

The Springer Series on Demographic Methods  
and Population Analysis 43

Alain Bélanger  
Patrick Sabourin

# Microsimulation and Population Dynamics

An Introduction to Modgen 12

**EXTRAS ONLINE**

 Springer

# **The Springer Series on Demographic Methods and Population Analysis**

Volume 43

**Series Editor**

Kenneth C. Land, Duke University

In recent decades, there has been a rapid development of demographic models and methods and an explosive growth in the range of applications of population analysis. This series seeks to provide a publication outlet both for high-quality textual and expository books on modern techniques of demographic analysis and for works that present exemplary applications of such techniques to various aspects of population analysis.

Topics appropriate for the series include:

- General demographic methods
- Techniques of standardization
- Life table models and methods
- Multistate and multiregional life tables, analyses and projections
- Demographic aspects of biostatistics and epidemiology
- Stable population theory and its extensions
- Methods of indirect estimation
- Stochastic population models
- Event history analysis, duration analysis, and hazard regression models
- Demographic projection methods and population forecasts
- Techniques of applied demographic analysis, regional and local population estimates and projections
- Methods of estimation and projection for business and health care applications
- Methods and estimates for unique populations such as schools and students

Volumes in the series are of interest to researchers, professionals, and students in demography, sociology, economics, statistics, geography and regional science, public health and health care management, epidemiology, biostatistics, actuarial science, business, and related fields.

More information about this series at <http://www.springer.com/series/6449>

Alain Bélanger • Patrick Sabourin

# Microsimulation and Population Dynamics

An Introduction to Modgen 12



Springer

Alain Bélanger  
Urbanisation Culture Société  
Inst nat de la recherche scientifique  
Urbanisation Culture Société  
Montréal, Québec, Canada

Patrick Sabourin  
Urbanisation Culture Société  
Inst nat de la recherche scientifique  
Urbanisation Culture Société  
Montréal, Québec, Canada

Additional material to this book can be downloaded from <http://extras.springer.com>, <http://www.statcan.gc.ca/>.

ISSN 1389-6784 ISSN 2215-1990 (electronic)  
The Springer Series on Demographic Methods and Population Analysis  
ISBN 978-3-319-44662-2 ISBN 978-3-319-44663-9 (eBook)  
DOI 10.1007/978-3-319-44663-9

Library of Congress Control Number: 2016948704

© Springer International Publishing Switzerland 2017

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by Springer Nature  
The registered company is Springer International Publishing Switzerland  
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

*This new type of model consists of various sorts of interacting units which receive inputs and generate outputs. The outputs of each unit are, in part, functionally related to prior events and, in part, are the results of a series of random drawings from discrete probability distributions.*

—Guy H. Orcutt (1957)

# Foreword

This book is intended for anyone who needs a practical introduction to microsimulation in general and to the Modgen program in particular. We have selected Modgen, a Statistics Canada software package, as the platform for this initiation in microsimulation because of its flexibility, performance, accessibility and multifunctional capacity. Our view is that although Modgen is increasingly being used outside Statistics Canada itself, it is still under-exploited. Recent European microsimulation projects have invested significant resources in building their own microsimulation platforms, but these have not always proved to be as flexible or to perform as well. The underuse of Modgen is probably due mainly to the fact that the documentation produced by Statistics Canada is essentially aimed at those who are already experienced Modgen users. We believe that there is a need for a training manual designed for beginners, offering a simple but realistic microsimulation model. This book is intended to fill this gap. It provides a series of exercises which are specifically tailored to enable the beginner to build up his or her first dynamic microsimulation program step by step. To create a dynamic microsimulation model, a designer has to begin by identifying the events which will affect a population and which will drive its evolution over time. Whether the population is made up of businesses, households, people or other elements (or cases, to use the technical term), these individual elements all have to come into existence (or be born), evolve (grow older) and disappear (or die). Each particular population model, then, exists in a particular setting. In this book we use examples of human populations and build a model of demographic projection which simulates the evolution of the population of the different regions of Canada, based on certain demographic characteristics. Although we will be using some of the concepts and methods of demography, the non-demographer should find it relatively easy to practise the exercises and then to transpose his or her skills in using Modgen into the development of models in another expert field, such as economics, sociology, biology or elsewhere.

The idea for this book emerged from a seminar organised by the Population Change and Lifecourse Strategic Knowledge Cluster (PCLC Cluster). Because of a lack of documentation on Modgen, the Cluster decided to sponsor the production of a training manual to encourage Modgen users to learn how to use the software

program, and this task was given to us. Through the Cluster, this book has therefore received financial backing from the Canadian Social Sciences and Humanities Research Council (SSHRC). The authors would like to acknowledge the support of SSHRC and of its director, Professor Roderic Beaujot. We would also like to thank Samuel Vézina for proofreading the original manuscript and Julien De Gouffe from Statistics Canada for his precious technical support. The book was originally written in French and has been translated into English by Duncan Fulton, whom we also wish to thank for his work.

Of course, any errors or imprecisions that you may encounter in this book are entirely ours. We are always happy to receive suggestions and comments, and you can write to us at [comments@microsimulationandpopulationdynamics.com](mailto:comments@microsimulationandpopulationdynamics.com).

Québec, Canada

Alain Bélanger  
Patrick Sabourin

# Installing Modgen 12

Modgen is a microsimulation package based on the C++ programming language. The Modgen software is designed to be used in conjunction with Microsoft Visual C++.

Fortunately, Visual C++ is now available at no cost, as Microsoft has released a free version of its Visual Studio suite (Community Edition). You must download and install Visual Studio Community Edition before installing Modgen.

You will then be able to download and install the Modgen 12 software which is also available for free on the Statistics Canada website.

It is important that you install Visual Studio before Modgen, since Modgen will introduce some of its components in the Visual Studio files. Both installations are straightforward and should not be problematic. Complete installation instructions, along with download links to Visual Studio and Modgen can be found on the book website: [www.microsimulationandpopulationdynamics.com](http://www.microsimulationandpopulationdynamics.com).

Modgen 12 is scheduled to be released sometime in the Winter of 2017. Should it still be unavailable by the time you buy this book, you can write to [statcan.microsimulation-microsimulation.statcan@canada.ca](mailto:statcan.microsimulation-microsimulation.statcan@canada.ca) and ask for an advanced copy of the program.

Users of previous versions of Modgen will notice that some substantial changes have been made in version 12. Most notably, the Modgen toolbar has been removed, and precompilation (more on that topic in the book!) is now performed automatically upon compilation with the Visual C++ compiler. The developer's guide and other files, previously accessible through the Modgen toolbar, are now only available through the start menu.

For up-to-date information on the book and on Modgen, visit <http://www.microsimulationandpopulationdynamics.com/>.

# Contents

<b>1</b>	<b>Creating a Basic Cohort Model .....</b>	<b>1</b>
1.1	Using the New Model Wizard .....	2
1.2	The ModgenExample.mpp File (Appendix 1.1).....	6
1.3	The PersonCore.mpp File (Appendix 1.2).....	8
1.4	Compiling and Running the Simulation Program .....	16
1.5	Summary .....	21
	Appendices.....	21
	Appendix 1.1 ModgenExample.mpp.....	21
	Appendix 1.2 PersonCore.mpp.....	24
<b>2</b>	<b>The Life Table.....</b>	<b>29</b>
2.1	Adding a Classification.....	30
2.2	Adding a Numerical Range .....	32
2.3	Adding an Event: The Birthday Event.....	33
2.4	Modifying the <i>Start</i> Function .....	35
2.5	Modifying the Mortality Event Function.....	37
2.6	Adding Values in the Parameters File.....	39
2.7	Generating New Tables: The Elements of the Life Table.....	41
2.7.1	Derived States in the Table .....	43
2.8	Modifying Parameters and Reorganising Tables with the User Interface.....	50
2.9	Summary .....	55
	Appendices.....	55
	Appendix 2.1 PersonCore.mpp.....	55
	Appendix 2.2 Base(PersonCore).dat.....	61
<b>3</b>	<b>The Multiple Increment-Decrement Life Table .....</b>	<b>63</b>
3.1	Self-Scheduling Events: The <i>self_scheduling_int</i> Function .....	64
3.2	Creating a New Module for Interprovincial Migration.....	66
3.3	Assigning a Province of Birth Using a <i>cumrate</i> Parameter and the <i>Lookup</i> Function .....	71
3.4	Adding an Internal Migration Event.....	74

3.5	Modifying the Mortality Parameter .....	78
3.6	Multi-regional Tables.....	79
3.7	Manipulating Tables in Modgen (Advanced Topic) .....	83
3.8	Adding a Parameter File to a Scenario .....	86
3.9	Summary .....	88
	Appendices.....	89
	Appendix 3.1 PersonCore.mpp.....	89
	Appendix 3.2 Migration.mpp .....	93
	Appendix 3.3 Base(Migration).dat .....	97
<b>4</b>	<b>Modelling Fertility</b> .....	99
4.1	Adding Calendar Time.....	100
4.2	Creating a Fertility Module .....	103
4.3	Modifying the <i>PersonCore</i> Module and the <i>Start</i> Function .....	109
4.4	Bringing the Tables Together in a Results Module.....	114
4.5	Looking at the Results .....	117
4.6	Summary .....	120
	Appendices.....	121
	Appendix 4.1 PersonCore.mpp.....	121
	Appendix 4.2 Fertility.mpp.....	125
	Appendix 4.3 Tables.mpp .....	127
	Appendix 4.4 ModgenExample.mpp.....	128
<b>5</b>	<b>The Base Population</b> .....	131
5.1	Preparing a Microdata File .....	132
5.2	Importing External Data into Modgen.....	133
5.3	Integrating Import Functions into the Main File .....	134
5.4	Using the Imported Data and Modifying the <i>Start</i> Function.....	135
5.5	Modifying Calendar Time .....	140
5.6	Modifying the Scenario and Running the Model .....	141
5.7	Summary .....	143
	Appendices.....	144
	Appendix 5.1 ModgenExample.mpp.....	144
	Appendix 5.2 PersonCore.mpp.....	147
	Appendix 5.3 Tables.mpp .....	151
<b>6</b>	<b>International Migration</b> .....	153
6.1	Creating an International Migration Module.....	154
6.2	The Emigration Event.....	157
6.3	Modifying the <i>Start</i> Function .....	158
6.4	Modifying the Simulation Loop to Include Immigration .....	161
6.5	Adjusting Weights.....	165
6.6	Generating Results.....	166

6.7 Summary .....	169
Appendices.....	170
Appendix 6.1 InternationalMigration.mpp .....	170
Appendix 6.2 ModgenExample.mpp .....	171
Appendix 6.3 PersonCore.mpp .....	175
Appendix 6.4 Tables.mpp .....	179
<b>Conclusion .....</b>	<b>181</b>
<b>Appendix: Coding Standards .....</b>	<b>185</b>
<b>Glossary .....</b>	<b>187</b>
<b>References .....</b>	<b>189</b>
<b>Index.....</b>	<b>191</b>

# List of Figures

Fig. 1.1	Opening a new Visual C++ project .....	2
Fig. 1.2	Creating a new Modgen project .....	3
Fig. 1.3	Selecting the language for a model .....	5
Fig. 1.4	Visual Studio programming interface .....	5
Fig. 1.5	Compiler output messages .....	17
Fig. 1.6	Executable file in the project folder .....	18
Fig. 1.7	Modgen model user interface: opening the base scenario .....	19
Fig. 1.8	Table of results .....	19
Fig. 1.9	Modifying the set-up parameters .....	20
Fig. 1.10	Modifying the mortality parameter in the model .....	20
Fig. 2.1	Accessing the Modgen developer's guide .....	46
Fig. 2.2	The derived states section of the developer's guide .....	46
Fig. 2.3	Section of developer's guide explaining the use of the <i>duration</i> derived state .....	47
Fig. 2.4	Mortality parameters .....	50
Fig. 2.5	Scenario parameters .....	51
Fig. 2.6	Life expectancy at birth by sex .....	52
Fig. 2.7	Comparing methods for calculating life expectancy .....	52
Fig. 2.8	Survivors by age and sex .....	53
Fig. 2.9	Table properties .....	54
Fig. 2.10	Survivors, deaths and person-years lived by age .....	54
Fig. 3.1	Accessing the menu to add a new Modgen module .....	67
Fig. 3.2	Adding a Modgen module .....	67
Fig. 3.3	The empty migration module .....	68
Fig. 3.4	Menu for adding a new parameters file .....	68
Fig. 3.5	Creating a new parameters file .....	69
Fig. 3.6	New parameters file for the Migration module .....	69

Fig. 3.7	Adding a parameters file to a scenario .....	87
Fig. 3.8	User interface, multi-regional tables .....	87
Fig. 3.9	Multi-regional life expectancy table .....	88
Fig. 4.1	Demographic parameters according to event distribution .....	107
Fig. 4.2	Adding the fertility parameters file .....	118
Fig. 4.3	Table of births and age-specific fertility rates .....	118
Fig. 4.4	Comparison of simulated fertility rates and input fertility rates for the province of Québec (100,000 cases for the whole of Canada).....	119
Fig. 4.5	Population, births, deaths, exits and entries, by province .....	120
Fig. 5.1	Format of data in the base population file.....	133
Fig. 5.2	Modifying the number of cases in the scenario .....	142
Fig. 5.3	Table of results .....	142
Fig. 6.1	Diagram of microsimulation algorithm .....	162
Fig. 6.2	Number of immigrants per five-year period .....	168
Fig. 6.3	Canadian population according to high (300,000), medium (250,000) and low (200,000) immigration scenarios.....	169

# Introduction

IF WE ASK A DEMOGRAPHER to describe the future evolution of a given population, he or she will start by gathering two types of information. The first of these concerns the actual population numbers by age group and sex (and perhaps by region), which are the starting point for an elementary form of population forecast or projection. The demographer's next question will be about the frequencies with which the main demographic events occur – births, migrations and deaths. These frequencies will be used to calculate rates for each age and gender group. So our imaginary demographer will be building a model whose operationalisation requires population numbers for every group considered as “homogeneous” in some way (e.g. in terms of age or sex) and rates which represent the degree of risk (of mortality, birth, migration, etc.) that these groups are exposed to. The number of such events projected per year is obtained by multiplying the population numbers by the risk in question. Based on analysis of more or less recent trends or on comparison with trends observed in other regions or countries, hypotheses are formed about the likely evolution of these rates in the future.

## From Multi-state Macro Models to Microsimulation

This projection methodology is determinist in nature. The uncertainty attached to the future evolution of the various rates is estimated by generating a consistent set of scenarios which incorporate different plausible hypotheses about future rates of fertility, migration and mortality. These scenarios allow us to generate a set of projections; as future events unfold, their actual values should fall within the limit values of these projections. A probabilistic component can also be added to the model by estimating, in addition to the central tendency, the variance of each parameter (e.g. based on a chronological series). In this way the same projection can be repeated several times, giving the rates a random value depending on the variance attributed to each of them.

Any model is necessarily an abstraction from reality, but these kinds of models, whether they are deterministic or probabilistic, have several major limitations. Firstly, all of them assume that each sex and age group is internally homogeneous, that is, that all the individuals in a group experience the same risk of undergoing a particular type of event (e.g. that all women aged 22 have the same risk of dying, migrating or giving birth). Such a hypothesis rarely if ever holds true in practice, and this is bound to result in a bias in the projected results. To illustrate this, suppose that a population is composed of two subgroups (e.g. with different languages or religions), which are different from each other in terms of their fertility (and, to keep things simple, solely in that respect). Assuming that fertility rates remain constant, the more fertile subgroup will gradually increase in its relative size within the population as a whole. But if the relative size of this more fertile group within the population increases, then the fertility of the population as a whole will automatically rise, even though there has been no change in the behaviour of the women of either of the sub-groups. This will mean that the further into the future the projections extend, the more this simple macro model will tend to under-estimate the number of future births.

Another limitation of traditional macro models is linked to the nature of the projection hypotheses. According to these, the probability of an event occurring in the future depends entirely on present conditions. To use the example of fertility again, the probability of a woman giving birth is assumed to depend solely on her age and the region where she is living at time  $t$ , without taking account of her country of birth or her parity (the number of children she has). But if fertility behaviour is partly a result of the social and cultural environment in which a person was brought up, then a woman born in a country with higher fertility will have, on average, higher fertility than one born in a region with lower fertility, irrespective of where she is living at time  $t$ . But a simple cohort-components model is unable to take these differences into account, because the fertility of both women are assumed to be the same as the fertility of all women of the same age in their region of residence.

We should point out that it is theoretically possible to make the cohort-components model more complex, so as to account for the underlying heterogeneity of populations (for example in terms of ethnic group or region of birth). But the difficulty here is more technical, and results from the fact that component-based projections usually depend on matrix computation. Without going into details, the key point here is that the size of the matrix increases exponentially with the number of variables being considered. So the model quickly becomes unworkable (Van Imhoff and Post 1998).

Microsimulation enables us to overcome these limitations by creating a randomised series of experiences for each individual in the projection and so building up a unique biography for each simulated case. Looking again at the fertility example, we can trace the stages in a simple microsimulation model. The first step is to compile a representative sample of the female population of the region being studied (e.g. using publicly available micro-level data from a census). For each woman in the sample and for each year in the simulation, a randomised experiment is then carried out whose probability of success is a function of the level of fertility at each age. To put this in concrete terms, a random value between 0 and 1 is compared to the probability of giving birth, for each woman in the sample. If the random value

is lower than the probability of giving birth, an individual aged 0 is added to the simulation. If the opposite is true, there is no birth and the model moves on to another event. The randomized experiment is repeated for every fertile age, varying the probability of giving birth. The result for the population as a whole is obtained by aggregating all the individual results, using appropriate weighting.

Access to increasingly powerful computers at ever lower cost, as well as improved availability of survey microdata, has meant that microsimulation has become increasingly widely used over recent decades. The development of programming languages and of software such as the Modgen program presented in this book has made it easier to create microsimulation models and has helped to increase the numbers of people using them. However, even with the help of Modgen, developing a dynamic microsimulation model is, initially, a potentially more demanding task than developing a cohort-component projection model or even a multiregional model. Considerable resources have to be devoted to programming, establishing parameters and validating the model.

But despite the effort needed to set them up, microsimulation models are nevertheless superior to macromodels. One of their main advantages, as argued by Orcutt (1957) in his original article, is theoretical in nature. The most powerful theoretical models for explaining human behaviour, such as decisions to have a child or to migrate, operate at the level of the individual. So it makes sense to simulate these behaviours at the individual level. The same argument explains another strength of microsimulation which we have already touched on. This concerns changes in the composition of the population which are hard to capture in macro models but which can be easily taken into account by a microsimulation model. Behaviours like those involving fertility and mobility can be made to vary according to a large number of relevant variables such as country of birth, level of education or religion. At the analytical level, microsimulation offers greater flexibility and is usually the only way to obtain consistent results, especially when a number of different characteristics have to be projected at the same time. For example, the microsimulation model developed at Statistics Canada to forecast the ethnocultural diversity of the Canadian population (Alain Bélanger and Caron Malenfant 2005) required simultaneous projection of several ethnic variables: immigration status and immigration period, membership of a visible minority group, religious denomination and mother tongue as well as age, sex and region of residence. Two other modelling options were available to the researchers developing this model, as alternatives to microsimulation – but one of them would have been impossible to implement, and the other would have yielded inconsistent results. If we had attempted to carry out this projection using a multistate model, the transition matrix would have had several hundred million cells, much more than the total number of inhabitants of Canada. This matrix would have been so big that it would have been impossible to create and manipulate. The other option would have been to carry out separate projections for each dimension of cultural diversity. Such projection models might have been manageable, but their results would not have been internally consistent. This is because there is no reason to assume that population projections based, for example, on immigrant status would arrive at the same totals as those based on visible minority group

membership or religious affiliation. Furthermore, this approach would not have allowed cross tabulation of all the projected variables, and this would necessarily have reduced the quality of the projection itself as well as the richness of the results produced. It is much easier to introduce a number of explanatory factors for each type of behaviour into a microsimulation model, because there is no need to increase the state space (the matrix) exponentially. A new explanatory factor (a state) can be simply added to the existing model without having to reorganise everything that has been done up to that point.

Conditional relationships (interactions) between explanatory variables are practically impossible to account for in a conventional projection model. For example, the relationship between the fertility of immigrant women, their ethnic origin and their religion can be modelled relatively easily in a microsimulation model, but significantly increases the number of cells in the transition matrix of a cohort component-model (it would be multiplied by the number of possible states in the immigrant status times the number of possible ethnic origins times the number of included religions). In the same way, modelling variables which vary over time is theoretically possible as an extension of multistate models, but in practice this would also lead to an exponential increase in the size of the transition matrix or to such complications that it is hardly ever attempted (Wolf 2001). The healthy immigrant effect—the effect of duration of residence of immigrants in the host country on their relative mortality risk—is an example of the implementation of a continuous variable in a projection model.

Finally, a multistate model does not allow for the modelling of possible interactions between the actors (in a model, individuals are referred to as “actors” or “cases”). There are numerous social processes in which the interactions between individuals are essential elements in understanding the phenomenon (and therefore in modelling it). For example, in family demography, modelling the formation of unions may require projecting the entire set of individuals in a population so as to enable a marriage market to be built up and interactions between the members of a couple to be generated. Modgen for instance enables us to build agent-based microsimulation models where individual agents may interact (by exchanging information, for instance), but these latter models are not dealt with in this introductory book. Our focus will be placed exclusively on the modelling of the life courses of non interacting individuals.

## What Is Modgen?

Modgen is a framework for the development of microsimulation models. It includes a specialised microsimulation programming language, a runtime library and a suite of standalone applications which can be used with Modgen modules. Modgen was designed at Statistics Canada in 1994 to facilitate the programming and modification of microsimulation models. The aim was to develop a generic programming environment which would allow all kinds of microsimulation projects to be carried out.

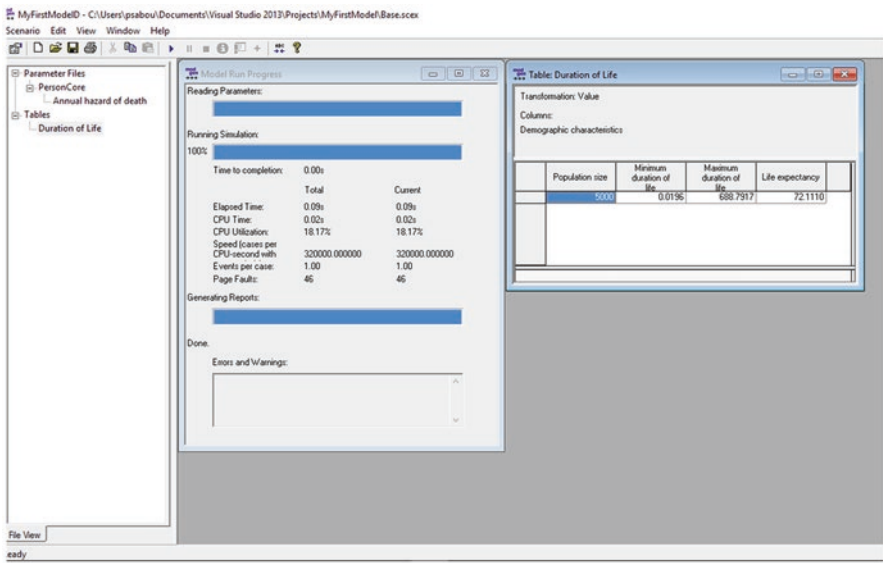


Fig. 1 The user interface of a Modgen model

**The Modgen Language** is a superset of the C++ programming language, so that Modgen functions are formed of C++ functions. The elements of the language specific to Modgen are processed by a precompiler which converts Modgen code into C++ code. The code is then processed by the C++ Visual Studio compiler to create a functional program. That is why Modgen 12 requires Visual Studio to be installed. There is no need to be an expert in C++ programming in order to use this book, but a basic knowledge of programming concepts will be very useful for understanding the code in the models. **The library** contains elements which are common to all the microsimulation models created using Modgen, such as the **user interface** (Fig. 1). Whereas the structure of the user interface is common to all Modgen models, its content is individualised for each one. It enables the user to run the model, consult and modify the parameters, control the simulations and manage the scenarios, as well as examine and export the tables generated by the model. With this interface it is possible to use projection models without necessarily knowing how to design them, in other words without needing to know the Modgen programming language. So a model can be distributed to external users, such as experts and decision makers, who can themselves set up their own scenarios and derive projections from them.

**The Modgen Toolkit** includes numerous optional applications, some installed with Modgen and others available online. The most commonly used tools are Modgen BioBrowser and the Modgen Web Interface. Microsimulation programs often require large databases to be created in which output information on the simulated individuals is stored; analysis of results then has to be done separately using external database software. With Modgen the simulation takes place at the individ-

ual level, but the tables generated by the model show aggregated results according to the designer's specifications. So Modgen generates tables in real time during the simulation. This means that there is no need for output databases, which can take up a large amount of time and processing power. But Modgen still maintains the option of observing individual life courses, thanks to the monitoring option provided by Modgen BioBrowser. BioBrowser allows a subset of the simulated population of individual life courses to be observed graphically, with its dimensions and variables specified by the user. Developers more interested by a life course perspective than by the aggregated projection results per se will find the BioBrowser very useful. The BioBrowser is not covered in this book, but information on this topic can be found in the Modgen developer's guide.

The Modgen Web Interface has been developed by the Canadian Partnership Against Cancer. It enables the user to operate a Modgen model from a website. This interface means that the user can see and modify the parameters, see the results in graphic or table form as well as compare the results of different scenarios.

## Types of Models

Modgen has been designed to be very flexible, and the software allows for both static and dynamic<sup>1</sup> models, although dynamic models are the most commonly used. For dynamic models, Modgen adopts a competing-events approach which is general enough to permit the use of continuous-time, discrete-time, or combined models.

So-called "time-based" models, in which an entire population is simulated simultaneously so as to take into account the interactions between its members, are possible. These models are often known as multiagent or agent-based. So-called "case-based" models, which we will use for the examples in this book, are more usual. In these models the entire population is simulated, but one case at a time, with each single case representing an individual (or some other unit, a family or a firm for instance).

---

<sup>1</sup>Typically, static models use weighted aging techniques to display the population over the course of the simulation. The characteristics of the actors do not change, but their weighting in the population is modified. In practice these models use macrodemographic projections carried out separately, usually by an official body, to modify the weighting of individuals in each age group. Static models are often used to measure the anticipated effect of a change in public policy, for example a fiscal measure, on individuals in terms of winners and losers.

