently 41).

MAX_SUBSAMPLES_PLUS1

This constant, equal to MAX_SUBSAMPLES+1, is useful for creating subsample totals in a case-based model.

Functions used for case weighting and population scaling

Modgen allows the use of case weights for tabulation purposes. Case weights are used to scale all additive tabulated quantities (including **unit**) but not the “min” and “max” type quantities (such as **min_value_in**, or **max_delta**). They are also used for population scaling (the sum of case weights is used instead of the number of processed cases). Each case starts with a default weight of 1, which remains in effect until the **SetCaseWeight()** function is called. Case weights have no utility in time-based models.

A special state called **actor_weight** is available for every actor. A call to the **SetCaseWeight()** function also sets the value of the **actor_weight** state for every actor in the case. The **actor_weight** state may be tracked and displayed with the BioBrowser application.

Another special state called **actor_subsample_weight** can also be tracked by BioBrowser. This state is equal to **actor_weight** multiplied by the number of sub-samples. This state indicates the weight applied to sub-sample quantities (**actor_weight** is applied to total quantities).

void **GetCaseWeight**(double **pdCaseWeight*, double **pdCaseSubsampleWeight*);
Returns the case weight and case sub-sample weight through pointer arguments.

long **GetPopulation**();
Returns the target population size used for population scaling.

bool **GetPopulationScalingRequired**();
Returns TRUE if the user has checked the box “Population Scaling” on the scenario settings dialog box of the visual interface, and FALSE otherwise.

void **NoteCaseWeight**(double *dCaseWeight* <, double *dSubsampleWeight* >);
Notes the case weight and subsample weight. The case weight used for population scaling is determined in the following order:

1. the last weight supplied by **NoteCaseWeight()**
2. the weight in effect at the end of the case (if **NoteCaseWeight()** was not previously called)

The subsample weight (second argument) is optional; if it is not used then the subsample weight noted will be equal to the case weight multiplied by the number of subsamples, which is appropriate for most cases. However if the subsample weight is assigned a non-default value and cloning is used, **NoteCaseWeight()** should then be called with two arguments.

void **SetCaseWeight**(double *dCaseWeight* <, double *dSubsampleWeight* >);
Sets the case weight and case subsample weight.

long **SetPopulation**(long *lValue*);

Seta a default value (*lValue)*for the total population which is then used to scale tabulation results. This value is used as long as the "Population scaling" field is checked AND the "Population value" field has a value of 0, both within the scenario settings via the visual interface for the model. If these two fields are used in this way but **SetPopulation()** is not used in the model code, the population size is instead set to the number of cases being processed.

Functions used in secondary loops of events

The following functions are useful when implementing secondary loops of events, such as in the spouse generation portion of Statistics Canada's LifePaths model. Together, these functions can be used to determine if the next event is to be implemented in the secondary loop or if it is time to stop simulating events in the secondary loop and return to the main loop of events

event *EventQueue::**GetCurrentEvent()**

Allows the model to peek at the event being currently implemented. The result is a pointer to the event that is retrieved.

event *EventQueue::**GetNextEvent()**

Allows the model to peek at the next event in the queue (but without actually implementing that next event). The result is a pointer to the event that is retrieved.

void EventQueue::**WaitUntilThisActor(**double *dTime,* ActorClass **poThisActor***)**

Allows Modgen to advance the time for only one actor in models without the **just_in_time** algorithm. Such a function should be used only in secondary loops of events. Advancing time for all actors in secondary loops can lead in general to inconsistencies in tabulation; this function helps you avoid such problems.

Functions associated with date and time

Modgen supplies a number of functions that assist in the handling of sub-annual calendar time. These functions use three different ways of expressing time, referred to below as "**model time**", "**day number**" and "**calendar**". Most of the functions in this section allow users to convert from one way of expressing time to another.

"**Model time**" is time expressed as a continuous real number where a "year" has a value of 1.0.

"**Day number**" is an integer counter of days. It uniquely identifies and sequentially counts elapsed days, starting from a fixed day in the distant past.

"**Calendar time**" re-expresses the current day into the corresponding year, month and day of the month.

There are two different calendars which have been incorporated in these functions. The first is the "Gregorian Standard Year" which has 365.2524 days, i.e. the average length of a year in the Gregorian calendar used today. (Gregorian calendar years themselves vary in length between either 365 or 366 days, making them unsuitable as a consistent unit of measurement.) The other calendar is the Leapless calendar. It has a constant year length of 365 days, which means that February 29th never occurs in this calendar. Both calendars are split into years, months and days of the month. The day of the week (e.g. Monday to Sunday) is also provided.

Note that in the Gregorian calendar, the integer part of the model time is not necessarily the associated calendar year, although it will always be very close. For example, January 1, 1988 is day number 2447162, with an associated model time of 1987.9997535884.

In the functions described below, note that month of the year, day of the month, and day of the week are expressed using 0 as the origin. In other words, January is month 0 of the year and December is month 11. Similarly, Monday is day 0 and Sunday is day 6. The first day of each month is numbered 0.

Certain other date-related quantities can be derived using these functions. For example, a continuously increasing counter of weeks can be constructed with an expression like "nDay/ 7". A continuously increasing counter of months can be constructed with an expression like"nYear * 12 + nMonthOfYear".

Gregorian functions

double **GregorianDayLength** ();

Returns the length of a day in "Gregorian Standard Years". This is the reciprocal of 365.2524, appropriately rounded for precision.

int **GregorianCalendarToDay** (int *nYear*, int *nMonthOfYear*, int *nDayOfMonth*);

Returns the day number associated with a given date (four-digit year, month of year [0-11] and day of month [0-30]) on the Gregorian calendar.

double **GregorianCalendarToTime** (int *nYear*, int *nMonthOfYear*, int *nDayOfMonth*);

Returns the model time associated with a given date (four-digit year, month of year [0-11] and day of month [0-30]) on the Gregorian calendar.

double **GregorianDayToTime** (int *nDay*);

Converts a day number, *nDay*, to the corresponding model time.

bool **GregorianIsLeapYear** (int *nYear*);

Given a four-digit Gregorian calendar year, returns TRUE if the year is a leap year.

int **GregorianTimeToDay** (double *dTime* <, **pnIsDayBoundary*>);
Converts a model time, *dTime*, to a day number. Any fractional part of the day is ignored, i.e. the time is rounded down to the nearest day. The second argument, which is optional, returns a flag which is set to TRUE if *dTime* corresponds to

an exact day boundary; otherwise the flag is set to FALSE.

void **GregorianDayToCalendar** (int *nDay*, int **pnYear*, int **pnMonthOfYear*, int **pnDayOfMonth*, int **pnDayOfWeek*);

Given a day number *nDay*, returns the corresponding year, month of year [0-11], day of month [0-30], and day of week [0-6] in the Gregorian calendar.

void **GregorianTimeToCalendar** (double *dTime*, int **pnYear*, int **pnMonthOfYear*, int **pnDayOfMonth*, int **pnDayOfWeek*);

Given a model time *dTime*, returns the corresponding year, month of year [0-11], day of month [0-30], and day of week [0-6] in the Gregorian calendar.

Leapless functions

double **LeaplessDayLength** ();

Returns the length of a day in "Leapless Standard Years". This is the reciprocal of 365, appropriately rounded for precision.

int **LeaplessCalendarToDay** (int *nYear*, int *nMonthOfYear*, int *nDayOfMonth*);

Returns the day number associated with a given date (four-digit year, month of year [0-11] and day of month [0-30]) on the Leapless calendar.

double **LeaplessCalendarToTime** (int *nYear*, int *nMonthOfYear*, int *nDayOfMonth*);

Returns the model time associated with a given date (four-digit year, month of year [0-11] and day of month [0-30]) on the Leapless calendar.

double **LeaplessDayToTime** (int *nDay*);

Converts a day number, *nDay*, to the corresponding model time.

bool **LeaplessIsLeapYear** (int *nYear*);

Always returns TRUE. This function is only present so that the "Leapless" family of functions contains an analogous function for each member of the Gregorian family of functions.

int **LeaplessTimeToDay** (double *dTime* <, **pnIsDayBoundary*>);

Converts a model time, *dTime*, to a day number. Any fractional part of the day is ignored, i.e. the time is rounded down to the nearest day. The second argument, which is optional, returns a flag which is set to TRUE if *dTime* corresponds to an exact day boundary; otherwise the flag is set to FALSE.

void **LeaplessDayToCalendar** (int *nDay*, int **pnYear*, int **pnMonthOfYear*, int **pnDayOfMonth*, int **pnDayOfWeek*);

Given a day number, *nDay*, returns the corresponding year, month of year [0-11], day of month [0-30], and day of week [0-6] in the Leapless calendar.

void **LeaplessTimeToCalendar** (double *dTime*, int **pnYear*, int **pnMonthOfYear*, int **pnDayOfMonth*, int **pnDayOfWeek*);

Given a model time, *dTime*, returns the corresponding year, month of year [0-11], day of month [0-30], and day of week [0-6] in the Leapless calendar.

Functions used for transformations between integers and classifications

Modgen has functions that facilitate the use of classifications which implicitly map to integer values. An example of such a classification is a selected subset of years or a subset of ages.

For each classification having at least one level whose name ends in a number, Modgen will create three functions. These functions are named **IntIs_xxx**(*integer*), **IntTo_xxx**(*integer*), and **IntFrom_xxx**(*level*), where "xxx" is the name of the classification. For the purposes of these functions, each level of the classification is associated with a numeric value corresponding to the trailing numeric digits of the level's name. For example, the classification level YR_1991 would be associated with the integer 1991. The function **IntIs_xxx**(*integer*) returns TRUE if *integer* is found among the levels of classification xxx. The function **IntTo_xxx**(*integer*) will return the level of classification xxx that corresponds to *integer*. If *integer* is not found in classification xxx, **IntTo_xxx**(integer) will return the last level that has no associated numeric value (e.g. YR_NA). If no such non-numeric level exists in classification xxx, the last level of the classification is returned. If multiple levels share the same integer value, the first such level is returned. The **IntTo_xxx**(*integer*) function returns a value of type xxx, which means that no casting of the return value is normally necessary. Finally, the **IntFrom_xxx**(*level*) function converts a level from the classification xxx to the corresponding integer value. If the level does not have an associated numeric value (e.g. YR_NA), the value –1 is returned.

The use of these functions is most easily understood through examples. In the code that follows, note how the **IntIs_CENSUS_YEAR()** function is used in the table filter to restrict the table to report on Census years only, and how the **IntTo_CENSUS_YEAR()** function is used to map the integer state **year** to the classification **CENSUS_YEAR**.

Example of transformation functions between integers and classifications

```
classification CENSUS_YEAR    //EN Census Year

{

      YR1971, //EN 1971
      YR1976, //EN 1976
      YR1981, //EN 1981
      YR1986, //EN 1986
      YR1991, //EN 1991
      YR1996, //EN 1996
      YR2001, //EN 2001
      YR2006  //EN 2006
};

actor Person
{
      CENSUS_YEAR census_year = IntTo_CENSUS_YEAR( year ); //EN Census Year
};

table Person CENSUS_COUNTS //EN Census Counts
[ dominant && IntIs_CENSUS_YEAR( year ) ]
{
      census_year * { unit }
};
```

The following example shows how the **IntFrom_ xxx**() function can be used to generate a table using levels selected from a larger classification.

```
classification FOS      //EN Field of study
{
      FOS1, //EN Mathematics
      FOS2, //EN Economics
      FOS3, //EN Sociology
      FOS4, //EN Anthropology

      FOS5, //EN Psychology

      FOS6, //EN Kinesiology

      FOS7 //EN Ontology

};


classification FOS_SEL  //EN Field of study (selected)

{

      FOS_SEL2, //EN Economics

      FOS_SEL3, //EN Sociology

      FOS_SEL5 //EN Psychology

};


actor Person

{

      FOS fos; //EN Field of study

      FOS_SEL fos_sel = IntTo_FOS_SEL( IntFrom_FOS( fos ) ); //EN Field of study

};


table Person FOS_ALL //EN Graduates

[ dominant ]

{

      fos * { unit }

};


table Person FOS_SELECTED //EN Graduates

[ dominant && IntIs_FOS_SEL( IntFrom_FOS( fos ) ) ]
```

```
{
      fos_sel * { unit }
};
```

.

Technical topics in Modgen

Aggregations of classifications

The derived state specification **aggregate()** converts values in one classification to values in a second classification, provided that the second is a strict aggregation of the first. To perform this transformation, a map between the first and second classification states had to have been provided first in the mpp source code, using the **aggregation** command.

Syntax of the aggregation command:

```
aggregation aggregated_classification, detailed_classification {
    aggregated_item, detailed_item,
    aggregated_item, detailed_item,
    …
    aggregated_item, detailed_item
};
```

In this syntax, “aggregated_classification”, “detailed_classification”, “aggregated_item” and “detailed_item” refer to previously defined aggregated and detailed classifications and their items. Note that there is no comma after the final detailed_item.

If an aggregation has been defined from classification `xxx to` classification `yyy` and if ***`stateXXX`*** is a classification state of type `xxx` then

`aggregate``(`***`stateXXX,`***` yyy)`

returns the corresponding state value from the classification yyy. Derived states using **aggregate()** can be used directly as dimensions of tables, in expressions, and in functions that you write as the model developer.

Modgen also creates functions that can be used to map one classification to another. For every pair of classifications xxx and yyy for which an aggregation has been defined, Modgen creates a function, **XXX_To_YYY(eLevel),** that takes a level of classification xxx as its argument and returns a level of classification yyy. These functions can be used in **PreSimulation()** and **UserTables()** as well as in expressions and in actor functions, but cannot be used as dimensions of tables.

Example of coding an aggregation of classifications

The following example illustrates how aggregations of classifications can be made to work. Sample results are shown after the code.

```
classification PROVINCE //EN Province

{

      NFLD,  //EN Newfoundland

      PEI,   //EN P. E. I.

      NS,    //EN Nova Scotia

      NB,    //EN New Brunswick

      QUE,   //EN Quebec

      ONT,   //EN Ontario

      MAN,   //EN Manitoba

      SASK,  //EN Saskatchewan

      ALTA,  //EN Alberta

      BC,    //EN British Columbia

      YUK,   //EN Yukon

      NWT    //EN N. W. T.

};


classification REGION   //EN Region

{

      R_ATLANTIC,  //EN Atlantic

      R_QUEBEC,    //EN Quebec

      R_ONTARIO,   //EN Ontario
      R_PRAIRIES,  //EN Prairies
      R_ALBERTA,   //EN Alberta,
      R_BC         //EN British Columbia
};

aggregation REGION, PROVINCE  //EN Province to region
{
      R_ATLANTIC, NFLD,
      R_ATLANTIC, PEI,
      R_ATLANTIC, NS,
      R_ATLANTIC, NB,
      R_QUEBEC,   QUE,
      R_ONTARIO,  ONT,
      R_PRAIRIES, MAN,
      R_PRAIRIES, SASK,
      R_ALBERTA,  ALTA,
      R_BC,       BC,
      R_BC,       YUK,
      R_ALBERTA,  NWT
};

actor Person
{
      PROVINCE province;      //EN Province
      REGION region = aggregate( province, REGION ); //EN Region
      REGION region2 = PROVINCE_To_REGION( province ); //EN Region
};
```

```
table Person TB_PROV
[ dominant ]
{
      province+
      * {
            duration()
      }

};

table Person TB_REGION
[ dominant ]
{
      aggregate(province,REGION)+  //EN Region
      * {
            duration()
      }

};
```

The two resulting tables from a test model are shown below.

```
Table: TB_PROV
```

Province	**Duration()**
Newfoundland	6.0

P. E. I.	6.0
Nova Scotia	6.0
New Brunswick	6.5
Quebec	7.0
Ontario	7.5
Manitoba	7.5
Saskatchewan	7.5
Alberta	7.5
British Columbia	7.0
Yukon	6.5
N. W. T.	6.0
All	81.0

```
Table: TB_REGION
```

Region	Duration()
Atlantic	24.5
Quebec	7.0
Ontario	7.5
Prairies	15.0
Alberta	13.5
British Columbia	13.5
All	81.0

Declaring and using function hooks

Modgen allows for a single actor function to be split amongst more than one file by providing function hooks. This is especially useful for certain standard functions common to all model type such as **Start()** and **Finish()**. Different modules of a model may need to add some code to such functions; they can do so by defining function hooks. A function hook is an actor function that takes no parameters and return no result. The prototype and definition of such functions is similar to the other actor functions. One additional definition (the hook definition) is required inside an actor definition. The syntax is as follows.

<u>Syntax of the hook definition:</u>

hook *HookFrom* , *HookTo* <,*Priority*>;

where *HookFrom* must be an actor function which takes no arguments and returns no result, and *HookTo* can be any actor function or event implementation function. The hook can optionally be given a priority to ensure it is executed before other hooks. The Modgen pre-compiler emits a warning if the hook order is ambiguous with respect to *HookTo*.

A function can be hooked to more than one other function, and more than one function can be hooked to the same function. The following is an example from Statistics Canada's LifePaths model.
Example

```
actor Person
{
...
      void    StartEducation();
      hook    StartEducation, Start;
...
};


void Person::StartEducation()
{
      prov_of_study = (PROV_OF_STUDY) min( prov_of_res, SIZE( PROV_OF_STUDY )-1
);
      school_year_start = next_year + (TIME) ( 5 + 8.0 / 12.0 );
      school_year_end = school_year_start + (TIME) ( 10.0 / 12.0 );
}
```

If the function to which other functions are hooked has a preferred place where all hooks should be inserted, then it should include the following instruction in that preferred place:

IMPLEMENT_HOOK();

A function can only have at most one instance of **IMPLEMENT_HOOK()**. If **IMPLEMENT_HOOK()** is not used, then all hooks are inserted at the end of the function.

The order in which the hooks are implemented is controlled by the *Priority* given in the hook declaration, whether or not **IMPLEMENT_HOOK()** is used. The Modgen pre-compiler will display the order in case of ambiguity, to allow ordering them explicitly.

Controlling precision in continuous time models

Some anomalies can occur within continuous time models due to the precision of time and rounding errors. Modgen provides two functions that control the precision of floating point calculations--.**SetMaxTime()** and **CoarsenMantissa().**

Using SetMaxTime() to control precision

You may estimate the maximal value for the simulated time, the **time** state, (after which Modgen will take care of rounding all event times) plus the **age** state to the appropriate precision. This approach eliminates certain observed anomalies e.g. the value of **age** at the person's 61st birthday will be exactly 61 and not something like 60.999999999997. The function to estimate this maximal time value is **SetMaxTime()**.

Syntax of the SetMaxTime() function:

void **SetMaxTime**(double *dMaxValue*);

In a case-based model, an ideal location in which to specify this time value is the **PreSimulation**() function, which is called only once for each case before the simulation starts. As an example, the following line could be added to the **PreSimulation()** function (where MaxLife is a parameter containing the maximum life expectancy):

```
SetMaxTime( MAX( YEAR ) + MaxLife );
```

Using CoarsenMantissa() to control precision

There are still other situations where the precision control of floating point variables may be desired to ensure that arithmetic operations on **time** and **age** are performed consistently. To handle such situations, Modgen supplies another function, **CoarsenMantissa()**.

Syntax of the CoarsenMantissa() function:

double **CoarsenMantissa**(double *dValue* <, double *dMaxValue*>);

CoarsenMantissa() returns the rounded value, based on the maximal absolute value for *dvalue*. The second argument (*dMaxValue)* is optional; if it is not supplied, the value set by the **SetMaxTime()** function is used.

CoarsenMantissa() uses the value specified in a previous call to **SetMaxTime()** to deliberately limit the precision of all waiting times, as well as the starting values of **time** and **age**. In effect, internal operations related to **time** and **age** are performed at fixed precision rather than floating precision, even if **time** and **age** are floating-point quantities.

Associated with this fixed precision is a minimum indivisible atomic unit of time. The value of this minimum indivisible atomic unit of time can be retrieved by calling the function **GetMinimumTimeIncrement()**. The value depends on the argument supplied to a previous call to **SetMaxTime()**, and also depends on whether **time_type** is float or double.

CoarsenMantissa() and **GetMinimumTimeIncrement()** each return a result of type double in models where **time_type** is declared as double, whereas they each return a result of type float when **time_type** is declared as float in the model. If **SetMaxTime()** has not been previously called, **CoarsenMantissa()** returns its argument unchanged and **GetMinimumTimeIncrement()** returns zero, because there is no minimum indivisible atomic unit of time.