Dynamic models work by continuously updating each of the characteristics of each individual across time: this is the type of model described in this book. Individuals evolve over time, experiencing life cycle events such as unions and their breakdowns, births, migratory movements, educational advancements, entry into and exit from the labor market etc. up to the time of death or the horizon of the projection. Each of these events occurs with a probability varying according to the characteristics of the actor. A specific risk is therefore calculated for every event which each actor in the projection may encounter, based on their current characteristics.

Finally, whatever the type of model chosen, the simulated populations may be synthetic (determined by the designer) or based on empirical micro-data (e.g. census data). There are two main types of dynamic model: population models and cohort models. These two types use similar processes for aging and for modification of the characteristics of individuals, but differ in terms of the population used as the starting point for the simulation. A cohort model uses a cohort of individuals all born at the same time as its starting population. So the base population of a cohort model is a synthetic population. A population model instead takes as its starting point a representative sample of the projected population, generally derived from publicly available census data. The first four chapters of this book deal with developing a cohort model, and the two remaining chapters with a model of the Canadian population.

## Structure of the Book

When a Modgen model is first created, the software automatically generates a default model that projects a population cohort subject to a fixed death rate. The program also calculates the life expectancy of this imaginary population and creates a table to present this result. We begin, in Chap. 1, by describing how to set up this automatically generated program and how to specify some of its parameters. We use this simple model to introduce certain fundamental ideas and to describe a number of essential functions for the development of all microsimulation models using Modgen.

Chapter 2 deals with the further development of the model automatically generated by the creation wizard. In this chapter we introduce the basic factors enabling some variation in the risk of death, namely age and sex. This produces a more realistic microsimulation model by cohort which simulates life expectancy and the other elements of the life table, using mortality data from the Canadian province of Ontario in 2006. The programming exercise developed throughout this chapter will teach the reader to add new states to a model, such as sex (male or female), and to create a new event function, such as the birthday. The latter will provide the age of the actor in completed years, thus enabling us to distinguish it from exact age, which is also used for the purposes of calculating the life table. Finally, the reader will learn how to modify an existing event function and how to add new dimensions to a results table.

Chapter 3 continues the development of this cohort model by integrating a regional dimension, thus transforming the life table created in Chap. 2 into a “multistate” table. This chapter’s example will help the reader to learn how to create a new event function that is more complex than the birthday event. This event is part of a module on interregional mobility, where internal migration is modelled in two stages. First, the probability of an individual leaving his or her place of residence is compared to a random number. If the Monte Carlo experiment succeeds (if the random number is lower than the probability of leaving), the model will randomly

assign a new region of residence to the individual according to a distribution specific to each region of origin. Through reproducing the example in this chapter, the reader will have the opportunity to become familiar with a type of parameter (*cum-rate*) and a function (*lookup*) which are specific to Modgen. These functions enable a new value to be randomly assigned to a state variable according to an arbitrary distribution matrix. In practice, this model will generate results which are equivalent to the results obtained by using a multiregional table.

In succeeding chapters, the cohort model will be gradually transformed into a population model. In Chaps. 4 and 5, we add a fertility module and build in a starting population. This makes it possible to carry out closed multi-regional projections i.e. without new entries resulting from international migration. International migration is included in the model in Chap. 6, making it possible to do “open” multi-regional projections.

Data search and estimation of parameters are of course an important part of the creation of a microsimulation projection model, but this laborious task is not necessary in order to learn how to use Modgen. The data used in our examples are provided in Excel files which are available on the book website at <http://www.microsimulationandpopulationdynamics.com/>. The data is taken from widely available microdata files from the 2006 Canadian census and from vital statistics data available on the Statistics Canada website.

All the files used in this book are available on the Internet at the following address: <http://www.microsimulationandpopulationdynamics.com/> or on Springer Extras Online. For each chapter, the Modgen code for the example (i.e. the complete Visual Studio solution) and the Excel files needed for the parameters are provided. The code for the relevant files for the first two exercises is given in full in the appendices to Chaps. 1, 2 and 3 to help the reader become familiar with the structure of the program. In succeeding chapters, the reader who is by then initiated will sometimes be required to look up the code directly in the solution files on the website.

\*\*\*

The reader who works through all the examples in the book will learn how to create a dynamic projection model using relatively complex microsimulation tools and, what is more, a model that would have been difficult to implement using the traditional cohort-component method. By this time he or she will have enough working knowledge of Modgen to be able to develop his or her own model of dynamic microsimulation.

As a final note, the model developed in this book is a slimmed-down version of a projection model developed by Alain Bélanger and his team in the Dynamic Simulation Laboratory. This model enables a simulation of the Canadian population and its subregions based on a representative sample of the population, along a set of demographic and ethnocultural dimensions (immigration status, visible minority membership, mother tongue, home language, religion, etc.).

# Chapter 1

## Creating a Basic Cohort Model

### Aims of This Chapter

- Creating a first microsimulation model using the Modgen wizard
- Learning about the structure of a Modgen model
- Learning some key Modgen functions
- Experiencing the Visual Studio environment and compiling your first Modgen model
- Running a Modgen model and getting to know the user interface

In this chapter our main aim is to introduce you to the Modgen software and to the Visual Studio programming environment, guiding you through the creation of a first basic model. This model will be based on one single type of event: death. Although this will yield only a small number of results, such as life expectancy at birth, it will allow us to demonstrate and to practise the use of several of the basic ideas in Modgen. We will move on to the calculation and presentation of other elements of the life table in the next chapter.

Calculating a life table is a good way to become familiar with Modgen. When demographers do this, they work through a series of stages. First, current or period mortality rates have to be translated into probabilities of dying<sup>1</sup>; when these are applied to the population in the table,<sup>2</sup> they give the number of deaths in each year of age, and therefore the number of survivors at each succeeding exact age.<sup>3</sup> Assuming a random distribution of deaths between birthdays, the number of person-years lived between each age interval can be summed and then be divided by the

---

<sup>1</sup>The probability of dying represents the probability for an individual at a given exact age  $x$  of dying before his or her next birthday.

<sup>2</sup>In other words, an imaginary cohort whose individual members were all born at the same time.

<sup>3</sup>Exact age is the age of an individual on the day of his or her birthday.

number of survivors at each exact age to give the life expectancy by age – the mean number of years of life remaining to the survivors of a cohort at a given exact age.

This kind of calculation can be easily reproduced in a microsimulation model using Modgen. Our Modgen model, created using the new model wizard, here generates a table showing the life expectancy of a cohort subjected from birth to a constant risk of dying of 0.014. For each case simulated, the time duration before death is calculated using a random variable and a mathematical equation specified in the model. Since the model operates in continuous time (we will explain this later on), Modgen calculates the exact time of death, so life expectancy can be easily derived by taking the average lifetime (time passed before death) of all the actors in the simulation.

## 1.1 Using the New Model Wizard

Although a Modgen microsimulation model can be created from scratch, we recommend using the model wizard, since this will automatically generate a Visual Studio “project” (or “solution”) in which all the files needed to build a basic model can be found. Some of these files can be modified later to make the model more complex, for example by creating new states, events or tables.

The first step is of course to install Visual Studio and Modgen (see Installing Modgen 12 at the beginning of the book). Once the installation is complete, Visual Studio can be started (optimised for programming in Visual C++) and a new project created by selecting New -> Project, as shown in Fig. 1.1 below.

In the *New Project* window, select Modgen in the Visual C++ folder. A list appears (Fig. 1.2) offering a choice of two microsimulation models, each available in either French or English. Modgen allows us to create two types of models, one based on cases and the other on time (see Box 1.1). In this book we are developing

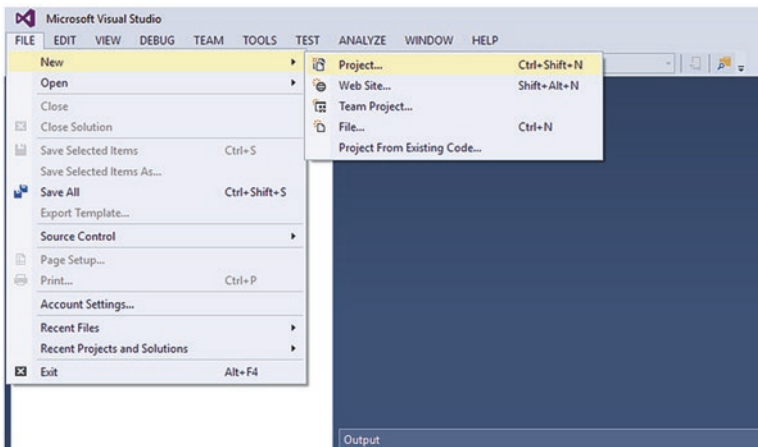


Fig. 1.1 Opening a new Visual C++ project

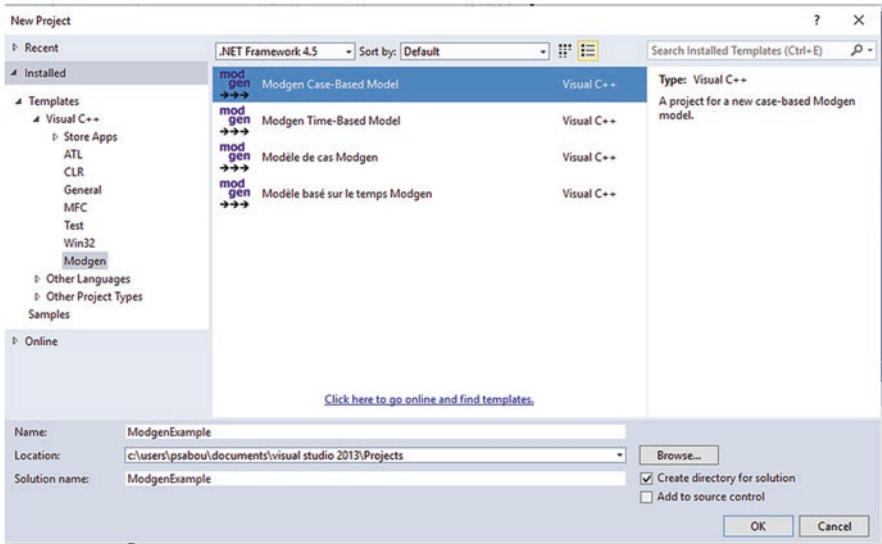


Fig. 1.2 Creating a new Modgen project

**Box 1.1: Definition of Some Key Concepts: Case-Based and Time-Based Models, Actors, Events and States**

Cases, time, actors, events and states are key concepts which will be used throughout this book. So it is worth taking a moment to define and explain them in more detail. In a microsimulation model, each simulation unit is called an **actor** and is simulated individually. The actor's characteristics or **states** are modified by the occurrence of an **event** which is simulated by the model (marriage, birth, migration etc). The simulation of all the events affecting an actor constitutes a "case". In a more complex model, a case can have several actors, as we will see later (imagine a model in which an actor may give birth to another actor; both will be simulated in the same case). Generally in demographic models an actor represents a person or a household, but there is no reason why an actor could not represent a virus, an animal, or another non-living entity. States are the characteristics which define an actor. For example, age, sex, place of residence or immigrant status are all state variables; their values are set when the actor is initially created, and are liable to change whenever a relevant event takes place, such as a birthday which increases the actor's age, or a move which changes the place of residence. Those who are familiar with C++ will recognise the concept of actor as being akin to the concept of class, while events and states would correspond to members of a class. Each actor in an actual simulation is therefore an object of the class *Person* (a description of the actor in our model).

(continued)

**Box 1.1** (continued)

In the example given in this chapter, the actor is subject only to the risk of dying. Death is therefore an “event” whose consequence will be a change of state from alive to dead. In practice, the death event will make the value of a logical variable change from true (alive) to false (deceased). The death event will also put an end to the simulation of this case; in a case-based model, the programme will then move on to the next case, and so on until the final case in the simulation round. In a case-based model, then, the actors are simulated one at a time until their death or up to the time horizon of the projection, or for a specific duration (a number of years) which has been pre-determined for this particular projection.

Because the actors are simulated separately, a **case-based model** does not allow for interactions between cases. If a model of interactions between actors is needed, **time-based model** must be used. In this kind of model, all the actors are simulated simultaneously up to the time horizon for the projection. A time-based projection requires greater material resources, because information covering the entire set of actors has to be maintained in the memory throughout the course of the simulation. Events occurring in a particular actor thus have the capacity to affect events in other actors.

a case-based model, so we select this option as shown in Fig. 1.2. This window also allows us to give the project a name and to specify the name of the sub-directory in which the files we create will be stored. Here we call the project “ModgenExample”.

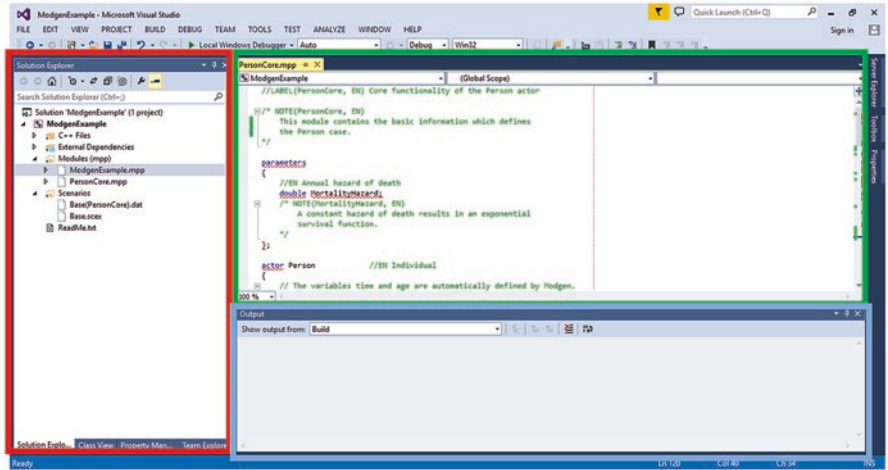
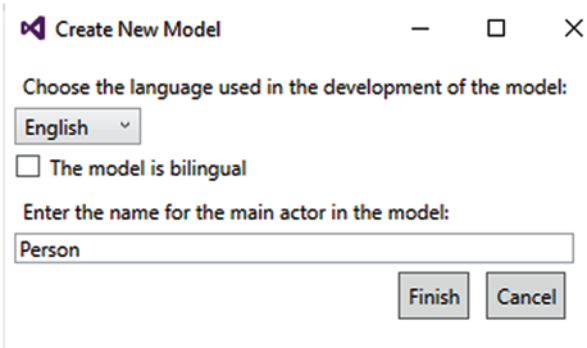
A new dialog box opens and enables the user to choose the language in which the model will be developed (Fig. 1.3). Since Modgen has been developed by Statistics Canada, which is itself subject to the bilingualism regulations of the Canadian government, all the elements of Modgen are available in both English and French. A bilingual model can also be created, which would contain both the English and the French versions of the notes and the titles of the variables. Here for simplicity we develop an English-only model. To do this, select English in the dropdown menu and do not check the *The model is bilingual* box.

The dialog box also allows us to give a name to the “main actor” in the model. The “main actor” (see Box 1.1) is a C++ class which will serve as the mould or blueprint for all the simulated cases. In this exercise and in the following ones, we will be simulating individuals only, so it seems appropriate for this principal actor to be named “Person”. So leave the default name “Person” as it is and click the *Finish* button.

After clicking *Finish*, you will find a set of files in the Visual Studio *Solution Explorer* which are the skeleton of your model (if the *Solution Explorer* does not appear, press CTRL+ALT+L). Figure 1.4 shows an overall view of the interface and of the files making up the microsimulation model.

All the files for our model can be found on the left of the window in the *Solution Explorer* (Fig. 1.4). They are sorted automatically into many folders, only two of which are relevant for model development.

**Fig. 1.3** Selecting the language for a model



**Fig. 1.4** Visual Studio programming interface. *Left panel*: file explorer for the microsimulation model. *Upper-right panel*: text editor window. This is where the Modgen code is written. *Lower-right panel*: output window. Messages referring to compilation errors will appear in this window

The *Modules* folder contains the Modgen code files (.mcpp) which make up the different modules of the microsimulation model. These are the files that will be modified by the developer to add events or states to the model. The Modgen pre-compiler (which we will talk about later) converts Modgen instructions in each .mcpp file into C++ code which is compatible with the Visual Studio compiler. The second important folder is the *Scenarios* folder which contains the files storing the parameters of the base scenario (.dat and .scex files). These files store the parameters of the simulation such as fertility rates, out-migration rates, mortality rates, the simulation horizon, etc. The *Base.scex* file also contains information on the base scenario, but it is modified using the Modgen user interface and not Visual Studio (as we will see later). A file giving a summary of the project (*ReadMe.txt*) is also generated automatically. This last file provides a brief description of the program structure.

The other folders contain files specific to C++ (.cpp,.h and others): these are always generated automatically when a Modgen programme is precompiled, and the model developer never has to modify them directly.

For the moment, we will focus on three files: the two.mpp files, *ModgenExample.mpp* and *PersonCore.mpp*, and the file in the *Scenarios* folder called *Base(PersonCore).dat*. In the next section we describe how to view and modify the content of these files in the main Visual Studio window. To make the code appear in the text editor window (green box, Fig. 1.4) simply double-click on the file name in the *Solution Explorer*.

## 1.2 The ModgenExample.mpp File (Appendix 1.1)

The name of this file corresponds to the name given to the project (see Fig. 1.2) and contains the code which is the motor of all Modgen microsimulation models. It has about 90 lines of code most of which appear in green: these are simply comments which help to understand the program and are not compiled.<sup>4</sup> There are also some lines which define the model, and two functions: *Simulation* and *CaseSimulation*. These are essential for understanding the structure of a Modgen microsimulation model.

The first lines in the file define the model version, the type of simulation (by case, or *case\_based*) and the development language (English). There is also the *time* variable which is defined as a *double* type of number. The *time* variable refers to the model time (or the model clock) and the *double* type means that the time is measured continuously with double precision.<sup>5</sup> Continuous time management means that the wait time for an event is calculated precisely, as opposed to a discrete model (non-continuous), where an event takes place during a unit of time specified in the model (generally a year). A particular feature of Modgen is that it makes modelling in continuous time easier in terms of programming. In later chapters we will see that this special feature is particularly useful for managing competing events efficiently.<sup>6</sup>

---

<sup>4</sup>See Box 1.2 and consult a C++ manual for more details.

<sup>5</sup>A *double* type variable contains 64 bits, which corresponds to a number with 15 digits. See a C++ manual for more details.

<sup>6</sup>Suppose that during a simulation a birth and a death happen in the same year. The order in which these events take place will have an effect on the results of the simulation. If the mother's death happens first, there will be no birth. On the other hand, if the birth happens first, a new actor will be added to the simulation. The risk of dying and the risk of giving birth are here defined as competitive. In a discrete-time model, because the time unit is generally one year, events forecast for the same year conflict; in other words, the time until the two events take place is identical and there is no way of knowing which one will take place first. The model designer therefore has to add supplementary rules to determine which event will take precedence. In a continuous-time model, the probability of two events happening simultaneously is infinitely small so there is no need to draw up supplementary rules for ordering conflicting events.

Reading these first lines of code, note that each instruction ends with a “;” and that an instruction can be written over several lines, as in this example<sup>7</sup>:

```

ModgenExample.mpp
17  languages {
18      EN // English
19  };

```

This could equally well be written as:

```

Languages { EN }; // English

```

It is good practice to put the code between curly brackets { } on separate lines, especially if the enclosed code stretches over several lines.

Next we have the code for the two main functions of the model, *Simulation* and *CaseSimulation*, which in conjunction generate and govern each of the simulated cases. In this sense these are the functions which are the motor of the microsimulation model. *Simulation* loops through all the cases and calls *CaseSimulation* which starts the simulation of each of these cases.

The *void CaseSimulation* function initialises the principal actor in a case by using the following code:

```

ModgenExample.mpp
27  // Initialize the first actor in the case.
28  Person *poFirstActor = new Person();
29  poFirstActor->Start( );

```

Using C++ programming jargon, we could say that this code creates a *poFirstActor* pointer (represented by the sign “\*”) to an object of the *Person* type (we will see the class which defines this object in another file later on). Within this *Person* object, all possible characteristics of the actors (states) will be defined, as well as the events which these actors may experience in the course of their lives. The *Start* function, which is called using “->” launches the simulation of this actor. In the comments, we see that *CaseSimulation* simulates a single case, but also initialises and generates the “first actor in this case”, which implies that a case can include several actors. In a model simulating births, for example, a case with a female actor can contain a number of actors larger than one, the female herself and also each of the virtual children she will give birth to. However, in the example which concerns us in this chapter, there is only one actor per case. As well as initialising the actor, this function also contains a loop<sup>8</sup> which actually manages all the possible future events affecting the actor in the case concerned, from the nearest event in time to the

<sup>7</sup>Remember that the complete code can be found in the appendix to this chapter. Numbers refer directly to the corresponding lines in the appendix.

<sup>8</sup>A loop is a sequence of code which is repeated a predetermined number of times. Consult a C++ programming instruction manual for more details.

**Box 1.2: Some Color Conventions for Visual C++**

In Visual Studio, the code can appear on the text editor screen in one of three colors: green, blue or black.

Code in green is for comments added by the programmer to document the program. There are two ways of adding comments to a program: either by using double forward slashes (“//”), which changes the rest of the line into comment, or by framing the comment text between a “/\*” to open the comment section and a “\*/” to close it. This code is ignored by the C++ compiler and is used solely to document the program, an essential task often neglected even by otherwise competent programmers. Since all rules must have exceptions, Modgen does use some information from the comments to create labels for parameters and tables in the user interface. We will see more of this in Chap. 2. Code in blue represents Modgen and C++ keywords. These keywords are words reserved for functions or data types that are an integral part of the programming language. To avoid confusion, these words cannot be used as names of variables in a user model. Finally, code in black is added by the user and is not made up of C++ or Modgen keywords. This is programming code which consists mainly of names of variables, functions and operators.

most distant (lines 37–56 in *ModgenExample.mpp*). In our first model, only a single type of event, death, is liable to affect the actor. In a more complex model, a number of competing events may occur (where an actor is subject to all the different risks simultaneously), and the code for this loop will automatically order these events in time. So of all the possible events, the one with the shortest wait time will be selected as the next event to occur. For now, there is no need to understand the detail, but you should get a general sense of what the *CaseSimulation* function does.

The *Simulation* function is a simple loop which calls the *CaseSimulation* function a number of times specified in advance by the user as a scenario parameter. When covering the inclusion of a base population in Chap. 5, we will see how this loop works.

### 1.3 The PersonCore.mpp File (Appendix 1.2)

The second file created by the Modgen wizard contains the key elements of the description of the actor. The *Start* and *Finish* functions initialise and terminate each simulated case, while the event functions enable mortality to be modelled. This file already contains most of the elements that will need to be modified to develop a new model.

The *PersonCore.mpp* file includes four sections: the first, the header, contains the statement of the actor’s characteristics (like sex or place of residence); the second contains the definition of the parameters which will be used to simulate the events

(the inputs, such as the mortality rate); the third contains the statement of the actor *Person* as well as of each of the event functions associated with the actor (duration before an event and the action to be taken when an event occurs); finally the fourth and last section contains the definition of the tables of results (the outputs, see Appendix 1.2). We will now look in more detail at each of these sections.

At present the header contains only comments. In this first example, the actor has no special characteristics at all – neither sex nor place of residence. We will add these characteristics as we go through the following chapters.

The “parameters” section contains the definition of the parameters to be used in the module, such as the mortality and fertility rates. The model wizard automatically generates a mortality risk which it calls “*MortalityHazard*”. At present this risk is simply a constant which does not vary with either age or sex, or any other determinant of mortality. The purpose of the wizard is not to create a realistic model, but a functional one. In the next chapter we will see how to add in factors which bring the model closer to reality.

It is important to note that the value of the parameter is not shown in this file. The parameter is declared in the *PersonCore.mpp* file, but its actual value is defined in the *Base(PersonCore).dat* file which is found under the *Scenarios* thumbnail in the *Solution Explorer*. In our example, the mortality hazard is constant and equal to 0.014.

The parameter values are found in a different file from *PersonCore.mpp* simply because parameters must be externally modified by users to create a range of scenarios. In other words, the *.dat* file is not part of the C++ program itself but acts as a database for the parameters. If the value of the parameter was shown in the program itself, it would not be possible to change it after compiling the program. In later chapters we will see how to modify the scenario file, and we will also see how to make the parameters vary according to the factors which influence the risk of dying, such as age and sex.

Going back to the description of the contents of the *PersonCore.mpp* file, we see that the third section includes the definition of the *Person* actor. All the characteristics of the actor are found here, and also the events likely to affect him (lines 18–106, which include the description of the functions declared in the *Person* class, see Appendix 1.2). The comments tell us that two variables, *time* and *age* have been created automatically (lines 20–33, see Appendix 1.2). The base unit for these two variables is a year.

The first instruction encountered in the actor *Person* creates a logical (or Boolean) variable called “alive”, whose default value is *TRUE* or 1 (*FALSE* being equal to 0).

PersonCore.mpp

```
36    logical alive = {TRUE};
```

The value of this variable will be modified at the time of death, before closing the simulation of a case. Since for the moment no other variable is needed to create a basic functioning model, the wizard produces only this one. In the next chapters we will add other variables in addition to the fact of being alive; these may be less critical but are still important!

The definition of the actor also includes a declaration of the events that might occur during the simulation. In Modgen, all the events are defined using two functions: the first, the time function, determines the simulation time when the event is expected to take place, while the second, the event function, contains the changes to be made to the actor's characteristics once the event has occurred. Note that here the functions are merely declared: they will be described in greater detail later in the file.

```

                                                    PersonCore.mpp
44  event      timeMortalityEvent,      MortalityEvent;    //EN
      Mortality event

```

Two last functions are essential components of the actor. The *Start* function is applied right at the beginning of the simulation of the actor (remember that the *ModgenExample.mpp* file contains a call for this function). In addition to launching the simulation, it enables the initialisation of actor characteristics, such as sex, place of birth, or any other relevant characteristics included in the model. As this is a very simple cohort model, all the actors may be in only a single state (alive or dead), and all of them start the simulation at time 0 and age 0, as we can see in the definition of the *Start* function further on in the file (lines 84–96). The *Finish* function, in turn, simply removes the actor from the simulation so that Modgen can launch a new actor.

The following extract from the code shows the time and event functions which together simulate the mortality event.

```

                                                    PersonCore.mpp
57  // The time function of MortalityEvent
58  TIME Person::timeMortalityEvent()
59  {
60      TIME tEventTime = TIME_INFINITE;
61
62      // Draw a random waiting time to death
63      // from an exponential distribution
64      // based on the constant hazard MortalityHazard.
65      tEventTime = WAIT( - TIME( log( RandUniform(1) ) /
66          MortalityHazard ) );
67
68      return tEventTime;
69  }
70
71  // The implement function of MortalityEvent
72  void Person::MortalityEvent()
73  {
74      alive = FALSE;
75
76      // Remove the actor from the simulation.
77      Finish();
78  }

```

The first function, the “time function”, determines the timing of an event for which the hazard is constant. It is very important to understand how this function works, as it will be used and adapted to create new and more sophisticated events where the hazard may vary according to individual characteristics such as age and sex.

We should first note that *TIME*, *TIME\_INFINITE* and *WAIT* are key words which are specific to Modgen. For the moment all we need to emphasise is that *TIME* is a Modgen data type or macro-command defining a time variable which is used to manage continuous time efficiently and easily in a simulation.

The first line of the function is as follows: *TIME Person::timeMortalityEvent()*. The “*::*” operator (a scope resolution operator) indicates that the *timeMortalityEvent()* function is a member (or is part of) the actor (or the class, in C++ jargon) *Person*. The keyword *TIME* indicates that this function returns a value of type *TIME*, which is to be expected since this function is meant to calculate waiting times before an event.

In the function, the time variable *tTimeEvent* is initialised using the pre-defined constant *TIME\_INFINITE*. This means that if the value of the variable is not modified, the event will never take place (*TIME\_INFINITE* representing an extremely high value). But *TIME\_INFINITE* is simply an initialisation value, and the next line will specify the time of death using the *WAIT* function. This function converts the estimated wait time before death into a figure which can be easily interpreted by the Modgen event manager. The length of time before the death is estimated using the equation found in the *WAIT* function brackets (see Box 1.3 for a fuller explanation). Once the wait time has been calculated and stored in *tTimeEvent*, the variable is sent back to the Modgen events manager using the C++ *return* command. To summarise, a variable containing the wait time is first defined, the wait time is then calculated and its value converted using the *WAIT* function, and finally this value is sent back to the Modgen event manager.

The moment when the death occurs has now been calculated, and Modgen moves the simulation forward to the next event. Because only mortality is being simulated, Modgen moves the simulation time forward to the age established at the time of death, and carries out the function associated with this event. It is only within an event function that the actor’s characteristics can change value. Here the “alive” state of the actor is changed from *TRUE* (1) to *FALSE* (0) (line 74) and the case is completed by calling the *Finish* function (line 77).

The final section of the *PersonCore.mpp* file includes the definition of the output tables storing the results of the simulation. One of the advantages of using Modgen, as we have said before, is that it makes the management of events much easier. Another advantage becomes clear with the *table* command, which aggregates the results of the simulation as each case progresses, without having to generate a database containing all the detail of the simulation. There are two advantages to this way of storing results. First, there is no need to take up a lot of time during the running of the simulation, as reading and writing in a file are always relatively slow. Modgen keeps its tables in memory. Second, there is therefore no need to use separate software to exploit the database and create synoptic tables.

The command syntax for a table has three sections. The first has only one line and is the header of the table (line 111). It contains the Modgen keyword *table* (indicating that a table definition is coming), the name of the actor (*Person*) and the actual name of the table (*DurationOfLife*). The second section (lines 113–123) contains the data to be tabulated. Generally, data are gathered through Modgen instructions called “derived states”<sup>9</sup> which derive information about actor states (we will come back to this throughout this book). In the table created by the wizard, four of these derived states are used: *value\_in()*, *min\_value\_out()*, *max\_value\_out()* and *duration()*.

```

                                                    PersonCore.mpp
108    /*NOTE(DurationOfLife, EN)
109        This table contains statistics related to the dura-
            tion of life.
110    */
111    table Person DurationOfLife //EN Duration of Life
112    {
113        {
114            //EN Population size
115            value_in(alive),
116            //EN Minimum duration of life decimals=4
117            min_value_out(duration()),
118            //EN Maximum duration of life decimals=4
119            max_value_out(duration()),
120            //EN Life expectancy decimals=4
121            duration() / value_in(alive)
122        }
123    } //EN Demographic characteristics
124    };
```

*Duration* (line 121) is probably one of the most useful derived states. It will be used either alone or in conjunction with other derived states. It supplies the value of the time spent in the state specified in brackets (i.e. as an argument). Where no state is specified, it provides the lifespan of the actor in the simulation. In its simplest version, where *duration* is used without argument and is not itself the argument of another derived state, it enables us to add together the time spent by all the actors in the simulation. When we want to calculate the duration in a specific state, we use arguments in brackets to allow us to specify the name of the state variable and its status. We will come back to this in the next chapter.

The keyword *value\_in()* provides the value of the state when the actor enters the simulation. Here the state variable *alive* is a logical type of variable which has the

---

<sup>9</sup>Derived states such as the ones used in tables can be thought of as functions extracting information about an actor’s states, such as the duration of time spent in a given state or the number of times the value of a given state has changed. In that sense, even though they are called derived states, it would be more appropriate to think of them as derived “from” states.

value 1 when the actor is still alive. Writing *value\_in(alive)* in the table description calls for the sum of the simulated cases, because each of the actors enters the simulation with *alive = 1* (or *TRUE*). In this cohort model, the number of simulated cases is equivalent to the population total, which corresponds to the root of a life table. Two other features of the table depend on derived states – *min\_value\_out* and *max\_value\_out* – which give, respectively, the minimum and maximum values for the state specified in brackets, for all the simulated cases. So in the present case they give the age at death of the simulated cases, with the shortest and the longest lifespan as calculated by the derived state *duration*.

The comment lines above each derived state are important, as they will be used as labels for the table. This is one case where comments are not ignored by the Modgen precompiler. So, for instance, for the last derived state, the label in the table will be *Life Expectancy*, and the number of decimals appearing in the data cell will be four.

To recapitulate: the first element of the table, *value\_in(alive)*, calculates the number of actors taking part in the simulation; the next two, *min\_value\_out(duration())* and *max\_value\_out(duration())*, determine the shortest and the longest lifespan; finally, the fourth element of the table, *duration()/value\_in(alive)*, divides the sum of the years spent alive for all the actors by the number of actors, and so gives the life expectancy. The tables and the derived states are defined in a rather non-intuitive but logical way. The description of the table we have given here is relatively simple. In the next chapters we will make the tables more complex and go into the subject in more depth.

The third section of a table usually contains the dimensions along which the various derived states are to be tabulated. We could for instance make all the above calculations for men and women separately by adding a *sex* dimension. Note however that the model generated by the wizard does not yet contain states other than *alive*, so the table cannot include additional dimensions such as age or sex.

### Box 1.3: How Is the Wait Time Before an Event Set?

In a discrete-time microsimulation model, whether an event takes place or not is determined at each time change. If the smallest unit of time is 1 year, and the probability of dying is 0.01, each actor will have one chance in a hundred of dying during the year. If the event does not take place at time *t*, the model goes forward one step and repeats the experiment, until the event occurs or the simulation ends. This method is intuitive and easy to execute, but has a number of major disadvantages. Firstly, a randomized experiment has to be done at each time interval, which can turn out to be expensive in computing power. Next, several events which are subject to competing risks may occur during the same time interval. If that happens, how should we select which event occurs first? Should childbirth and death be scheduled to occur in a given year, the outcome will differ depending on which event occurs first: if death comes first, childbirth will never occur.

(continued)

**Box 1.3** (continued)

A model which functions in continuous time allows us to get around this problem. Rather than evaluating the risk that an event occurs at each time interval, Modgen calculates a precise duration before the occurrence of the event, or a wait time. As it is possible to calculate this duration on the basis of a given hazard, the two methods are mathematically equivalent.

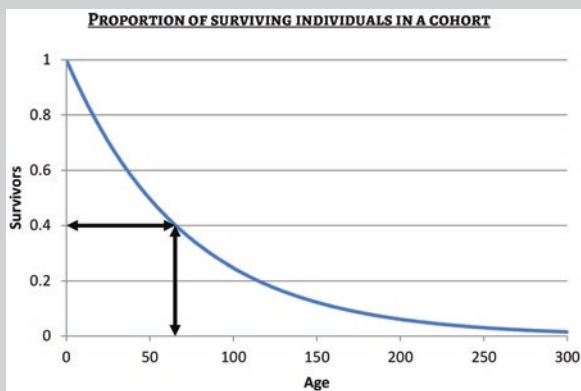
The Modgen wizard defines a force of mortality ( $\mu$ ), or a hazard of death, of 0.014 per year. The model in this chapter assumes that the risk of dying is constant and that therefore the population at time  $t$  ( $P_t$ ) is equal to the population at the starting time ( $P_0$ ) multiplied by  $e^{-0.014t}$  (or 0.9861 where  $t=1$ ).

$$P_t = P_0 e^{-\mu t}$$

Because the population is closed and there are no births, we can obtain the proportion of individuals still alive at time  $t$  :

$$P_t / P_0 = e^{-\mu t}$$

The graph of this equation, using the value  $\mu=0.014$  looks like this:



The derivative of the curve represents the number of deaths per years of age when the force of mortality remains constant throughout the lifespan. In general the graph can be interpreted as follows: the y axis shows the proportion of members of a cohort who are still alive at a given age, for example 40% at age 65 (see the arrows on the graph). But the curve can also be interpreted from an individual point of view, so that each point on the y axis represents the individual's probability of living until age  $x$  (on the  $x$  axis). Since everyone has to die (perhaps unfortunately), one can determine a random age at death by picking a random number between 0 and 1 (a point on the y axis) and finding the corresponding age on the  $x$  axis – or, in microsimulation language,

(continued)

**Box 1.3** (continued)

the duration before the “death” event. The horizontal axis point can be found mathematically using the equation presented above. If the vertical axis point chosen at random is represented by a uniform random variable  $Y$ , we have :

$$Y = e^{-\mu t}$$

Isolating  $t$ , the time elapsed before the mortality event,

$$\ln(Y) = \ln(e^{-\mu t})$$

$$\ln(Y) = -\mu t$$

$$t = -\ln(Y) / \mu$$

When the force of mortality is known and the distribution of deaths follows an exponential risk model, then a random wait time can be attributed to an actor using this formula. And this is exactly what Modgen does in the model created by the wizard.

```

                                                PersonCore.mpp
65         tEventTime = WAIT( - TIME( log( RandUniform(1) ) /
66         MortalityHazard ) );
```

We can see that the expression  $\log(\text{RandUniform}(1))/\text{MortalityHazard}$  (the wait time) is written as an argument to the *TIME* function, itself an argument of the *WAIT* function. Remember that the *tEventTime* variable is a *TIME* type of variable. The *TIME()* function is then used to convert a number of *double* type (a number with 25 significant digits after the decimal point) into a *TIME* type of number, which has special properties in Modgen. The *WAIT* function for its part transforms the wait time into absolute time, to make it compatible with the model’s internal clock.

The *RandUniform(1)* function chooses a random number between 0 and 1 ( $Y$ , in our example) and the *MortalityHazard* variable contains the force of mortality ( $\mu$ ). Note that the argument of *RandUniform(1)* has the sole function of initialising the random number generator in C++ and is automatically assigned by Modgen during the precompilation. When this function is inserted into the code, the bracket should be left empty and the function written simply *RandUniform()*.

Once the random wait time has been calculated, it is returned by the *timeMortalityEvent* function and sent to the event queue managed by Modgen. Management of wait time is invisible to the model developer, but he or she still has to establish the wait time of the different event to be simulated using

(continued)

**Box 1.3** (continued)

risk models. In our example, the risk model is the exponential distribution of events, but we could equally well have used a Weibull or a log-normal distribution.

Finally, we should remember that as the duration before each event is determined randomly, different simulations will give different results. The error or variance which is produced by this stochastic process is called the Monte-Carlo error. This error is not present in deterministic models, in which a given mortality rate always implies the same number of deaths when the size of the population at risk is constant. The Monte-Carlo error can be reduced by taking the mean of the results of several microsimulations, or simply by increasing the number of cases.

## 1.4 Compiling and Running the Simulation Program

As we have said before, Modgen is a superset of C++, meaning that in addition to the basic C++ functions, it also contains some specific high-level functions. These functions are not recognised as such by the C++ Visual Studio compiler, and this is why the compilation is performed in two distinct stages.<sup>10</sup> Firstly, the Modgen programme has to be precompiled; this means that its high level functions need to be translated into C++ to make them interpretable by the Visual Studio compiler. Next the precompiled program has to be compiled by Visual Studio to generate an “autonomous” model in the form of a program file.

In previous Modgen versions, precompilation had to be done manually using a button on a Modgen toolbar. In Modgen 12, precompilation is done automatically at the time of compilation. The code is compiled by selecting *Build Solution* in the *Build* menu in Visual Studio (or simply by pressing the F7 key).

When a program is compiled, Visual Studio sends error messages to an output window. If you do not see the output window, you can make it appear by pressing Alt+2. Figure 1.5 below shows the output when precompilation and compilation are performed without errors, as indicated by the lines of the output message (*Modgen: 0 errors – 0 warnings* for precompilation and *Build: 1 succeeded, 0 failed [...]* for compilation).

The compiler creates an executable file (.exe) whose name is the name of the project plus a suffix (usually the letter *D* or *E*). To run the program, we have to go back to the project folder using Windows Explorer and double-click on the

---

<sup>10</sup>The action of compiling transforms the high level code (Modgen, C++) written by the programmer into an independent executable file. Here, the executable file is the actual microsimulation model, which includes the execution library and the user interface.

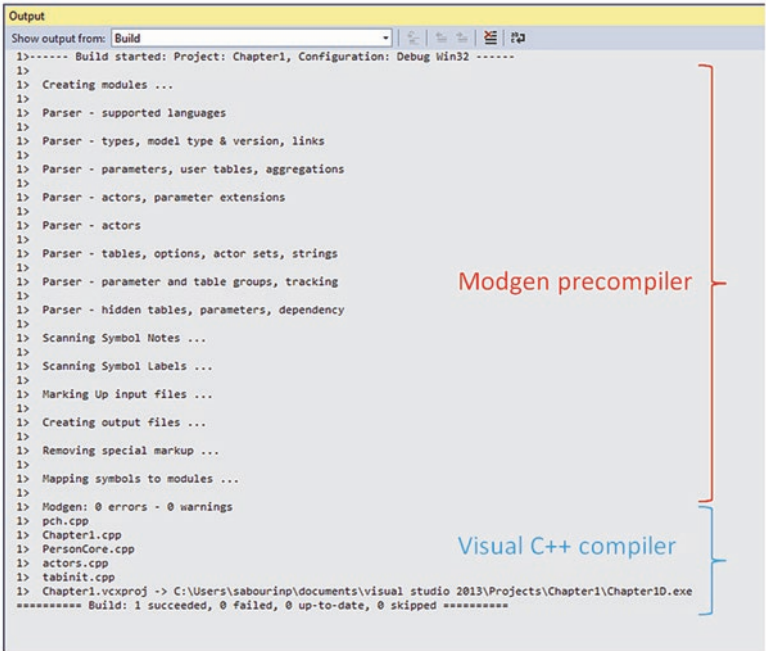


Fig. 1.5 Compiler output messages

*ModgenExampleD.exe* file. If the file extension does not appear, you can find the right file in the *Type* column: it is the only *Application* type file in the folder (Fig. 1.6). Remember that the project folder was specified when the model was first created (see Fig. 1.2).

Double-clicking on the model program file opens the user interface created automatically by Modgen for all microsimulation projects. Through this interface, the user can create scenarios, modify parameters, run the model and manipulate result tables. The wizard automatically creates a scenario which can be opened by clicking on *Open...* in the *Scenario* menu and selecting the *Base.scex* scenario (Fig. 1.7).

As there has not been a simulation yet, Modgen notifies the user that the output database (the results table) is missing and that the simulation has to be launched in order to create it. Click on *OK* to make this message disappear and click on ► in the toolbar to run the program. A window appears showing the progress of the simulation, and posts a message when it is finished. The title of the single table produced by this model (Duration of Life) then appears in the window on the left of the screen (*File View* tab) under *Tables*. To open it, just double-click on the title of the table and the results will appear.

In Fig. 1.8 we can see that the simulation involved 5000 cases (the size of the cohort), that the shortest lifespan was 0.02 years and the longest 688.8 years (!), and that life expectancy was just over 72 years. Notice that the title over each column corresponds to the comment written above each derived state in the table description.

Name	Date modified	Type	Size
Debug	2016-02-05 11:23 ...	File folder	
ipch	2016-02-05 11:23 ...	File folder	
ModgenExample	2016-02-01 2:20 PM	File folder	
ACTORS	2016-02-01 3:47 PM	CPP File	16 KB
ACTORS	2016-02-01 3:41 PM	H File	6 KB
app	2015-07-31 7:08 AM	ICO File	25 KB
Base(log)	2016-02-01 3:47 PM	Text Document	1 KB
Base(PersonCore)	2016-02-01 2:20 PM	DAT File	1 KB
Base(tbl)	2016-02-01 3:47 PM	Microsoft Access ...	432 KB
Base	2016-02-01 2:20 PM	SCEX File	1 KB
Base.scvx	2016-02-01 3:48 PM	SCVX File	2 KB
default	2015-11-23 12:20 ...	SQL Server Comp...	384 KB
MODEL	2016-02-01 3:47 PM	H File	1 KB
MODEL	2016-02-01 3:47 PM	RC File	1 KB
Modgen.Build	2015-11-23 12:22 ...	Project Property File	3 KB
Modgen.Configuration	2015-11-23 12:22 ...	Project Property File	4 KB
Modgen.targets	2015-07-31 7:08 AM	TARGETS File	1 KB
ModgenExample	2016-02-01 3:47 PM	CPP File	3 KB
ModgenExample	2016-02-01 3:41 PM	Text Document	2 KB
ModgenExample	2016-02-01 3:41 PM	MPP File	3 KB
ModgenExample	2016-02-04 8:12 AM	SQL Server Comp...	75,200 KB
ModgenExample	2016-02-01 2:20 PM	SLN File	1 KB
ModgenExample	2016-02-01 3:43 PM	VCXPROJ File	2 KB
ModgenExample.vcxproj	2016-02-01 2:20 PM	VC++ Project Filte...	2 KB
ModgenExampleD	2016-02-01 3:47 PM	Application	6,185 KB
ModgenExampleD	2016-02-01 3:47 PM	Incremental Linke...	6,147 KB
modgenexampled	2016-02-01 3:47 PM	Program Debug D...	15,220 KB
ModgenLog	2016-02-01 3:47 PM	HTML Document	2 KB
PARSE	2016-02-01 3:47 PM	Setup Information	12 KB
pch	2015-07-31 7:08 AM	CPP File	1 KB
PersonCore	2016-02-01 3:47 PM	CPP File	4 KB
PersonCore	2016-02-04 8:12 AM	MPP File	4 KB
ReadMe	2016-02-01 2:20 PM	Text Document	4 KB
TABINIT	2016-02-01 3:47 PM	CPP File	1 KB
TABINIT	2016-02-01 3:47 PM	H File	1 KB

**Fig. 1.6** Executable file in the project folder

The results of the simulation depend on the parameters specified by the user. These can be modified by clicking on *Settings* in the *Scenario* menu. The default values for the simulation are 5000 cases and a *Population size* of 5000. The *Population size* field is used to rebalance the simulated population (Modgen refers to this as *Population scaling*) to make the results representative of a given population. In our example, as the number of cases is equal to the population size, each case has a weight of 1, and thus scaling is unnecessary. Uncheck the *Population scaling* box (Fig. 1.9).

Because the radix (the starting population) of a life table is generally 100,000, the number of cases needs to be raised to 100,000 (Fig. 1.9). Raising the number of

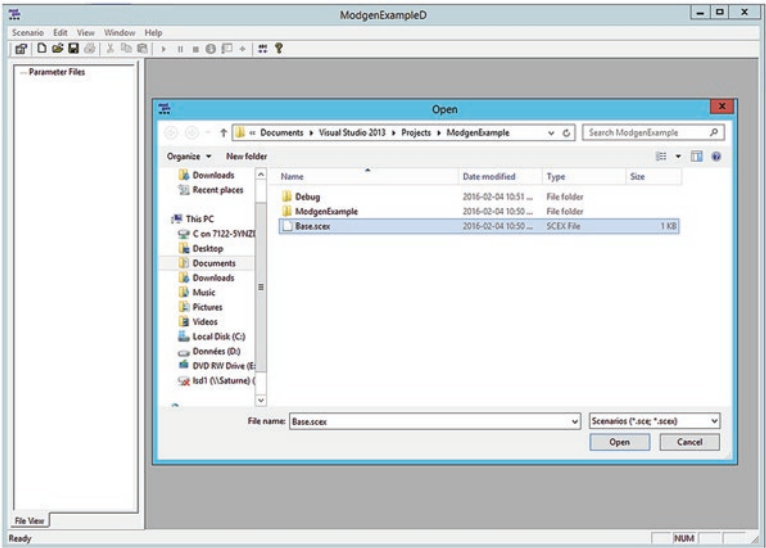


Fig. 1.7 Modgen model user interface: opening the base scenario

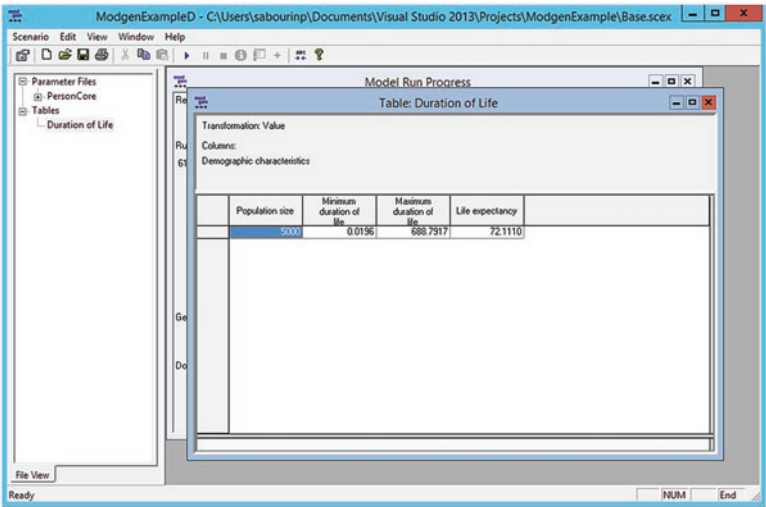


Fig. 1.8 Table of results

cases to 100,000 will reduce the Monte Carlo error (see Box 1.3) and will slightly alter the results shown in the *Duration of Life* table in Fig. 1.8.

Clicking on ► again in the toolbar launches the simulation of this new scenario and produces fresh, and more precise, results.

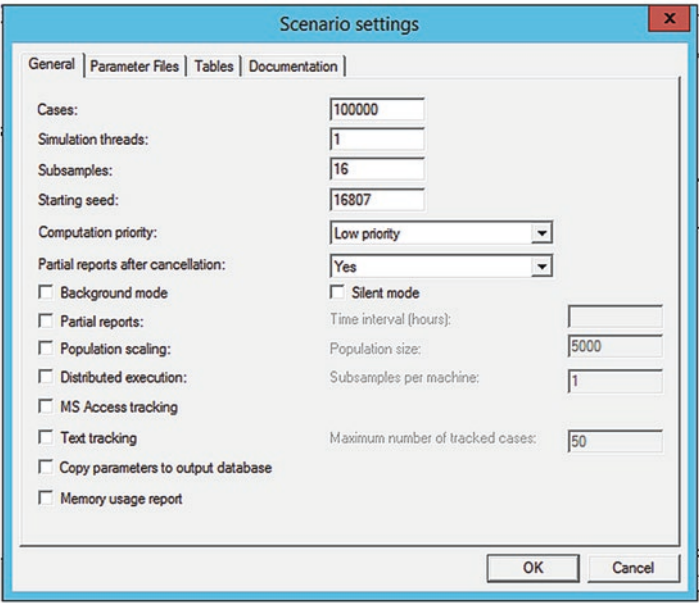


Fig. 1.9 Modifying the set-up parameters

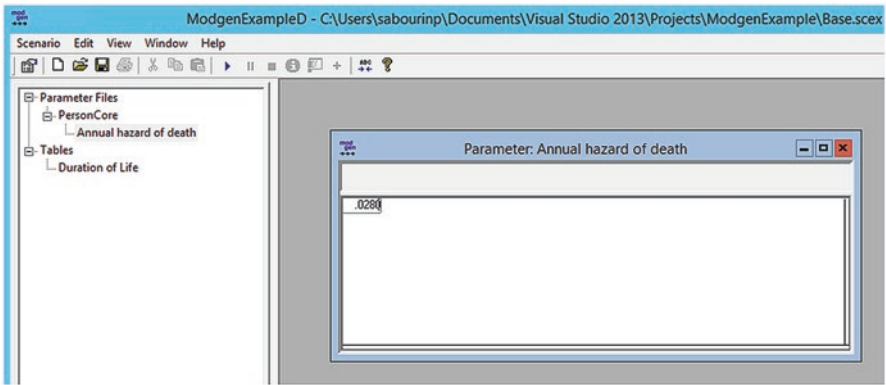


Fig. 1.10 Modifying the mortality parameter in the model

A new scenario can also be created by modifying the mortality parameter. To do this, we just double-click on *Parameter files* -> *PersonCore* -> *Annual hazard of death* in the *File View* window: we will double the mortality hazard by changing its value from 0.014 to 0.028 (Fig. 1.10).

To keep the parameters of the previous scenario, we save this new scenario under a new name, by clicking on *Save* in the *Scenario* menu. Let's call the new scenario *Base2X* to distinguish it from the previous one. If we click again on ►, the results of this new simulation will appear in the *Duration of Life* table with, essentially, values

which are halved for life expectancy and for maximum and minimum lifespan. It is possible to modify the mortality parameters without creating a new scenario file, but it is still good practice to create one file per scenario in order to avoid any confusion, especially when working with dozens of parameters in complex models.