Example: (from Statistics Canada's LifePaths model):

```
birth = CoarsenMantissa( birth );
```

Independent random number streams in Modgen models

Modgen uses independent random number streams in order to eliminate spurious 'noise' and reduce the variance of inter-run differences.

When Modgen scans the mpp files, it looks for occurrences of **RandUniform(), RandLogistic()** and **RandNormal()** function calls without arguments and assigns a unique index as an argument to those functions. The changes are written back to the mpp files before the corresponding cpp files are generated. The arguments added by Modgen to **RandUniform()**, **RandLogistic()** and **RandNormal()** calls are used internally to ensure the independence of all random number streams used by the model.

For example, assuming the next available unique index was 3, the following piece of code:

```
      sex = ( RandUniform() <= FemaleProp ) ? FEMALE : MALE;
```

would be changed in the .mpp file by Modgen to:

```
      sex = ( RandUniform(3) <= FemaleProp ) ? FEMALE : MALE;
```

When you write new code (e.g. adding a new equation or a new event), all calls to **RandUniform()** , **RandLogistic()** and/or **RandNormal()** should have no arguments; Modgen will assign them at parse time. Modgen does not modify

RandUniform(), **RandLogistic()** and **RandNormal()** calls that already have a numeric argument if duplicate arguments are detected (e.g. as a result of having copied a block of code containing indexed **RandUniform()** Modgen will issue a warning message. To eliminate the warning and obtain unique numeric arguments for all streams, edit the model source, delete one of the duplicate arguments and recompile.

The random number generator needs to remember the seeds and deviates of all random number streams used in a model. To simplify the use of the random number generator state, a Modgen-defined class called CRandState should be used. Variables of type CRandState can be declared in any function of the model and passed as arguments to the **GetRandState()** and **SetRandState()** functions. The class also supports the assignment of one CRandState to another.

<u>Syntax of the GetRandState() and SetRandState() functions (to get and set the complete set of seeds and deviates:</u>

void **GetRandState**(CRandState *rRandState*);

void **SetRandState**(CRandState *rRandState*);

Example (from Statistics Canada’s LifePaths model):

```
void CaseSimulation()
{
	CRandState	rRandState;
	int		nCloneNumber = 0;
	int		nClonesRequested = 0;

	while ( nCloneNumber <= nClonesRequested )
	{
		if ( nCloneNumber == 0 )
		{
			GetRandState( rRandState );
		}
		else
		{SetRandState( rRandState );
		}
		…


	}
	…
}
```

Cloning observations in case-based models: A special technique

The purpose of cloning is to obtain a bigger count of an interesting population (only for a case-based model) and then apply different algorithms (e.g. interventions for lung cancer patients in Statistics Canada’s Pohem model) on different clones of the same object. Cloning is not part of Modgen but it can be easily implemented in Modgen-generated models.

Here is the code required to implement cloning in both the LifePaths and Pohem models developed by Statistics Canada. First, the following two states are added to the Person actor:

```
int CloneNumber;        //EN clone number

int ClonesRequested;    //EN requested number of clones
```

CloneNumber is assigned only once (at the beginning of the Person actor’s life, shortly after the **Start()** function is called). The **ClonesRequested** state is initialized to 0 and incremented during the simulation if some event of interest occurs.

The following code fragment of **CaseSimulation()** takes care of cloning:

```
void CaseSimulation()

{

      …

      CRandState  rRandState

      int         nCloneNumber = {0};

      int         nClonesRequested = {0};

      Person      *prDominant = {NULL};

      …

      while ( nCloneNumber <= nClonesRequested )

      {

            if ( nCloneNumber == 0 )

            {

                  GetRandState( rRandState );
            }
            else
            {
                  SetRandState( rRandState );
            }
            …
            prDominant = new Person();
            prDominant->Start( … );
            // nCloneNumber and nClonesRequested are arguments to Start()
            …
            // event loop for current case
            …

            nCloneNumber = prDominant->CloneNumber + 1;
            nClonesRequested = prDominant->ClonesRequested;
      }
      …
}
```

At the beginning of each simulated case **CloneNumber** is set to 0 and **ClonesRequested** is set to 0. If model code sets **ClonesRequested**, for the dominant actor, to a non-zero value then the current case is re-simulated, using the same starting seed. This re-simulation increments **CloneNumber**. Model code and tables can be made sensitive to **CloneNumber** in order to generate different case-specific trials and report on their effects.

Controlling pre-compiled models from outside applications

Pre-compiled Modgen models can be controlled, and their inputs and outputs manipulated, by outside applications. This controllability can be used, for example, to implement tracking techniques to perform model calibration. In this case, a controlling application runs the model in a loop, searching the model's parameter space in order to match historical control totals.

Modgen as an in-process automation server

Since one of the most difficult tasks for such an application is the reading (parsing) and writing of the .dat parameter files, this functionality has been provided as an in-process automation server within the ModAuto12.dll library. This library must be registered before the server can be used. The Modgen setup program performs the necessary registration.

The programmatic identifier of the server is “Modgen12.AutoParser”. Models should use this identifier to create the server object. Use this identifier for late binding of the server.
e.g. Set objParser = CreateObject("Modgen12.AutoParser") 'late binding

For more information, refer to the OLE documentation within your development environment.

The type library file ‘ModAuto12.tlb’ is also distributed with the Modgen setup. Applications may use this file with object browsers to access information about the exposed methods.

Methods exposed by the parsing server

All methods return a result code (0 means success). Error codes and their descriptions are found in the documentation database generated by the model.

[id(1)] short **Init**(BSTR *szDocDb*);
Initializes the automation server for a specific model using information stored in the documentation database file *szDocDb*.

[id(2)] short **ResetData**();
Resets all parameters. This method should be used if the data files need to be re-parsed.

[id(3)] short **ReadFiles**(BSTR *szFileSpec*);
Parses file(s) indicated by *szFileSpec* (wildcards allowed).

[id(4)] short **SaveFiles**(BSTR *szFileSpec*);
Saves the contents of files indicated by *szFileSpec*.

[id(5)] short **GetParameterCell**(BSTR *szName*, long *lCell*, VARIANT **pVariant*);
Obtains the data value (as a string) for cell *lCell* (ravelled index in display order) for parameter indicated by *szName*. The value is returned as a string in *pVariant*.

[id(6)] short **SetParameterCell**(BSTR *szName*, long *lCell*, BSTR *szData*);
Sets the data value (as a string) for cell *lCell* (raveled index in display order) for parameter indicated by *szName*. *szData* contains the new value.

[id(7)] short **GetStoredParameterSlices**(BSTR *szName*, long **plSlices*);

Gets the number of first dimension slices stored for an extendible parameter indicated by *szName*.

[id(8)] short **SetStoredParameterSlices**(BSTR *szName*, long *lSlices*);

Updates the number of first dimension slices stored for an extendible parameter indicated by *szName*.

[id(9)] short **ParameterUpdateRequired**(BSTR *szName*, short **pnUpdate*);

Checks if an extendible parameter needs an update. The result is returned in the variable pointed to by *pnUpdate* (1 if an update is required, 0 otherwise).

[id(10)] short **GetFileNote**(BSTR *szFile*, short *nLang*, VARIANT **pVariant*);

Gets the note associated with the file indicated by *szFile* in language *nLang* (0-based language index).

[id(11)] short **SetFileNote**(BSTR *szFile*, short *nLang*, BSTR *szNote*);

Sets the note associated with the file indicated by *szFile* in language *nLang* (0-based language index).

[id(12)] short **GetParameterNote**(BSTR *szName*, short *nLang*, VARIANT **pVariant*);

Gets the note associated with the parameter indicated by *szName* in language *nLang* (0-based language index).

[id(13)] short **SetParameterNote**(BSTR *szName*, short *nLang*, BSTR *szNote*);

Sets the note associated with the parameter indicated by *szName* in language *nLang* (0-based language index).

[id(14)] short **ChangeFileReference**(BSTR *szOldFile*, BSTR *szNewFile*);

This method should be used if a data file is parsed and then it needs to be saved under a different name. The server can only save files that were encountered by the parser. Each object (parameter, table exclusion, file note etc.) has a file reference attached to it so that the server knows which objects to save when saving a data file. If the "SaveFiles" method was used with a new file name, a blank file is created; however, the "ChangeFileReference" method solves this problem. It should be called just before the "SaveFiles" method. The “ChangeFileReference” method creates an empty file as a side effect; the "SaveFiles" method then adds contents to it. *szOldFile* is the original file parsed by the server, while *szNewFile* is the new file name.

[id(15)] short **GetParameterDecimals**(BSTR *szName*, short **pnDecimals*);

Gets the number of decimals stored for the parameter indicated by *szName* (a negative value in *pnDecimals* means that there is no limit).

[id(16)] short **SetParameterDecimals**(BSTR *szName*, short *nDecimals*);

Obtains and updates the number of decimals stored for the parameter indicated by *szName* (a negative value in *nDecimals* means that there is no limit).

Examining the individual lifetimes of actors

Examining actor histories

The life progression of a simulated actor can be examined through the use of the BioBrowser (Biographical Browser) application that comes with Modgen. The purpose of this tool is to aid in uncovering possible algorithmic errors in the model or to study some particularly interesting cases with respect to the specified model. To create the actor histories, use the **track** command within an .mpp file and re-compile the model to build a new executable; then launch the simulation run, after first selecting the “Microsoft Access tracking” output option from the **Scenario** > **Settings** dialog box in Modgen's visual interface.

When Microsoft Access tracking is selected, a database is created to contain actor tracking as defined in the model. This database has the name ‘scenario_name(trk).mdb’. The BioBrowser application requires the presence of this database file to enable you to visualize the histories of the actors.

It is also possible to obtain tracking output in text format by first selecting the “Text tracking” output option from the **Scenario** > **Settings** dialog box in Modgen's visual interface. The tracking file created in this way has the name ‘scenario_name(trk).txt’ and can be examined with your preferred text editor..

The track command

The activation of individual actor results is achieved through the use of the Modgen **track** command. Only one definition of tracking is allowed for each actor type in the model. However, two different tracking filters (both being logical expressions) can be used to limit the actors that are tracked. The initial filter activates tracking only while an actor is in a certain state. The optional final filter activates tracking only if the actor has ever been in a certain state at some point in the simulation run

Syntax of the track command

track actor_name [initial_filter<,final_filter>] { state_or_link , ... , state_or_link } ;

Instead of listing each state or link to be tracked, Modgen allows alternative keywords. For example, “all_derived_states” adds all of the actor’s derived states to the tracking database; “all_base_states” adds all simple states, excluding internally generated states associated with table filters, the **unit** keyword, or actor set filters; “all_internal_states” adds those internally generated states; and “all_links” includes all of the links established for the actor. All of these alternative keywords should be used with caution, however, as they can cause the tracking database to become very large very quickly.

Example 1

```
// In this example, the sex (and other states) of dominant actors are tracked.
// This filter can be helpful if you have a case-based model with many secondary
// actors but you want to restrict your tracking to dominant actors only
track Person
[ dominant ]
{
      sex,
      …
};
```

Example 2

```
// Both filters are used in this example. starting with only dominant actors
// and applying a final filter to only look at actors who were ever divorced
track Person
[ dominant, divorced ]
{
      sex,
      …
};
```

Example 3

```
// Final filter only

// This tracks everyone, dominant or otherwise, who was ever divorced

// at some point in their lifetime.

track Person

[, divorced ]

{

      sex,

      …

};
```

Using case_id and actor_id in tracking filters with the modulus (remainder) operator

The user of a model can specify a maximum number of cases to track, in a case-based model, but has no control over which cases are tracked. However, you, as the model developer, can control exactly which cases are tracked by using the state ***case_id*** in the initial filter in conjunction with the modulus operator %. The modulus operation produces the remainder when one number is divided by another. The value of the operation is zero when the numerator is exactly divisible by the denominator.

Example (to track states for every 10th case, where the case id ends in 0):

```
track Person
[ case_id % 10 == 0 ]
{

…

};
```

The state ***actor_id*** can be particularly interesting for time-based models. There is no concept of a case in a time-based model and so it is impossible to specify a maximum number of actors to track. However, to limit the costs associated with the tracking, you can choose to limit the tracked actors by using the state ***actor_id*** in the initial filter characteristics, also with the modulus operator.

Example (to track states for every 20th consumer only):

```
track Consumer

[ actor_id % 20 == 0 ]

{

…

};
```

Furthermore, in a time-based model, it is generally recommended that you ensure that tracking occurs in only one replicate. This avoids confusion when visualizing the results with the BioBrowser application. To do this, you can filter actors according to the replicate (with the assistance of Modgen's **GetReplicate()** function that returns the identifier of the current replicate.

Example to filter on one replicate only:

```
track Person
[ GetReplicate() == 0 ]
{

…

};
```

Costs associated with tracking

Tracking, while very useful, can also be costly. First, it can generate huge output files. (For efficiency reasons, the tracking file is initially generated in the TEMP folder of the machine running the simulation; this implies that the machine must have sufficient space.) Second, tracking can significantly slow down simulation runs.

For these reasons, you should try to limit the tracking that is done. Obviously, the more states or links that are tracked, the bigger the tracking will be, so it is useful to limit the states or links included in the tracking as much as possible

The initial and final filters can also be useful for limiting the size of the output file. However, only the initial filter helps make the simulation run more efficient. Indeed, with the initial filter, the actor will not be tracked before reaching a specified state and will stop being tracked when it is no longer in that specified state. On the other hand, with the final filter, Modgen must track the actor throughout the actor's life even if that tracking will not be used if the actor never reaches the specified state. For that reason, the final filter can even slow down the simulation run. If you choose to use the final filter, it is recommended that you also use an initial filter to improve efficiency.

Modgen-generated states for tables

For each table, Modgen generates a state that implements the table's filter expression and a state that implements the table's ***unit*** state (if the table uses **unit**). These states have predictable names that can be used in the **track** statement so that the states can in turn be exported to BioBrowser for further study. Filters have names of the form `table_filter_X` while units have names of the form `table_unit_X`, where `X` represents the name of the associated table. (It is also possible to use these states in subsequent tables, i.e. tables that occur afterwards in the source code module. However, such usage is recommended for debugging or exploratory purposes only and should not be used in production code.)

The following example shows the use of the ***table_filter_X*** and ***table_unit_X*** states:

Example

```
track Person

[ dominant ]

{

      province,

      region,

      table_unit_CENSUS_COUNTS,

      table_filter_CENSUS_COUNTS

};


table Person CENSUS_COUNTS //EN Census counts

[ dominant && IntIs_CENSUS_YEAR( year) ]

{

      census_year+

      * {

            unit,

            duration()

      }

};
```

```
table Person CENSUS_COUNTS_CHECK //EN Debug the Census table

[ dominant ]

{

      {

            duration()

      }

      * sim_year+

      * table_filter_CENSUS_COUNTS //EN Filter for CENSUS_COUNTS

};
```

The tables that result from a test model are shown below.

Table: CENSUS_COUNTS

Table Description: Census counts

Data: Value

	Selected Quantities	
Census Year	unit	duration()
1971	3	2.5
1976	3	3
1981	2	2
1986	2	2
1991	1	1
1996	1	1
2001	0	0
All	12	11.5

Table: CENSUS_COUNTS_CHECK

Table Description :Debug the Census table

Data: Value

	Filter for CENSUS_COUNTS	
Simulation year	False	True
1970	29	0
1971	0	2.5
1972	3	0
1973	3	0
1974	3	0
1975	3	0
1976	0	3
1977	3	0
1978	2.5	0
1979	2	0
1980	21	6
All	69.5	11.5

Debugging and optimizing tools and techniques

Introduction to debugging

This chapter introduces the topic of debugging a Modgen model in the Visual Studio C++ environment. It aims to categorize the different types of bugs that you might encounter, and where possible, outline the likely causes of such bugs and suggest tools or techniques to help you investigate and ultimately fix the problems. Such a discussion of course cannot be comprehensive in terms of incorporating every possible problem or solution but the aim is to cover the more common problems that you could experience while developing your model.

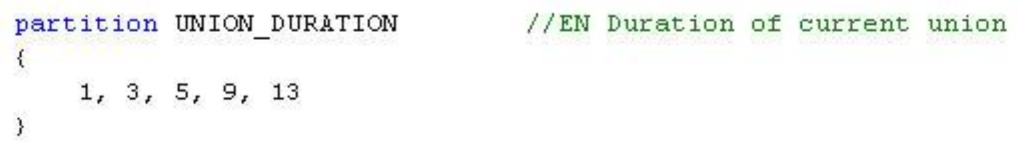
Bugs can be broken down into three broad categories:

- compile-time bugs that include errors returned by the Modgen pre-compiler and by the C++ compiler or the linker during the build process; these include syntax errors or mis-spelled keywords and are often relatively straightforward to fix
- run-time errors that occur after successful C++ compilation and that can cause the execution of your model to freeze or crash
- unexpected results in which your model compiles successfully and does not fail during execution but in which it does not return correct results either; reasons for this could include logic errors or unanticipated interactions with newer versions of Modgen or other code in your model

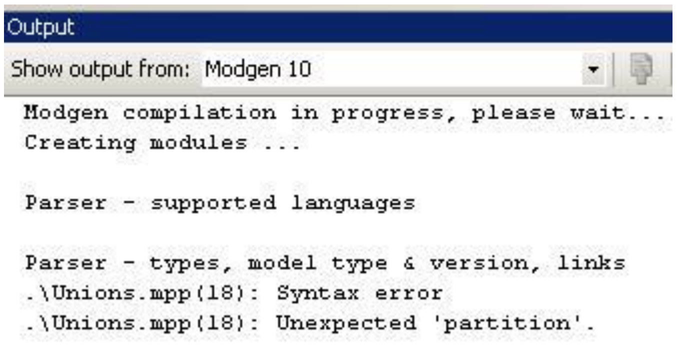
Compile-time bugs

Compile-time bugs can be returned by the Modgen precompiler (i.e. when running Modgen on your model) and by the C++ compiler or linker (i.e. when building or re-building your model within Visual Studio). The precompiler detects Modgen syntax errors in your .mpp files, while the errors reported during the build process are usually C++ syntax errors or misspelled C++ keywords.

The errors returned by the Modgen precompiler or C++ compiler are displayed in the output window of Visual Studio, as shown in the example below. (The code fragment with the error is shown first, in which the error was caused by a missing ';' after the '}' at the end of the partition statement; the corresponding output window is shown after the code fragment)



```
partition UNION_DURATION        //EN Duration of current union
{
    1, 3, 5, 9, 13
}
```



```
Modgen compilation in progress, please wait...
Creating modules ...

Parser - supported languages

Parser - types, model type & version, links
.\Unions.mpp(18): Syntax error
.\Unions.mpp(18): Unexpected 'partition'.
```

An important aspect of the Modgen architecture is that it enables you to debug your model code within the model's .mpp files. By clicking on the first line of the error message in the "Output" window, your cursor should be transferred immediately to the location of the error in your code (line 18 of 'Unions.mpp' in the example above), at which point you can make the correction. (Many of these errors are relatively straightforward to fix; they are often syntax errors generated by specifying Modgen or C++ instructions incorrectly)

The compile-time bugs that come from the linker are frequently one of the following two types:

- "xxx" already defined in "yyy"
- unresolved external symbol "zzz"

The former is usually caused because the "xxx" function was defined twice, often in two separate modules; the solution is to use Visual Studio's "find" capabilities to find all occurrences of the function name and then remove the bogus definition(s). The latter can also occur if a function was declared and defined with two different names (one having a typo), in which case you need to make the names consistent and accurate.

Run-time errors

Run-time errors can essentially occur because your model has encountered an exception, or it has frozen, or it has crashed after you started the simulation run. Irrespective of the cause, however, the general procedure is to try to isolate the problem and then understand its cause, after which you can take steps to fix it.

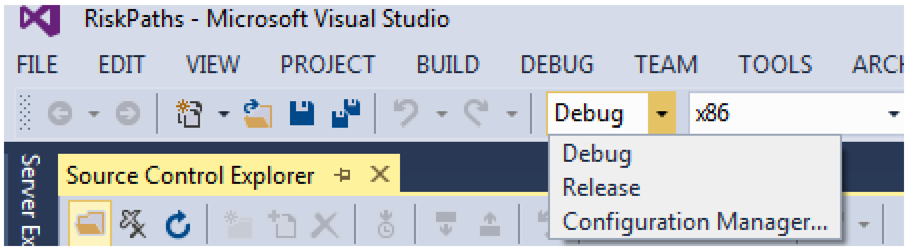
In release versions of models, a run-time error produces an informative message in the "Errors and Warnings" area at the bottom of the model progress window of the visual interface. The message is also written to the log file of the scenario. The model will not produce any output tables but will produce a tracking file if tracking was enabled. The simulation will end immediately but the visual interface will continue to function. (If the simulation was started in batch mode, it will exit with a return code of 2.)

When a run-time error occurs, the call stack should be examined (which can only be done in a debug version of your model). Sometimes the error occurs inside the Modgen library or inside ‘ACTORS.cpp’ or ‘TABINIT.cpp’. If this is the case, then the function from the .mpp file closest to the top of the stack should be examined. In most cases, this is enough to identify the problem.

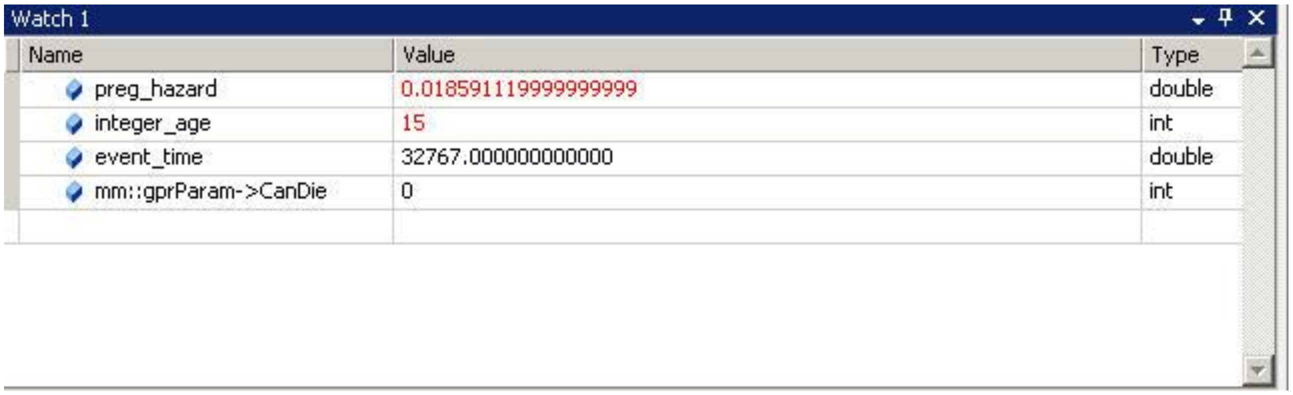
Creating a debug version of your model

To create a debug version of your executable, select the Debug item in the “Solutions Configurations” drop-down box on the Visual Studio toolbar and then build the executable by running Modgen and compiling your model. Note that the name of the resulting executable will finish with the letter ‘d’; for example, in the model RiskPaths, the debug version of the executable has the name ‘RiskPathsd.exe’.

In the following diagram, again corresponding to the model called RiskPaths, the possible choices of the Solution Configuration drop-down box are shown, with the “Debug” choice highlighted or selected.



To display the value of a state or variable, you need only type its name in the Watch window that appears in the bottom left of the Visual Studio debugging environment. (A sample Watch window is shown in the figure below). Special handling is required for parameter names; they must be prefixed by the character string “mm::gprParam->”. An example, which is also shown in the following figure, is mm::gprParam->CanDie, where “mm” is the C++ namespace used by Modgen to protect all symbols declared in model code.



Watch 1

Name	Value	Type
preg_hazard	0.018591119999999999	double
integer_age	15	int
event_time	32767.000000000000	double
mm::gprParam->CanDie	0	int

In debug mode, Modgen automatically verifies the integrity of certain internal memory structures and displays a message box if your code has compromised these structures. This verification is no guarantee that your code is error-free overall but rather provides a possible means of identifying if the code is responsible for anomalous behaviour such as crashes.

Exceptions

Modgen has code in place to prevent release versions of models from crashing when an exception occurs. The exceptions covered in this manner are:

- division by zero
- a bad pointer or index
- lack of sufficient memory
- invalid argument to the C++ **exp()**, **log()**, **pow()**, or **sqrt()** functions

In the release version of your model, the simulation will stop if an exception is encountered, and an informative message will be displayed in the Output area at the bottom of the model progress window in Visual Studio. (The same message is also written to the log file of the scenario.) The actual content of the message depends on the nature of the error, the type of model (case-based or time-based) and the phase of execution (PreSimulation, Simulation, PostSimulation, or UserTables).   The following example shows the message when an invalid argument is used for the **sqrt()** function in the Simulation phase of a case-based model:

```
Error:
      Type: invalid argument to sqrt()
      Phase: Simulation
      Subsample: 1
      Event: Person.AgeOneYearEvent(implement)
      case_seed = 2147500455
      time = 18.000000
      actor_id = 33
```

As the example illustrates, the name of the current event is displayed as well as an indication of whether the error occurred in the 'time' or 'implement' function of the event. Note that a run-time error which occurs during the implementation of an event might be only indirectly associated with the code in the implement function of the event. For example, if 'a' is a simple state and 'b' is a derived state defined as 'b = exp(a);', an event implementation function that sets 'a=1000;' will generate a run-time error associated with the event, even though the call to 'exp' that generated the error is not located in the implementation function of the event.

The error information may include the value of the state **actor_id** (also as illustrated in the example). This can be useful in identifying which actor was responsible for the error, since **actor_id** is an arbitrary unique identifier assigned to each actor in a simulation. To produce reproducible values for **actor_id**, the scenario starting seed must be identical, and the number of simulation threads in the scenario has to be set explicitly to 1 in the **Scenario** > **Settings** dialog box of Modgen's visual interface..

Virtually all run-time errors (including out of memory and reading or writing invalid memory locations) are caught and reported in the Release version of models. Most errors will be reported with comprehensible information under "Type:", but some rarer errors may require recourse to system documentation to understand. For example, the error EXCEPTION_STACK_OVERFLOW may result from model code that attempts to create an excessively large array in a temporary variable in a function.

For run-time errors that are due to simple programming errors or that occur in small pieces of code, you may be able to resolve the error by examining the offending code that generated it. If you are unable to do so, however, or if the offending code is either larger or complex in scope, a typical approach is to build the model in debug mode and then start debugging it from the **Debug > Start debugging** menu in Visual Studio. When each exception occurs, you will see a message offering to break at an appropriate location in the code. Because it can be a slow process to run a simulation through a debugger, you can obtain significant time savings if you can simulate just the case that produced the error. (You would do this through the **Scenario** > **Settings** dialog box, choosing the number of cases to be 1 and using the case seed from the error report as the starting seed)

Freezes and crashes

A run-time freeze generally indicates that there is an infinite loop somewhere in your code. Such a loop is often found within the event implementation function for a certain event, although it can also be caused if an event time function always returns the exact same time and is never reset. A run-time crash indicates a relatively serious error in your code because Modgen already captures most exceptions, meaning that such a crash would not be due to any of those exceptions. (Some such crashes, however, can be cleared by rebuilding your model so if you encounter a run-time crash, a good first step is to simply rebuild and rerun your model to see if the same crash still remains.)

The process of debugging either a freeze or a crash is not a mechanical one, but generally it involves the following steps:

- identify the case being simulated at the time of the freeze or crash. For a freeze, the INFO button in the visual interface indicates the current case being run; for a crash, the debugger is used to identify the problem case.
- next, identify or isolate where in the case (that is, in what event) the infinite loop or crash is happening. This is generally done using the Visual Studio debugger by putting breakpoints (right-click in the code window and select **Breakpoint** > **Insert Breakpoint)** in the implementation function for each event
- finally, look at the code of the offending event to try and determine why the freeze or crash has occurred. This is again generally done using the Visual Studio debugger, inserting a breakpoint at the beginning of the event; you can use the F5 command to move to the next breakpoint or the F11 ("Step into") command to step through each line of the function.

Unexpected results

As suggested by its name, it is very difficult to itemize all of the possible ways through which unexpected results can arise. Such results can broadly be categorized as:

- results that change after a new version of Modgen was used to compile your model
- results that change after new and seemingly unrelated changes were made to your model code
- unexpected results not associated with any change of Modgen or your model code

Unexpected results could occur after compiling your model with a new version of Modgen if the new Modgen version fixed a bug that previously affected your model's results or if some aspect of Modgen was redesigned or if a new bug was in fact introduced into Modgen itself. To see if this might have happened, you can check the release notes for the new version of Modgen (using the Release Notes icon on the Modgen toolbar in Visual Studio) to see if there are any relevant announcements. If nothing obviously relevant appears, you should contact microsimulation@statcan.gc.ca and provide details of your problem.

To deal with situations in which your results have changed after seemingly unrelated changes were made to your model, it is extremely useful to have multiple versions of your model available. You can then run the same scenario with different versions of your model and compare the results, using the Computed Tables feature from Modgen's visual interface. Also, while there should be relatively few public versions of your model overall, it is good practice to have many "in-between" interim versions with relatively few changes per version. In this way, if the results have in fact changed from the previous version, there are not as many modules that should need to be investigated. The investigation itself involves comparing the source code (remember that .mpp files are really text files). The final category (unexpected results not associated with any change of Modgen or your model code, which can include logic errors or incorrect table specifications) can be, somewhat predictably, the most difficult category to debug. . Use of the tracking database in conjunction with the BioBrowser application is appropriate for small runs. You can use different filters to observe small subsets of the population of interest, track as many states as you can, and always make sure to track the state ***case_seed***. For large runs, the expression **min_value_in(*case_seed*)** can be added to the tables of interest. It will display the smallest seed of the case that resulted in an entry to a cell that was expected to be blank. That seed can then be used to debug the problem case.

General hints to facilitate debugging of run-time errors

Irrespective of the type of bug that you have encountered, there are certain general principles to always keep in mind to simplify or make the debugging process easier.

- ensure that you are using the correct version of your model and the latest version of Modgen. You can verify both version numbers from the **Help** > **About** menu in Modgen's visual interface to your model
- ensure that you are using the correct scenario; once you have opened a scenario, its name appears in the title line of Modgen's visual interface
- before debugging, always ensure that you have chosen **Build** > **Rebuild model_name** from the Visual Studio menu, and not **Build** > **Build model_name**
- ensure that you are using only one thread when you are debugging

Clearly, a debugging process would also be improved if bugs were avoided in the first place! While it is usually impossible to achieve this phenomenon, there are techniques that can be adopted as "good coding practice" which should also reduce the frequency of creating new bugs.

- document your code with notes or labels which makes it easier to understand your code and avoid miscomprehensions (and which also contributes to the contents of the encyclopaedic documentation)
- apply consistent naming conventions so that you can tell at a glance is an entity is, for example, a symbol or a local variable

- apply appropriate indentation so that individual sections or blocks of code stand more clearly apart from their adjacent sections
- put the opening and closing curly brackets of blocks on separate lines and indent them equally to help make it easier to spot the block of code within the matching brackets
- always initialize your variables and simple states, enclosing the initial values within curly braces
- avoid global variables which can cause problems, for example when there is more than simulation thread
- ensure that states only change once per event; if interim state changes are allowed to occur before the event is completed, store them in local variables
- establish a code review process, with the review preferably performed by someone other than you, the developer of the model
- copy reuseable code from one module to another, rather than relying on your typing abilities

Specific tools and techniques to facilitate debugging of run-time errors

Bounds checking

Parameters: Modgen models compiled in debug mode check, when the simulation is running, that each index used in an array parameter does not exceed the correct bounds of the associated dimension. If a parameter array bound is violated, a message box describing the problem is displayed. The same information is written to the scenario log file. After the message box is displayed, a second dialog box enables you to exit the model, to ignore the error, or to invoke the debugger. This verification of the array parameter indexes is toggled on or off via the *index_errors* option. Such checking is not active in models compiled in release mode, and has no effect on performance in that mode.

This checking can be de-activated in debug mode by defining the manifest constant MG_NCHKBND in Visual Studio using “Preprocessor definitions” in the “C/C++” tab in Project Settings. You may wish to de-activate bounds checking in debug mode if you know that your model does not violate array bounds and if improving performance in debug mode is essential for troubleshooting a specific problem.

You are strongly encouraged to use debug mode to identify array bounds errors before releasing models to users or other developers. Otherwise the appearance of many error dialog boxes that are generated due to array bounds violations could make it difficult to use the model in debug mode. In addition, this may indicate that a serious error in logic is present.

Actor data member (state arrays): Modgen verifies that array bounds of actor data members are not violated at run-time for models compiled in debug mode. This feature operates in the same way as for array parameters. Bounds checking decreases the simulation speed of models compiled in debug mode by about 30%.

Classification or range state: Modgen can produce a run-time error message box in debug mode when an attempt is made to assign an invalid value to a classification state or a range state. The message indicates the state name and the value to be assigned. In release mode such values are silently coerced to the first and last value of the classification, or to the minimum and maximum allowed value of the range. The behaviour of deliberately generating run-time errors in debug mode if an invalid value is assigned to a range or classification state is enabled either by defining the manifest constant MG_NCHKLMT in Visual Studio using “Preprocessor definitions” in the “C/C++” tab in Project Settings, or by use of the *bounds_error* option.

Check Sums

Modgen can generate a "unique" number for each case simulated, based on the events that occur during the case and the time when each event occurs. This functionality can be activated both in release and in debug mode by defining the manifest constant MG_CHKSUM in Visual Studio using “Preprocessor definitions” in the “C/C++” tab in Project Settings or by using the *case_checksum* option. The case seed and check sum for each case will then be written in the debug file (ex. ‘Base(debug).txt’). One use of check sums could be to identify a case which is different between two simulations when it should instead be identical. Check sums are ignored in time-based models.

Event tracing

Modgen models can trace all the events occurring in a case, or in a portion of a case. This functionality can be activated in either debug or release mode by defining the manifest constant MG_TRCEVNT in Visual Studio using “Preprocessor definitions” in the “C/C++” tab in Project Settings or by using the *event_trace* option. You can also specify in the code when to start tracing events as well as when to stop tracing events. To do this, Modgen uses the two functions **StartEventTrace() and StopEventTrace(),** both which take no arguments and return no results. (These functions are intended for use in case-based models only.)

Tracing events does make the simulation extremely slow and can also lead to a huge debug file, so event tracing should be used with caution. When a model traces events, its debug file will contain an entry for each event which will include the case seed, the actor identifier, the name of the actor, the name of the event and the time when the event occurred. For scheduled events, instead of the name of the event itself, the debug file entry will contain « scheduled – » followed by the number identifying which scheduled event occurred This can be used, for instance, to identify the first event that differs after a case has been pinpointed with the check sum method,

SkipCases() function

This function allows you to skip the simulation of a given number of cases. **SkipCases()** starts cases and ends them so that the next seed is the same as if the cases had been simulated. **SkipCases()** facilitates debugging because by using it, you can simulate, for example, only the second half of a simulation or the last ten cases of a simulation.

Supposing you had a series of three cases with the following case seeds:

```
6589593
25768594
183475
```

Skipping one case would result in the simulation of a series of two cases with the following case seeds:

```
25768594
183475
```

SkipCases() also modifies the distribution of cases among the subsamples so that the distribution is the same as it would have been if the skipped cases had in fact been simulated. For example, if you requested 5000 cases and another 5000 cases were skipped, Modgen would distribute the cases among the subsamples as if 10000 cases had been requested--but it would then just simulate the last 5000 cases that you requested. (That is, the first 5000 cases would be skipped.)

SkipCases() has to be called before cases are simulated. To use it, create a parameter containing the number of cases to be skipped and call the function before **CaseSimulation()** in each thread. If a case has been simulated in the thread before **SkipCases()** is called, an error message will be shown and the simulation will continue without skipping cases in that thread.
Example: (in which CasesToSkip is a parameter containing the number of cases that should be skipped)

```
void Simulation()
{
	long		lCase;

	SkipCases(CasesToSkip);
	// case loop
	for ( lCase = 0; lCase < CASES() && !gbInterrupted && !gbCancelled &&
!gbErrors; lCase++ )
	{
		// simulate the next case
		StartCase();
		CaseSimulation();
		SignalCase();
	}
}
```

Note that the number of cases specified in the scenario does not include the skipped cases—nor are the skipped cases included in the population size (shown via **Scenario** > **Settings** in Modgen's visual interface).. Modgen will skip the number of cases specified as the argument of **SkipCases(),** then will simulate the number of cases specified on the scenario settings screen. For example, if you ask for 10 cases and set your CasesToSkip parameter to 1000, the model will skip 1000 cases and then simulate the 10 cases that you requested.

Skipped cases are excluded from tracking.

WriteDebugLogEntry() function

The function **WriteDebugLogEntry()** causes output to be directed to the file 'scenario_name(debug).txt'. It can be used in release mode as well as in debug mode. The easiest way to use it is to call it with a string argument.
Example:

```
CString szDebugExample = "";
szDebugExample = "Now in BirthdayEvent";
WriteDebugLogEntry(szDebugExample);
```

Note that the use of **WriteDebugLogEntry()**, will make the debug file huge. Thus, caution should be exercised when using this function.

Selectively executing a single replicate

A single case can be simulated in a case-based model by specifying a starting seed. Time-based models have no corresponding built-in ability to simulate a single replicate if multiple replicates were requested. Simulating a single replicate can sometimes be useful for debugging purposes. To simulate a single replicate, insert a line such as the following as the first line of the **Simulation()** function:

```
// Simulate only replicate #4

if ( 4 != GetReplicate() ) return;
```

Optimizing techniques

Running Modgen models reliably over a network

Modgen models often have long running times. If a model is run across a network (e.g. the executable file is located on a server and is run on a workstation), stale connections or other network curiosities can result in obscure run-time errors for older models that were not created with Modgen's new model wizard.

The approach to solve this problem involves a one-time modification to the Visual Studio properties of each Modgen project. Open the solution file for the project in Visual Studio, and pick **Project** > **Properties**. Under Configuration Properties on the left pane, click "Linker" and then select "System".   Then change the setting "Swap Run From Network" from "No" to "Yes (/SWAPRUN:NET)" and recompile the model.

It is also possible to set this option in an existing executable without re-compiling it. This can be useful for running old models (perhaps compiled with an earlier version of Modgen or Visual Studio) reliably across a network. To set the option in an existing executable without recompiling, open a command prompt and run the EDITBIN utility (supplied with Visual Studio) setting the SWAPRUN option as in the following example:

```
EDITBIN /SWAPRUN:NET riskpaths.exe
```

Increasing available memory to Modgen models on Windows x64 operating systems

The memory available to a Modgen model can be doubled if the model is run on a 64-bit Windows operating system such as Windows 7 64-bit. This can be particularly significant for time-based models where memory is at a premium.

This is already specified in the Models created with the New Modgen Wizard, but for previous versions, open the solution file for the project in Visual Studio, and pick **Project** > **Properties**. In the left pane, click "Linker" under "Configuration Properties" and select "System". Change the setting "Enable Large Addresses" to "Support Addresses Larger Than 2 Gigabytes (/LARGEADDRESSAWARE)". Recompile the model.

It's also possible to set this option in existing executables without re-compiling the model. Use the EDITBIN utility (supplied with Visual Studio) as in the following example:

```
EDITBIN /LARGEADDRESSAWARE riskpaths.exe
```

Using the *options* statement

The **option*s*** command is followed by one or more options, separated by white space. Each option consists of a keyword, followed by the equal sign, i.e. = , followed by the option value. Multiple options are permitted within one **options** statement, where each option is separated by a white space. The option settings are also cumulative in effect; if the same option is specified multiple times in the model source code, the last setting encountered by Modgen takes precedence. Certain Modgen capabilities that previously required you to change C++ compiler options can instead be enabled with the **options** command.
Example:

```
options bounds_errors=off packing_level=1;
```

The following table lists the options available in the current version of Modgen and the equivalent C++ manifest constant (if any).

Option	Default value	Description	Equivalent compiler option
bounds_errors = on | off	On	Controls run-time errors on out-of-bounds range or classification assignment (Debug version only)	MG_NCHKLMT (undocumented)
index_errors = on | off	On	Controls run-time errors on out-of-bounds array index (Debug version only)	MG_NCHKBND
case_checksum = on | off	off	Controls the production of a checksum for each case (Debug and Release versions)	MG_CHKSUM
event_trace = on | off	off	Controls the trace event capability (Debug version, also requires use of **StartEventTrace()** and **StopEventTrace()** helper functions).	MG_TRCEVNT
fp_consistency = on | off	off	Controls whether the C++ compiler generates more consistent but less efficient floating point code	none
packing_level = 0 | 1 | 2	0	Controls memory optimization of actor states	none

The first four options, which are all debugging options (bounds_error, index_errors, case_checksum, event_trace) were discussed earlier in the section "Specific tools and techniques to facilitate debugging of run-time errors". The remaining two options (*fp_consistency* and *packing_level*) have more of an optimization role and are discussed below.