## 1.5 Summary

The simulation model generated by the modelling wizard is simple and not very realistic. But it has enabled us to see how a Modgen microsimulation model written with Visual Studio works, to understand better how Modgen manages events, and to explore the thumbnails and buttons in the user interface of the compiled program. In its current form, the model is much too simple to be useful in any practical sense. New elements must be added, and this will be the subject of the next chapters.

In Chap. 2, we will add some more characteristics to the actor so as to make mortality vary according to age and sex. We will also learn how to generate an abridged life table and to derive all the elements needed for its construction.

## Appendices

### *Appendix 1.1 ModgenExample.mpp*

Code Sections: Header (lines 1 to 24), *CaseSimulation()* Function (lines 25 to 62), *Simulation()* Function (lines 63 to 97)

```

1  //LABEL(ModgenExample, EN) Core simulation functions
2
3  /* NOTE(ModgenExample, EN)
4      This module contains core simulation functions and
        definitions.
5  */
6
7  // The model version number
8  version 1, 0, 0, 0;
9
10 // The model type
11 model_type case_based;
12
13 // The data type used to represent time
14 time_type double;
15
```

```
16 // Supported languages
17 languages {
18     EN // English
19 };
20
21 // The CaseSimulation function simulates a single case,
22 // and is called by the Simulation function declared
23 // later
24 // in this module.
25 void CaseSimulation( )
26 {
27     // Initialize the first actor in the case.
28     Person *poFirstActor = new Person();
29     poFirstActor->Start( );
30
31     // Continue processing events until there are no
32     // more.
33     // Model code is responsible for ending the case
34     // by calling
35     // Finish on all existant actors.
36
37     // The Modgen run-time implements
38     // the global event queue gpoEventQueue.
39     while ( !gpoEventQueue->Empty() )
40     {
41         // The global variables gbCancelled and
42         // gbErrors
43         // are maintained by the Modgen run-time.
44         if ( gbCancelled || gbErrors )
45         {
46             // The user cancelled the simulation,
47             // or run-time errors occurred.
48             // Terminate the case immediately.
49             gpoEventQueue->FinishAllActors();
50         }
51         else
52         {
53             // Age all actors to the time of the
54             // next event.
55             gpoEventQueue->WaitUntil( gpoEvent-
56                                     Queue->NextEvent() );
57
58             // Implement the next event.
59             gpoEventQueue->Implement();
60         }
61     }
62 }
```

```
55         }
56     }
57
58     // Note that Modgen handles memory cleanup
59     // when Finish is called on an actor.
60 }
61
62
63 // The Simulation function is called by Modgen to simulate a set of cases.
64 void Simulation()
65 {
66     // counter for cases simulated
67     long lCase = 0;
68
69     // The Modgen run-time implements CASES (used below),
70     // which supplies the number of cases to simulate
71     // in a particular thread.
72     //
73     // The following loop for cases is stopped if
74     // - the simulation is cancelled by the user,
75     //   with partial reports (gbInterrupted)
76     // - the simulation is cancelled by the user,
77     //   with no partial reports (gbCancelled)
78     // - a run-time error occurs (gbErrors)
79     //
80     // The global variables gbInterrupted, gbCancelled and gbErrors
81     // are maintained by the Modgen run-time.
82     for ( lCase = 0; lCase < CASES() && !gbInterrupted
83         && !gbCancelled
84         && !gbErrors; lCase++ )
85     {
86         // Simulate a case.
87
88         // Tell the Modgen run-time to prepare to simulate a new case.
89         StartCase();
90
91         // Call the CaseSimulation function
92         // defined earlier in this module.
93         CaseSimulation();
94     }
```

```

94             // Tell the Modgen run-time that the case
             has been completed.
95             SignalCase();
96         }
97     }

```

## Appendix 1.2 *PersonCore.mpp*

Code Sections: Header, Parameters, Actor *Person* and functions, Tables

```

1    //LABEL(PersonCore, EN) Core functionality of the Person
    actor
2
3    /* NOTE(PersonCore, EN)
4        This module contains the basic information which
        defines
5        the Person case.
6    */
7
8    parameters
9    {
10        //EN Annual hazard of death
11        double MortalityHazard;
12        /* NOTE(MortalityHazard, EN)
13            A constant hazard of death results in an
            exponential
14            survival function.
15        */
16    };
17
18    actor Person          //EN Individual
19    {
20        // The variables time and age are automatically
            defined by Modgen.
21        // Model-specific labels and notes are supplied
            below.
22
23        //LABEL(Person.time, EN) Time
24        /*NOTE(Person.time, EN)
25            Time is a continuous quantity in this
            model.
26            A unit of time is a year.
27        */

```

```

28
29     //LABEL(Person.age, EN) Age
30     /*NOTE(Person.age, EN)
31         Age is a continuous quantity in this model.
32         A unit of age is a year.
33     */
34
35     //EN Alive
36     logical alive = {TRUE};
37     /*NOTE(Person.alive, EN)
38         Set to TRUE when the actor starts, and to
39         FALSE just
40         before the actor finishes. Since the numeric
41         value
42         of TRUE is 1 and FALSE is 0, this
43         variable
44         can also be used to count actors in tables.
45     */
46
47     event timeMortalityEvent, MortalityEvent;    //EN
48     Mortality event
49
50     //LABEL(Person.Start, EN) Starts the actor
51     void Start();
52
53     //LABEL(Person.Finish, EN) Finishes the actor
54     void Finish();
55 };
56
57 /*NOTE(Person.MortalityEvent, EN)
58     This event implements a constant hazard of death.
59 */
60
61 // The time function of MortalityEvent
62 TIME Person::timeMortalityEvent()
63 {
64     TIME tEventTime = TIME_INFINITE;
65
66     // Draw a random waiting time to death
67     // from an exponential distribution
68     // based on the constant hazard MortalityHazard.
69     tEventTime = WAIT( - TIME( log( RandUniform(1) )
70         /
71         MortalityHazard ) );
72

```

```
68         return tEventTime;
69     }
70
71     // The implement function of MortalityEvent
72     void Person::MortalityEvent()
73     {
74         alive = FALSE;
75
76         // Remove the actor from the simulation.
77         Finish();
78     }
79
80     /*NOTE(Person.Start, EN)
81         The Start function initializes actor variables
82         before simulation
83         of the actor commences.
84     */
85     void Person::Start()
86     {
87         // Modgen initializes all actor variables
88         // before the code in this function is executed.
89
90         age = 0;
91         time = 0;
92
93         // After the code in this function is executed,
94         // Modgen initializes events and tables for the
95         // actor.
96         // Modgen also outputs starting values to the
97         // tracking file if requested.
98     }
99
100    /*NOTE(Person.Finish, EN)
101        The Finish function terminates the simulation of
102        an actor.
103    */
104    void Person::Finish()
105    {
106        // After the code in this function is executed,
107        // Modgen removes the actor from tables and from
108        // the simulation.
109        // Modgen also recuperates any memory used by
110        // the actor.
111    }
```

```
108  /*NOTE(DurationOfLife, EN)
109      This table contains statistics related to the
        duration of life.
110  */
111  table Person DurationOfLife //EN Duration of Life
112  {
113      {
114          //EN Population size
115          value_in(alive),
116          //EN Minimum duration of life decimals=4
117          min_value_out(duration()),
118          //EN Maximum duration of life decimals=4
119          max_value_out(duration()),
120          //EN Life expectancy decimals=4
121          duration() / value_in(alive)
122      }
123      } //EN Demographic characteristics
124  };
```

## Chapter 2

# The Life Table

### Aims of This Chapter

- Learning how to use classifications and numerical ranges
- Creating new state variables for an actor
- Creating a new event
- Adding dimensions to an existing parameter
- Understanding how to use age as a continuous or discrete variable

With the help of the Modgen wizard, we have developed and successfully run our first microsimulation program. This simple model was able to calculate life expectancy for a cohort whose risk of death remains constant throughout the simulation. For demographers, and probably for most users, this unrealistic mortality hypothesis is unsatisfactory. As a minimum, even in a simple cohort model, mortality should vary with age and sex, thus enabling us to calculate all the components of a conventional life table.

In this chapter we will make the necessary modifications to the “wizard model” in order to generate a life table. Risk of death for an actor will now vary along two dimensions: sex and age. When dividing a population into subgroups of “men” and of “women”, we are in fact stratifying it using a categorical variable, namely that of sex. In Modgen, this kind of stratification takes a specific form which is called a “classification”. Age, on the other hand, takes the form of an integer number having a value comprised between 0 and the maximum lifespan. This kind of variable is defined in Modgen as a “range”. The range is simply defined by its minimum and maximum values.

We will also add a “birthday” event to enable us to increment the integer age of the actor (which is not the same as the *age* variable which is automatically put into the model which is a continuous variable). We will then need to make some modifications to the definition of the actor *Person* and to its *Start* function, which, as we

explained before, initializes each individual simulation. New data on age- and sex-related mortality will have to be introduced into the model. And finally we will create new output tables, using derived states similar to the “*duration*” derived state used in the Chap. 1 table. These tables will enable us to derive the individual components which make up the life table: the number of deaths between two exact ages, the number of survivors at each exact age, the number of person-years lived in an age interval, and age-specific life expectancy.

All these new elements will be added to the *PersonCore.mpp* file, in one of the four sections of code defined in the previous chapter: the header, the parameters, the actor *Person* and function definitions, and the tables. The complete file code can be seen in Appendix 2.1 at the end of this chapter, with each of the four sections of the program identified their line numbers. Since no modification will be made to the *ModgenExample.mpp* file, it is not reproduced in the appendices of Chap. 2. But remember that all the files from the complete program can be found on the book website (<http://www.microsimulationandpopulationdynamics.com/>) or on Springer Extras Online.

## 2.1 Adding a Classification

Adding a categorical state variable in a Modgen model is done in two stages. First the variable categories have to be defined in the header of the program, using the *classification* instruction. A classification can be thought of as a blueprint of a variable, or a form of data type. Second, a state variable of a type corresponding to the classification is declared in the *Person* class.

A classification is declared by using the following syntax:

```
classification NAMEOFCLASSIFICATION {list of categories separated with commas};
```

Note the use of curly brackets { } to enclose the values for all the possible categories of the classification, each separated from the previous one by a comma. The semi-colon indicates the end of the instruction to the compiler.

Building on the model created by the wizard, we will create a *SEX* classification and write its defining categories, namely “Female” and “Male”:

```
PersonCore.mpp
1  classification SEX {S_FEM, S_MAL};1
```

A classification is a type of data and not a state variable. In other words, the classification is the template or plan for a state variable. Just as an architect’s plan is not the same as a house, the classification is not the state variable. So the next step is to

---

<sup>1</sup>Remember that the numbers at the beginning of the lines are not part of the code: they refer to the corresponding line numbers in the Appendix.

create a variable which will be of the type *SEX*. As this state variable will contain the characteristics of the actor, its declaration has to be located in the *Person* section (i.e. in the class definition of the actor).<sup>2</sup>

```

                                                                    PersonCore.mpp
22  actor Person    //EN Individual
23  {
    ...

34  SEX sex; // Sex state variable

    ...

49  };

```

In this instruction we create a *sex* state variable (note the lower case) of the *SEX* type (the classification, in upper case). The *sex* state variable has two possible values: *S\_FEM* and *S\_MAL*, as stated above in the classification. *S\_FEM* and *S\_MAL* are the two possible values of the state variable *sex*, in the same way as the digits 0,1,2,3,4,5,... would be the possible values for an integer variable. All this may seem a little confusing at present but will become clear with practice.

It is important to remember that C++ and Modgen are case-sensitive – the same letter in upper and lower case denotes two distinct symbols. So *SEX*, *Sex* and *sex* would refer to three different entities. Capital letters are conventionally used exclusively to define groups or types such as here in *classification SEX*, and lower case letters exclusively for state variables. The values given to the categories of a classification will be in upper case, with the first letter referring to the name of the classification, followed by the first letters of the name of the category, separated by an underline. In the example above we have *S\_FEM* and *S\_MAL*, where *S\_* indicates that *FEM* and *MAL* are part of the *SEX* classification.

Appendix at the end of the book provides a short summary of coding standards in Modgen. Microsimulations are often team projects, so it is not unusual for several people to work on the same program. Coding standards help the programmer to find his way around his own program, but also enable other programmers to understand the coding logic more easily.

---

<sup>2</sup>Three dots (...) in the code mean that some intervening lines of code have been left out. See Appendix 2.1 for the complete code sequence.

## 2.2 Adding a Numerical Range

With Modgen, creating microsimulation models in continuous time is easy: the state variables *age* and *time* which are automatically created by the model wizard are continuous variables. A measure of age in whole numbers (called a discrete variable, because it is one which can only take certain values) is however almost always needed, because the input parameters themselves usually vary by year of age or calendar year.

By using the Modgen *range* command we can create a continuous series of whole numbers, following the same logic as for creating a classification.

```

                                                    PersonCore.mpp
3   range AGE {0,110};
```

The instruction above creates a numerical range comprising 111 integer values, from 0 to 110. Note that the Modgen *range* is just a special case of *classification*. While *classification* defines a type of categorical data, *range* defines a type of ordinal data, creating a range of consecutive whole numbers. We could equally well have created this number range using a classification defined as follows,

```
classification AGE {0, 1, 2, 3, ..., 110};
```

taking care to (tediously) replace the “...” by the complete sequence of digits up to 110.

In the same way as for classification, the numerical range created using the *RANGE* instruction does not generate a state variable but simply a type of data which can be applied to a state variable. So, just as we have done for the *SEX* classification, we have to create a state variable *age\_int* of the type *AGE*, as shown in the code excerpt below:

```

                                                    PersonCore.mpp
24   actor Personne //EN Individual
25   {
    ...

35   AGE age_int = {0}; // Age state variable

    ...

65   };
```

You will note that here, in addition to defining *age\_int* as being of type *AGE*, we initialize this state variable to zero. So by definition the actor being simulated has an

initial exact age of 0, and unless we create some kind of event that increments the age variable, the actor will remain of age 0. A simple way to increment the age state variable is to introduce a “birthday” event into the definition of the actor, so that we can increase its age by one unit every year on the day of its “birthday”.

## 2.3 Adding an Event: The Birthday Event

To create a birthday event, we have to declare a new event in the description of the *Person* actor. Note that a birthday is an event in the same sense as mortality, with the difference that the timing of the event is predetermined and the event is repeatable. The first thing to do is to make a declaration of the new event in the actor *Person*. We also initialize a *TIME* type variable which will define the moment at which the event takes place.

```

                                                                    PersonCore.mpp
22  actor Person                                //EN Individual
23  {
    ...

39      // Duration before next birthday
40      TIME tNextBirthday = { TIME_INFINITE };
41      // Birthday event: increment age_int by 1
42      event TimeBirthdayEvent, BirthdayEvent;

    ...

49  };

```

The first instruction (line 40) creates *tNextBirthday*, a *TIME* type variable which determines the duration of time until the next birthday. The variable is initialised with a *TIME\_INFINITE* value, a Modgen key word representing, as its name implies, an infinite value (in practice, a very large number). This means that the birthday event will never happen unless the *tNextBirthday* value is modified during the simulation. We will set its first value in the *Start* function described later.

The second instruction (line 42) specifies the name of the two functions which together will set the duration before the next birthday and change the actor’s age expressed in completed years. Notice that the syntax for the instruction is the same as for the mortality event predefined by the wizard. In general, all events are stated as follows:

```
event NameOfTimeFunction,EventName;
```

Event and time functions are simply declared in the *Person* class, so their complete content have to be written further down in the program, after the definition of the *Person* class, as shown in the code below.

```

                                                    PersonCore.mpp
75  // The birthday event: increments age_int by 1
76  TIME Person::TimeBirthdayEvent()
77  {
78      return tNextBirthday;
79  }
80  void Person::BirthdayEvent()
81  {
82      if (age_int == AgeMax)
83      {
84          alive = FALSE;
85          Finish();
86      }
87      else
88      {
89          tNextBirthday = WAIT(1);
90      };
91      age_int++;
92
93  }
```

As mentioned earlier, the birthday event differs from the mortality event in two important ways. First, it is a repeatable event whereas death, of course, occurs only once. Second, the duration before the event is not random but fixed, and is always 1 year. This is why there is no need to calculate a randomised wait time in the time function, as was done for the mortality event. Because the “risk” of celebrating a birthday does not vary with time, or with the characteristics of the actor, the time function simply returns the value of *tNextBirthday* to the events manager. If the value of *tNextBirthday* kept its default value (infinite, as defined above), the birthday would never take place. The value of *tNextBirthday* will be first set to 1 year in the *Start* function, and then reset every year in the birthday event function, as described below.

The function determining the birthday event itself has two parts. The first is a conditional statement checking whether the maximum allowable age has been reached (lines 82–90). If maximum age is reached, the current case is ended by modifying the state variable *alive* and then by calling the *Finish()* function, which terminates the case. If maximum lifespan is not reached, the duration before the next birthday is fixed at 1 year (line 89).

The second part is simply an instruction to increment the age in completed years,<sup>3</sup> or in other words to increase the *age\_int* variable by one unit (line 91).

Once the event function has been executed, the duration before the next occurrence of the birthday event is immediately and automatically computed by the *TimeBirthdayEvent* function. In this case, the time function simply collects the value of *tNextBirthday* (set to 1 year in the event) and sends it to the Modgen events manager, who will determine which event is meant to occur next.

You may have noticed that the condition in the birthday event refers to a variable which has not yet been defined: *AgeMax*.<sup>4</sup> We could have written the maximum age directly into the code, which would have given us *if (age\_int==110)*, but this would have prevented a future user from selecting a different maximum lifespan. A researcher working on mortality in the oldest old might want to extend this maximum age by a few years. So it is good practice not to insert constants directly into the code but instead to define them as parameters. *AgeMax* should then be defined in the *parameters* section of the *PersonCore.mpp* file. The value attributed to this parameter will be added later in the *Base(PersonCore).dat* file.

```

                                                    PersonCore.mpp
8   parameters
9   {
    ...

17  int AgeMax;           // Maximum lifespan

    ...

20  };

```

## 2.4 Modifying the *Start* Function

The time of occurrence of the birthday event (*tNextBirthday*) has been initially set to a very distant future (*TIME\_INFINITE*), which in practice means that the event will never take place and that the actor will never “age”. So where and how should the duration before the first anniversary be set? In the model for this chapter, we have to find a place to specify that the first birthday will occur 1 year after the creation of the actor. Note that the initial duration before the first anniversary does not always have to be 1 year: this is because in some models actors can be integrated into the simulation at other moments than at their birth. In a real situation, for

<sup>3</sup>The ++ symbol is used in C++ to increase a variable by one unit: *age\_int++* is equivalent to *age\_int = age\_int + 1*;

<sup>4</sup>Note the use of capital letters for the first letter of each word and the lack of spaces between words, as suggested for a parameter in the coding standards (see Appendix I).

example, the base population includes people of all ages. So the first birthday in the simulation happens less than 1 year after the introduction of the new actor. We specify the duration leading up to the first birthday in the *Start* function, which is the first function to be called when a case is created:

```

                                                    PersonCore.mpp
99   void Person::Start()
100  {
    ...

107  // Sets waiting time before first birthday
108  tNextBirthday = WAIT(1);

    ...

121  }
```

Remember that, just as for the event functions described earlier, the *Start* function needs to be declared first in the description of the actor *Person* (see line 45, Appendix 2.1). Its full code description is written further below using the scope resolution operator (*Person::*).

In addition to initializing the *tNextBirthday* variable, the *Start* function has other uses. Because we want to calculate life tables for men and women separately, we have to create a simple synthetic cohort comprised of members of both sexes. The *Start* function is the place where the sex of the actor is determined.

The actor's sex is attributed randomly using the *RandUniform()* function we introduced earlier in the mortality event function. It might seem tempting to make half the actors male, but in a normal population the number of male births is about 105 for every 100 female births. So the sex of the actors is attributed by way of a conditional statement which compares the ratio of male births (0.512) to a random number between 0 and 1 obtained by the *RandUniform()* function:

```

                                                    PersonCore.mpp
99   void Person::Start()
100  {
    ...

112      if (RandUniform() < ProbabilityMale)
113      {
114          sex = S_MAL;
115      }
116      else
```

```

117      {
118          sex = S_FEM;
119      };
120
121  }
```

Note that the *RandUniform* instruction bracket is left empty, so that Modgen will add a unique argument during precompilation. The argument initializes the pseudo-random number generator. If the same argument is used in different calls of *RandUniform*, the same sequence of numbers will be generated each time. It may be useful to be able to set a fixed sequence of random numbers so that the results can be reproduced. On the other hand, because it is important to avoid creating a correlation between random events in the program, the arguments for each *RandUniform()* function have to be different. This is why Modgen automatically inserts non-redundant arguments for all the *RandUniform()* functions in the model.

So how can different simulation outcomes be obtained if all the random number generators have fixed arguments? In fact it is possible to modify all the starting values of the random number generator when setting up the scenario in the user interface, as we will see later.

Finally, since we used a new parameter (*ProbabilityMale*) for the random sex assignment, it needs to be added to the *parameters* section of the program:

```

                                                                    PersonCore.mpp
8      parameters
9      {

          ...

18     double ProbabilityMale; // Probability of male birth
19
20     };
```

As for *AgeMax*, the precise value of *ProbabilityMale* will be inserted later in the *Base(PersonCore).dat* data file.

## 2.5 Modifying the Mortality Event Function

The final step is to modify the mortality parameter and event function so as to include the new dimensions of age and sex.

First, the parameter generated by the modelling assistant has to be changed from a scalar (constant) to a two-dimensional matrix of rates (according to age and sex). Dimensions are added to a parameter using square brackets [], as follows :

```

                                                    PersonCore.mpp
8   parameters
9   {
10  //EN Annual hazard of death according to sex and age
11  double MortalityHazard[SEX][AGE];

      ...

20  };

```

Note that classifications and numerical ranges (in upper case), and not state variables (in lower case), are inserted in the square brackets immediately following the parameter. Why should *SEX* and *AGE* be used rather than the state variables *sex* and *age\_int*? The first thing to understand here is that state variables generally contain a single value – the value of an actor’s state at a given moment – whereas classifications and ranges contain sets of values. Here, we want to tell Modgen that the *MortalityHazard* parameter’s dimension will be [2] by [111]. So when a classification or range appears inside the square brackets of a parameter, Modgen determines that the specified dimension of that parameter must be of a length equal to the length of the classification or range. In fact, the *SEX* classification has been defined in advance and contains two categories, while the *AGE* range contains 111 years of age. Therefore, Modgen determines that *MortalityHazard[SEX][AGE]* must be a 2 by 111 matrix containing a total of 222 values. In the same way as for *AgeMax* and *MaleProbability*, the *MortalityHazard* mortality rates will be integrated into the parameters file *Base(PersonCore).dat* later on.

Now we have to modify the mortality event function so that sex and age are taken into account. We must add sex and age dimensions to the *MortalityHazard* parameter in this function:

```
tEventTime=WAIT(-TIME(log(RandUniform())/MortalityHazard));
```

Once again, we do this using square brackets []:

```

                                                    PersonCore.mpp
53  TIME Person::timeMortalityEvent()
54  {

      ...

60  tEventTime = WAIT( - TIME( log( RandUniform() ) /
61  MortalityHazard[sex][age_int] ) );
62
63  return tEventTime;
64  }

```

Note that here *sex* and *age\_int* are written in lower case, because it is the age- and sex-specific value of the simulated case that has to be used to identify and select the parameter from the matrix of parameters *MortalityHazard* [*SEX*] [*AGE*]. Furthermore it is the discrete *age\_int* variable which is used here rather than the continuous *age* variable: elements of a matrix are always accessed through an integer (also called an *index*). A male actor aged 30, for example will be exposed to a mortality hazard equal to *MortalityHazard*[1][30]. Notice that the first value of a dimension in a matrix is accessed using the number 0 and not 1 (it is said to be “zero-based”). Modgen also automatically converts the elements of a classification into a sequence of integer numbers giving the first element the value zero, the second 1, the third 2 etc. In the *SEX* classification, then, *S\_FEM* has the value 0 and *S\_MAL* has the value of 1.

## 2.6 Adding Values in the Parameters File

All parameters must have default values so that the base scenario may be loaded in the user interface of the model. We therefore have to modify the instructions in the parameters file so as to add the values of the new elements: the sex ratio, the maximum age at death, as well as the age- and sex- related mortality rates.

Double-click on the *Base(PersonCore).dat* file in the *Scenarios* tab of the *Solution Explorer* to access these parameters (see Fig. 1.4). The .dat files contain the values for each of the parameters for the base scenario. Since the *PersonCore.mpp* file has three parameters, the *Base(PersonCore).dat* file also has to include these three parameters, with their values specified in curly brackets. These may be modified later via the user interface of the model.

Although specifying the value of a constant is simple, entering each of the 222 values needed to complete the mortality rate matrix by hand would be time consuming and prone to error. When a vector or an input data matrix is too big, it is better to work through the user interface to paste in values which have already been tabulated in an Excel spreadsheet. Some parameter values still need to be specified because the compiled program expects to read a predetermined number of values in the .dat file, and failure to do so will prevent it from loading the full scenario.<sup>5</sup> The simplest way to set parameters is to first insert temporary values in the parameters file and then replace them later via the user interface. In the example below, a value of 0.1 is assigned to all ages (111) and both sexes (2): the sequence {(222) 0.1,} means that the 0.1 value is repeated 222 times.

---

<sup>5</sup>Even in a case where the file does not contain the same number of values as that indicated in the parameters section of the .mpp file, it is still possible to compile the program without errors, because the .dat file is not compiled with the rest of the program. However, when trying to access a parameter through the user interface, a runtime error will appear showing that the program is expecting x values for a particular parameter but can read only a number of values which is different from x.

```

Base(PersonCore).dat
1  parameters {
2
3      int      AgeMax = { 110 }; // Maximum lifespan
4      double   ProbabilityMale = { 0.512 }; // Probability
           of male birth
5      // Mortality hazard by sex and age (initialized with
           arbitrary values)
6      double   MortalityHazard[SEX][AGE] = { (222) 0.1, };
7
8  };

```

### Box 2.1: Risk Recalculation and Competing Risks

In this chapter we have seen how to add dimensions to mortality so as to take age and sex into account. You may have noticed that there is no instruction in Modgen to indicate when the wait times have to be recalculated to take account of a change in risk, for instance with a change in age at an anniversary. When and how does Modgen calculate the wait times of events? Or in other words, when is the event's time function called by Modgen?