Option fp_consistency:

In Release versions of models, the C++ compiler normally optimizes the speed of floating point operations. A slight change in model code that does not affect the logic of the model (such as those produced by *packing_level*) can result in differences in results when floating point code is optimized. Setting *fp_consistency* to *on* will reduce or eliminate such effects, but the resulting model will run slower. The option has no effect on debug versions of models. The *fp_consistency* option can be useful when testing for differences between model versions.

Option packing_level:

The *packing_level* option controls how Modgen defines and orders actor states to reduce memory use. This option can be useful in time-based models with many actors. Higher values of *packing_level* reduce memory use, but at the expense of increased computation time. The default value of *packing_level* is 0, which means no packing is performed apart from efficient ordering of C++ members of actor-related structures and classes. A *packing_level* of 1 means that Modgen will use the smallest possible simple C++ integral type to store range, classification, partition, and logical states. A *packing_level* of 2 means that Modgen will pack variables of type range, classification, partition, and logical into the minimum number of bits possible. The available packing levels are summarized in the following table.

packing_level	Description
0	Use the C++ type int to store states of type range, classification, partition, and logical.
1	Use the smallest possible simple C++ type to store states of type range, classification, partition, and logical.
2	Use the smallest possible number of bits to store states of type range, classification, partition, and logical.

In release versions of models, differences in floating point-optimized code generated by the C++ compiler can lead to different model results, depending on the value of *packing_level*. However, this does not affect debug versions of models. In situations where consistency is more important than efficiency, you can use the *fp_consistency* option to turn off the optimization of floating point code in the release version of a model.

The Modgen memory usage report does not include the effect of *packing_level*.

Computation and memory efficiency

Use of index_type

For each table declared in a model, Modgen creates internal actor states that record the location of the current cell in the table. The C++ type used to represent these internal states can be controlled using the **index_type** statement. The values allowed for **index_type** are the C++ signed integer types char, short, int, long, and the C++ unsigned integer types uchar, ushort, uint, ulong (synonyms for unsigned char, unsigned short, unsigned int and unsigned long, respectively). If a model contains no **index_type** statement, these internal states are of type int.

While the **index_type** statement is useful in models where memory utilization is important, it is nevertheless your responsibility as the model developer to ensure that the maximum number of cells in selected tables does not exceed the limit of the C++ data type specified in the **index_type** statement. If more than one replicate is specified in a model scenario, the number of cells in each table needs to be multiplied by the number of replicates plus 1 when calculating the limit.

Example:

```
index_type ushort;
```

real, counter and integer types

These types can be used to declare actor states that contain floating point values (real), non-negative integral values (counter) or integral values (integer). (Note that they can be used as data types for states, but they cannot be used at all as types for parameters). The actual C++ types used to represent these states are controlled using the **real_type, counter_type,** or **integer_type** statements, respectively. For time-based models with many actors where memory usage is important, you may wish to consider declaring all floating point actor states as real, certain non-negative integral states as counter, and all integral actor states as integer. The corresponding type statements can then be used to optimize memory usage. The following table lists further characteristics for each of these three types.

	type real	**type counter**	**type integer**
C++ types allowed in type statement (with storage used by each type)	float (uses 4 bytes of memory, has 6 decimal digits of accuracy) double (uses 8 bytes of memory, has 15 decimal digits of accuracy)	char or uchar (1 byte) short or ushort (2 bytes) int or uint (4 bytes) long or ulong (4 bytes)	char (1 byte) short (2 bytes) int (4 bytes) long (4 bytes)
Default C++ type if model has no type statement	double	int	int

Example of type statement and type declaration in actor definition	`real_type float;`
	
	`actor Person {`
	`       real  earnings;`
	`};`
		`counter_type ushort;`
		
		`actor Person {`
		`       counter jobs;`
		`};`
			`integer_type short;`
			
			`actor Person {`
			`       integer`
			`gains_and_losses;`
			`};`

Comments on optimal use by model developer	Use double for maximum accuracy or float to minimize	Developer must ensure that maximum values attained	Developer must ensure that maximum and minimum

	memory use	by actor states fall within the C++ limits for the data type specified in the counter_type statement	values attained by actor states fall within the C++ limits for the data type specified in the integer_type statement

How to improve memory efficiency in time-based models

Time-based models with many actors make heavy demands on memory. The maximum number of actors that can be simulated is limited by available memory and the efficiency of use of that memory. This section contains a number of recommendations for improving memory efficiency in time-based Modgen models. These recommendations are subject to the particularities of a given model.

- Declare all floating point actor states as real rather than as double or float, and specify

  ```
  real_type float;
  ```

- Declare all integer actor states as ranges rather than as int, and specify

  ```
  options bounds_errors=off;
  ```

  and

  ```
  options packing_level=2;
  ```

- Specify either

  ```
  options packing_level=1;
  ```

  or

  ```
  options packing_level=2;
  ```

- Set the number of threads to 1 in **Scenario > Settings**, if the model can execute on a multi-processor or hyperthread-enabled machine.
- Specify

  ```
  time_type float;
  ```

- If the model permits, specify

  ```
  counter_type ushort;
  ```

- If the model permits, specify

  ```
  integer_type short;
  ```

- If the number of cells in any table is 65535 or less (the number of replicates requested in a scenario affects this), specify

```
index_type ushort;
```

The following recommendations are useful for any model, time-based or case-based.

- Examine the memory usage report to better understand what parts of the model consume the most memory. The use of distinct modules improves the usefulness of the report.
- Comment unneeded or rarely-used tables in model source code.
- Place rarely used tables and associated actor states in separate modules so that they can be excluded easily from the production version of the model.
- Consider logically-equivalent designs that reduce the number of events declared in the model.
- Consider logically-equivalent designs that reduce the number of states.

Memory Usage Report

During the simulation phase, significant memory use normally originates from four sources:

- parameters
- the storage of actors
- the event queue
- tables

A fifth potential source for significant memory use is actor sets (for models that make use of them).

Parameters

Parameters do not grow in size during a simulation, but may nevertheless consume significant memory if they contain many cells. This is only a problem during the simulation phase if the cells are referenced repeatedly. For example, a large microdata file represented as a parameter may not be a problem if it is read only once at the beginning of the simulation, because the operating system would “page out” the unreferenced memory to disk. If the machine has 2GB or more of physical memory, parameter memory use can reduce the memory available during the simulation phase, because paging cannot increase available memory beyond the 2GB addressing limit of 32-bit processors.

Actor memory

The memory used is the product of the number of actors and (roughly) the number of states. The type of state is also important, with floating point states requiring eight bytes and other states requiring four bytes. Hidden states associated with tables are also present in actors.

Event queue

The memory used is the product of the number of actors and the number of events (for each type of actor) multiplied by the size of the internal structure used for events (a model-independent fixed quantity).

Tables
Tables take up space, due to internal states created for each actor, plus the common area used to cumulate results. The common area used for tabulation may become large if there are many cells and if these cells are repeatedly updated during the simulation phase. The table cell area is allocated only for tables that are selected by the user at run-time, whereas the internal table states created in actors are present whether the table is selected or not. The space required for the table cell area also depends on the number of replicates or subsamples.

Actor sets
The memory used is (roughly) the product of the number of actors in the actor set and the size of the internal structure used to store references to actors in the actor set. Hidden internal states can also be created for actor sets for each actor, whether or not the actor is a member of the actor set.

The memory usage report identifies areas of the model that are susceptible to change by you, the model developer. Tables are emphasized because the elimination of a table is an easy way for you to free up memory with no change to the model structure.

There is a checkbox in the **Scenario > Settings** dialog box of Modgen's visual interface, labeled "Memory Usage Report". If checked, a separate report on memory usage is generated in a file named 'scenario_name(usage).htm'. The report is generated at the end of the simulation run, in the language of the model at the time of report generation. Note that the name is the same in English and in French. For distributed runs, only the statistics for the final "master" job (the one that merges the others) are saved, under the "master" scenario name.

A memory usage report has the following components or sections:

- general information
- six tables on memory usage (plus a seventh table if your model uses actor sets)
- three annexes

Each part of the memory usage report is explained below.

General information

The first part of the report contains general information about the simulation: date, executable, version, scenario, subsamples or replicates, table copies and simulation threads. The number of simulation threads corresponds to the real number of threads used for the simulation. For instance, if you specify a number of threads greater than the number of subsamples, the number of real threads created will be equal to the number of subsamples. The number of table copies is shown because Modgen keeps a copy of the table cells for each subsample. For case-based models, a copy of the table cells is also kept for the whole simulation.   The number of table copies is used in the memory usage table for the model's tables.

Table 1: Maximum instances

Following the general information is a table containing the maximum instances of actors for each type of actor declared in the model   For practical reasons, this is in fact the sum of the maximum "active" actors in each thread during the simulation. This information is used later in the actor memory usage table.

Table 2: Memory use: Parameters

The next table reports the memory usage of parameters in bytes per parameter. Each parameter is listed in alphabetical order.   The memory used by a parameter is the number of cells in the parameter multiplied by the memory used for one cell. The memory used for one cell depends on the type of the parameter. Information about the memory used for each type of parameter is given at the end of the report.

Table 3: Memory use: Actors and events

There is actually one of these tables for each type of actor in your model. In each table, the memory used is reported by module. There is one row for each module in which memory is used for actors and/or events. Each row contains the following information as its columns:

Declared states (bytes per actor)

This is the memory used by the states that you declared in the module. The memory is calculated for one actor only. For instance, the memory used by the state ***sex*** will be counted in the row for module "personCore" if the file 'PersonCore.mpp' includes the following declaration:

```
actor Person
{
      SEX   sex;
};
```

Internal states (bytes per actor)

This is the memory used by internal states that are created by Modgen for derived states declared in the module. The memory is calculated for one actor only. For instance, assume that the file 'PersonCore.mpp' includes the following declaration:

```
actor Person
{
      int   person_year = duration(life_status,ALIVE);
};
```

The memory used for the state ***person_year*** would be counted in the declared states for the module "PersonCore". However, because the derived state **duration(…)** was also specified, Modgen automatically creates one or more "internal states" which also use memory. This memory is counted in the internal states for the module "PersonCore". Information on the maximum memory used by each kind of derived state is provided at the end of the report.

Links (average bytes per actor)

This is the memory used by links declared in the module. For single links, the memory used is the same for all actors. For multiple links, however, the memory used depends on the number of linked actors. For each different link, the maximum actors linked is kept in memory and an average is calculated by taking the memory used by the link at the time it was taking the most memory and dividing by the maximum number of active actors as given in "Table 1: Maximum instances".

Total actor memory (bytes)

The total memory used for actors is the sum of the first three columns, multiplied by the maximum number of active actors in the simulation.

Event functions (number)

This is the number of events declared for that actor type in the module. For instance, this column would indicate that four event functions were declared in the module "PersonCore" if the file PersonCore.mpp contained this declaration:

```
actor Person
{
      event timeSchoolYearEvent,SchoolYearEvent,2;
      event timeSchoolEntryEvent,SchoolEntryEvent, 1;
      event timeElSecDropoutEvent,ElSecDropoutEvent, 1;
      event timeElSecReturnEvent,ElSecReturnEvent, 1;
};
```

Total event memory (bytes)

For one actor, the memory used in bytes corresponds to the memory used for one event multiplied by the number of event functions declared in the module.   This column gives the total memory used for events for all actors, which is the memory used for events for one actor multiplied by the maximum actor instances.

Total memory (bytes)

This is the sum of the total event memory and the total actor memory.

Table 4: Memory use: Tables

This table reports the memory used for tables. Each table is listed in alphabetical order. Each row contains the following information as its columns:

Table selected

Memory is used for table cells only if the table is selected in the simulation. This column contains an "X" if the table has been included in this simulation and is blank otherwise.

Table cells (bytes)

This is the memory used by one copy of the table cells if the table is selected. (Remember that multiple copies of the table cells are kept). This information is provided even if the table was not included in the simulation.

Internal states (bytes per actor) for tables

This is the memory used by states created by Modgen for derived states declared in the table. The memory is calculated for one actor only. For instance, assume that the following table declaration is in your model:

```
table Person X03_SeverelyDisabled
{
	{
		entrances( dfle, SEVERE_DIS )
	}
};
```

Modgen will create an internal state for the derived state **entrances(***dfle*, *SEVERE_DIS***)**. This memory is included in the corresponding row of the table.

Modgen also creates internal states for the table itself. This memory is also included in this column and the memory is used even if the table is not selected. The memory used for internal states created for a table, excluding all internal states associated with derived states, is given at the end of the report.

Total internal states memory (bytes) for tables

This corresponds to the memory used for internal states for one actor multiplied by the maximum number of active actors as given in "Table 1: Maximum instances".

Total (bytes) for tables

This is the total memory used for the table. If the table was not included in the simulation, the total memory is the same as the total internal states memory. If the table was included in the simulation, the total memory equals the internal states memory plus the table cells memory multiplied by the number of table copies.

Table 5: Memory use: Summary by module

This table provides a summary of the memory used, broken down by module. Each module is listed in alphabetical order. Each row has three columns--the memory used for:

- parameters
- actors and events
- tables declared in the module

(Potential) Table: Memory use: Actor sets

This table is only generated if your model contains at least one actor set. The table reports the memory used by those actor sets. Each actor set is listed in alphabetical order. For each actor set, the following information is provided:

Maximum members

This is the maximum number of members the actor set contained during the simulation.

Members (bytes)

Some of the memory used by an actor set depends on the membership of actors in the actor set. For each actor member in the set, memory is needed to keep the position of the actor in the set, which is the memory reported in this column. More specifically, this column reports the memory used for each member of the set multiplied by the maximum number of members the set had during the simulation run. The memory used for one member of a set is given at the end of the report.

Internal states (bytes per actor) for actor sets

Depending on the declaration of the actor set, Modgen sometimes creates internal states for the actor set. For instance, assume that the following actor set has been declared in your model:

```
actor_set Host asAllHosts
order RandomStream();
```

In this example, Modgen creates an internal state to keep the random number associated with the actor member of the set, asAllHosts.

It is also possible for derived states to be declared in an actor set. When this happens, the corresponding memory is reported here but is given for one actor only. This memory is used, even if the actor is never a member of the set.

Total internal states memory (bytes)

This is the "internal states memory" multiplied by the maximum number of active actors given in "Table 1: Maximum instances". This figure is multiplied by the maximum number of active actors, and not by the maximum number of members in the set, because the memory is used whether or not the actor is in the set.

Actor set cells

This is the number of cells in the actor set multiplied by the memory taken by one actor set cell. This memory is used even if the actor set is empty. The memory taken by one actor set cell is given at the end of the report.

Total (bytes)

This is the sum of the memory taken by members, internal states and the actor set cells.

Table 6: Memory use: Summary

The last table in the report is a summary. It reports the total memory used for parameters, actors and events, actor sets, and tables.

Annex A – parameter and state sizes

There are three different annexes that appear after the tables that report memory use. The first annex is a list of possible data types with the memory used by one state or parameter of this type. For instance, a state declared with the type int uses four bytes, whereas a state declared with a type double uses eight bytes.

The size of the type TIME depends on the declaration of **time_type** that you provide. For instance, a state declared with the type TIME uses four bytes if your model has the following declaration:

```
time_type int;
```

Annex B – internal states

For each derived state specification, the maximum number of bytes used by internal states is given. For instance, Modgen creates one internal state of type int for each derived state **changes** (e.g. changes(dfle)).

Sometimes the number and type of internal states created by Modgen depend on the context. For instance, Modgen creates two internal states of type double and one internal state of type TIME for a **weighted_duration** derived state if it is used in the declaration of a state However, if the **weighted_duration** is only used inside a table, Modgen only creates one internal state of type double and one internal state of type TIME. Neveertheless, this annex displays the maximum number of bytes.

Also reported in this annex is the number of internal states created for a table. This only includes internal states created for the state itself, not internal states created for derived states declared in the table.

Annex C – structures

This annex provides a list of different structures that are referred to in the report and gives the memory used for the structure itself.   This contains the actual size of the actor class created by Modgen for each type of actor. This size includes the effect of your chosen *packing_level* option, but unlike Table 3 ”Memory use: Actors and events”, it does not include dynamically-allocated memory used within actors, e.g. the memory used to store links to multiple actors.

Documentation of a Modgen model

The documentation of a model can be categorized at the following levels:

- Primary: a broad overview of the model and of its underlying assumptions or equations
- Encyclopedic: the detailed structure of the model
- Scenario: notes to describe scenarios or parameter estimation methods

Modgen provides tools or facilities to deal with all of these levels of documentation. This chapter discusses what you need to know and do, in your role as model developer, to ensure that all of this documentation is properly established for your model. Then, since Modgen was designed to enable models to be multilingual, the chapter provides an overview of translation issues with respect to each level of documentation.

There is one other important aspect to a model that also requires documentation, namely the visual interface to running the model. Because such a visual interface is generic to all Modgen models, however, this documentation need is fulfilled by the "Guide to the Modgen Visual Interface". Thus, as a developer, you need not say anything further about this interface as part of your own model documentation.

Primary documentation

As the model developer, you must write your own documentation to provide an overview of your model. You may also choose to provide more technical documentation on the equations used in the model as well—the choice is yours. (Modgen provides substantial assistance to create encyclopedic documentation, as will be seen in the next section, but Modgen itself cannot help you create the contents of the primary documentation)

However, Modgen can integrate any type of documentation that you desire into the model's overall help system. To take advantage of this capability, the documentation that you add must be a .chm (Compiled Help) file format with the name: '<model_name>1_<language_code>.chm'. (As an example, such a help file was created for Statistics Canada's

LifePaths model; it has the name: ‘LifePaths1_EN.chm’) As a rule of thumb, the primary documentation links to the encyclopedic documentation file of the same language, which is a file with the name: ‘<model_name>2_<language_code>.chm’. By virtue of such a link, the encyclopedic documentation thus also becomes available through Modgen’s visual interface.

Whenever a user of the model asks for help, Modgen searches for and tries to open the primary documentation file, if it exists. Otherwise, Modgen tries to open the encyclopedic documentation file, assuming it has been generated and is up-to-date. (If that assumption is false, Modgen gives the model user an opportunity to re-create the file in the current language of the visual interface, thus ensuring that the user is not unknowingly viewing out-of-date help information.)

Encyclopedic documentation

Modgen has a very powerful, built-in capability to automatically generate a documentation file that describes the model’s entire structure. Such documentation is called the “encyclopedic documentation” of a model. Its building blocks include first the languages defined for the model, and then special comments, representing either labels or notes for each symbol, that are inserted into the model’s code.

The following first explains how straightforward it is to generate encyclopedic documentation for a model. Next, there is additional information on the building blocks (languages, notes, labels) of the encyclopedic documentation and a detailed outline of the documentation’s structure or contents.

Creating a model’s encyclopedic documentation

The encyclopedic documentation can be created and viewed through the **Help** menu of a model's visual interface, as described in the Appendix of the “*Guide to the Modgen Visual Interface*”. It can also be generated, in all of the model's languages through the batch mode command line. This section describes the latter two methods.

Creating the documentation using the batch mode command line

The batch execution mode of Modgen models allows you to create encyclopedic documentation in all of the model's languages by entering only one command on the command line.

<u>Syntax to create encyclopedic documentation:</u>

```
model_name.exe –help <-s>
```

Example (to generate English and French help files for Statistics Canada’s LifePaths model):

```
C:\LifePaths\LifePaths.exe –help
```

Once the documentation has been generated in all of the model's languages, a dialog box appears to confirm that the help systems were generated successfully; if “-s”was entered on the command line, however, the dialog box is suppressed.

Known languages of a model

All Modgen models must have at least one language explicitly defined for viewing and documenting the model. You can specify one or more languages by entering the following **languages** definition statement:

Syntax:

```
languages
{
        langcode, // name of language
        …
        langcode // name of language
};
```

It is recommended for simplicity that each language code be two characters in length and entered in upper case, e.g. EN or FR, although you are not forced to use this convention. The model's working language is the first language listed. Example:

```
languages
{
      EN, // English
      FR // Français
};
```

Symbol labels

A label is essentially a brief description of the symbol. If an abbreviation has been used in the symbol's name, the label can potentially be the unabbreviated name. The label can even simply be the name of the symbol itself. But irrespective of its content, a label should be explicitly provided for each symbol in the model. (Modgen will create a label for any symbol that does not have one but since the labels are a key foundation for creating the encyclopedic documentation of a model, it is excellent coding practice to include a label for each symbol in the model. Explicitly-specified labels are also required if you wish to use Modgen's Translation Assistant tool to assist with translating your labels into other languages of your model.)