In fact Modgen may recalculate the wait time of an event in three circumstances: (1) at the beginning of the simulation, (2) each time there is a change in a variable which features in the time function or (3) after the occurrence of an event. For example, whenever a birthday occurs, the age of the actor is increased and the duration to the next birthday is set at 1 year. As soon as the birthday event has occurred, Modgen calls the time function to fetch the duration before the next birthday. As it turns out, this duration is simply given by the *tNextBirthday* time variable whose value was set to 1 year.

Because the birthday event modifies the *age\_int* variable, and because this variable features in the mortality time function, Modgen recalculates the wait time to death according to the new age-dependent mortality parameter.

In one sense, the mortality event competes with the birthday event. If the birthday event occurs first, the age is modified and the duration until death has to be recalculated using the new mortality parameter. If the death occurs first, the case terminates and the birthday will never happen. In other words, for a death to occur, the wait time of the mortality event must be inferior to 1 year, otherwise the anniversary will take place first and the duration until death will be recalculated.

Therefore, Modgen is implicitly and automatically managing changes in risk and competing risks.

## 2.7 Generating New Tables: The Elements of the Life Table

To make the most of the new functionalities inserted in the model, we will have to modify the output tables. We will first create a new table which will contain some elements of the life table (see code below): survivors at an exact age  $x$  ( $l_x$ ), deaths within an age interval ( $d_{x, x+5}$ ), person-years lived during each age interval ( $L_{x+5}$ ).<sup>6</sup> Calculating the number of person-years to be lived from an exact age  $x$  (the  $T_x$ ) is more complex and we will come back to it later.

```

                                                    PersonCore.mpp
133  table Person DeathsAndSurvivors // Elements of the life
      table
134  {
135      {
136
137          value_in(alive), // Survivors ( $l_x$ )
138          entrances(alive, FALSE), // Deaths ( $d_x$ )
139          duration() // Person-years lived ( $L_x$ ) decimals=4
140
141      }
142      *split(age_int, AGE_GROUP) + // by age
143      *sex + // and sex
144  };

```

Let's go back briefly over each of the elements found in the above table. The first line (line 133) contains the table statement (*table Person*) followed by its unique name (*DeathsAndSurvivors*). It basically says that the table *DeathsAndSurvivors* is part of the actor *Person*.

All the table elements appear between curly brackets (between line 134 and 144). These elements are grouped into two sections. First, the output items taken from the derived states appear between their own curly brackets (lines 135–141). Derived states are separated from each other by a comma. Second, additional dimensions are added using the *\** operator, which can be seen here before the expressions *split* (142) and *sex* (143). So the table above will show the data by sex and age group for the three derived states. In summary, a table has this general structure:

---

<sup>6</sup>For more details on the life table, consult a basic demography text such as Rowland (2003).

```

table Person NameOfTable
{
    {DerivedState1, DerivedState2, ... , DerivedStateN}

    *Dimension1
    *Dimension2
    ...
    *DimensionM
};

```

Note that the table statement with its derived states and additional dimensions may be considered as a single Modgen instruction and is ended by a semi-colon. Also, the order in which the dimensions are written will be the default order in which they appear in the user interface. We will see later how the order can be rearranged directly within the user interface.

In Box 1.2 we mentioned that Modgen sometimes uses the information in comments, for example to create a help file or explanatory tabs in the user interface. The comment following the table statement (line 133) also contains the name of the table as it will appear in the user interface (*Elements of the life table*). If a title has not been provided in the comment, the actual name of the table is used (*DeathsAndSurvivors*). The comments specified in lines 137–143 will be the titles for the different derived states and dimensions of the table.

Note that the content of the comments may be preceded by the *EN* abbreviation showing that these are for the English version of the model. If we had built a bilingual model, it would have been possible to include tabs for the French version as well.

Finally, notice the statement *decimals=4* in the comments on line 139. This command specifies the degree of precision with which the data will be displayed in the table (in this case, one ten thousandth of a year). The number of decimal places can also be modified within the user interface.

Now let's take a closer look at the age and sex dimensions of the table.

Because the age dimension ranges from 0 to 110 years, a detailed life table would be unnecessarily large (111 lines). The table can be abridged by showing data only for ages ending in 0 or 5. To do this, we have to build age groups according to limits which are defined in what is called a *partition*. The *partition* instruction is placed in the header of the *PersonCore.mpp* file (see Appendix 2.1), with classifications and numerical ranges, so that they can be easily found as the program becomes more complex.

```

                                                    PersonCore.mpp
5    partition AGE_GROUP {5, 10, 15, 20, 25, 30, 35, 40, 45,
6        50, 55, 60, 65, 70, 75, 80, 85};

```

It is important to understand that the *partition* instruction does not create age groups directly, but defines limits at exact ages. A partition must be used in conjunction with the *split* function in order to create groups. So in the *DeathsAndSurvivors*

table, the *split* function groups the values of the *age\_int* state variable within the *AGE\_GROUP* limits defined by the *partition* instruction. Therefore the first group includes ages zero (the minimum age) to four, the second group includes ages five to nine, and so on. Because the final limit is set at age 85, the last group will be of those actors between 85 and 110 (the maximum age). In other words, the *split* function “cuts” the number range at intervals defined by the partition. If we modified the above partition as follows,

```
partition GROUPE_AGE {1, 5, 10, 15, 20, 25, 30, 35, 40, 45,
50, 55, 60, 65, 70, 75, 80, 85, 100};
```

the first age group would include actors aged zero (the minimum age), the second those aged 1–4, the third those aged 5–9, and so on up to the last age groups aged 85–99 and 100–110.

Partitions are convenient because they allow the model developer to make groupings without having to define new (and redundant) state variables.

The + symbol after the *sex* state variable and the expression *split(age\_int, AGE\_GROUP)* indicates that, in addition to the categories defined for each of the state variables (each sex and age group), the table must also include values for all the categories of each dimension taken together (for both sexes and for all age groups).

### 2.7.1 Derived States in the Table

As we have seen, the values which appear in the table cells by sex and age group are defined using expressions known as derived states, separated by commas and placed between curly brackets { }<sup>7</sup> (lines 135–141). The table *DeathsAndSurvivors* contains three expressions of this kind. The expression *value\_in(alive)* instructs Modgen to add up the values of the state variable *alive* for all the simulated actors reaching an age group defined by the *split* function (0 years, 5 years etc). Because the state variable *alive* takes the value *TRUE* when the actor is alive (which equates to the numerical value 1), the expression delivers the number of survivors at exact ages 0, 5, 10, ... for each sex.

To calculate the number of deaths occurring in a 5 year interval (the  $d_{x,x+5}$ ), the program has to sum all state changes of the *alive* variable in the course of an age interval. To be more precise, it has to count the number of actors whose state variable *alive* takes on the *FALSE* value during a particular age interval. To do this we will use the derived state

```
entrances(statevariable, state)
```

---

<sup>7</sup>Note that there is no comma after the final expression of derived state in the table.

which counts the number of times a given variable (*statevariable*) takes (or “enters”) a specified state value (*state*). In our example (line 138), the derived state *entrances* counts up the transitions of the state variable *alive* into the *FALSE* value – i.e. the number of deaths.

The number of person-years lived in the course of each age interval is obtained by using the *duration* derived state, which we saw in the previous chapter. The table then had no dimensions, so *duration* simply measured the lifespan of the actor. Here, because the table has sex and age group as additional dimensions, the derived state *duration* gives the length of time spent alive in each combination of sex and age group. The same expression of a derived state may thus provide different results depending on the table dimensions. We will come back to the results of these derived states at the end of the chapter, after we have compiled and run the model.

All this is very useful, but the *DeathsAndSurvivors* table does not provide the key indicator in a life table: life expectancy. Remember that life expectancy is defined as the mean amount of time left to live for all the survivors of a cohort at a given exact age. Because the *duration* function can give only the number of person-years for each combination of states (here, sex and age group), it is not possible to obtain this average directly from a given exact age. The *duration* function used with the age group and sex dimensions can tell us the number of person-years lived between 5 and 10, for example, but not the number of person-years still to be lived from age 5 until the death of the actor.

In demographic terms, the derived state *duration* used in conjunction with the age group and sex dimensions provides the  $L_{x+5}$  of the life table. To obtain the life expectancy at a given age, we need instead to calculate the  $T_x$ , or the total number of years of life remaining starting from an exact age  $x$  (i.e. the sum of the  $L_{x+5}$  starting from age  $x$ ). Life expectancy at age  $x$  ( $e_x$ ) is obtained by dividing the  $T_x$  (the total number of person-years lived after exact age  $x$ ) by the number of survivors at exact age  $x$  ( $l_x$ ).

It would be easy to import the results from the *DeathsAndSurvivors* table into an Excel spreadsheet and to calculate the life expectancy at each age by summing the  $L_{x+5}$ . But it is always a good idea to try and limit the amount of external manipulation of data, and so we try to use the full potential of the tools provided by Modgen. So how can we use a combination of derived states to obtain sex and age-related life expectancy without resorting to external software?

First, we can get sex-specific life expectancy at birth simply by suppressing the age dimension: *duration* then provides, as we have seen, the number of years of life starting from birth (the  $T_0$  of the life table), which enables us to obtain life expectancy at birth of men and women:

```

                                                    PersonCore.mpp
146   table Person LifeExpectancyBirth // Life expectancy at
        birth
147   {
148       {
149           // Life expectancy decimals=4
150           duration() / value_in(alive)
151       }
152       *sex +           // By sex
153   };

```

It is possible to add an age filter to the previous table to obtain life expectancy at a given age. A table filter is simply a conditional statement constraining the elements of a table to certain specified state values. A filter is inserted next to the table statement between square brackets. The filter `[age_int >= 5]` inserted under the table *LifeExpectancy5* (see below) creates a table presenting life expectancy at age 5 by specifying that only actors at least 5 years old may be used in the calculations. In demographic notation, *duration* here gives the  $T_5$  and *value\_in(alive)* gives the  $I_5$ . Note that any arbitrary condition may be used as a filter in a table.

```

155   table Person LifeExpectancy5 // Life expectancy at 5 yo
156   [age_int >= 5]
157   {
158       {
159           //Life expectancy at 5 yo decimals=4
160           duration() / value_in(alive)
161       }
162       *sex +
163   };

```

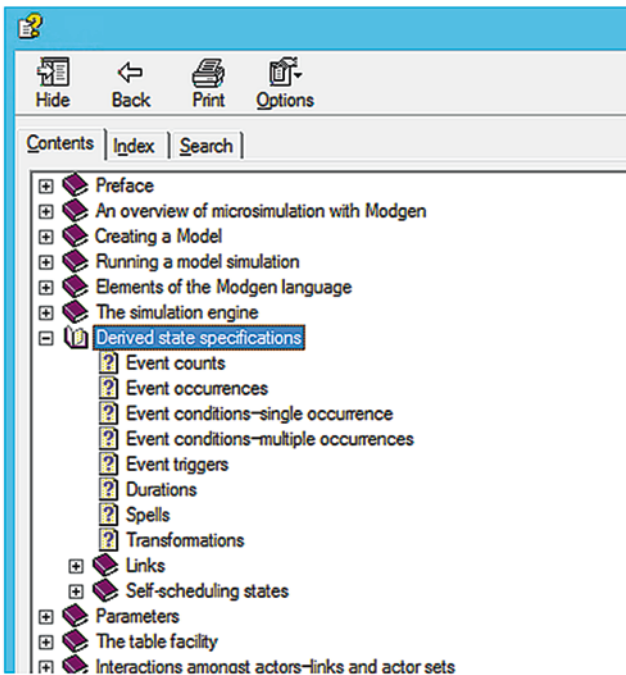
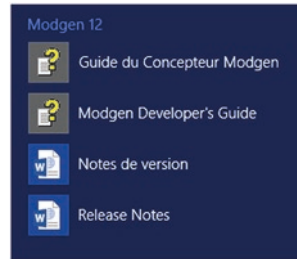
To obtain life expectancy at each age, we could simply copy the above instructions, changing the age specification of the filter. But this would be an inefficient way of doing things, because of the number of lines of code involved and the number of results tables needing to be manipulated. It would be better to create a table which gives life expectancy at each age directly. To do this, it appears that the derived states we have introduced so far are insufficient. Fortunately, Modgen provides dozens of derived states which are described in the developer's guide, the reference work developed by the Modgen team at Statistics Canada.

Let's first open the developer's guide, which is located in the Modgen application group (this was inserted in the start menu by the installer) (Fig. 2.1).

The curious reader is encouraged to take a look at the *Derived states specification* section under the *Contents* tab (Fig. 2.2).

We will here concentrate on the derived states used in this chapter. First, we search for *duration* in the index (Fig. 2.3).

**Fig. 2.1** Accessing the Modgen developer's guide



**Fig. 2.2** The derived states section of the developer's guide

The developer's guide explains that the *duration* derived state can be used without argument (as we have used it up to now) or with arguments. In the latter case, the function takes the form *duration(observed\_state, value)* and measures the length of time spent by an actor in a given state. In our model, the derived state *duration(alive, TRUE)* would be strictly speaking equivalent to *duration()* because both measure the time spent by the actor in the simulation.

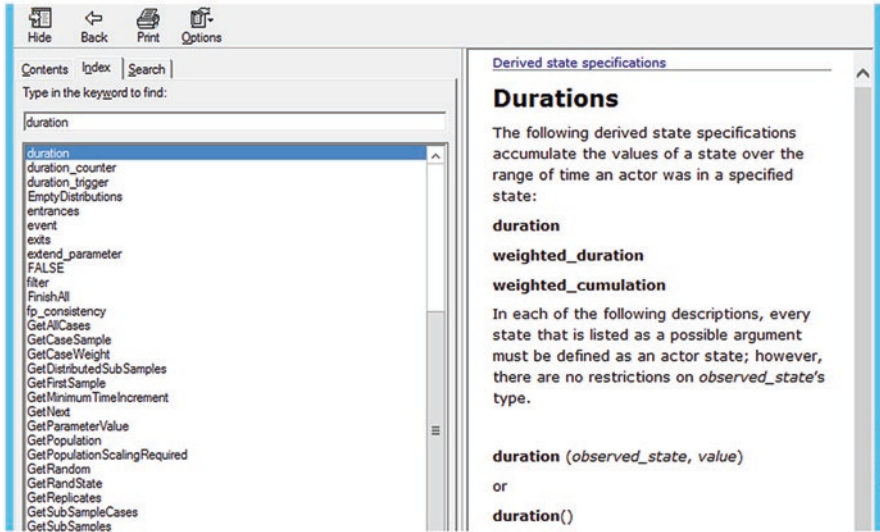


Fig. 2.3 Section of developer's guide explaining the use of the *duration* derived state

The variable used in the function *duration(variable, state)* can be a state variable or a derived state. If we could insert an on-off switch which would make the *duration* function count years only after a certain age, we would have an effect equivalent to the filter we just saw in the *LifeExpectancy5* table, but directly integrated into the derived state. In concrete terms this would mean that a derived state would take on the value *TRUE* only after a given age has been reached.

In fact Modgen does contain a derived state of the *duration* type which can create this kind of switch. This is the derived state *duration\_trigger*:

```
duration_trigger(observed_state, value, time_interval)
```

This derived state takes on the value *TRUE* once an actor has spent a certain amount of time (*time\_interval*) in a predetermined state (*value*), and the value *FALSE* if not. The derived state

```
duration_trigger(alive, TRUE, 5)
```

thus takes the value *TRUE* when an actor has remained in the state *alive = TRUE* for a duration of 5 years or more, and *FALSE* if not. By combining *duration\_trigger* and *duration*, one can calculate the time spent in the state *alive = TRUE* from a certain age onwards:

```
duration (duration_trigger(alive,TRUE,5),TRUE)
```

The expression above measures the time during which the derived state *duration\_trigger(alive, TRUE,5)* has the value *TRUE*, starting from the moment when the actor reaches the age of five. In the life table, this expression corresponds to  $T_5$ . Any  $T_x$  can be obtained by modifying the *time\_interval* argument in the derived state *duration\_trigger*.

Note that it would also have been possible to create a switch in the model itself, by creating a logical variable which can take the *TRUE* value from a given age onwards. *Duration* would then have had the form *duration(Switch, TRUE)*, which would have simplified the description of the table. But then we would have had to create a state variable for each  $T_x$  and then add the relevant code in the anniversary event so that the switches take on the value *TRUE* at the relevant ages. This way of doing it would have overburdened the code unnecessarily. It is always better to exploit the power of the derived states provided by Modgen as much as possible, rather than creating extra functions and state variables.

We now have a way to measure the length of life of an actor after a certain exact age  $x$ . To find life expectancy we also need the number of survivors at this exact age ( $l_x$ ). The number of survivors is quite easy to find by modifying the *entrances* function we used earlier. It is simply a question of counting the number of actors who reach ("enter") a given age:

```
entrances(age_int,5)
```

Here the derived state *entrances* counts the number of actors reaching the age of 5 ( $l_5$  in the life table). Combining all the functions we have seen so far can give us the life expectancy at age 5 ( $T_5/l_5$ ):

```
171 duration(duration_trigger(alive,TRUE,5),TRUE) /
172 entrances(age_int,5)
```

To make a life table by age (the  $T_x/l_x$  or the  $e_x$  of the life table), all we have to do is to copy this instruction for each 5 year segment:

```

165   table Person LifeExpectancyByAge // Life expectancy by
      age
166   {
167     {
168       // Life expectancy at birth decimals=4
169       duration() / value_in(alive),
170       // Life expectancy at 5 yo decimals=4
171       duration(duration_trigger(alive, TRUE, 5), TRUE) /
172       entrances(age_int, 5),
173       // Life expectancy at 10 yo decimals=4
174       duration(duration_trigger(alive, TRUE, 10), TRUE) /
175       entrances(age_int, 10),
176       // Life expectancy at 15 yo decimals=4
177       duration(duration_trigger(alive, TRUE, 15), TRUE) /
178       entrances(age_int, 15),
      ...
      ...

215       // Life expectancy at 80 yo decimals=4
216       duration(duration_trigger(alive, TRUE, 80), TRUE) /
217       entrances(age_int, 80),
218       // Life expectancy at 85 yo decimals=4
219       duration(duration_trigger(alive, TRUE, 85), TRUE) /
220       entrances(age_int, 85)
221     }
222   }
223   *sex +
224   };

```

Note now that only the *sex* dimension needs to be added, because the age dimension is implicitly included in the specification of the derived states.

Derived states and table coding is often a difficult and unintuitive task, so don't worry if at this point Modgen tables still appear a bit confusing. In the following section we will examine the actual output of the tables, and hopefully this will shed some light on the code we have just presented.

Now we can precompile, compile and run our model to examine and compare the results of our life tables. If you have forgotten how to do this, consult the section on Precompiling, Compiling and Running the simulation program in the previous chapter.

## 2.8 Modifying Parameters and Reorganising Tables with the User Interface

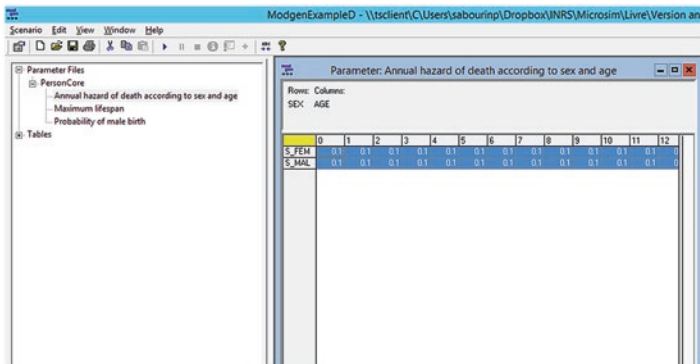
We can now open the user interface by double-clicking on the compiled program icon in the solution folder (to refresh your memory on the procedure, look for Fig. 1.6 of the Precompiling, Compiling and Running the Program section of Chap. 1.) The whole model along with the executable file can also be found on the book website (<http://www.microsimulationandpopulationdynamics.com/>) or on Springer Extras Online.

Once in the user interface, the base scenario (base.scex) can be loaded using the *Scenario* menu or by pressing the *ctrl + o* keys. In the base.scex file, we find the basic simulation parameters (number of cases, scaling, etc.), as well as links to the model parameters files such as *Base(PersonCore).dat*.

Output tables are only available after the first simulation run of the model. So Modgen may alert you to start the simulation, in order to update the results tables. But before launching the simulation, we still have to put in real age- and sex- related mortality parameters. Remember that we inserted some temporary parameter values from Visual Studio, setting a constant risk of dying of 0.1 for each age. You can confirm this by double-clicking on the item *Parameter Files -> PersonCore-> Annual hazard of death according to sex and age* in the left panel of the window.

New parameter values can be entered one by one in this table or, more simply, can be cut and pasted from an Excel spreadsheet. To do this, first copy the cells from an Excel file containing the parameters, and then select the parameter cells in the table (by clicking on the rectangle just above S\_FEM in Fig. 2.4), and finally paste the information.

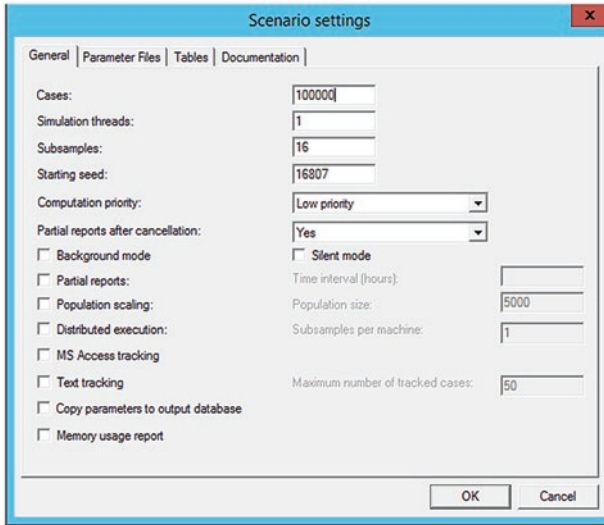
Here we will use the mortality risks observed for Ontario in the year 2006. You can find these parameters in the *Data\_Example2.xlsx* file in the Chap. 2 folder (website: <http://www.microsimulationandpopulationdynamics.com/>).



The screenshot shows the Modgen user interface. On the left, a tree view shows the 'Parameter Files' structure, with 'PersonCore' expanded to show 'Annual hazard of death according to sex and age'. The main window displays a table titled 'Parameter: Annual hazard of death according to sex and age'. The table has columns for AGE (0-12) and rows for S\_FEM and S\_MAL. The values are all 0.1.

Row\Columns	AGE	0	1	2	3	4	5	6	7	8	9	10	11	12
S_FEM		0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
S_MAL		0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

Fig. 2.4 Mortality parameters



**Fig. 2.5** Scenario parameters

These values are saved by pressing *ctrl-s*. Saving them will modify the values of the parameters in the *Base(PersonCore).dat* file.

Next, we make sure that the general scenario parameters are correctly specified (as shown in Fig. 2.5), and launch the simulation by clicking on the *play* (▶) button on the toolbar or on *Run/resume* in the *Scenario* menu.

Once the simulation has run its course and the tables are updated, we can start comparing the results from each of our four life tables. Let's remind ourselves of these tables: *Elements of the life table* shows the constitutive elements of the life table (number of deaths, survivors and person-years lived); *Life expectancy at birth* calculates life expectancy at birth for men and women separately using the expression *duration()/value\_in(alive)*; *Life expectancy at 5 yo* calculates life expectancy at age 5 using a filter excluding actors aged 4 and under; and finally *Life expectancy by age* calculates age-specific life expectancy using a combination of the derived states *duration*, *duration\_trigger* and *entrances*.

Double-clicking on Tables -> Life Expectancy at Birth opens the table shown on Fig. 2.6.

This table shows life expectancy at birth for women (*S\_FEM*) and for men (*S\_MAL*), as well as life expectancy for the population as a whole (*All*). The *All* column is present because the “+” sign had been added after the *sex* dimension in the table definition. We must stress here that this column is not simply the mean of the *S\_FEM* and *S\_MAL* values: it is the life expectancy at birth of the total population both male and female, a mean weighted in accordance to the sex ratio at birth.

We can compare life expectancy at birth with life expectancy at age five by double-clicking on the *Life Expectancy at 5 yo* and *Life expectancy by age* tables. Note that the tables are similar, but that life expectancy at age 5 is lower than life expectancy at birth, which is to be expected in a population with low infant mortality.

The values we find in the *Life expectancy by age* table at birth and at age five should normally be identical to the life expectancy from the other two tables, and this is easy to confirm by comparing the results of three tables (Fig. 2.7).

We can also take a look at the elements of the life table by double-clicking on the *Elements of the life table* (Fig. 2.8).

The table which comes up gives the numbers of survivors by sex (columns) and age (rows). Deaths and person-years lived can be viewed by selecting the relevant item in the *Selected Quantities* drop-down menu (Fig. 2.8).

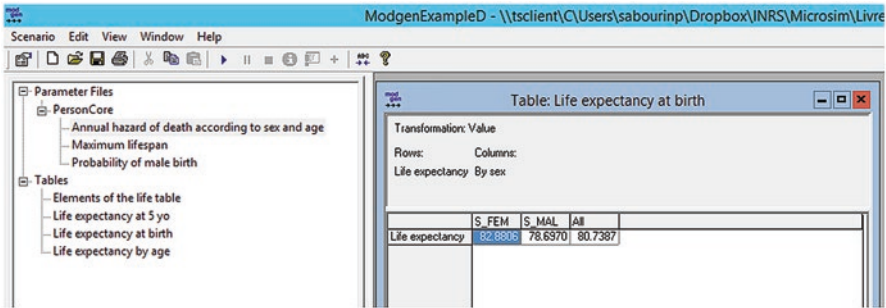


Fig. 2.6 Life expectancy at birth by sex

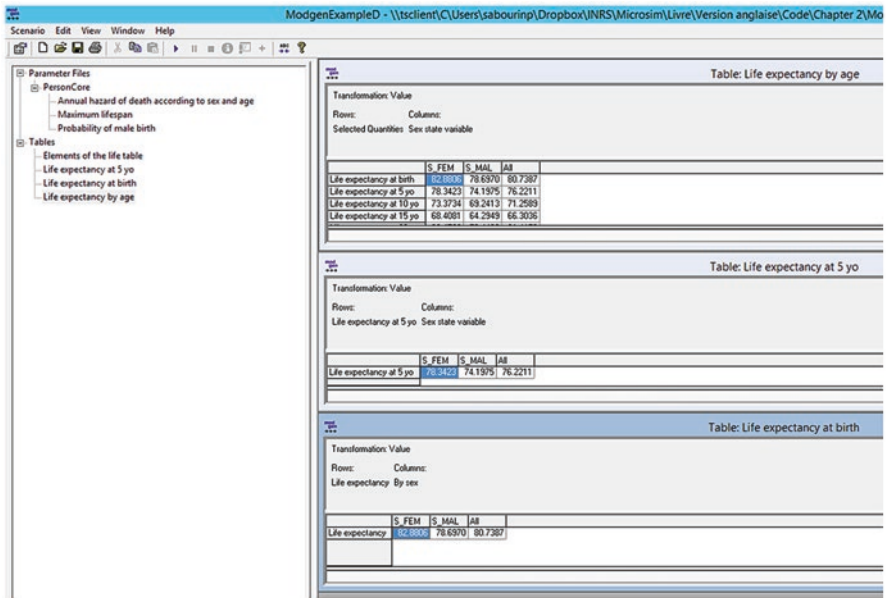


Fig. 2.7 Comparing methods for calculating life expectancy

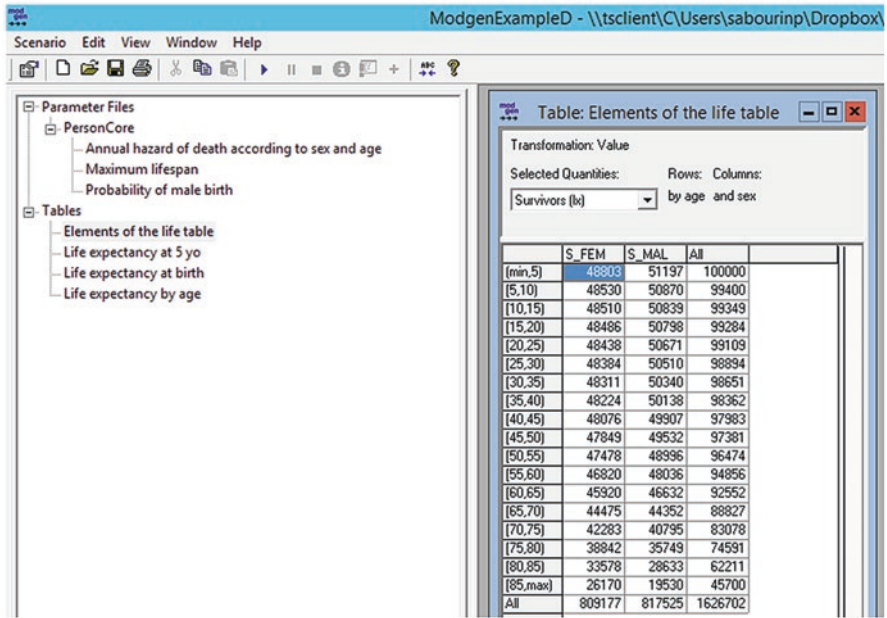


Fig. 2.8 Survivors by age and sex

We can see that the root of the table (the starting population) is 100,000 as specified in the scenario, but that the number of male actors does not exactly match the sex ratio specified in the parameters (0.512). This is explained by the fact that sex attribution, which is done at random in the *Start* function, is subject to the Monte-Carlo error. If we want a life table whose root total population is made up of exactly 100,000 men and 100,000 women, we would have to take each sex separately. Then we can get round the Monte-Carlo error by specifying a sex ratio of 1, which will give us a table for men alone (with a root of 100,000). Repeating the simulation with a sex ratio of 0 will give us a table for women alone.

It may sometimes be useful to reorganise the dimensions of the table so as to make certain characteristics stand out. In our example, it would be useful to show the table contents (in columns) by age (in rows) for men and for women in two distinct tables. To modify the order of the dimensions, click on the table, and select *Properties* in the *View* menu (you can also find *Properties* by right-clicking anywhere on the table) (Fig. 2.9).

Under the *Dimensions* tab of the *Properties* window, we can find the table contents (*Selected Quantities*), age and sex. The dimensions can be moved using the arrows on the right of the window. The last dimension in the list always represents the columns, and the next to last the rows of the table. To display the table contents by age, then, we move *Selected Quantities* to the last position, and *Age* to the last but one. This will give the desired table as shown in Fig. 2.10. Note that you can also modify the number of decimals displayed for each quantity by clicking on the *Decimals* tab.

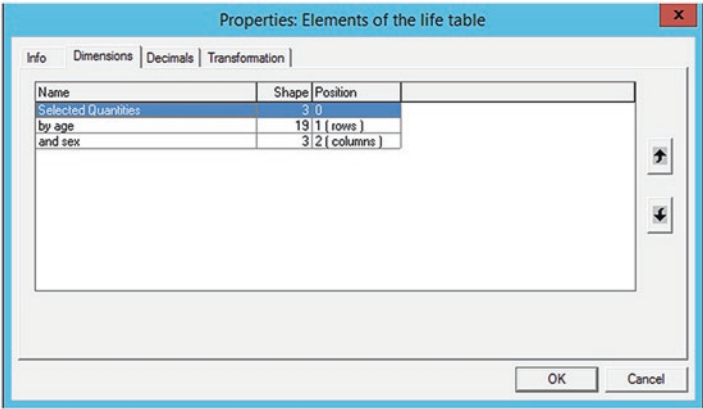


Fig. 2.9 Table properties

Fig. 2.10 Survivors, deaths and person-years lived by age

Transformation: Value  
and sex: Rows: Columns:  
S\_FEM by age Selected Quantities

	Survivors (lx)	Deaths (dx)	Person-years lived (Lx)
(min,5)	48800	273	242873.4705
(5,10)	48530	20	242604.6206
(10,15)	48510	24	242508.9713
(15,20)	48486	48	242321.0838
(20,25)	48438	54	242040.6749
(25,30)	48384	73	241736.6966
(30,35)	48311	87	241335.7928
(35,40)	48224	148	240766.9149
(40,45)	48076	227	239916.6935
(45,50)	47849	371	238325.3210
(50,55)	47478	658	235870.8721
(55,60)	46820	900	231962.9978
(60,65)	45920	1445	226253.5409
(65,70)	44475	2192	217247.1846
(70,75)	42283	3441	203452.3242
(75,80)	38842	5264	182043.1746
(80,85)	33578	7408	150677.5975
(85,max)	26170	26170	182385.1851
All	809177	48803	4044823.3167

Note that the sex dimension now appears in the drop-down menu, so this table can be displayed either for women or for men.

The tables can be exported simply, by cutting and pasting the required cells into a spreadsheet program. The entire table can also be cut and pasted by right-clicking on any cell and selecting *Special Copy -> All Pages*. Finally it is possible to export one or more tables of results directly into an Excel format: just select *Export* in the *Scenario* menu and follow the instructions on screen.

## 2.9 Summary

In this chapter we have seen how to create classifications, numerical ranges and state variables. We have put age and sex into our model and added new parameters. We have also added an event, the birthday, and modified the mortality event so that variations in risk by age and sex may be taken into account. We have learned to work with new derived states (*duration*, *entrances*, *duration\_trigger*) and with filters, to produce more useful results tables. Finally we have learned how to personalise the parameters and the tables through the user interface. In the next chapter we will add a spatial dimension to the model, which will enable us to model the risk of migration from one region to another. With this model we will be able to reproduce the results of a multiple increment-decrement life table, also known as multi-regional or multi-state table.

## Appendices

### *Appendix 2.1 PersonCore.mpp*

Code Sections: Header (lines 1 to 7), Parameters (lines 8 to 21), Actor *Person* and functions (lines 22 to 132), Tables (lines 133 to 224)

```

1  classification SEX{ S_FEM, S_MAL };
2
3  range AGE{ 0, 110 };
4
5  partition AGE_GROUP{ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50,
6  55, 60, 65, 70, 75, 80, 85 };
7
8  parameters
9  {
10     //EN Annual hazard of death according to sex and age
11     double MortalityHazard[SEX][AGE];
12     /* NOTE(MortalityHazard, EN)
13        The hazard of death according to sex and age;
14        results in an exponential survival function.
15     */
16
```

```

17     int AgeMax;                // Maximum lifespan
18     double ProbabilityMale;    // Probability of male birth
19
20 };
21
22 actor Person                    //EN Individual
23 {
24
25     //EN Alive
26     logical alive = {TRUE};
27     /*NOTE(Person.alive, EN)
28         Set to TRUE when the actor starts, and to FALSE just
29         before
30         the actor finishes. Since the numeric value of TRUE
31         is 1 and
32         FALSE is 0, this variable can also be used
33         to count actors in tables.
34     */
35
36     SEX sex;                    // Sex state variable
37     AGE age_int = { 0 }; // Age state variable
38
39     event timeMortalityEvent, MortalityEvent; //EN Mortality
40     event
41         // Duration before next birthday
42         TIME tNextBirthday = { TIME_INFINITE };
43         // Birthday event: increment age_int by 1
44         event TimeBirthdayEvent, BirthdayEvent;
45
46         //LABEL(Person.Start, EN) Starts the actor
47         void Start();
48
49         //LABEL(Person.Finish, EN) Finishes the actor
50         void Finish();
51 };
52
53 // The time function of MortalityEvent
54 TIME Person::timeMortalityEvent()
55 {

```

```
55     TIME tEventTime = TIME_INFINITE;
56
57     // Draw a random waiting time to death from
58     // an exponential distribution based on the
59     // constant hazard MortalityHazard.
60     tEventTime = WAIT( - TIME( log( RandUniform(1) ) /
61         MortalityHazard[sex][age_int] ) );
62
63     return tEventTime;
64 }
65
66 // The implement function of MortalityEvent
67 void Person::MortalityEvent()
68 {
69     alive = FALSE;
70
71     // Remove the actor from the simulation.
72     Finish();
73 }
74
75 // The birthday event: increments age_int by 1
76 TIME Person::TimeBirthdayEvent()
77 {
78     return tNextBirthday;
79 }
80 void Person::BirthdayEvent()
81 {
82     if (age_int == AgeMax)
83     {
84         alive = FALSE;
85         Finish();
86     }
87     else
88     {
89         tNextBirthday = WAIT(1);
90     };
91     age_int++;
92 }
93 }
94
95 /*NOTE(Person.Start, EN)
96     The Start function initializes actor variables before
97     simulation
98     of the actor commences.
99 */
```

```

99 void Person::Start()
100 {
101     // Modgen initializes all actor variables
102     // before the code in this function is executed.
103
104     age = 0;
105     time = 0;
106
107     // Sets waiting time before first birthday
108     tNextBirthday = WAIT(1);
109     // Sets age_int to zero (cohort model)
110     age_int = 0;
111
112     if (RandUniform(2) < ProbabilityMale)
113     {
114         sex = S_MAL;
115     }
116     else
117     {
118         sex = S_FEM;
119     };
120
121 }
122
123 /*NOTE(Person.Finish, EN)
124     The Finish function terminates the simulation of an actor.
125 */
126 void Person::Finish()
127 {
128     // After the code in this function is executed,
129     // Modgen removes the actor from tables and from the
130     // simulation.
131     // Modgen also recuperates any memory used by the actor.
132 }
133 table Person DeathsAndSurvivors // Elements of the life table
134 {
135     {
136
137         value_in(alive), // Survivors (lx)
138         entrances(alive, FALSE), // Deaths (dx)
139         duration() // Person-years lived (Lx) decimals=4
140
141     }
142     *split(age_int, AGE_GROUP) + // by age

```

```

143      *sex +                                // and sex
144  };
145
146  table Person LifeExpectancyBirth // Life expectancy at birth
147  {
148      {
149          // Life expectancy decimals=4
150          duration() / value_in(alive)
151      }
152      *sex +                                // By sex
153  };
154
155  table Person LifeExpectancy5 // Life expectancy at 5 yo
156  [age_int >= 5]
157  {
158      {
159          //Life expectancy at 5 yo decimals=4
160          duration() / value_in(alive)
161      }
162      *sex +
163  };
164
165  table Person LifeExpectancyByAge // Life expectancy by age
166  {
167      {
168          // Life expectancy at birth decimals=4
169          duration() / value_in(alive),
170          // Life expectancy at 5 yo decimals=4
171          duration(duration_trigger(alive, TRUE, 5), TRUE) /
172          entrances(age_int, 5),
173          // Life expectancy at 10 yo decimals=4
174          duration(duration_trigger(alive, TRUE, 10), TRUE) /
175          entrances(age_int, 10),
176          // Life expectancy at 15 yo decimals=4
177          duration(duration_trigger(alive, TRUE, 15), TRUE) /
178          entrances(age_int, 15),
179          // Life expectancy at 20 yo decimals=4
180          duration(duration_trigger(alive, TRUE, 20), TRUE) /
181          entrances(age_int, 20),
182          // Life expectancy at 25 yo decimals=4
183          duration(duration_trigger(alive, TRUE, 25), TRUE) /
184          entrances(age_int, 25),
185          // Life expectancy at 30 yo decimals=4
186          duration(duration_trigger(alive, TRUE, 30), TRUE) /
187          entrances(age_int, 30),

```

```

188      // Life expectancy at 35 yo decimals=4
189      duration(duration_trigger(alive, TRUE, 35), TRUE) /
190      entrances(age_int, 35),
191      // Life expectancy at 40 yo decimals=4
192      duration(duration_trigger(alive, TRUE, 40), TRUE) /
193      entrances(age_int, 40),
194      // Life expectancy at 45 yo decimals=4
195      duration(duration_trigger(alive, TRUE, 45), TRUE) /
196      entrances(age_int, 45),
197      // Life expectancy at 50 yo decimals=4
198      duration(duration_trigger(alive, TRUE, 50), TRUE) /
199      entrances(age_int, 50),
200      // Life expectancy at 55 yo decimals=4
201      duration(duration_trigger(alive, TRUE, 55), TRUE) /
202      entrances(age_int, 55),
203      // Life expectancy at 60 yo decimals=4
204      duration(duration_trigger(alive, TRUE, 60), TRUE) /
205      entrances(age_int, 60),
206      // Life expectancy at 65 yo decimals=4
207      duration(duration_trigger(alive, TRUE, 65), TRUE) /
208      entrances(age_int, 65),
209      // Life expectancy at 70 yo decimals=4
210      duration(duration_trigger(alive, TRUE, 70), TRUE) /
211      entrances(age_int, 70),
212      // Life expectancy at 75 yo decimals=4
213      duration(duration_trigger(alive, TRUE, 75), TRUE) /
214      entrances(age_int, 75),
215      // Life expectancy at 80 yo decimals=4
216      duration(duration_trigger(alive, TRUE, 80), TRUE) /
217      entrances(age_int, 80),
218      // Life expectancy at 85 yo decimals=4
219      duration(duration_trigger(alive, TRUE, 85), TRUE) /
220      entrances(age_int, 85)
221
222  }
223  *sex +
224  };

```

***Appendix 2.2 Base(PersonCore).dat***

```
1 parameters {
2
3     int    AgeMax = { 110 }; // Maximum lifespan
4     double ProbabilityMale = { 0.512 }; // Probability of male
        birth
5     // Mortality hazard by sex and age (initialized with arbitrary values)
6     double    MortalityHazard[SEX][AGE] = { (222) 0.1, };
7
8 };
```

## Chapter 3

# The Multiple Increment-Decrement Life Table

### Aims of This Chapter

- Replacing the birthday event by the *self\_scheduling\_int* function
- Creating a new Modgen module to simulate interregional mobility
- Using a *cumrate* type parameter and the *Lookup* function to randomly attribute a new state to an actor
- Creating customized tables

In the mid 1970S, Professor Andrei Rogers (1975) proposed adapting the life table methodology for regional analysis. The classic demographic approach often abstracted migration, which was thus considered to be a disrupting phenomenon. Rogers had the idea that migration could be taken into account explicitly by using a multi-regional table which would make it possible to calculate, for example, the share of life expectancy spent in a specific region (or country). The multi-regional model he developed is now an integral part of the demographer's toolkit.

The multi-regional model is appropriate for analysing any phenomena involving transitions between different modalities of a state variable. The multi-regional table therefore becomes a “multi-state” table, or a multiple increment and decrement table. This type of model, which is more general in scope, has contributed to analysing a wide range of phenomena such as healthy life expectancy (Rogers et al. 1990; Bélanger et al. 2002), active life expectancy (Bélanger and Larrivée 1992; Willekens 1980) or the analysis of married life (Zeng et al. 2012; Willekens et al. 1982).

In this chapter we will be modifying the program we developed in the previous chapter, to convert it into a multi-regional model; this is a transposition of the original work by Rogers into the world of microsimulation. To do this, we will be adding new elements to the model to account for interprovincial mobility. The first step will be to add a new state to the actor, namely his or her province of residence, and initialise it at the time of birth. We will do this using the *Lookup* function and a new type of Modgen parameter, the *cumrate*. Used together, *Lookup* and *cumrate* allow

us to assign states to actors randomly according to a predetermined distribution. We will also add a migration event for which we will need new parameters, namely province-specific outmigration rates and origin-destination matrices. Finally, we will add a dimension to the mortality parameter to make the rates specific to each province.

As in previous chapters you will find the complete code of the relevant files at the end of the chapter. You can also access the files on the book's website (<http://www.microsimulationandpopulationdynamics.com/>) or on Springer Extras Online. But before we start to add a new migration module, we will simplify the code in Chap. 2 by replacing the birthday event by a convenient Modgen element called a self-scheduling state.

### 3.1 Self-Scheduling Events: The *self\_scheduling\_int* Function

In the previous chapter we saw how to increment the age of an actor by using the anniversary event. If the maximum age was reached when this event occurred, the *alive* state variable was switched to FALSE and the actor was withdrawn from the simulation. This repeatable event with fix duration was a convenient way to introduce the new user of Modgen to the creation of a new event. However, the *self\_scheduling\_int* function is a simpler way to do the same thing.

Modgen allows the value of a variable to be automatically increased without having to use an explicit event function. The *self\_scheduling\_int* function automatically increments a state variable by one unit for each year that passes. The moment when this increase takes place is determined by the value of another variable which changes in continuous time (such as the *time* and *age* variables for example, which are automatically defined by Modgen). Each time the integer part of a reference variable increases, *self\_scheduling\_int* increments a designated state variable. In the example below, the *self\_scheduling\_int* function automatically modifies the value of the variable *age\_int* according to the evolution of the continuous variable *age*.

```
int age_int = self_scheduling_int(age);
```

For example, when the *age* variable reaches the 2 year point, the *self\_scheduling\_int* function automatically updates the value of the *age\_int* variable, so that its value changes from 1 to 2.

The *self\_scheduling\_int* function (or its sister *self\_scheduling\_split*, which won't be discussed here) used to be a well-kept secret and was not documented in previous versions of the Modgen User's Guide. However it was described in an article by Martin Spielauer on Modgen and the Riskpath microsimulation model (Spielauer 2009).

It is a very useful function, and has been used to replace the birthday event which we introduced in the second chapter. To be precise, we have removed the birthday event (the declaration of the event in the actor description as well as the event code itself) and replaced it by the following line of code in the *Person* class:

```

                                                    PersonCore.mpp
26    // Age state variable, auto-increment
27    AGE age_int = COERCE(AGE, self_scheduling_int(age));

```

You will notice that the *self\_scheduling\_int(age)* function is inserted in another function, *COERCE*, which has not been introduced yet. This function acts to restrict a value so that it is confined within the limits defined by a Modgen range. Because the numerical range *AGE* has values between 0 and 110, the *COERCE* function will prevent the *self\_scheduling\_int(age)* function from returning a value above 110 or below 0. If an individual were to attain the age of 111, the *coerce* function would force the *age\_int* variable to take the maximum value allowed for the *AGE* range: 110 years.

A good modeller will normally ensure that the state variables cannot take on values outside the defined limits of a numerical range. However, the *coerce* function adds an extra layer of security.

In Chap. 2, age was constrained by the birthday event, which included a conditional statement to check whether the actor had reached the maximum age. Because we have removed the birthday event, we must now relocate this task to another event, and the mortality event seems the most appropriate for this:

```

                                                    PersonCore.mpp
43    // If max age is reached, death event occurs immediately
44    if (age_int == AgeMax)
45    {
46        tEventTime = WAIT(0);
47    }
48    else
49    {
50        // Draw a random waiting time to death from
51        // an exponential distribution based on the
52        // constant hazard MortalityHazard.
53        tEventTime = WAIT(-TIME(log(RandUniform(1)) /
54        MortalityHazard[sex][age_int][prov]));
55    };

```

This conditional statement simply states that if the maximum age has been reached, the death event occurs immediately (*WAIT(0)*); otherwise a random waiting time is computed using a Monte-Carlo experiment. What used to be done in the birthday event occurs now in the mortality event.

## 3.2 Creating a New Module for Interprovincial Migration

Having introduced these preliminary modifications into the code, we are now ready to add a new module to the model. But first, let's clarify what a module is and what it is not.

It is important to understand that a Modgen module is not strictly speaking a distinct programming block or entity, in the same way that a function or a class in C++ would be. Instead, a module is an element of organisation which enables us to put different functional parts of the model into separate files. So, for instance, we could put the code relating to mobility into one module and the code relating to fertility into another. In addition to allowing us to organise the code, modules make working in parallel possible, so that several modules can be developed separately by different members of a team.

Organizing the code into modules is not mandatory. It would be possible, although not very practical, to build a complex model without creating modules, by putting all the code into the *PersonCore.mpp* file. But it is easy to imagine how difficult this would be to achieve: a very large file, the risk of a poorly organised code, and no possibility of parallel working. It is good practice, even for smaller models, to create a specific module for each specific category of event.

A new Modgen module is simply a new.mpp file integrated into the Visual Studio project of a model. The file is created from the Visual Studio *Solution Explorer*: position the mouse pointer on the *Modules(mpp)* sub-directory and right-click to access the contextual menu and then select *Add -> New Item* (Fig. 3.1). Then click on “*Modgen12EmptyModuleVide*” which is available under the *Modgen* heading (Fig. 3.2). Give a name (*Migration*) to this new module and make sure that the location of the file corresponds to the project working directory. Then click on *Add*.

The new “*Migration*” module now appears in the *Solution Explorer*. If you double-click on it, a blank file opens (Fig. 3.3).

Because the simulation of migration will require new parameters, it is recommended to create a parameters file specifically for this new module. To do this, you need to create a new .dat file in the *Scenarios* folder. Right-click on the *Scenarios* folder in the *Solution Explorer* and then on *Add-> New Item* (Fig. 3.4).

In the new window, select *Text File* in the *Utility* section under Visual C++. Write the name of the file: *Base(Migration).dat* would be a good choice as it indicates that this is the parameters file of the *Migration* module for the base scenario. But ultimately the choice of the name is up to the developer. Finally click on *Add* (Fig. 3.5).

The new empty parameters file appears in the main window and in the *Solution Explorer*. We will close it for the moment, and come back to it later (Fig. 3.6).