It is possible, however, to label the same symbol more than once. If this happens, the final label that is found by Modgen (which looks at modules in alphabetical order) is the one that takes precedence.

There are two methods to label a symbol:

- Adding the label as a special comment with the definition of the symbol
- Using the LABEL instruction

The first method (adding the label as a special comment immediately after the definition of the symbol) is generally the preferred method of adding a label because it documents the symbol's meaning in the model's code itself, thus making the code easier to understand. However, a model's modules (i.e. its .mpp files) clearly cannot be labeled via this method since the modules are not defined in the code—instead, they actually contain the code.

The second method (using the LABEL instruction) can be used for all symbols, including modules. Its syntax is:

```
//LABEL ( symbol_name, language_code ) text_of_label
```

In this syntax, *language_code* is the code defined in the **languages** statement for the language, e.g. EN for English or FR for French, while *symbol_name* is simply the symbol name, entered (for the majority of symbol categories) exactly the same way as it is in the .mpp file However, there are exceptions to how *symbol_name* is specified:

- if the symbol is a member of an actor (i.e. a state, event, or actor function), then *symbol_name* starts with the actor's name followed by a period and the actual name of the state, event, or actor function.
- if the symbol is a classificatory dimension of a table, then *symbol_name* starts with the table name followed by a period and the numbered dimension (Dim0, Dim1, Dim2, etc.).
- if the symbol is a table expression, then *symbol_name* starts with the table name followed by a period and the numbered expression (Expr0, Expr1, Expr2, etc.).
- if the symbol is an aggregation, then *symbol_name* is aggregatedclassificationname_detailedclassificationname (note that the two classifications are joined with an underscore)

The *symbol_name* can also be replaced by the keyword **model** if you wish to enter a label for the model itself. The label text (text_of_label) has a maximum of 255 characters per label.

Often, labels are specified in the working language of the model using the first method (thus improving the readability of the code), while the second method is mostly used to label symbols in the other languages of the model. The latter are usually grouped into separate .mpp files (one file for each other language of the model, with a module name that incorporates the model's name and the language). For example, if the languages of a model are English and French, with a working language of English, then by using this approach, there would be a separate 'model_nameFR.mpp' module used to store the various LABEL instructions containing the model's French labels.
Examples:

In the following examples (based on a model with languages English and French, where English is the working language of the model), each first block of code shows the definition of the English labels, using the first method of defining labels. Each second block of code shows the definitions of the corresponding French labels, using the second method—each of these second code blocks would appear within the model's 'model_nameFR.mpp' module.
Example 1 (specifying symbol labels when the symbol name is entered the same way as it is in the .mpp file):

```
classification SEX      //EN Gender
```

```
{
      S_FEMALE,      //EN Female
      S_MALE         //EN Male
};


//LABEL ( SEX, FR ) Sexe
//LABEL ( FEMALE, FR ) Féminin
//LABEL ( MALE, FR ) Masculin
```

Example 2 (specifying symbol labels when the symbol is an actor member):

```
actor Person            //EN Individual
{
…
      int               earnings = {0};         //EN Lifetime earnings to date
      DFLE_STATE    dfle = {NO_DIS};            //EN Disability State
      LIFE_STATE    life_status = {ALIVE};      //EN Life Status
      …
};


//LABEL ( Person, FR ) Individu
//LABEL ( Person.earnings, FR ) Gains totaux à ce jour
//LABEL ( Person.dfle, FR ) Situation-handicapé
//LABEL ( Person.life_status, FR ) Situation-vie
```

Example 3 (specifying symbol labels when the symbol is either a table classificatory dimension or table expression):

```
table Person X08_AvgEarnByAge //EN Cases and their average earnings by age group
{
      split( age, AGE_GROUPS )+  //EN Age Group
      * {
            unit,                  //EN Persons
            earnings / duration()  //EN Avg earnings
      }
};


//LABEL ( X08_AvgEarnByAge, FR ) Cas et gains moyens par groupe d'âge
//LABEL ( X08_AvgEarnByAge.Dim0, FR ) Groupe d'âges
//LABEL ( X08_AvgEarnByAge.Expr0, FR ) Personnes
//LABEL ( X08_AvgEarnByAge.Expr1, FR ) Gains moyens
```

Symbol notes

As a developer, you can also create an additional note for a symbol. There are two types of symbol notes—notes in .mpp files which should describe the usage of a particular symbol, and notes in .dat files which should describe specific parameter values contained therein. (The latter are called value notes in the property windows of Modgen's visual interface).

While there are two methods to create labels, there is only one method or syntax to create a symbol note:

```
/* NOTE (symbol_name, language_code)
   note_text
*/
```

In this syntax, *language_code* is the code defined in the **languages** statement for the language, e.g. EN for English or FR for French, while *symbol_name* is simply the symbol name, entered (for the majority of symbol categories) exactly the same way as it is in the .mpp file   The exceptions for entering *symbol_ name* in this way are identical to those in place when defining labels with the LABEL statement:

- if the symbol is a member of an actor (i.e. a state, event, or actor function), then *symbol_name* starts with the actor's name followed by a period and then the actual name of the state, event, or actor function.
- if the symbol is a classificatory dimension of a table, then *symbol_name* starts with the table name followed by a period and then the numbered dimension (Dim0, Dim1, Dim2, etc.).
- if the symbol is a table expression, then *symbol_name* starts with the table name followed by a period and then

the numbered expression (Expr0, Expr1, Expr2, etc.). Thus, if the expression axis, for example, is the last dimension of the table (i.e. the table's columns), this makes it possible to have detailed notes for each column of the table

- if the symbol is an aggregation, then *symbol_name* is aggregatedclassificationname_detailedclassificationname (note that the two classifications are joined with an underscore)

The *symbol_name* can also be replaced by the keyword **model** if you wish to enter a note for the model itself.

Symbol notes are usually placed near the symbol declaration. This makes the code easier to understand and the note easier to maintain.

Example of a symbol note:

```
/* NOTE (Mortality, EN)

      This module implements death according to a person's mortality
      hazard.  There is also a maximum life expectancy incorporated
      into the model.

*/
```

Model notes can be used to elaborate the use of a symbol, the content of a module, or the details on when a certain event can take place and what happens when that event does take place. However, the notes should not simply reiterate information stored elsewhere, such as in the labels or even other notes; otherwise, they do not really add a lot of significant value to the overall documentation of the model.

Formatting of symbol notes

Symbol notes can be formatted to a certain extent; for example, they can contain paragraphs and lists. The following formatting rules are in place that affect how a note is displayed:

- Spaces and tabs at the beginning of a line are deleted from each line
- A paragraph must be surrounded by blank lines, i.e. there must be at least one blank line before the paragraph and one blank line after it, to confirm that the text will be treated as a paragraph
- A line that starts with a hyphen "-" is considered to be one line or item of a list
- A line that starts with the character ">" will cause all subsequent text on that line, including any spaces or tabs, all to appear, starting on a new line. (The ">" character itself, however, does not get displayed). This is an especially useful mechanism to include indented code inside a note.

Example 1—Text that is entered as:

```
/* NOTE (ExampleSymbol, EN)
     Here is an imaginary paragraph
     which has been indented for illustrative
     purposes.  It is identified as a paragraph because it
     is terminated by a blank line, i.e. \n\n.


    - This is item 1 of an ordered list,
      which has been wrapped because
      it is long.
    - This is item 2 of the list

     Here we have another paragraph.  Had this paragraph started
     immediately following the list, an empty line should still have been
     inserted since paragraphs should have one blank line between themselves
     and preceding or following text.
*/
```

would in fact be displayed as the following:

```
Here is an imaginary paragraph which has been indented for illustrative
purposes.  It is identified as a paragraph because it is terminated by a blank
line, i.e. \n\n.

 - This is item 1 of an ordered list, which has been wrapped because it is long.

 - This is item 2 of the list

Here we have another paragraph.  Had this paragraph started immediately
following the list, an empty line should still have been inserted since
paragraphs should have one blank line between themselves and the preceding or
following text.
```

Example 2—Text that is entered as:

```
/* NOTE ( ExampleSymbol, EN )
Here is an imaginary paragraph which has been indented for illustrative
purposes.  It is identified as a paragraph because it is terminated by a blank
line, i.e. \n\n.

            This is a new paragraph because it is surrounded by blank lines but
>     this line is separated from the rest.

Here we have another paragraph.  Had this paragraph started immediately
following the list, an empty line should still have been inserted since
paragraphs should have one blank line between themselves and preceding or
following text.

*/
```

would be displayed as:

```
Here is an imaginary paragraph which has been indented for illustrative
purposes.  It is identified as a paragraph because it is terminated by a blank
line, i.e. \n\n.

This is a new paragraph because it is surrounded by blank lines but
      this line is separated from the rest.

Here we have another paragraph.  Had this paragraph started immediately
following the list, an empty line should still have been inserted since
paragraphs should have one blank line between themselves and preceding or
following text.
```

Contents of the encyclopedic documentation

The encyclopedic documentation generated by Modgen is produced interactively in a file with the name: ‘<model_name>2_<language_code>.chm’, assuming that the machine has a .chm compiler installed. (Otherwise, the documentation is generated as HTML files inside of folders called “Gen<language_code>” with a start file called ‘2.htm’.)

The encyclopedic documentation documents the model's internal structure and therefore includes all Modgen symbols. Each Modgen symbol in the model has its own documentation page, and links are established at each appropriate location between the different symbols by way of hyperlinks between the various symbol pages. With these links, the reader can understand the relationships between the model's various symbols. The encyclopedic documentation also includes a hierarchical tree of all the symbols in a separate navigational panel.

For each Modgen symbol, the documentation includes the following:

- symbol name
- symbol label
- symbol note (if a note has been defined)
- module in which the symbol has been defined (not applicable for symbols that are modules)

In addition, the following information is included for each symbol category:

Symbol category	**Documentation content**
States	Actor Type State tracking information Dependent derived states Events that read the value Functions that read the value Tables that use it Modules in which the state is read Table representing state dependencies
Simple states	Events that modify the state value Functions that modify the state value

	Modules in which the value is modified
Derived states	Expression text
Data member[1]	Actor
Events	Actor Priority Time function name States read in the time function States read in the implement function States modified in the implement function Functions hooked to the event
Member functions	Actor States read in the function States modified in the function Functions and events to which the function is hooked Functions hooked to this function
Parameters	Type Groups of which the parameter is a member Dependent parameters Extending parameters Dimension information
Groups	Group members
Modules	Types defined in the module Parameters defined in the module Parameter groups defined in the module Tables defined in the module User tables defined in the module

[1] A data member is a multidimensional state that can be modified in the model's code but that, unlike simple states, cannot be used in expressions.

	Table groups defined in the module Simple states read in the module Derived states read in the module Simple states modified in the module
Tables (including user tables)	Table rank Table groups to which the table belongs Classification dimensions Analysis dimensions
Tables	Actor Table filter
Types	States for which this is the type Parameter of which this is the type Parameters of which this is a dimension User tables of which this is a dimension Tables in which the type is used
Classifications	Classification levels
Ranges	Minimum value Maximum value
Partitions	Partition values in the form of intervals
Classification levels	Value Classification
Aggregations	Aggregated classification Detailed classification Mapping between the levels of the aggregated classification and the detailed classification

In addition to symbols, the encyclopedic documentation includes a page to track every actor for which tracking has been defined. This page includes the following information:

- module in which tracking has been defined

- tracking filter
- final tracking filter
- list of tracked states

Documentation related to a model’s scenarios

While primary and encyclopedic documentation are completed during a model's development and creation, the documentation presented in this section is created by the model users or the model developer as part of the basic scenarios provided with the model. When users create a scenario, it is always useful to document that scenario. For example, users may want to document the fact that scenario *x* is used to produce results for report *y*. This will prevent the users from hesitating when questions are asked about the scenario used years after report *y* has been released. Users may also want to document the reason why a certain value of a specific parameter was chosen. Finally, it may be useful for users who group the model's parameters within several .dat files to indicate the logic used to create those groupings of parameters.

Modgen enables three levels of scenario documentation, each initiated through the visual interface of a Modgen model and entered as notes, in order to accommodate the types of situations described above:

- scenario documentation
- parameter file documentation
- parameter value documentation

Irrespective of what type of scenario note you are entering, you can only use the current language of the visual interface. Creating multilingual documentation therefore requires changing the language of the visual interface to enter the documentation in all known languages. Once you are done, you also need to save the scenario afterwards in order to preserve the notes that you have entered. The various rules and techniques outlined in the “Formatting of symbol notes” section also apply to all of these levels of scenario notes as well.

Scenario documentation

First, it is possible to document information about a scenario itself. Such scenario documentation might contain, for example, how the scenario is used or how the parameters were changed to create that scenario. You can enter both a label and a more detailed note for a given scenario; both items are entered via the Documentation tab of the **Scenario > Settings** dialog box in Modgen’s visual interface.

Parameter file documentation

Next, a model user can choose to individually document each parameter file included in the scenario. For example, users can choose to indicate in such a note what categories of parameters are contained in the file. Also, in a complex model with many parameters, users might prefer to indicate, within a note for each parameter file, all of the changes made to the parameters to create the scenario (rather than recording this information in the scenario note). These notes are generated by first highlighting the parameter file in the visual interface and then entering the information into the **View >**

Properties dialog box. (Alternatively, with the parameter file highlighted, you can instead right-click your mouse and then select Properties to access the same dialog box.)

Parameter value documentation

Finally, a model user can choose to document the specific parameter values in a parameter file. This documentation helps users who are returning to a previously used scenario to remember the reason why parameter *x* has value *y*. Again, a basic documentation target is that users who have not worked with a model for a few years but then return to it, should be able to quickly understand what a given scenario contains and why.

There are two types of parameter-related notes. The first is a note about the parameter itself which you defined as part of your model code, using the NOTE syntax, in your role as model developer. The other is a note about some or all of the specific data values for that parameter in the given scenario; such a note is entered from the visual interface. To document parameter values, users must first open the parameter by double-clicking on its name from the tree containing the tables and parameters in the left pane of the visual interface. At that point, by right-clicking in the Parameter window and selecting Properties, a dialog box appears that includes space into which the desired “Value Note” can be entered.

Distribution of a Modgen model

One of the goals in creating a model is often to be able to distribute it to other policy analysts or researchers, either internal or external to your organization. Each organization also usually has its own rules and standards that must be followed in distributing or releasing any product for use by others. The purpose of this section, then, is to review the general issues that should be considered before releasing a model—but while the discussion is essentially general, any specific requirements imposed by Modgen are also included. For example, a model can have (but does not have to have) release notes, and if they exist, their contents are basically at your discretion; however, if they exist, there is a file-naming convention that must be followed so that Modgen's visual interface can display the notes.

The issues to address after a model has been created but before you release it to others are:

- documenting and translating the model
- writing the release notes
- preparing a licence for the use of the model
- determining the exact model contents to release
- creating an installation procedure for the model
- testing the model and the installation procedure

Each of these issues is now discussed in turn.

Documenting and translating the model

The documentation specific to a model has three essential categories or levels:

- Primary: a broad overview of the model and of its underlying assumptions or equations
- Encyclopedic: the detailed structure of the model
- Scenario: notes to describe scenarios or parameter estimation methods

As the model developer, you have to prepare both the Primary overview material and the Scenario notes directly, whereas Modgen can generate the detailed encyclopedic documentation, based on notes and labels that you have provided elsewhere throughout your model code.

The languages of a model are specified via the **languages** command which is usually inserted in the main module of the model. At least one language must be explicitly defined for each model. If there is more than one language, each category of documentation needs to be translated as well. Translations for all of the labels and notes are usually placed in one .mpp file bearing the name of the model and underlying language, i.e. '<model_name>_<language_code>.mpp'**.** The translation for each label and note can also appear beside the definition of the symbol in the corresponding .mpp file. However, the Modgen Translation Assistant helps in the creation of the file '<model_name>_<language_code>.mpp'. The translation of the primary documentation must be placed in a file named '<model_name>1_<language_code>.chm'**.** The translations of the scenario notes must be input through Modgen's visual interface to the model

You should also invoke the "**Help on <ModelName>**" menu item to generate an up-to-date encyclopedic documentation .chm file that can be shipped with your model. This should be done for each language supported if multiple languages are supported by the model. The generated .chm file is named '<ModelName>2_<language_code>.chm', where <ModelName> is the name of the model, and <language_code> is the string identifying the language for models that support multiple languages. For example, the English version of the generated documentation for Statistics Canada's LifePaths model is named 'LifePaths2_EN.chm'. If you do not include an up-to-date .chm file, users of your model will be told the file is out-of-date and be asked if they wish to generate a new file. Thus, there is no danger that they would unknowingly be using an out-of-date help file; nevertheless, it is good practice to ensure that the .chm file they see is already up-to-date before distributing a new release of your model.

Writing the release notes

Each release of your model could include a set of release notes. If your model has release notes, Modgen imposes no restrictions on the contents of these notes but it does require them to exist in a file '<model_name>Notes_<language_code>.htm' so that they can be accessed from the Help menu in Modgen's visual interface to your model. This file must be placed in the same folder that has the executable file for your model.

Again, Modgen does not impose any rules with respect to the exact contents of your model's release notes. As guidelines, however, release notes often include:

- a list of new or modified features contained in this version of the model
- the bugs or problems resolved by this version of the model
- an indication of the known problems or issues that still remain
- a link to the Statistics Canada web page containing the Modgen software

- information on how to contact the model's development team (phone and/or email)
- general instructions on where to find more information about the model

Preparing a license for the use of the model

In many cases, a license agreement also has to be included with a distributed model. The role of the license is that each potential user of your model should first be presented with a copy of its contents; that individual would then have to review and accept the terms and conditions presented in the license before being allowed to proceed with downloading your model. The licence often first focuses on exactly what the user will be downloading (e.g. is the source code included or not?) The license also frequently outlines permitted uses of the model, copies that can be made of the model, ownership of the model, warranty and liability issues, etc.

If a license exists for your model, it can be made accessible in the visual interface through the **Help > About `<model_name>`** menu. In this case, the Modgen interface will look for a file called '<model_name>Licence_<language_code>.htm' in the same folder as your model executable,

Determining the exact model contents to release

There are many different considerations in determining exactly what entities should be contained in the public release of your model. A first consideration is that some of the tables might not be appropriate for releasing, either because they may contain confidential data or because they were creating solely for debugging purposes, You might also wish to hide certain tables or parameters (using the **hide** command) so that the hidden functions and/or parameters would not even appear in the visual interface; this would keep the model's structure intact but simplify the model's visual interface.

You also have the decision of how detailed a tracking file (e.g. how many actors, how many states) you wish to include with your model

From the perspective of appearance, you can also modify the icon that is associated with your model. The default file (which is in the same folder as your model's executable file after you run Modgen for the first time) is called 'app.ico'. You are free to change this file as you wish (as long as you do not change its name); however, after any change, you need to run Modgen 11 and then re-compile your model in order to generate a new version of your model's executable.

Each model also requires at least one scenario (.scex) file, usually called a base scenario file, so you have to decide what parameter values and scenario settings to choose for this base scenario file. Furthermore, it is also your decision whether the results of the simulation run with this base scenario are included with the model, or whether the person downloading the model has to subsequently perform the simulation run directly. In addition, as the developer, you can choose whether or not to also distribute the source code (.mpp files) of your model with the model release.

Model version number

Modgen allows you to assign a version number to your model, and you should have a separate version number for each version of your model that you prepare for distribution. The version number can have up to four separate digits (e.g. version 4.2.5.0). The first digit is the primary or major version number of your model; minor updates of a major version are reflected by keeping the same first digit and modifying either the second or third digits (depending on the nature of the minor updates). Some installation packages only allow three digits for a version number so it usually makes sense to avoid the fourth digit when identifying versions of your model for distribution. However, the fourth digit can be useful to help you distinguish between different internal and interim versions of your model that you work on before ultimately distributing the next official public version.

The version number is set by using the **version** command, usually inside the main module of your model (i.e. the module named 'model_name.mpp'). The command takes an argument of the four separate digits, each separated by a comma. Example :

```
version 4,2,5,0
```

This version number can be accessed from the **Help > About `<model_name>`** menu (or the final icon which is in the shape of a question mark) in Modgen's visual interface to your model. (The corresponding version of Modgen that was used to create your model is shown on the same dialog box.