Let's go back now to the new empty migration module. Its structure will be identical to the structure of the *PersonCore.mpp* file. It will include a header (with declarations of classifications and ranges, for example), a section for the parameters, a section for the description of the actor class (*Person*) and a section for the tables. All these sections mirror the equivalent sections in *PersonCore.mpp*. In fact, even though all the elements of a model may be divided into several modules, you can

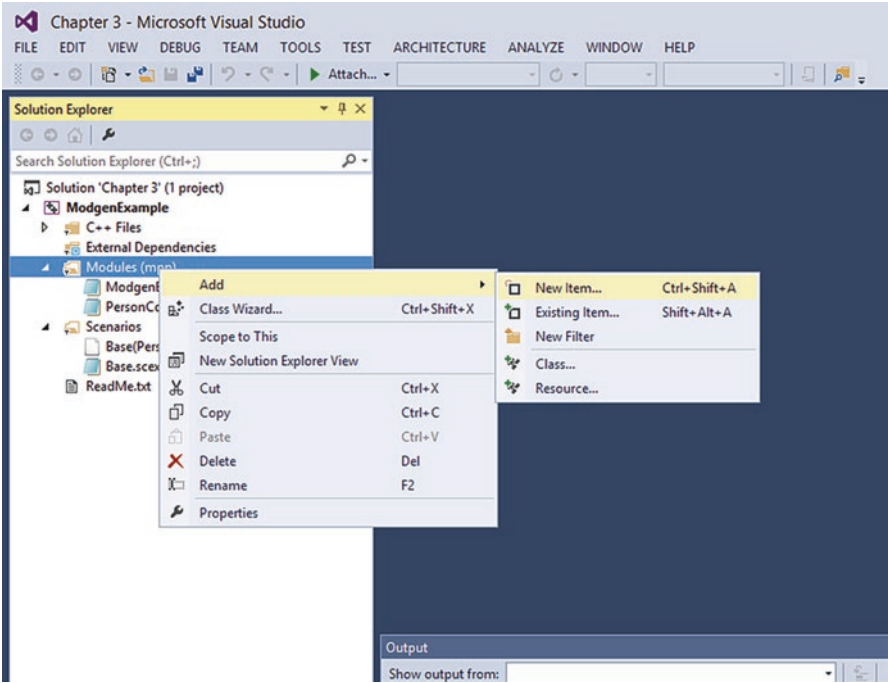


Fig. 3.1 Accessing the menu to add a new Modgen module

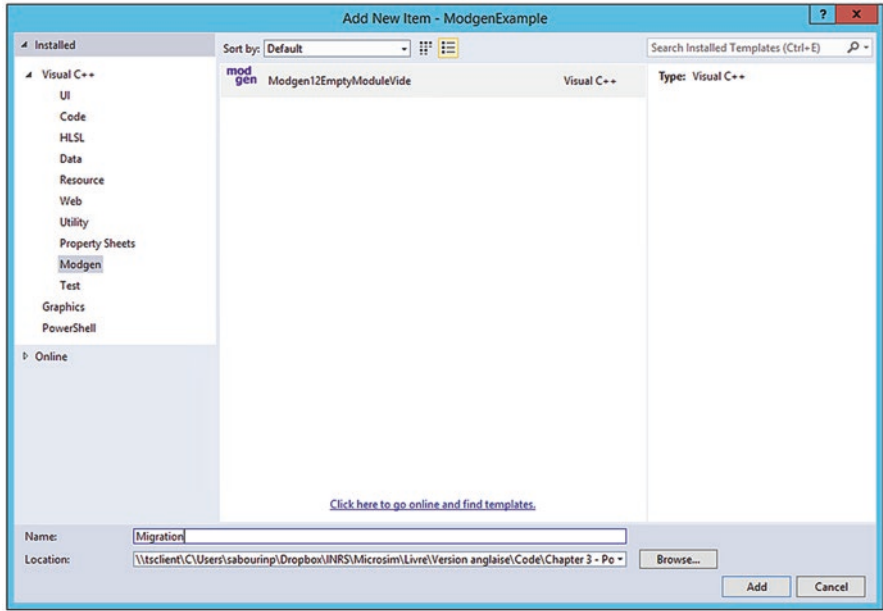


Fig. 3.2 Adding a Modgen module



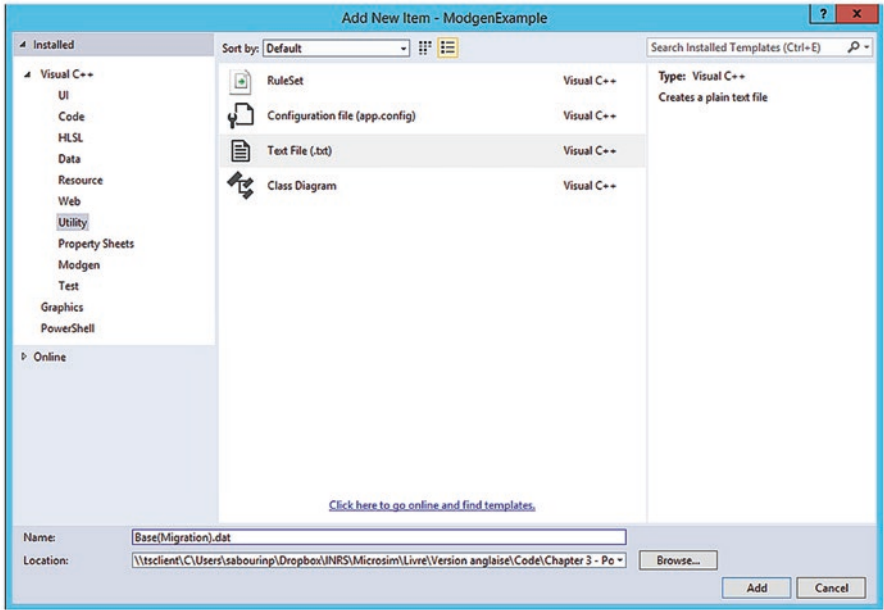


Fig. 3.5 Creating a new parameters file

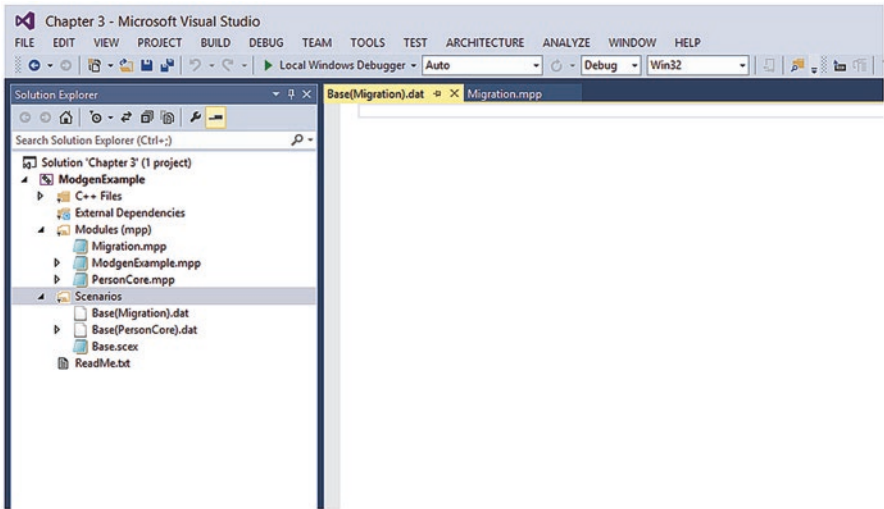


Fig. 3.6 New parameters file for the Migration module

think of the model as having a single parameter section, a single header section, and so forth. The elements of the modules are combined during compilation, so that code which is already in one file should not be repeated in others. Consequently, all the elements contained in one module are accessible in all the others. This is very important. For example, the classification *PROVINCE* which we are going to create in the *Migration.mpp* module may also be used in the main module (*PersonCore.mpp*) to add a new dimension to the mortality parameter or a new dimension to a table. Equally, the classifications declared in *PersonCore.mpp* will also be available in the migration module, so there is no need to declare the AGE classification again in the migration module. In fact, if we did this, the compiler would signal an error because the same classification would be defined twice in the same project.

So what should we include in a mobility module? First of all, it is useful to start by describing the content of the module in the file header. The description should explain broadly and succinctly the purpose of the module.

```

                                                                Migration.mpp
1  /* Internal Migration Module
2  This module contains all the elements of the mobility
    event.
3  The Lookup function is used in conjunction with an
4  origin-destination matrix (cumrate) to determine
5  the choice of a destination province.
6  */
```

Next the classifications and numerical ranges used in the new module are stated. In the case of a mobility module, we will need a state variable to hold the place of residence or the place of birth of the actor. Because our microsimulation model is Canadian, we will choose the province as the geographical unit, but the choice could have been different depending on the available data (the state, the county or the department, for example), and on the analytical decisions made by the modeller (what research questions do we want this microsimulation model to help us answer?). So we create a “Province” classification covering the ten Canadian provinces (the territories are not included in this model).

```

                                                                Migration.mpp
8  classification PROV{ P_NFL, P_PEI, P_NOV, P_NEB, P_QUE,
    P_ONT,
9  P_MAN, P_SAS, P_ALB, P_BRC };
```

Note again the use of capital letters in the declaration of a classification and the building of modalities based on the first letter of the classification (P) and three significant letters from the province name. The first three letters of the province would be a good choice, but sometimes it is more meaningful to use some other

acronym (NFL for Newfoundland and Labrador for instance is more meaningful than NEW). So the modeller can depart from writing conventions (by not taking the first three letters of the province for example), but should make sure to retain a degree of logic and clarity in the naming system.

Now that we have a classification for the province, we have to give the actor a province of birth and a province of residence. But in order to do this, we first have to introduce two new elements of the Modgen programming language: the *cumrate* parameter and the *Lookup* function.

### 3.3 Assigning a Province of Birth Using a *cumrate* Parameter and the *Lookup* Function

Suppose we want to assign a value to a state variable according to a pre-defined distribution, for example to assign a sex value at birth according to a sex distribution of 105 men to 100 women (the natural sex ratio at birth). This is what we did in the previous chapter when we assigned a sex to the actor in the *start()* function. Because the *sex* variable had only two modalities, we used a simple condition to assign sex: if the value of a random number was lower than the probability of being male, the actor's sex was male. If not, the actor was female. But what if the number of modalities of a state variable is higher than two, as in the case of region of birth, where there are 10 modalities? Do we have to multiply the conditional statements accordingly? What should we do if we want to use a distribution which has several dimensions? For example a destination province might be assigned taking account of the province of residence, age and sex. This could make assigning a state quite complicated.

Fortunately the *Lookup* function and the *cumrate* parameter in Modgen make it possible to manage all this automatically.

The *cumrate* parameter (from “cumulative rate”) is a statistical distribution containing the relative frequencies of the modalities of a given state variable. Let's take the example of the province of birth. We have just defined a province classification (*PROV*) which has ten modalities (the ten provinces of Canada). When an actor is created in the *start()* function, a province of birth has to be assigned to him or her. This province of birth is also the province of residence at the time of birth. The distribution of births is contained in a *cumrate* parameter.

```

Migration.mpp
17 // Distribution of births according to province
18 cumrate ProvBirth[PROV];
```

This command creates a parameter along a single dimension (*PROV*, because the birth province is one of the ten provinces). This distribution is specified in the parameter file associated with the mobility module (see Appendix 3.3, *Base(Migration).dat*) :

```

Base(Migration).dat
8   cumrate ProvBirth[PROV] = {1,1,1,1,1,1,1,1,1,1};

```

Notice first that the distribution has not been standardised (meaning that the sum of its elements is not equal to one): Modgen itself standardises it when selecting a random state with the *Lookup* function. Any number repeated ten times would yield the same result. In our example for this chapter, we assume that births are randomly distributed across all the provinces. So we will use this distribution to construct the multi-regional life tables. However, we might have wanted to assign a birth province based on real data. In that case we could have used the numbers of births in each province to construct the distribution, which would have given us the following<sup>1</sup>:

```

cumrate ProvBirth[PROV] =
    {4455,1423,8572,6826,88250,142448,16237,
    15367,56582,43738};

```

The sum of all these numbers is 385,937, and so this distribution implies that the probability of having Newfoundland and Labrador as a province of birth (the first in the list) is  $4455/385937=0.012$ . For Ontario (the sixth), the probability is  $142448/385937=0.369$  and so on.

You may have noticed that a new classification has been created to define the provinces (*PROV*), but that no corresponding state variable has been added to the actor definition. We can create these now.

```

Migration.mpp
22   actor Person           // Individual
23   {
24       ...
25       PROV prov;
26       PROV prov_birth;
27   };

```

We have added two new state variables (in lower case letters), the province of residence and the province of birth, both based on the *PROV* classification (in capitals). Remember that the case is important. *PROV* and *prov* are two distinct entities: the first refers to an invariable classification (a type of data), while the second refers to a state variable.

These state variables have to be initialised at the beginning of the simulation, in the *start* function of the actor. Remember that the *start* function is defined in the main *PersonCore.mpp*, and not in the migration module.

We can then use the information contained in the *cumrate* parameter to randomly select the actor's birth province in the *start* function. It is at this point that the

<sup>1</sup>According to Statistics Canada data, 2013–2014: <http://www.statcan.gc.ca/tables-tableaux/sum-som/102/cst01/demo04a-fra.htm>. Consulted 21 May 2015.

*Lookup* function comes into play. Its syntax is rather unusual (and not self-explanatory), so we will go through it step by step.

```

                                                                    PersonCore.mpp
70  void Person::Start()
71  {
    ...

91      // A temporary variable to store the province value
92      int prov_temp = {0};
93      // Lookup picks a random province according to cumrate
        distribution
94      Lookup_ProvBirth(RandUniform(5), &prov_temp);
95      // Casts the (integer) prov_temp value into the
96      // prov state variable (of PROV type)
97      prov = (PROV) prov_temp;
98      // Province of residence at birth is the province of
        birth
99      prov_birth = prov;
100
101 }
```

The use of the *Lookup* function is generally done in three stages. First a temporary variable is created (line 92) which will eventually contain the state picked randomly by the *Lookup* function. This variable is of type integer (int), because *Lookup* returns the state value in the form of a whole number (remember that the elements of a classification are ordered, with the first element taking the value 0, the second 1, and so on). So in our example, because the *cumrate ProvBirth* parameter allows for ten possible values (corresponding to the 10 provinces), the *Lookup* function will return a whole number between 0 and 9.

Next the *Lookup* function is called to return a randomly selected province in the previously defined temporary variable (line 94). Finally we transfer the value of the temporary variable *prov\_temp* into the state variable *prov* (line 97) and define the birth province as the province of residence at birth (line 99).

Let's go back over the last two stages (creating a temporary variable is relatively trivial and doesn't require further explanation). We will take a closer look at the syntax of the *Lookup* function:

```
Lookup_[cumrate    distribution](RandUniform(),    &dest_vari-
able);
```

We can see that the *Lookup* function is tied to a *cumrate* parameter using an underline: in our example, *Lookup* is tied to the province of birth distribution (*Lookup\_ProvBirth*). The *Lookup* function paired with a *cumrate* also takes two

arguments. The first, *RandUniform()*, is a function returning a random number: it is the argument which will enable the *Lookup* function to select a province randomly. The second argument is a pointer to a temporary variable: it indicates to Modgen the variable in which the randomly selected value is to be stored. In our example, the temporary variable is called *prov\_temp*. Because the argument of the function has to be a pointer, don't forget to add an ampersand (&) in front of the name of the variable (&*prov\_temp*, see line 94).<sup>2</sup> To re-emphasise, the *Lookup* function does not return the value of a state, but a whole number value: this is why *prov\_temp* is an int type variable and not a *PROV* state variable. So next we must give the *prov* state variable the value selected by *Lookup* and stored in *prov\_temp*. Because the *prov\_temp* variable is a whole number (*int*) and the state variable is a *PROV* classification, we convert the whole number type into a *PROV* type using a "casting". A casting is a conversion from one type of data to another. For example the value "0" may simply signify zero (nil value) as a whole number, but may equally represent the value *P\_NFL* or *Newfoundland and Labrador* as a *PROV* type variable. To convert an integer into a *PROV* type, the destination type (*PROV*) is written in parentheses in front of the name of the variable to be converted.

```
PersonCore.mpp
```

```
97     prov = (PROV) prov_temp;
```

Because place of residence is equivalent to place of birth at the time of birth, the value of the place of residence is simply given to the *prov\_birth* state variable (line 99).

We now have assigned a birth province, which will remain the same throughout the actor's life, and a province of residence, which will have to change with successive migrations. And these migrations will be determined by a mobility event, which is the subject of the next section.

### 3.4 Adding an Internal Migration Event

Internal migration can be seen as a decision made in two separate steps: the first step is linked to the decision to change province of residence (to become mobile), while the second step is linked to the choice of a destination province. In Modgen, modeling this kind of process is done using *event* functions, which we have already used in previous chapters to model mortality and birthdays. The time function of the event enables us to determine the random wait time before the mobility event, and the event function lets us assign a new province of residence to the actor.

---

<sup>2</sup>In concrete terms, &*prov\_birth* corresponds to the memory address where the content of the *prov\_birth* variable is located. So it indicates the memory address where the *Lookup* function can place the result of the random selection. For more information on pointers, consult a C++ guide.

First we have to declare the event function in the actor class of the new migration module.

```

                                                                    Migration.mpp
22  actor Person                // Individual
23  {
24      event TimeIntMigEvent, IntMigEvent;

      ...

27  };

```

The two aspects of the event are stated using the *event* function. The complete functions are defined a little further on in the code. The random time elapsed before the mobility event is calculated in the event time function (*TimeIntMigEvent*), while the destination province is assigned in the event function itself (*IntMigEvent*). Let's first have a look at the time function.

```

                                                                    Migration.mpp
29  TIME Person::TimeIntMigEvent()
30  {
31      TIME tEventTime = TIME_INFINITE;
32
33      // A waiting time is picked at random
34      tEventTime = WAIT(-TIME(log(RandUniform(3)) /
35                          (IntMigHazard[sex][age_int][prov] +
36                          0.0000000001)));
37
36
37      return tEventTime;
38  }

```

As we might have expected, the time function for the mobility event is similar to the one used for the mortality event. A *TIME* type variable containing the wait time is first initialised with an infinite value. The value of this variable is later replaced by a random wait time generated from the mobility rates written into the parameters section (we will come to this later). The value of the random wait time is then sent back to the events manager using the *return* command. So this time function is indeed similar to the mortality event function in Chap. 2, but there are two important differences.

First, the *IntMigHazard* parameter (stated in the parameters section) contains three dimensions rather than two: sex, age and province of residence – a state variable which was added in the migration module. This new parameter is declared in the *Parameters* section of the migration module, and its values are initialised in the *Base(Migration).dat* parameters file.

```

Migration.mpp

11  parameters
12  {
13      // Outmigration hazard
14      double IntMigHazard[SEX][AGE][PROV];
15      ...
19
20  };

Base(Migration).dat

1  parameters
2  {
3      // Outmigration hazard
4      double IntMigHazard[SEX][AGE][PROV] = { (2220) 1, };
5      ...
9  };

```

The new parameter is filled with values equal to 1. Those serve only the purpose of initializing the parameter and will be replaced from the user interface with plausible values.

A second difference from the mortality time function is the addition of a very small number (0.00000000001) to the risk of out-migration (*IntMigHazard*). Why is this? Some of the exit rates to be used will have a nil value: extremely old people, for example, have a practically zero risk of migration. But because the value of the exit rate is part of the denominator of the equation, there is a risk of computing a division by zero (an indeterminate result that leads to a runtime error). By adding a negligible quantity to the value of the parameter, we avoid generating an error when running the model. We could equally have proceeded by replacing the nil values directly in the parameters file, which would have avoided the trouble of inserting the correction in the code. However, adding the correction directly into the code is a way of ensuring that any nil rate which may have been overlooked in the parameters file will be rectified.

We now move on to the *IntMigEvent* function, which determines the actor's destination province. We will again use the *cumrate* and *Lookup* function as we did before, but now adding an extra complication. The destination province is actually selected not on the basis of a distribution vector, as for the birth province, but on the basis of an origin-destination matrix. This is because the choice of a destination province also depends on the province of origin: for each province of residence, there are nine possible destination provinces. So we need to declare a two-dimensional *cumrate* parameter.

```

Migration.mpp
11  parameters
12  {
    ...
15      // Interprovincial migration, origin-destination
        matrix
16      cumrate OrDestMat[PROV][PROV];
    ...
20  };

```

As usual, the parameter values have to be initialised in the migration module parameters file.

```

Base(Migration).dat
6      cumrate OrDestMat[PROV][PROV] = {(100) 1, };

```

Here we enter the same value 100 times (because we defined a ten by ten matrix), which represents a purely random distribution. We can add real values from the user interface once we run the model. So we have to remember that the final dimension of the parameter is in columns and the second to last is in rows. In other words, the provinces of origin are in rows and the destination provinces in columns.

Following a mobility event, we determine the destination province by a procedure which is analogous to the one we used to determine the province of birth: we use the *cumrate* parameter in conjunction with the *Lookup* function.

```

Migration.mpp
40  void Person::IntMigEvent()
41  {
42      int prov_dest = 0;
43      Lookup_OrDestMat(RandUniform(4), prov, &prov_dest);
44      prov = (PROV)prov_dest;
45  }

```

As we did before, a temporary variable is first created (*prov\_dest*) in which the value of the randomly selected destination province will be stored. *Lookup* is then called together with the origin-destination matrix (*Lookup\_OrDestMat*), and the value of the province of residence (*prov*) is finally replaced by the value of the picked destination province (*prov\_dest*). Notice that *Lookup* now takes an extra argument (now three instead of two). This is because in addition to a random variable and a temporary variable, we have to specify the province of residence. The number of arguments following the random number generator has to correspond to the number of dimensions in the *cumrate* parameter, and they must appear in the same order. As arguments, state variables provide a constraint to the random draw while pointers to temporary variables are randomly assigned a value by the *Lookup* function based on given constraints and the *cumrate* distribution. So in our example,

the province of residence state variable acts as a constraint so that *Lookup* may randomly find a value along the destination province dimension. The *Lookup* function is very flexible: it can accommodate a *cumrate* parameter with an arbitrary number of dimensions.

### 3.5 Modifying the Mortality Parameter

In a multi-regional life table, the death rates vary according to region of residence. The assumption is that all individuals in the same territory are also subject to the same behaviours and socio-economic conditions influencing life expectancy. In Chap. 2, we modified the mortality parameter to take account of age and sex in determining the risks of dying. To take the province of residence into account, we add a dimension to the mortality parameter.

```

                                                    PersonCore.mpp
10 // Annual hazard of death according to sex, age and
    province
11 double MortalityHazard[SEX][AGE][PROV];
```

It is simply a matter of adding the name of the PROV classification in square brackets after the SEX and AGE dimensions. The number of elements in the parameters matrix now rises from 222 to 2220, because there are ten provinces. We must then re-initialise the values in the *Base(PersonCore).dat* file, so that the parameter contains the right number of values.

```
double MortalityHazard[SEX][AGE][PROV] = {(2220) 0.001, };
```

Once the parameter has been modified and its values have been re-initialised to take account of the new dimension, all that remains is to modify the code in the mortality time function.

```

                                                    PersonCore.mpp
53      tEventTime = WAIT(-TIME(log(RandUniform(1)) /
54      MortalityHazard[sex][age_int][prov]));
```

We add the new dimension to the mortality parameter in the event function. The appropriate mortality rate is accessed using the state variables *age\_int*, *sex* and *prov*, or the province of residence of the actor (again, remember, not the *PROV* classification).

Here we can see the power of microsimulation using Modgen. Revising the model to take account of an extra dimension turns out to be quite simple: we just add an extra dimension to a parameter, modify the event function, and the job is done. Now that our model contains all the elements required to produce multi-regional tables, we can define the appropriate tables in the *table* section of the mobility module.

## 3.6 Multi-regional Tables

Our microsimulation model now allows for the simulation of interprovincial mobility, but proper output tables still need to be constructed to collect the relevant data and use them to produce a multi-regional life table. But before thinking about multi-regional tables, it would be useful just to take a look at the number of in- and out-migrations for each province, and also calculate the net migration. This is what the *NetMigIP* table does.

```

                                                                    Migration.mpp
47  table Person NetMigIP // Exits, entrances and net migra-
      tion by province
48  {
49    {
50      exits(prov, P_NFL), // Exits from Newfoundland
51      exits(prov, P_PEI), // Exits from PEI
52      exits(prov, P_NOS), // Exits from Nova Scotia
53      exits(prov, P_NEB), // Exits from New- Brunswick
54      exits(prov, P_QUE), // Exits from Québec
55      exits(prov, P_ONT), // Exits from Ontario
56      exits(prov, P_MAN), // Exits from Manitoba
57      exits(prov, P_ALB), // Exits from Alberta
58      exits(prov, P_SAS), // Exits from Saskatchewan
59      exits(prov, P_BRC), // Exits from British Columbia
60
61      entrances(prov, P_NFL), // Entrances in Newfoundland
62      entrances(prov, P_PEI), // Entrances in PEI
63      entrances(prov, P_NOS), // Entrances in Nova Scotia
64      entrances(prov, P_NEB), // Entrances in New-Brunswick
65      entrances(prov, P_QUE), // Entrances in Québec
66      entrances(prov, P_ONT), // Entrances in Ontario
67      entrances(prov, P_MAN), // Entrances in Manitoba
68      entrances(prov, P_ALB), // Entrances in Alberta
69      entrances(prov, P_SAS), // Entrances in Saskatchewan
70      entrances(prov, P_BRC), // Entrances in British
                                Columbia
71
```

```

72     // Net migration Newfoundland
73     entrances(prov, P_NFL) - exits(prov, P_NFL),
74     // Net migration PEI
75     entrances(prov, P_PEI) - exits(prov, P_PEI),
76     // Net migration Nova Scotia
77     entrances(prov, P_NOS) - exits(prov, P_NOS),
78     // Net migration New-Brunswick
79     entrances(prov, P_NEB) - exits(prov, P_NEB),
80     // Net migration Québec
81     entrances(prov, P_QUE) - exits(prov, P_QUE),
82     // Net migration Ontario
83     entrances(prov, P_ONT) - exits(prov, P_ONT),
84     // Net migration Manitoba
85     entrances(prov, P_MAN) - exits(prov, P_MAN),
86     // Net migration Saskatchewan
87     entrances(prov, P_SAS) - exits(prov, P_SAS),
88     // Net migration Alberta
89     entrances(prov, P_ALB) - exits(prov, P_ALB),
90     // Net migration British Columbia
91     entrances(prov, P_BRC) - exits(prov, P_BRC)
92
93 }
94
95 *split(age_int, AGE_GROUP) +
96 *sex +
97 };

```

The derived states *exits* and *entrances* count the number of times an actor enters or leaves a given state of a state variable. So it takes as arguments the name of a state variable and the name of a particular state. For example, *exits(prov, P\_NFL)* counts the number of times the state variable *prov* leaves the *P\_NFL* state, or in plain language, the number of times an actor migrates out of the province of Newfoundland and Labrador.

The calculation of net migration by combining the *exits* and *entrances* derived states is straightforward. All that is needed is to subtract *exits* from *entrances*: for the province of Ontario for example, the formula is *entrances(prov, P\_ONT) – exits(prov, P\_ONT)*. Note also that the elements of the table are disaggregated by age group (using the *split* function, see Chap. 2) and by sex.

Having generated these descriptive data, we can now move on to the main topic of this chapter: the construction of a multi-regional life table. There are many ways of using this type of table, but here we will limit ourselves to measuring the share of life expectancy lived in each of the provinces for an actor born in a given province.

First we have to establish the number of survivors at an exact age  $x$ , by sex and birth province. We then need to calculate the number of years lived in each of the provinces from exact age  $x$ , by sex and birth province. The ratio of years of life remaining in each province to the number of survivors will give the multi-regional life expectancy.

The two components can be easily obtained via the derived states we introduced in this chapter and in the previous one. Survivors are obtained using the *Small\_lx* table below.

```

                                                    Migration.mpp
99      table Person Small_lx // Small lx (multiregional)
100     [age_int >= agex]
101     {
102
103         sex +
104         *{
105             value_in(alive)          // Survivors at age x
106                                     decimals=4
107         }
108         *prov_birth
109     };

```

Here we recognise the derived state *value\_in(alive)* which counts the number of times an actor “enters” a combination of states. Because sex and birth province are fixed characteristics which do not vary with time, *value\_in* simply counts the number of actors entering the model. Note that a filter *[age\_int>=agex]* has been inserted after the title of the table so that only individuals reaching a particular age  $x$  (line 100) are counted, which enables us to calculate the multi-regional life expectancy at age  $x$  (when  $x=0$ , we have the life expectancy at birth). While *age\_int* is a state variable that is already defined, *agex* is a new parameter which should be declared in the *parameters* section of a module. We will integrate it into the *PersonCore.mpp* file, because this is the module in which variables linked to age are declared. But we could equally well have added it to the migration module; this is for the model builder to decide. As usual, we will add the *agex* parameter to the *Base(PersonCore).dat* parameters file (the code for this file is not reproduced at the end of this chapter).

```

                                                    PersonCore.mpp
14      int agex; // Age x (for multiregional life expectancy)

                                                    Base(PersonCore).dat
int agex = 0; // Age x (for multiregional life expectancy)

```

The *Small\_lx* table then provides the number of survivors at age *age<sub>x</sub>*, by birth province and sex, a 10 by 2 table (actually 10 by 3 because the sum of both sexes is also included by the addition of the + symbol). To complete the table, we must now derive the number of years lived in each of the provinces from exact age *age<sub>x</sub>*. To do this we use the *duration* derived state as we did in the previous chapters.

```

                                                                    Migration.mpp
111   table Person Big_Tx // Big Tx (multiregional)
112   [age_int >= agex]
113   {
114
115       {
116           duration() //EN Person-years to live from age x
                        (multiregional)
117
118       }
119       *sex +
120       *prov +
121       *prov_birth
122   };

```

The *Big\_Tx* table gives the number of years lived from exact age *x*, by sex (*sex*) and birth province (*prov\_birth*), in each province of residence (*prov*). To obtain the multiregional life expectancy at age *x*, all that needs to be done is to divide the number of years lived in each province from exact age *x* by the number of survivors at exact age *x*.