Copyright Notices

You can add a copyright notice to your model for each of the languages allowed by the model. Modgen enables this capability by supplying a string, S_COPYRIGHT1 whose contents can be specified in an .mpp file (usually the main module of your model, called 'model_name.mpp'). Two lines of text are allowed for each notice, and "\n" is permitted as a line separator. If you specify a copyright notice, it can be viewed via the **Help> About `<model_name>`** menu (or the final icon which is in the shape of a question mark) in Modgen's visual interface to your model
Examples :

```
string S_COPYRIGHT1; //EN LifePaths™ published by authority of the Minister
responsible for Statistics Canada.\n© 1998-2011 Minister of Industry.
```

```
string S_COPYRIGHT1; //FR LifePaths™ est une publication autorisée par le
ministre responsable de Statistique Canada.\n© 1998-2011 Ministre de l'Industrie
```

Creating an installation procedure for your model

There are many commercial applications available to create installation programs for your model, and the purpose of this section is not to focus on any specific application, but instead to discuss the Modgen-related principles that may need to be addressed or incorporated to create a successful installation procedure for your model.

The first step that you should undertake immediately before preparing the installation procedure is to recreate your model (i.e. run Modgen on your source code, then recompile the model again, and run the simulation with the scenario(s) that will be distributed). You should also then recreate the encyclopedic documentation in all of the languages of the model. (It is also good practice to make a copy of your model's .mpp files). In this way, you can make sure that the model's executable is consistent with its source, that the results are consistent with the executable, and that any user of the executable will have access to up-to-date encyclopedic help.

As for the installation procedure that you generate, it should allow the installing individual to work in any of the languages supported by the model. It should not only force the individual to view the license but also force that individual to perform an action (e.g. clicking a button) to actually accept the license. (That is, the default choice behaviour should not be that the license is automatically accepted by simply advancing to the next screen.). It should also check that the appropriate version of Modgen Prerequisites is already installed on the machine, given that no Modgen-generated model can run without Modgen Prerequisites. (If Modgen Prerequisites is not present on the machine, you can also choose whether to include a link to allow it to be downloaded or whether to simply terminate the installation process.)

You also have to decide whether different versions of your model can exist on the same machine side-by-side or whether only one installation of your model is allowed per machine. If more than one version is allowed, the distinction is made internally by a product identifier that you provide to the installation application; it is usually represented externally via the first digit of the model's version number. For example, if you allowed more than one version of your model and if the last three consecutive released versions of your model were 2.7.0.0, 2.8.0.0, and 3.0.0.0, then you would allow someone to install version 3.0.0.0 separately from either 2.7.0.0 or 2.8.0.0 (assuming that they could go directly from 2.7.0.0 to 3.0.0.0 while bypassing 2.8.0.0). As a result, you would also need to incorporate that first digit as part of your model's name so that the two versions could be easily distinguished from one another.

However, in the same example, you would not allow someone to have separate versions of 2.7.0.0 and 2.8.0.0—an individual at 2.7.0.0 who wanted 2.8.0.0 would lose version 2.7.0.0 by virtue of installing version 2.8.0.0 on that machine. (In such a situation, version 2.8.0.0 would likely have bug fixes and/or minor enhancements compared to version 2.7.0.0 so there would be no need to have both versions running on the same machine. As the developer, you would keep the same internal product identifier used for version 2.7.0.0 when you were creating the installation package for version 2.8.0.0.)

In a similar manner, given the same three version numbers, i.e. 2.7.0.0, 2.8.0.0 and 3.0.0.0, and a decision NOT to allow more than one version of your model to exist on the same machine, then the individual who upgraded to version 3.0.0.0 would also end up removing version 2.8.0.0 from his or her machine.

Testing the model and installation procedure

Once the installation procedure has been created, you must ensure that it actually installs the model and enables a simulation to take place under each operating system that you choose to support. It is not necessary to test Modgen's visual interface to the model but you will want to ensure that all of the links (e.g. to your help files or release notes or license text) are functional. You should also ensure that the base scenario(s) provided with your model can be opened and that the resulting simulations can run completely and produce the correct results (thus also assuring you that all of the required .dat files are available)

Reference material

Modgen command summary

This section lists Modgen definitions, keywords, expressions and derived state specifications by the nature of their usage. Its purpose is a syntactic summary of the Modgen language.

The only Modgen functionality available at run-time is the ability to change parameter values and their display order. Any other alterations imply a re-compilation of the model. For example, changing the rank (number of dimensions), shape and type of a parameter will involve a re-compilation. Changing the track statement for the BioBrowser tracking file (e.g. altering the filter or adding another track state for an actor) also involves re-compilation. On the output side, you cannot change tables without re-compilation. However, you can specify at run-time which tables and parameters are to be sent to output.

Definitions and keywords

Definition/keyword	Description	Type of file with definition
actor actor_name { elements};	Declares an actor	.mpp
aggregation aggregated_classification, detailed_classification {…	Declares an aggregation	.mpp
classification classification_name { elements };	Declares a classification type	.mpp
cumrate parameter_name …	Declares a cumrate model parameter	.mpp, .dat
dependency(table_name1, table_name2, <table_name3, ... >);	Establishes a dependency relationship of table_name1 on the remaining tables in	

	the arguments	
event time_function,implementation_function<,priority_code>	Declares an event within an actor declaration	.mpp
extend_parameter parameter1<,parameter2>;	Extends the stored parameter data	.mpp
hide(parameter_name, <parameter_name, parameter_name, …>);	Hides the parameters included in the argument from the simualtion run	
hide(table_name, <table_name, table_name, …>);	Hides the tables included in the argument from the simulation run	.mpp
hook from_function_name, to_function_name <,Priority>;	Declares a function hook into a member function or an event implementation function, with optional relative priority to other hooks to to_function_name.	.mpp
languages …	Declares the languages used by the model	.mpp
link actor_name1 actor_name2;	Establishes a link between actors	.mpp
logical variable_name…	Declares a logical state or parameter	.mpp, .dat
model_type case_based/time_based/cell_based;	Specifies the model type	.mpp
model_generated parameter_name…	Declares a model parameter which is generated	.mpp
parameter_group group_name {name,…,name};	Declares a parameter group	.mpp
parameters { model_parameters };	Declares a set of model parameters	.mpp, .dat
partition partition_name { elements };	Defines a model partition	.mpp

piece_linear parameter_name…	Declares a piece_linear model parameter	.mpp, .dat
range range_name { elements };	Defines a model range	.mpp, .dat
table <sparse> actor_name <table_name> <[filter]> { elements };	Declares a table specification	.mpp
table_group group_name {name,…,name};	Declares a table group	.mpp
TIME state_name…	Declares a state to be type TIME	.mpp
time_type value;	Specifies the type of microsimulation model	.mpp
track actor_name [intial_filter<,final_filter>] { elements };	Declares states to be tracked for a given actor	.mpp
user_table <sparse> table_name {elements};	Declares a user table	.mpp
version value;	Assigns a version number to the model	.mpp

Note 1: If descriptive labels are required, you must use multiple lines.

Note 2: < > denotes optional arguments, … indicates an incomplete statement.

Derived state specifications used in table and actor definitions

Derived state specification	Description	Return type
active_spell_delta(*observed_state, value, differenced_state*)	Change in *differenced_state* since the value of *observed_state* became *value*	real or integer (depends on type of *differenced_state*)
active_spell_duration(*observed_state, value*)	Length of time elapsed since the value of *observed_state* became *value*	TIME
active_spell_weighted_duration (*observed_state, value, weighting_state*)	Sum of *weighting_state* since *obsesrved_state* became *value*	real
aggregate(*observed_state, classification_name*)	Transforms *observed_state* into a state of type *classification_name* according to predefined aggregation rules	classification_name
changes*(observed_state*)	Counts the number of times that the value of *observed_state* has changed	counter
completed_spell_delta(*observed_state, value, differenced_state*)	Change in *differenced_state* over the last period when *observed_state* was *value*	real or integer (depends on type of *differenced_state*)
completed_spell_duration (*observed_state, value*)	Length of time of the last period when *observed_state* was *value*	TIME
completed_spell_weighted_duration (*observed_state, value, weighting_state*)	Sum of *weighting_state* over the last period when the value of *observed_state* was *value*	real
duration*(observed_state, value*)	Total length of time the value of *observed_state* has been *value*; it equals the total length of time the actor has existed so far if *observed_state* and *value* are both omitted	TIME
duration_counter (*observed_state, value, time_interval, max_time_interval*)	Counts the number of time intervals of size *time_interval* that have elapsed since *observed_state* became *value*, up to the optional *max_time_interval*	counter
duration_trigger(*observed_state, value, time_interval)*	TRUE or 1 once the length of time that *observed_ state* has been *value* exceeds *time_interval* (up until *observed_state* changes	logical

	its value again)	
entrances(*observed_state*,*value*)	Counts the number of times that *observed_state* has become *value* so far in the actor's simulated life	counter
exits(*observed_state*,*value*)	Counts the number of times that *observed_state* stopped being *value* so far in the actor's simulated life	counter
self_scheduling_int(*observed_state*)	The integer part of *observed_state*. Creates an event at exact boundaries for coherence. Possible values for *observed_state* are restricted, e.g. age, time, duration().	counter
self_scheduling_split(*observed_state, partition_name*)	Partitions *observed_state* according to the values defined in *partition_name* in order to create groups. Creates an event at exact boundaries for coherence. Possible values for *observed_state* are restricted, e.g. age, time, duration().	*partition_name* (when used in table declaration); integer when used in explicit declaration of derived state
split(*observed_state, partition_name*)	Partitions *observed_state* according to the values defined in *partition_name* in order to create groups	*partition_name* (when used in table declaration); integer when used in explicit declaration of derived state
transitions(*observed_state value1,value2)*	Count the number of times that *observed_state* changed from *value1* to *value2* so far in the actor's simulated life	counter
trigger_changes(*observed_state*)	TRUE whenever the value of *observed_state* changes; FALSE as soon as the time changes again	logical
trigger_entrances(*observed_state, value*)	TRUE whenever the value of *observed_state* becomes *value*; FALSE as soon as the time changes again	logical

trigger_exits(*observed_state, value*)	TRUE whenever the value of *observed_state* stops being *value*; FALSE as soon as the time changes again	logical
trigger_transitions(*observed_state, value1,value2*)	TRUE whenever the value of *observed_state* changes from *value1* to *value2*; FALSE as soon as the time changes again	logical
undergone_change(*observed_state*)	FALSE until *observed_state* changes for the first time; TRUE thereafter	logical
undergone_entrance(*observed_state, value*)	FALSE until *observed_state* becomes equal to *value*; TRUE thereafter	logical
undergone_exit(*observed_state, value*)	FALSE until *observed_state* stops being equal to *value*; TRUE thereafter	logical
undergone_transition(*observed_state, value1,value2*)	FALSE until *observed_state* changes from *value1* to *value2*; TRUE thereafter	logical
value_at_changes(*observed_state, summed_state)*	Sum of the values of *summed_state* over each time that the value of *observed_state* has changed its value in the actor's simulated life.	real or integer (depends on type of *summed_state*)
value_at_entrances*(observed_state, value, summed_state*)	Sum of the values of *summed_state* over each time that the value of *observed_state* has become *value* in the actor's simulated life.	real or integer (depends on type of *summed_state*)
value_at_exits(*observed_state, value, summed_state*)	Sum of the values of *summed_state* over each time that the value of *observed_state* stopped being *value* in the actor's simulated life.	real or integer (depends on type of *summed_state*)
value_at_transitions(*observed_state, value1, value2,summed_state*)	Sum of the values of *summed_state* over each time that the value of *observed_state* changed from *value1* to *value2* in the actor's simulated life.	real or integer (depends on type of *summed_state*)
value_at_first_change(*observed_state, returned_state*)	Value of *returned_state* the first time that *observed_state* changed in the actor's simulated life	same type as *returned_state*
value_at_first_entrance*(observed_state, value, returned_state)*	Value of *returned_state* the first time that *observed_state* became *value* in the actor's	same type as *returned_state*

	simulated life	
value_at_first_exit*(observed_state, value, returned_state)*	Value of *returned_state* the first time that *observed_state* stopped being *value* in the actor's simulated life	same type as *returned_state*
value_at_first_transition(*observed_state ,value1, value2,returned_state*)	Value of *returned_state* the first time that *observed_state* changed from *value1* to *value2* in the actor's simulated life	same type as *returned_state*
value_at_latest_change(*observed_state, returned_state*)	Value of *returned_state* the latest time that *observed_state* changed in the actor's simulated life	same type as *returned_state*
value_at_latest_entrance*(observed_state , value, returned_state)*	Value of *returned_state* the latest time that *observed_state* became *value* in the actor's simulated life	same type as *returned_state*
value_at_latest_exit*(observed_state, value, returned_state)*	Value of *returned_state* the latest time that *observed_state* stopped being *value* in the actor's simulated life	same type as *returned_state*
value_at_latest_transition(*observed_state,value1, value2,returned_state*)	Value of *returned_state* the latest time that *observed_state* changed from *value1* to *value2* in the actor's simulated life	same type as *returned_state*
weighted_cumulation*(cumulated_state, weighting_state)*	Sum of the product of *cumulated_state* and *weighting_state* over the entire simulated life of the actor	real
weighted_duration(*observed_state, value, weighting_state*)	Sum of *weighting_state* over the periods of time that *observed_state* has been *value* so far in the actor's life, summed over all such time periods; the sum is over the actor's entire simulated life if *observed_state* and *value* are both omitted.	real

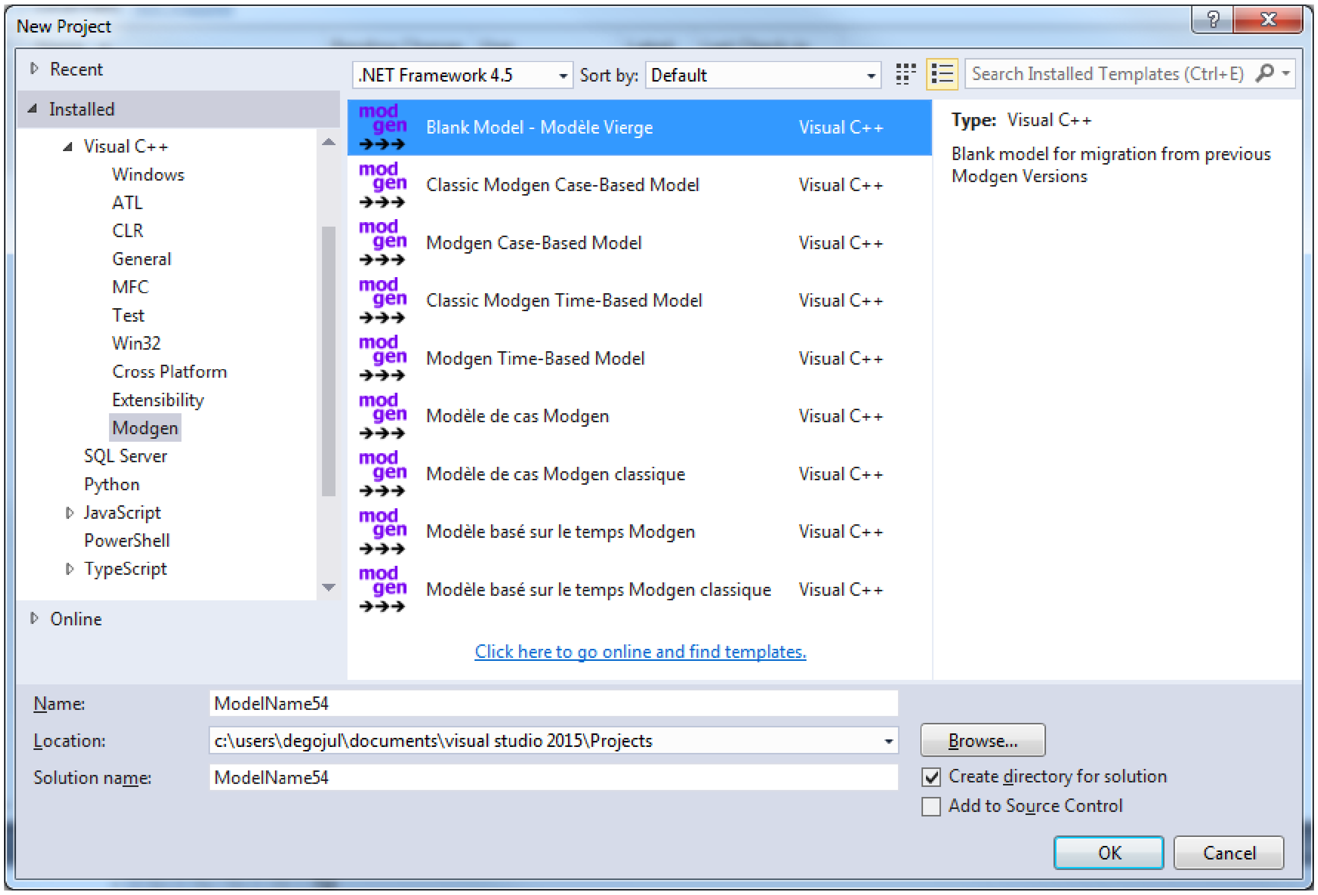
Commands and functions used with links

Command/function	Description
Add(actor_pointer)	Adds an actor of the right type to the link
count(*link_name*)	Returns the number of actors linked by *link_name*
IMPLEMENT_HOOK	Location of the function hook
FinishAll( )	Call the finish function
***GetNext**(int *nInitPos*, int **pnPos*)	Returns the position of the next actor in a multiple link
max_over(*link_name*, *maxed_state*)	Maximum value of *maxed_ state* over all actors linked by *link_name*
min_over*(link_name*, *minned_state*)	Minimum value of *minned_ state* over all actors linked by *link_name*
Remove(int nPosition)	Removes the actor at the specified position from the link
RemoveAll( )	Removes all the actors from the link
sum_over(*link_name*, *summed_state*)	Sum of the values of *summed_state* for each actor currently linked by *link_name*

Appendix: Converting Modgen models to Visual Studio 2015

If you have any models created with Modgen 11, their corresponding projects need to be converted to Visual Studio 2015 so that the models can be compiled under Modgen 12. This section explains how to create a new version of a model created or updated with Visual Studio 2010 to one usable with Visual Studio 2015 and Modgen 12.

Start the Visual Studio 2015 application. Choose **New** and then **Project** from the **File** menu. In the New Project dialog's left-hand tree, navigate to Templates > Visual C++ > Modgen. Select the Blank Model template and specify a folder and a name for the new Model:


New Project

- Recent
- Installed
  - Visual C++
    - Windows
    - ATL
    - CLR
    - General
    - MFC
    - Test
    - Win32
    - Cross Platform
    - Extensibility
    - Modgen
  - SQL Server
  - Python
  - JavaScript
  - PowerShell
  - TypeScript
- Online

.NET Framework 4.5 Sort by: Default

Search Installed Templates (Ctrl+E)

Blank Model - Modèle Vierge	Visual C++
Classic Modgen Case-Based Model	Visual C++
Modgen Case-Based Model	Visual C++
Classic Modgen Time-Based Model	Visual C++
Modgen Time-Based Model	Visual C++
Modèle de cas Modgen	Visual C++
Modèle de cas Modgen classique	Visual C++
Modèle basé sur le temps Modgen	Visual C++
Modèle basé sur le temps Modgen classique	Visual C++

Click here to go online and find templates.

Type: Visual C++

Blank model for migration from previous Modgen Versions

Name: ModelName54

Location: c:\users\degojul\documents\visual studio 2015\Projects

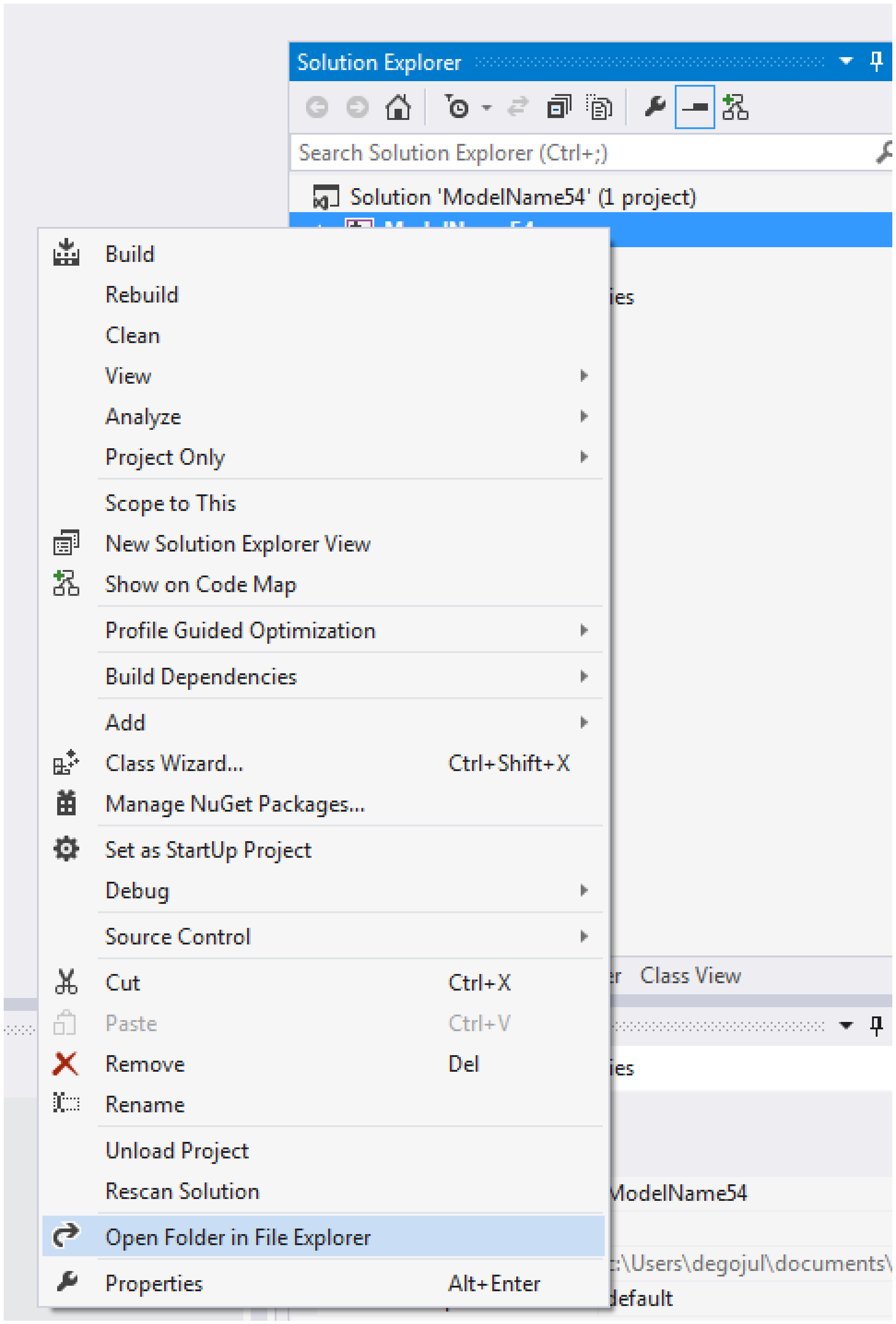
Solution name: ModelName54

Create directory for solution [✓]

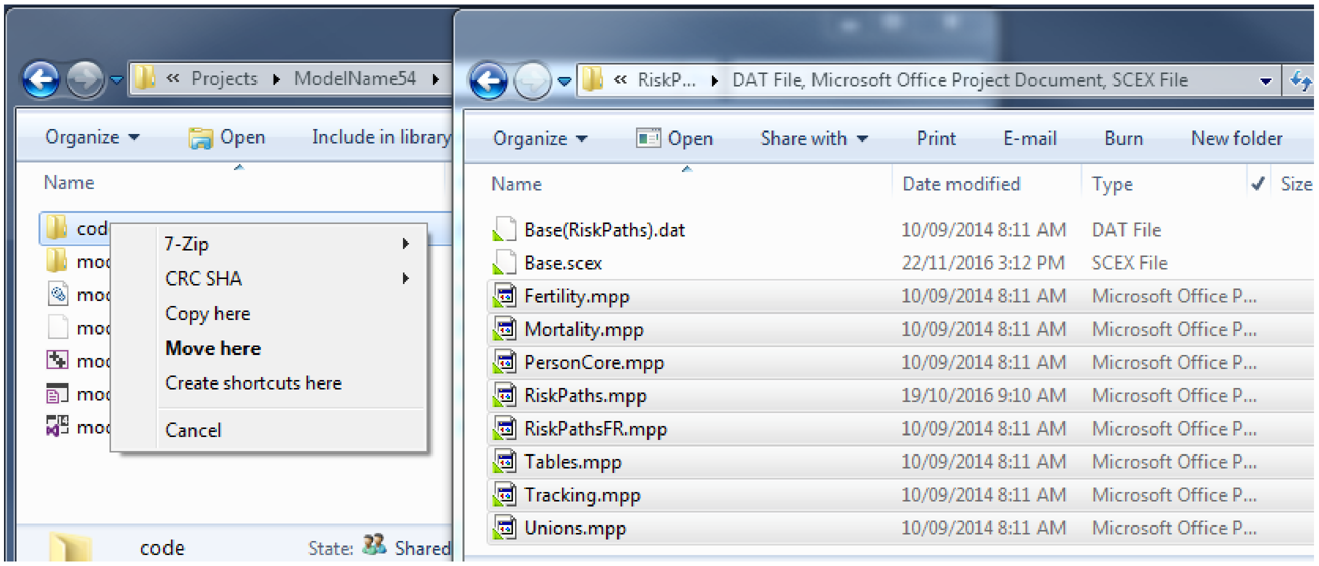
Add to Source Control [ ]

Click “OK” to proceed. This will create a new empty model.

Right-click on the Project in Solution Explorer and select Open Folder in File Explorer:

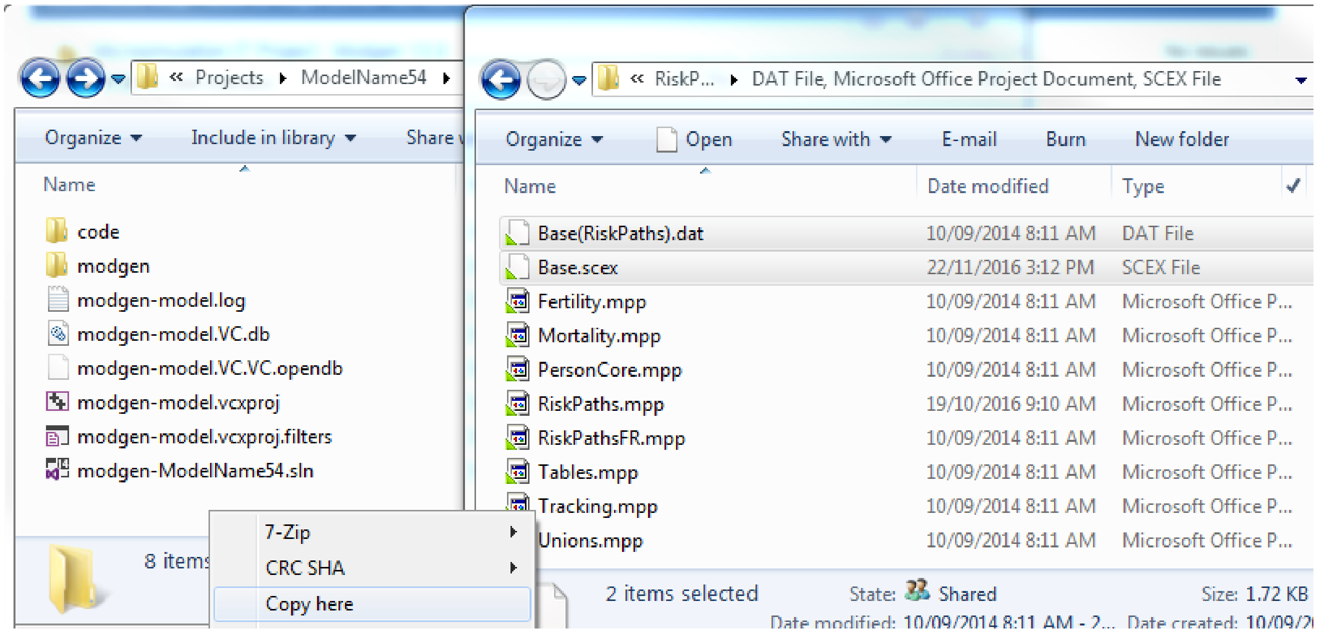


Copy all the mpp files to the **code** subdirectory of the opened folder:


« Projects ▸ ModelName54 ▸
Organize ▾ Open Include in library
7-Zip
CRC SHA
Copy here
Move here
Create shortcuts here
Cancel
code State: Shared
« RiskP... ▸ DAT File, Microsoft Office Project Document, SCEX File
Organize ▾ Open Share with ▾ Print E-mail Burn New folder

Name	Date modified	Type
Base(RiskPaths).dat	10/09/2014 8:11 AM	DAT File
Base.scex	22/11/2016 3:12 PM	SCEX File
Fertility.mpp	10/09/2014 8:11 AM	Microsoft Office P...
Mortality.mpp	10/09/2014 8:11 AM	Microsoft Office P...
PersonCore.mpp	10/09/2014 8:11 AM	Microsoft Office P...
RiskPaths.mpp	19/10/2016 9:10 AM	Microsoft Office P...
RiskPathsFR.mpp	10/09/2014 8:11 AM	Microsoft Office P...
Tables.mpp	10/09/2014 8:11 AM	Microsoft Office P...
Tracking.mpp	10/09/2014 8:11 AM	Microsoft Office P...
Unions.mpp	10/09/2014 8:11 AM	Microsoft Office P...

Copy the scex and dat files to the project directory:

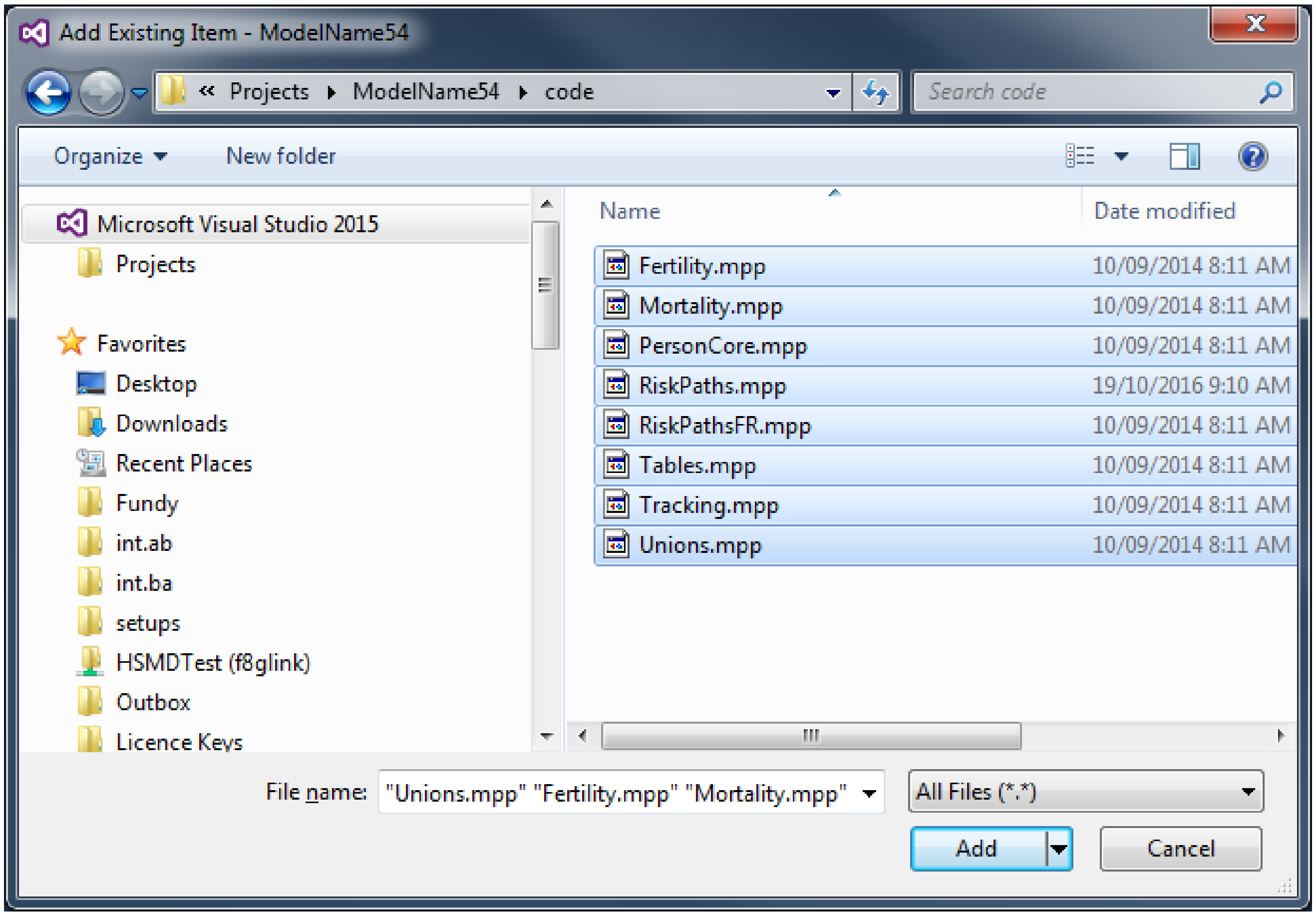

« Projects ▸ ModelName54 ▸
Organize ▾ Include in library ▾ Share
8 items
7-Zip
CRC SHA
Copy here
« RiskP... ▸ DAT File, Microsoft Office Project Document, SCEX File
Organize ▾ Open Share with ▾ E-mail Burn New folder

Name
code
modgen
modgen-model.log
modgen-model.VC.db
modgen-model.VC.VC.opendb
modgen-model.vcxproj
modgen-model.vcxproj.filters
modgen-ModelName54.sln

Name	Date modified	Type
Base(RiskPaths).dat	10/09/2014 8:11 AM	DAT File
Base.scex	22/11/2016 3:12 PM	SCEX File
Fertility.mpp	10/09/2014 8:11 AM	Microsoft Office P...
Mortality.mpp	10/09/2014 8:11 AM	Microsoft Office P...
PersonCore.mpp	10/09/2014 8:11 AM	Microsoft Office P...
RiskPaths.mpp	19/10/2016 9:10 AM	Microsoft Office P...
RiskPathsFR.mpp	10/09/2014 8:11 AM	Microsoft Office P...
Tables.mpp	10/09/2014 8:11 AM	Microsoft Office P...
Tracking.mpp	10/09/2014 8:11 AM	Microsoft Office P...
Unions.mpp	10/09/2014 8:11 AM	Microsoft Office P...

2 items selected State: Shared Size: 1.72 KB
Date modified: 10/09/2014 8:11 AM - 2... Date created: 10/09/2

Back in Visual Studio, right-click on Modules (mpp) and select Add > Existing Item:



Name	Date modified
Fertility.mpp	10/09/2014 8:11 AM
Mortality.mpp	10/09/2014 8:11 AM
PersonCore.mpp	10/09/2014 8:11 AM
RiskPaths.mpp	19/10/2016 9:10 AM
RiskPathsFR.mpp	10/09/2014 8:11 AM
Tables.mpp	10/09/2014 8:11 AM
Tracking.mpp	10/09/2014 8:11 AM
Unions.mpp	10/09/2014 8:11 AM

File name: "Unions.mpp" "Fertility.mpp" "Mortality.mpp"

All Files (*.*)

Select all the mpp files in the **code** subdirectory and click Add.

Repeat the last two steps to add the scex and dat files to the **Scenarios** filter.

The model is ready to be compiled and run.

Special Considerations for Models that Contain Custom .h and .cpp Files

Some models make use of libraries or types that are defined outside of .mpp files. These files also need to be copied to the new model directory in the same way as the mpp files shown in the previous section, in the **code** subdirectory of the model.

The cpp files should be added in the same way as for .mpp files, but under the C++ filter instead of Modules (mpp). This is not necessary for h files.

Appendix: Modgen databases

Modgen Documentation Database

The Modgen model designer has the ability to document, through a short descripitive label as well as through more thorough descriptive text, every element (actors, states, parameters, events, tables, etc.) of a particular model. If the designer provides this descriptive text in multiple languages, the resulting model can produce outputs in the language selected by the user a run-time. The documentation database captures all such information as an MS Access database consisting of the following tables:

Table: *VersionInfo*
Description: File version.number of the documentation database (not of Modgen nor of the model itself). The current version number is 1.7.
Fields: *Version*

Table: *LanguageDic*
Description: Dictionary of supported languages. Selected run-time language is indicated in *Selected* field. Translations of a few important Modgen keywords are also included here (in fields *All, Min, Max*, *Contents*, *Parameters*).
Fields: *LanguageID* (numeric), *LanguageCode* (e.g. "EN" for English), *LanguageName*, *All, Min, Max, Contents, Parameters*

Table: *ModelDic*
Description: Information to document the model itself
Fields: *Name, Description, Note, DescGenerated, NoteGenerated, ModelType, Version, LanguageID, ContextID*
where:

DescGenerated has the values:

0	label is specified in that language by the user
1	label is generated by taking the label specified in the default language
2	label is generated by using the name of the model

NoteGenerated has the values:

0	note is generated in that language by the user
1	note is generated by taking the note specified in the default language

Table: *AutoParserErrorDic*

Description: Descriptions of errors returned by Modgen Autoparser OLE server.

Fields: *ErrorID, ErrorDescription, LanguageID*

Table: *TypeDic*

Description: Dictionary of all variable types used in a model. Includes simple types, classifications, ranges and partitions. For each type more information can be obtained from the more specific dictionary (identified by *DicID* field) by using *TypeID* field.

Fields: *TypeID, DicID* (0 - *SimpleTypeDic*, 1 - *LogicalDic*, 2 - *ClassificationDic*, 3 - *RangeDic*, 4 - *PartitionDic,* 5 - *LinkTypeDic*)

Table: *SimpleTypeDic*

Description: Dictionary of simple data types: int, long, float, double, TIME, cumrate, piece_linear.

Fields: *TypeID* (same as in *TypeDic*), *Name*

Table: *LogicalDic*

Description: Contains the values and descriptions associated with “logical” type.

Fields: *TypeID, Name, Value, ValueName, ValueDescription, LanguageID*

Table: *ClassificationDic*

Description: Dictionary of all classifications.

Fields: *TypeID, Name, Description, Note, SourceFileID, NumberOfValues, LanguageID, ContextID*

Table: *AggregationDic*

Description: Dictionary of all aggregations.

Fields: *Name, AggregatedClassificationID, DetailedClassificationID*

Aggregation tables (named according to *AggregationDic*)

Description: Detailed mapping of a specific aggregation.

Fields: *DetailedValue, AggregatedValue*

Table: *RangeDic*

Description: Dictionary of all ranges.

Fields: *TypeID, Name, Description, Note, SourceFileID, Min, Max, LanguageID, ContextID*

Table: *PartitionDic*

Description: Dictionary of all partitions.

Fields: *TypeID, Name, Description, Note, SourceFileID, NumberOfValues, LanguageID, ContextID*

Table: *LinkTypeDic*

Description: Dictionary of all link types.

Fields: *TypeID, Name, LinkedActorID*

Table: *ClassificationValueDic*

Description: Dictionary of all members of all classifications.

Fields: *TypeID, EnumValue, Name, Description, Note, LanguageID, ContextID*

Table: *PartitionValueDic*

Description: Dictionary of all breakpoints of all partitions. Each breakpoint is stored in numeric and string format.

Fields: *TypeID, Position, Value, StringValue*

Table: *ParameterDic*

Description: Dictionary of all parameters.

Fields: *ParameterID, Name, Description, Note, SourceFileID, TypeID, Rank, NumberOfDatapoints,* *NumberOfCumulatedDimensions*, *ModelGenerated* (Yes/ No), *Extendible* (Yes/ No), *Hidden* (Yes/ No), *RelatedParameterID, LanguageID, ContextID*

Table: *ParameterDimensionDic*

Description: Dictionary of all dimensions of all parameters

Fields: *ParameterID, Position*, *TypeID, DisplayPosition*

Table: *ParameterDependencyDic*

Description: Dictionary of parameter dependencies in parameter extensions

Fields: *ParameterID, DependentParameterID*

Table: *ParameterGroupDic*

Description: Dictionary of all parameter groups, including model-generated parameter groups

Fields: *ParameterGroupID*, *Name*, *Description*, *ModelGenerated* (Yes/ No), *Note, LanguageID, ContextID*

Table: *ParameterGroupMemberDic*

Description: Dictionary of all members of all parameter groups.

Fields: *ParameterGroupID*, *Position*, *ParameterID, MemberGroupID*

Table: *ActorDic*

Description: Dictionary of all actors.

Fields:

- ActorID (integer)
- Name (text)
- Description (text)
- Note (memo)
- LanguageID (integer)
- ContextID (long integer)
- DescGenerated (integer)
- NoteGenerated (integer)

Table: *ActorStateDic*

Description: Dictionary of all states of all actors.

Fields: *StateID* (unique across all actors), *Name*, *Description*, *Note*, *SourceFileID*, *ActorID*, *TypeID*, *KindOfState* (0 - simple, 1 - derived, 2 – identity expression, 3 - linked, 4 – generated expression for tabulation or cumulation operators), LanguageID, *ContextID*

Table: *ActorStateDependencyDic*

Description: Dictionary of state dependencies.

Fields: *StateID, DependentStateID*

Table: *ActorUnknownStateDic*

Description: Dictionary of unknown actor states encountered while generating the list of states read/set by a function or an event. This could be due to complicated pointer expressions involving actor states.

Fields: *StateID, Name*

Table: *ActorEventDic*

Description: Dictionary of all events of all actors.

Fields:

- ActorID (integer)
- EventID (integer)
- WaitFnName (text)
- ImplementFnName (text)
- Priority (integer)
- Description (text)
- Note (memo)
- SourceFileID (of declaration) (integer)
- LanguageID (integer)
- ContextID (long integer)
- DescGenerated (integer)
- NoteGenerated (integer)

Table: ActorEventStateDic

Description: Dictionary of input states of all actor events.

Fields: *ActorID*, *EventID*, *StateID*

Table: ActorEventStateReadDic

Description: Dictionary of all states read by the implement function of all actor events

Fields: *ActorID*, *EventID*, *StateReadID*

Table: ActorEventStateSetDic

Description: Dictionary of all states set by the implement function of all actor events.

Fields: *ActorID*, *EventID*, *StateSetID*

Table: *ActorFunctionDic*

Description: Dictionary of all actors functions (excluding events).

Fields: *ActorID*, *FunctionID*, *Name*, *Description*, *Note*, *SourceFileID* (of declaration), *Declaration*, *LanguageID*, *ContextID*

Table: *ActorFunctionStateReadDic*

Description: Dictionary of all states read by all actor functions.

Fields: *ActorID*, *FunctionID*, *StateReadID*

Table: *ActorFunctionStateSetDic*

Description: Dictionary of all states set by all actor functions.

Fields: *ActorID*, *FunctionID*, *StateSetID*

Table: *ActorFunctionHookedFunctionDic*

Description: Dictionary of all function hooks to actor functions.

Fields: *ActorID*, *FunctionID*, *HookedFunctionID*

Table: *ActorEventHookedFunctionDic*

Description: Dictionary of all function hooks to actor events.

Fields: *ActorID*, *EventID*, *HookedFunctionID*

Table: *ActorLinkDic*

Description: Dictionary of links between the actors.

Fields: *ActorID, LinkID, TypeID, Name, Description, Note, SourceFileID, Single* (Yes/ No), *LinkedActorID, LinkedLinkID, LanguageID, ContextID*

Table: *TableDic*

Description: Dictionary of all output tables.

Fields: *TableID*, *Name*, *Description*, *Note*, *SourceFileID*, *ActorID*, *Rank*, *AnalysisDimensionPosition, AnalysisDimensionName*, *AnalysisDimensionDescription*, *AnalysisDimensionNote*, *Sparse*, *Hidden* (Yes/ No), *LanguageID, ContextID*

Table: *TableClassDimDic*

Description: Dictionary of all classificatory dimensions of all tables.

Fields: *TableID, Position*, *StateID*, *Name*, *Description*, *Note*, *TypeID*, *Totals* (Yes/ No), *LanguageID*

Table: *TableDependencyDic*

Description: Dictionary to document dependencies between tables and other tables or table groups. Each entry in the table includes the dependent table identifier plus the identifier of the table or table group on which it depends

Fields: *DependentTableID, TableID*, *TableGroupID*

Table: *TableExpressionDic*

Description: Dictionary of all expressions of all tables.

Fields: *TableID*, *ExpressionID*, *Name*, *Description*, *Note*, *ExpressionString, Scale, Decimals, LanguageID, DescGenerated*

Here are the values of DescGenerated and their meanings. (Note that this information is primarily of use for tools that use the documentation database to facilitate the translation of a model into a different language.)

0 = the label was specified in the code (in the same language)

1 = the label was generated by Modgen (e.g. for a simple state, the label is the name of the state)

2 = the label was generated by Modgen by taking the label specified in the code for another language

3 = the label was generated by Modgen by taking the label of a state in the same language

4 = the label was generated by Modgen by taking the label of a state in another language

5 = the label was generated by Modgen by taking the Modgen-generated label of another state

Modgen creates descriptive labels according to the following logic:

- If there is a label for the dimension, then DescGenerated is set to 0.
- If there is no label for the dimension in the required language, but there is a label for the dimension in the default language, then this label is shown and DescGenerated is to 2.
- If there is no label in the required language nor in the default language then:
    - if there is a label for the state in the required language, that label is used and DescGenerated is set to 3 if it's not a generated label :
    - If the label is the generated label, then DescGenerated is set to 5.
- If there is no label for the state in the required language, then the label for the default language is used:
    - DescGenerated is set to 4 if that's not a generated label
    - DescGenerated is set to 5 if it is a generated label

Table: *TableRefStateDic*
Description: Dictionary of all states referenced in all tables.
Fields: *TableID, StateID*

Table: *TableGroupDic*
Description: Dictionary of all table groups.
Fields: *TableGroupID, Name, Description, Note, LanguageID, ContextID*

Table: *TableGroupMemberDic*
Description: Dictionary of all members of all table groups.
Fields: *TableGroupID, Position, TableID, MemberGroupID*

Table: *ModuleDic*
Description: Dictionary of all source files (usually mpp files) with file notes attached. This table was formerly known as SourceFileDic.
Fields: *SourceFileID, FileName, FileNote, LanguageID, ContextID*

Table: *UserTableDic*

Description: Dictionary of all user tables.

Fields: *TableID* (starts at 1000), *Name*, *Description*, *Note*, *SourceFileID*, *Rank*, *AnalysisDimensionPosition*, *AnalysisDimensionName*, *AnalysisDimensionDescription*, *AnalysisDimensionNote*, *Sparse*, *Hidden* (Yes/ No), *LanguageID*, *ContextID*

Table: *UserTableClassDimDic*

Description: Dictionary of all classificatory dimensions of user tables.

Fields: *TableID*, *Position*, *Name*, *Description*, *Note*, *TypeID*, *Totals* (Yes/ No), *LanguageID*

Table: *UserTableExpressionDic*

Description: Dictionary of all expressions of user tables.

Fields: *TableID*, *ExpressionID*, *Name*, *Description*, *Note*, *Scale*, *Decimals*, *LanguageID*, *DescGenerated*

Here are the values of DescGenerated and their meanings. (Note that this information is primarily of use for tools that use the documentation database to facilitate the translation of a model into a different language.)

0 = the label was specified in the code (in the same language)

1 = the label was generated by Modgen (e.g. for a simple state, the label is the name of the state)

2 = the label was generated by Modgen by taking the label specified in the code for another language

3 = the label was generated by Modgen by taking the label of a state in the same language

4 = the label was generated by Modgen by taking the label of a state in another language

5 = the label was generated by Modgen by taking the Modgen-generated label of another state

Modgen creates descriptive labels according to the following logic:

- If there is a label for the dimension, then DescGenerated is set to 0.
- If there is no label for the dimension in the required language, but there is a label for the dimension in the default language, then this label is shown and DescGenerated is to 2.
- If there is no label in the required language nor in the default language then:
    - if there is a label for the state in the required language, that label is used and DescGenerated is set to 3 if

    it’s not a generated label :

    - If the label is the generated label, then DescGenerated is set to 5.

- If there is no label for the state in the required language, then the label for the default language is used:
    - DescGenerated is set to 4 if that’s not a generated label
    - DescGenerated is set to 5 if it is a generated label

Modgen Output Database

The Modgen output database is an MS Access database which contains the following tables:

Table: *VersionInfo*
Description: File version number (of the output database, not of Modgen nor of the model itself). The current version number is 1.7.
Fields: *Version*

Table: *LanguageDic*
Description: Language dictionary, same as the one in documentation database. Contains the list of supported languages and translations of some Modgen-generated words encountered in outputs.
Fields: *LanguageID, LanguageCode, LanguageName, All, Min, Max*

Table: *ModelDic*
Description: Contains information on the model itself
Fields: *Name, Description, Note, ModelType (0=case-based, 1=time_based), Version, LanguageID*

Table: *SimulationInfoDic*
Description: Information about the run of the simulation which created the output database
Fields: *Time, Directory, CommandLine, CompletionStatus, FullReport, Cases, LanguageID*

where *Cases* is the number of cases (case-based model) or replicates (time-based model) completed in the simulation

Table: *TableDic*

Description: List of non-excluded output tables

Fields: *TableID, Name, Description, Note, Rank, AnalysisDimensionPosition, AnalysisDimensionName, AnalysisDimensionDescription, AnalysisDimensionNote*, *Sparse, LanguageID*

Table: *TableClassDimDic*

Description: List of classificatory dimensions for all output tables.

Fields: *TableID*, *Position*, *Name*, *Description*, *Note*, *TypeID*, *Totals* (Y/N), *LanguageID*

Table: *TableExpressionDic*

Description: List of expressions used in all output tables.

Fields: *TableID, ExpressionID, Name*, *Description*, *Note*, *Decimals, LanguageID*

Table: UserTableDic

Description: Dictionary of all user tables.

Fields: *TableID* (starts at 1000), *Name, Description, Note, Rank, AnalysisDimensionPosition*, *AnalysisDimensionName, AnalysisDimensionDescription, AnalysisDimensionNote*, *Sparse, LanguageID*

Table: *UserTableClassDimDic*

Description: Dictionary of all classificatory dimensions of user tables.

Fields: *TableID*, *Position*, *Name*, *Description*, *Note, TypeID, Totals* (Yes/ No), *LanguageID*

Table: *UserTableExpressionDic*

Description: Dictionary of all expressions of user tables.

Fields*: TableID, ExpressionID, Name, Description, Note, Decimals, LanguageID*

Table: <*table_name*>

Description: The actual data for the output table indicated by <*table_name*> (regular or user table).

Fields: *Dim0*, ..., *DimN*, *Value*, *CV*, *SE*

If the simulation was run with more than 1 sub-sample and the model type is not cell-based then additional fields containing sub-sample values are present: *Value0, Value1, ..., ValueM* where *M* is one less than the number of sub-samples.

Table: *ParameterDic*

Description: List of output parameters.

Fields: *ParameterID, Name, Description, Note, ValueNote, DataFileID,, TypeID, Rank, LanguageID*

Table: *ParameterDimensionDic*

Description: List of dimensions for all output parameters.

Fields: *ParameterID, DisplayPosition, TypeID*

Table: <*parameter_name* >

Description: The actual data for the output parameter indicated by <*parameter_name*>

Fields: *Dim0, ..., DimN, Value*

Table: *ScenarioDic*

Description: Information on the scenario, including all of that found in the "General" and "Documentation" tabs of the Scenario/Settings dialog box in the visual interface

Fields: *Name, Description, Note, DescGenerated, NoteGenerated, Subsamples, Cases, Threads, Seed, ComputationPriority, BackgroundMode, PartialReports, TimeInterval, PopulationScaling, PopulationSize, DistributedExecution, SubsamplesPerMachine, AccessTracking, TextTracking, MaxTrackedCases, CopyParameters, MemoryReport, LanguageID where:*

DescGenerated has the values:

3	label is specified in that language by the user
4	label is generated by taking the label specified in the default language
5	label is generated by taking the name of the scenario

NoteGenerated has the values:

2	note is generated in that language by the user
3	note is generated by taking the note specified in the default language

Subsamples is the number of subsamples (case-based model) or the number of replicates (time-based model)

Cases is the number of cases (case-based model, as integer) or the end time (time-based model, as double)

PopulationScaling is used only for case-based models and remains empty for time-based models

PopulationSize is used only for case-based models and remains empty for time-based models

MaxTrackedCases is used only for case-based models and remains empty for time-based models

Table: *TypeDic*

Description: List of all types used by output tables and/or output parameters.

Fields: *TypeID, DicID* (0 - *SimpleTypeDic*, 1 - *LogicalDic*, 2 - *ClassificationDic*, 3 - *RangeDic*, 4 - *PartitionDic*, 5 - *LinkTypeDic*)

Table: *SimpleTypeDic*

Description: List of simple data types used by output tables and/or output parameters.

Fields: *TypeID, Name*

Table: *LogicalDic*

Description: Contains the values and descriptions associated with "logical" type.

Fields: *TypeID, Name, Value, ValueName, ValueDescription, LanguageID*

Table: *ClassificationDic*

Description: List of classifications used by output tables and/or output parameters.

Fields: *TypeID*, *Name*, *Description*, *Note*, *NumberOfValues*, *LanguageID*

Table: *AggregationDic*

Description: Dictionary of all aggregations.

Fields: *Name, AggregatedClassificationID, DetailedClassificationID*

Aggregation tables (named according to *AggregationDic*)

Description: Detailed mapping of a specific aggregation.

Fields: *DetailedValue, AggregatedValue*

Table: *ClassificationValueDic*

Description: List of all classification members.

Fields: *TypeID*, *EnumValue*, *Name*, *Description*, *Note*, *LanguageID*

Table: *RangeDic*

Description: List of ranges used by output tables and/or output parameters.

Fields: *TypeID, Name, Description, Note, Min, Max, LanguageID*

Table: *PartitionDic*

Description: List of partitions used by output tables and/or output parameters.

Fields: *TypeID, Name, Description, Note, NumberOfValues, LanguageID*

Table: *PartitionValueDic*

Description: List of breakpoints of all partitions.

Fields: *TypeID, Position, Value, StringValue*

Table: *LinkTypeDic*

Description: List of link types used by output tables and/or output parameters.

Fields: *TypeID, Name, LinkedActorID*

Table: *DataFileDic*

Description: File notes for all data files.

Fields: *DataFileID, FileName, FileNote, LanguageID*

Format of parameters in output database

Note--if parameters are copied to the output database they are stored in the sparse format (i.e. zeroes are not stored) thus significantly reducing the size of output database for models containing large parameters. In the case of Statistics Canada's LifePaths model, the database size went down from 79 MB to 36 MB. Large savings should occur for the POHEM model as well.

When parameters are exported from the output database to MS Excel format all missing zeroes are added (except in the sparse format of the pivot table style).

Modgen Tracking Database

This appendix describes release 1.2.0.0 of the tracking database file. All tables except the **History** table within this database are identical in design to the tables in the Modgen Documentation database (refer to Modgen Documentation Database). Within the tracking database, the tables may have fewer fields and fewer records depending on the Modgen track statement used for database creation.

The **History** table contains the actual data for all tracked states. Its fields are:

ObjectID – a long integer

Time – a double

StateID – a long integer

Value – a double

SequenceCnt – a long integer (no repeats over all records).

The Modgen BioBrowser uses SQL queries against this History table to build its graphic displays. To optimize the speed of BioBrowser queries, Modgen indexes this table by ObjectId/ StateId as a combined index and by SequenceCnt as a separate index.

Modgen uses the following rules when creating the tracking file:

- An instance of an actor is uniquely identified by an ObjectID in the History table.
- For each Object, a record is written to the History table for every tracked state only when its value changes. For simultaneous events (same ObjectID, Time & StateID, different Value field), the SequenceCnt field determines the order of display by the BioBrowser.
- Each actor must have a logical state whose Name is "tracking" in the state dictionary ActorStateDic. This is a Modgen generated state (a user generated state can not be called "tracking"). Note that the "Name" field in ActorStateDic is a programmatic identifier for a state and is language independent (unlike the Description and Note fields in that table).
- Every actor will have an internally generated state whose Name is "case_id". This state is not available for display by the BioBrowser (i.e. it is ignored as an available state in the **Add/insert states** dialog window). A user generated state can not be called "case_id".
- For simple state types, the Value field contains the value of that state at that Time. For classifications, it contains the EnumValue in the ClassificationValueDic. For logical states, it contains the Value field in the LogicalDic. For ranges, it contains an integer value between the Min and Max (inclusive) in the RangeDic. The PartitionDic and PartitionValueDic are always empty in the tracking file (states based on partitions are always written as int's to the tracking file, the integer being the position in the partition). The Value field for Link states is described in detail below.
- Every tracked state (except link states) must have a value at the earliest tracking time. When ordered by SequenceCnt the tracking state will be first and last for all states for that instance of the actor. Its value will be 1 when tracking begins and 0 when tracking ends. Each BioBrowser query contains the tracking state as well as the requested state (this allows the BioBrowser to break up the graphic as the tracking state is shut on and off). The BioBrowser sorts its History table queries by SequenceCnt rather than Time since there are too many ties in the Time field.

- Each StateID in the actor state dictionary and LinkID in the actor link dictionary must be unique across all actors e.g. if a person actor and a child actor both have a state called Gender they will each have their own record and unique StateID in the state dictionary. In addition, the LinkID's in ActorLinkDic, can not be equal to any of the StateID's in the ActorStateDic.
- All dictionaries with language dependent text contain 1 record for each language specified in the LanguageDic.
- Link states are special states: the StateID field in the History table contains the LinkID value for that linked state in ActorLinkDic. In addition, their value field contains a pointer (ObjectID) to the linked object. A link state's Value field contains only -/+ObjectID. This is the only state type that does not contain a value in the History table at the earliest tracking time. The Value field contains ObjectID when that linked object was first added and –ObjectID when that link state was removed.

Appendix: Table of old and new names

As Modgen has continued to be developed and enhanced over the years, some features have had name revisions. Only the revised or current names are used throughout the Modgen Developer Guide. However, for those who might be maintaining older models, the following table displays the former and revised names for each changed entity.

Current Name	Previous Name	Where Referenced?
duration_trigger	duration_dummy	Derived state specifications (self-scheduling states—duration counters)
active_spell_duration	current_spell_duration	Derived state specifications (spells)
active_spell_weighted_duration	current_spell_weighted_duration	Derived state specifications (spells)
active_spell_delta	current_spell_delta	Derived state specifications (spells)
completed_spell_duration	previous_spell_duration	Derived state specifications (spells)
completed_spell_weighted_duration	previous_spell_weighted_duration	Derived state specifications (spells)
completed_spell_delta	previous_spell_delta	Derived state specifications (spells)
LABEL(module_name,language_code)	FILE_LABEL(language_code)	Documentation of a Modgen model (symbol labels)
NOTE(module_name,language_code)	FILE_NOTE(language_code)	Documentation of a Modgen model (symbol notes)
ModuleDic	SourceFileDic	Modgen Documentation Database
RandNormal()	RandDeviate()	Run-time functions (functions associated with

		the random number generator)
RandUniform()	RATE(LongRand())	Run-time functions (functions associated with the random number generator)
undergone_change()	first_changes()	Derived state specifications (event occurrences)
undergone_entrance()	first_entrances()	Derived state specifications (event occurrences)
Actor_id	ID (no longer in Modgen)	
mm	modgen_model	Creating a model (How Modgen code becomes a model executable file)
SPLIT	Split	Derived state specifications (transformations—difference between SPLIT and split)

Appendix: Command line arguments of modgen.exe

The Modgen compiler processes all files with an extension of .mpp in the current working directory. The command line arguments are as follows:

modgen.exe [-D output_directory] [-i input_directory] –EN|–FR

There is one mandatory argument, which must be either –EN or –FR. This argument specifies the language used for compiler output, English or French. There are two optional arguments, which specifies the directory to write compiler outputs, and read the compiler inputs. If either argument is omitted, compiler defaults to using the current working directory.

Launching the Model from the Command Line

To launch the simulation in batch mode directly on the command line, supply the scenario file with the –sc command line argument.

Syntax:

modelexe_name –sc *scenario_file* [-*computational_priority*] [-s]

modelexe_name –doc

• *scenario_file* is mandatory the name of the scenario file. It will have either a .sce or .scex file extension.

• the *computational_priority* is optional and can be either ‘normal’ or ‘low’; (if not specified, the default is ‘low’)

• The optional flag –s means ‘silent’. It supresses the beep sound normally produced when a simulation launched from the command line completes.

• The second form specifies the option –doc and no other options. When invoked this way, the model will generate a documentation database containing all model metadata, and then exit.