At this point, one might ask a legitimate question: why didn't we do this division in the *Big\_Tx* table? Why not just use the *duration()/value\_in(alive)* operation directly in this table, like we did before? This question takes us back to the way the derived state *value\_in* works.

Remember that the *value\_in* derived state enumerates all the instances in which an actor of a given sex and birth province “enters” a combination of given states. In the *Big\_Tx* table, the derived state *value\_in* would count the number of times an actor of a given sex and birth province made an entry into a new province of residence (after a mobility event). So for example, an actor who leaves his or her birth province and makes a return migration would be counted twice in a single province (once at birth and another time at the return migration). Because the *Big\_Tx* table has an extra dynamic dimension, the derived state *value\_in* would therefore count not the number of **survivors** of a given sex and birth province **at age *x***, but rather the number of **entries** in each province for an actor of a given sex and birth province **starting from age *x***. So we have to create two separate tables in order to derive the number of survivors and the number of years lived. To obtain a final compact multiregional life table, we can copy and paste the results of the two tables into an Excel spreadsheet and calculate life expectancy for ourselves by dividing *Tx* by *lx*. This extra step is not very complicated, but for those who want to avoid using Excel,

Modgen does offer advanced tools for manipulating table results. These tools are described in the next section, which is aimed at more advanced (and more motivated) users.

### 3.7 Manipulating Tables in Modgen (Advanced Topic)

Because we can't build a multi-regional life expectancy in a single table, we would normally have to import the results into an Excel spreadsheet and carry out the additional calculations separately. However Modgen does enable us to manipulate pre-defined table elements in order to create new customized tables.

The idea is to create first a blank table into which arbitrary values may then be inserted. The data from the existing tables can then be extracted, transformed and inserted in this new blank table.

To create a blank table, the command *user\_table*, rather than *table*, must be used. We give the new customized table the title *MultiregionalLifeExpectancy*.

```

                                                                    Migration.mpp
126  // Creates an empty canvas for the multiregional
    lifetable
127  user_table MultiregionalLifeExpectancy // Multiregional
    life exp. at age x
128  {
129      {
130          LIFE_EXPECTANCY // Life expectancy decimals=4
131      }
132
133      *SEX + // sex
134      *PROV + // province of residence
135      *PROV // province of birth
136  };

```

The table is set up in the same way as a normal table, except that instead of derived states we enter titles which will enable us to access the cells of the table later on. In addition, the dimensions are not specified using state variables but using classifications, which is why capital letters are used here. We have to tell Modgen the dimensions of the blank table and not give it state variables to tabulate. A customized table is passive: it is an empty shell that contains none of the actor's derived states. In our example, *MultiregionalLifeExpectancy*, the table cells are accessed by way of the *LIFE\_EXPECTANCY* heading. The *SEX* and *PROV* classifications define the dimensions of the table: in this case *SEX* + \* *PROV* + \* *PROV* gives  $3 \times 11 \times 10$  (remembering that the + symbol adds the sum of the states), and so the table will have 330 cells.

We now have a blank table with three dimensions, with each cell containing one result (*LIFE\_EXPECTANCY*). To enter real values into the blank table, the *UserTables()* function must be used. This generally contains nested loops as an efficient way of accessing and filling all the boxes of the customized table (in our example, filling them all one by one would require 330 lines of code!). So to fill this blank table, we use the function *UserTables()*.

The code which enables us to complete our multi-regional life expectancy table is reproduced below. Each of the steps will be explained afterwards.

```

Migration.mpp
138  // This function fills the empty cells of the multire-
    gional life table
139  void UserTables()
140  {
141
142      double TValue;
143      double lValue;
144
145      // Loops through all three specified dimensions to fill
    the cells
146      for (int nSex = 0; nSex <= SIZE(SEX); nSex++) {
147
148          for (int nProvN = 0; nProvN < SIZE(PROV);
    nProvN++) {
149
150              for (int nProvR = 0; nProvR <= SIZE(PROV);
    nProvR++) {
151
152                  lValue = GetTableValue("Small_lx.Expr0",
    nSex, nProvN);
153                  TValue =
154                      GetTableValue("Big_Tx.Expr0",
    nSex, nProvR, nProvN);
155                  SetTableValue("MultiregionalLifeExpe
    ctancy.LIFE_EXPECTANCY",
156                      TValue / lValue, nSex, nProvR, nProvN);
157
158              }
159
160          }
161
162      }

```

In line 139 we see the statement of the function *UserTables()*. All the elements of this function are shown between curly brackets (lines 140 and 162). At lines 142 and 143 two variables are declared: *TValue* and *lValue*. These will be used to temporarily store the values of the big *T<sub>x</sub>* and the small *l<sub>x</sub>*, which can be found in the *Small\_lx* and *Big\_Tx* tables. Because our table has three dimensions, we will use three nested loops to cover all the cells. The maximum size of the index, which terminates the loop, is defined using the *SIZE(Classification)* function. This function returns the number of states included in the classification specified as an argument. For example, the *SEX* classification has two possible values, so the *SIZE(SEX)* function will return the value two. The following loop:

```

Migration.mpp
146   for (int nSex = 0; nSex <= SIZE( SEX ); nSex++ ) {

```

will therefore run three times (with the index passing through the values  $nSex=0=female$  and  $nSex=1=male$  and  $nSex=2=sum\ of\ the\ dimension$ ). Notice that the condition determining the end of the loop is either exclusive ( $<$ , which makes the loop run a number of times equal to *SIZE(classification)*) or inclusive ( $<=$ , which makes the loop run a number of times equal to *SIZE(classification) + 1*), depending on whether the  $+$  option (the sum total of a dimension) has been specified in the table description or not. The indexes of these three nested loops enable us to access each of the cells of the customized table and of the results tables *Small\_lx* and *Big\_Tx*. To complete each box in the customized table, first we have to find the corresponding big *T<sub>x</sub>* and small *l<sub>x</sub>* using the *GetTableValue* function.

```

Migration.mpp
152   lValue = GetTableValue( "Small_lx.Expr0", nSex, nProvN);
153   TValue =
154       GetTableValue( "Big_Tx.Expr0", nSex, nProvR,
                       nProvN);

```

The *GetTableValue* function allows us to access an element in a pre-defined results table. It takes the name of the table as an argument, as well as indices to access the relevant cell. Let's look at line 152. The *GetTableValue* function takes *Small\_lx.Expr0* as its first argument. *Small\_lx* refers to the title of the table which gives the number of survivors at age  $x$ . The term *Expr0* refers to the first derived state in the *Small\_lx* table. Subsequent derived states could then be accessed using the expressions *Expr1*, *Expr2* etc. But in the case of *Small\_lx*, because only one derived state is defined, only *Expr0* is valid. The function's other arguments specify the indices which give access to elements in the table. For example, if  $nSex=0$  and  $nProvN=1$ , the function will return a value of  $l_x$  for women born in Prince Edward Island. The value will be stored in the temporary *lValue* variable. The principle is the same to retrieve  $T_x$  (lines 153–154), the only difference being that *GetTableValue* then takes an extra argument to indicate the province of residence (because the *Big\_Tx* table has three dimensions rather than two).

Now that we have retrieved the  $l_x$  and the  $T_x$  elements of the table, we need to calculate the life expectancy and enter it into our customized table. The *SetTableValue* function (line 155) lets us do this. The function's first argument gives us access to the customized table we created (*MultiregionalLifeExpectancy.LIFE\_EXPECTANCY*). Instead of using the *Expr0* expression as we did in the *GetTableValue* function, this time we access the customized table item using the label we have specified (*LIFE\_EXPECTANCY*). The second argument of this function contains the value to be entered in the table. In our example, this is the life expectancy at age  $x$ , or  $T_x/l_x$  (in the code, *TValue/lValue*). The other three arguments are the indices providing access to each of the cells in the table, according to the three dimensions specified in the table description (sex, province of residence and birth province). It is important to use the same order of dimensions as in the definition of the customized table. So once the model is compiled and run, Modgen will execute the *UserTables()* function and the new customized table will appear as a regular table in the user interface. We will see this in the next section.

### 3.8 Adding a Parameter File to a Scenario

The multiregional table is now complete and you can precompile, compile and open the microsimulation program as we did in the two previous chapters (see Precompiling, compiling and running the microsimulation program). Then open the base scenario as we did before. Before doing anything, new information must be added to the scenario. First, we have to indicate that a new parameters file (*Base(Migration).dat*) must be included. Although this new file was added to the Visual Studio project, it still has to be manually integrated into the base scenario. We add a parameter file by accessing the thumbnail *Parameter Files* in the *Settings* (*Scenario -> Settings -> Parameter Files*, Fig. 3.7).

Click on the *Add* button, select the *Base(Migration).dat* file and finally click *OK*. Then click on one of the parameters files (in the left hand panel of the main window) to make the parameters appear (Fig. 3.8).

Next we have to modify the values of the parameters that were added to the model, which were initially set up to be constants (a model with constant value parameters would not give very interesting results). These new parameters are the risk of migration between provinces, the origin-destination matrix and age  $x$  for the multi-regional table. We can also modify the distribution of births by province, but because we are calculating multi-regional life tables, the number of births in each province does not make much difference, given that these births are sufficiently numerous to minimise the Monte-Carlo error (it is the average duration of life in each province that is of interest to us). We also have to enter new mortality rates because these now vary by province of residence. Canadian values for these parameters are available on the book website (<http://www.microsimulationandpopulation-dynamics.com/>). You may notice in the File View pane that the parameters are grouped by file name. This is the advantage of using one file of parameters per

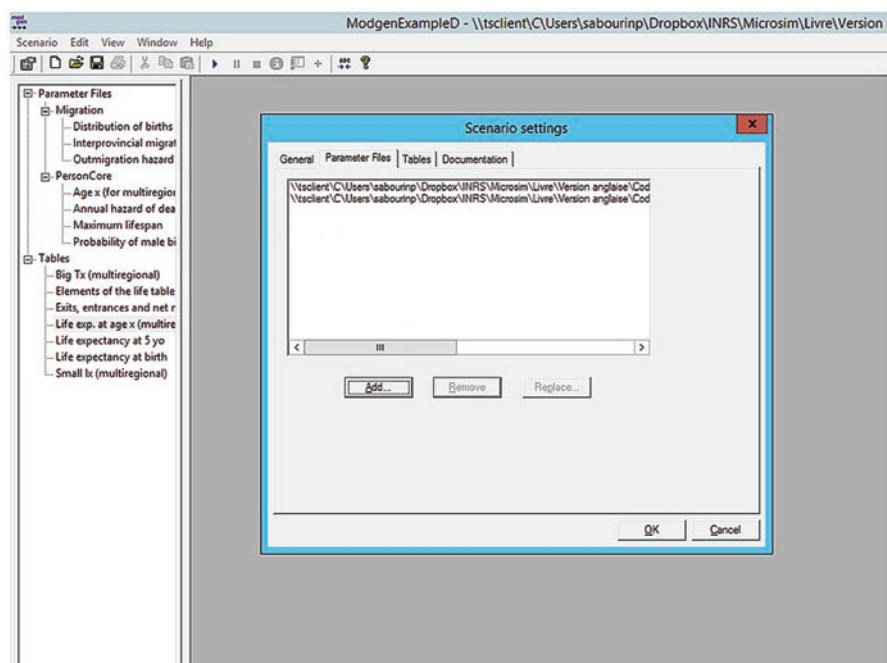


Fig. 3.7 Adding a parameters file to a scenario

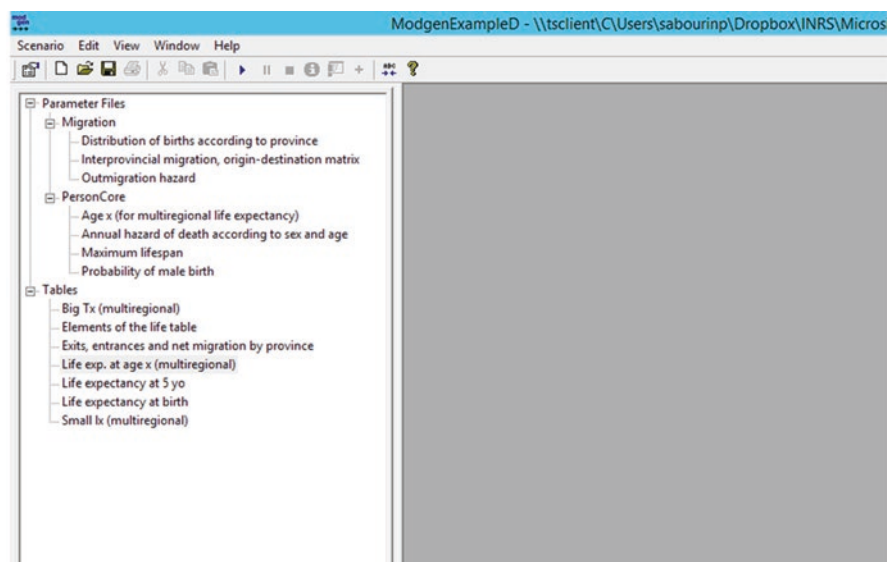


Fig. 3.8 User interface, multi-regional tables

ModgenExampleD - \\tsclient\\C\\Users\\sabourin\\Dropbox\\INRS\\Microsim\\Live\\Version anglaise\\C

Scenario Edit View Window Help

Parameter Files

- Migration
  - Distribution of births according to province
  - Interprovincial migration, origin-destination matrix
  - Outmigration hazard
- PersonCore
  - Age x (for multiregional life expectancy)
  - Annual hazard of death according to sex and age
  - Maximum lifespan
  - Probability of male birth
- Tables
  - Big Tx (multiregional)
    - Elements of the life table
    - Exits, entrances and net migration by province
    - Life exp. at age x (multiregional)
    - Life expectancy at 5 yo
    - Life expectancy at birth
    - Small lx (multiregional)

Table: Life exp. at age x (multiregional)

Transformation: Value

sex: Rows: Columns:

province of residence province of birth

	P_NFL	P_PEI	P_NOS	P_NEB	P_QUE	P_ONT	P_MAN	P_SAS	P_ALB	P_BRC
P_NFL	65.5110	1.6738	2.0789	1.2578	0.1401	0.8110	0.4924	0.3882	0.6342	0.3861
P_PEI	0.6550	35.6032	0.3020	0.9743	0.0635	0.1604	0.1298	0.1309	0.1724	0.1284
P_NOS	4.4873	7.7908	37.2500	5.1266	0.5016	1.5421	1.0569	1.1026	1.2458	1.3637
P_NEB	2.1839	4.8267	4.0473	40.6522	0.8353	0.9553	0.8237	0.5596	0.8162	0.5560
P_QUE	2.3607	3.4333	3.8164	6.7021	64.7703	4.3321	2.3010	1.7166	2.2281	2.0758
P_ONT	19.5688	14.0885	17.0678	14.2379	8.9866	60.9187	13.1318	9.5353	10.7208	10.0255
P_MAN	1.1979	0.8773	1.2986	1.1259	0.3574	1.1074	36.9485	3.8262	2.3977	1.6991
P_SAS	0.6681	0.8122	1.0228	0.6348	0.2402	0.7002	3.5760	31.6951	3.6820	1.8396
P_ALB	5.2864	4.6816	4.5102	3.7875	1.1827	3.4408	8.8836	18.4413	40.0795	9.4782
P_BRC	6.4730	5.7231	7.6923	4.5891	2.4531	6.2126	12.5603	14.6707	18.1055	52.6588
All	79.5323	79.4806	79.8443	79.5380	79.5408	80.2016	79.6982	80.0264	80.0873	80.2382

Fig. 3.9 Multi-regional life expectancy table

module. In a more complex model, these headings are useful for finding particular parameters more easily.

Run the model to create or refresh the table results (*Scenario -> Run/Resume*) and take a look at the multi-regional life expectancy table (Fig. 3.9).

Notice that the last line of the table (*All*) gives total life expectancy for each birth province. A native of Manitoba, for example, has a life expectancy of 79.7 years. Each item in a column indicates the number of years an actor born in a given province can expect to spend in each of the provinces. The proportion of the life expectancy spent in the birth province is low in those provinces where there are generally high rates of out-migration. In the Atlantic provinces, an individual can expect to spend on average less than half of his or her life in his province of birth. On the other hand, the share of life expectancy spent in one's province of birth will be greater where the overall rate of exit from the province is low: Quebec and Ontario are examples of this. You can explore for yourself the results of the different tables we have created in this chapter. As an exercise, you could show using Excel that the multi-regional life expectancy table can be recreated using the survivors table (*Small lx*) and the person-years left to live (*BigTx*).

### 3.9 Summary

In this chapter we have learned how to use the *self\_scheduling\_int* function and how to add a new module to a Modgen microsimulation program. In the course of doing this we have added a new event – migration between provinces of Canada – to the model and calculated the elements of a multi-regional life table. We have also learned how to generate a customized table with *UserTables*.

The multi-regional model proposed by Rogers has been developed in the context of analysing migration between the regions of a country. As a generalisation, regions of a country can be thought of as separate states, between which we can observe movements. So it is easy to adapt the multi-regional table model developed here to treat any phenomenon where there are transitions between a defined number of mutually exclusive states. For example, such multi-state models, also known as multiple increment and decrement tables, enable us to analyse life expectancy according to marital status, activity status or health condition.

The results from a microsimulation model should be identical to those obtained from a determinist model based on matrix mathematics as proposed by Rogers, except of course for the Monte-Carlo error, which decreases as the number of cases increases. Because it is generally easy to increase the number of cases while maintaining a reasonably low model run time, the Monte-Carlo error can be reduced to a negligible level.

There are two major advantages to using microsimulation rather than a matrix method. Firstly, in the matrix method, the size of the state space grows exponentially as the number of states increases, so the model quickly becomes mathematically and computationally hard to manage. This does not happen with microsimulation, which can accommodate a large number of states. Secondly, the maths involved in a conventional model can soon become complicated, especially if one has to take a number of competing risks into account or use more complex risk models. For example, if we wanted to produce multi-state tables in which populations could move in various ways between ten provinces, three states of health and two activity statuses, it would become rather tedious to define the elements of the matrices in a satisfactory way. Adding new dimensions (new states, or new events) to a microsimulation model is made easy by Modgen's modelling flexibility and its mode of operation in continuous time.

## Appendices

### *Appendix 3.1 PersonCore.mpp*

Code Sections: Header (lines 1 to 7), Parameters (lines 8 to 17), Actor *Person* and functions (lines 18 to 113), Tables (lines 114 to 146)

```

1  classification SEX{ S_FEM, S_MAL };
2
3  range AGE{ 0, 110 };
4
5  partition AGE_GROUP{ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55,
6    60, 65,
7    70, 75, 80, 85 };

```

```

8 parameters
9 {
10     //EN Annual hazard of death according to sex, age and
        province
11     double MortalityHazard[SEX][AGE][PROV];
12     int AgeMax;                // Maximum lifespan
13     double ProbabilityMale; // Probability of male birth
14     int age_x; // Age x (for multiregional life expectancy)
15
16 };
17
18 actor Person                //EN Individual
19 {
20
21     //EN Alive
22     logical alive = {TRUE};
23
24     SEX sex;    // Sex state variable
25
26     // Age state variable, auto-increment
27     AGE age_int = COERCE(AGE, self_scheduling_int(age));
28
29     event timeMortalityEvent, MortalityEvent; //EN Mortality
        event
30
31     //LABEL(Person.Start, EN) Starts the actor
32     void Start();
33
34     //LABEL(Person.Finish, EN) Finishes the actor
35     void Finish();
36 };
37
38 // The time function of MortalityEvent
39 TIME Person::timeMortalityEvent()
40 {
41     TIME tEventTime = TIME_INFINITE;
42
43     // If max age is reached, death event occurs immediately
44     if (age_int == AgeMax)
45     {
46         tEventTime = WAIT(0);
47     }
48     else
49     {
50         // Draw a random waiting time to death from

```

```
51         // an exponential distribution based on the
52         // constant hazard MortalityHazard.
53         tEventTime = WAIT(-TIME(log(RandUniform(1)) /
54             MortalityHazard[sex][age_int][prov]));
55     };
56
57     return tEventTime;
58 }
59
60 // The implement function of MortalityEvent
61 void Person::MortalityEvent()
62 {
63     alive = FALSE;
64
65     // Remove the actor from the simulation.
66     Finish();
67 }
68
69
70 void Person::Start()
71 {
72     // Modgen initializes all actor variables
73     // before the code in this function is executed.
74
75     age = 0;
76     time = 0;
77
78     // Sex is randomly attributed according to parameter
79     // ProbabilityMale
80     if (RandUniform(2) < ProbabilityMale)
81     {
82         sex = S_MAL;
83     }
84     else
85     {
86         sex = S_FEM;
87     }
88
89     // The following lines attribute province of birth
90     // and province of residence to the actor
91
92     // A temporary variable to store the province value
93     int prov_temp = {0};
94
95     // Lookup picks a random province according to cumrate
96     // distribution
```

```

94     Lookup_ProvBirth(RandUniform(5), &prov_temp);
95     // Casts the (integer) prov_temp value into the
96     // prov state variable (of PROV type)
97     prov = (PROV) prov_temp;
98     // Province of residence at birth is the province of birth
99     prov_birth = prov;
100
101 }
102
103 /*NOTE(Person.Finish, EN)
104     The Finish function terminates the simulation of an actor.
105 */
106 void Person::Finish()
107 {
108     // After the code in this function is executed,
109     // Modgen removes the actor from tables and from the
110     // simulation.
111     // Modgen also recuperates any memory used by the actor.
112 }
113
114 table Person DeathsAndSurvivors // Elements of the life table
115 {
116     {
117         value_in(alive), // Survivors (lx)
118         entrances(alive, FALSE), // Deaths (dx)
119         duration() // Person-years lived (Lx) decimals=4
120     }
121     *split(age_int, AGE_GROUP) + // by age
122     *sex + // and sex
123 };
124
125
126
127
128
129 table Person LifeExpectancyBirth // Life expectancy at birth
130 {
131     {
132         // Life expectancy decimals=4
133         duration() / value_in(alive)
134     }
135     *sex + // By sex
136 };
137

```

```

138 table Person LifeExpectancy5 // Life expectancy at 5 yo
139 [age_int >= 5]
140 {
141     {
142         // Life expectancy at 5 yo decimals=4
143         duration() / value_in(alive)
144     }
145     *sex +
146 };

```

### ***Appendix 3.2 Migration.mpp***

Code Sections: Header (lines 1 to 10), Parameters (lines 11 to 21), Actor *Person* and functions (lines 22 to 46), Tables (lines 47 to 164)

```

1  /* Internal Migration Module
2  This module contains all the elements of the mobility event.
3  The Lookup function is used in conjunction with an
4  origin-destination matrix (cumrate) to determine
5  the choice of a destination province.
6  */
7
8  classification PROV{ P_NFL, P_PEI, P_NOV, P_NEB, P_QUE, P_ONT,
9      P_MAN, P_SAS, P_ALB, P_BRC };
10
11 parameters
12 {
13     // Outmigration hazard
14     double IntMigHazard[SEX][AGE][PROV];
15     // Interprovincial migration, origin-destination matrix
16     cumrate OrDestMat[PROV][PROV];
17     // Distribution of births according to province
18     cumrate ProvBirth[PROV];
19
20 };
21
22 actor Person // Individual
23 {
24     event TimeIntMigEvent, IntMigEvent;
25     PROV prov;
26     PROV prov_birth;
27 };
28
29 TIME Person::TimeIntMigEvent()

```

```

30 {
31     TIME tEventTime = TIME_INFINITE;
32
33     // A waiting time is picked at random
34     tEventTime = WAIT(-TIME(log(RandUniform(3)) /
35         (IntMigHazard[sex][age_int][prov] + 0.0000000001)));
36
37     return tEventTime;
38 }
39
40 void Person::IntMigEvent()
41 {
42     int prov_dest = 0;
43     Lookup_OrDestMat(RandUniform(4), prov, &prov_dest);
44     prov = (PROV)prov_dest;
45 }
46
47 table Person NetMigIP // Exits, entrances and net migration by
    province
48 {
49     {
50         exits(prov, P_NFL), // Exits from Newfoundland
51         exits(prov, P_PEI), // Exits from PEI
52         exits(prov, P_NOS), // Exits from Nova Scotia
53         exits(prov, P_NEB), // Exits from New-Brunswick
54         exits(prov, P_QUE), // Exits from Québec
55         exits(prov, P_ONT), // Exits from Ontario
56         exits(prov, P_MAN), // Exits from Manitoba
57         exits(prov, P_ALB), // Exits from Alberta
58         exits(prov, P_SAS), // Exits from Saskatchewan
59         exits(prov, P_BRC), // Exits from British Columbia
60
61         entrances(prov, P_NFL), // Entrances in Newfoundland
62         entrances(prov, P_PEI), // Entrances in PEI
63         entrances(prov, P_NOS), // Entrances in Nova Scotia
64         entrances(prov, P_NEB), // Entrances in New-Brunswick
65         entrances(prov, P_QUE), // Entrances in Québec
66         entrances(prov, P_ONT), // Entrances in Ontario
67         entrances(prov, P_MAN), // Entrances in Manitoba
68         entrances(prov, P_ALB), // Entrances in Alberta
69         entrances(prov, P_SAS), // Entrances in Saskatchewan
70         entrances(prov, P_BRC), // Entrances in British
            Columbia
71
72         // Net migration Newfoundland

```

```

73     entrances(prov, P_NFL) - exits(prov, P_NFL),
74     // Net migration PEI
75     entrances(prov, P_PEI) - exits(prov, P_PEI),
76     // Net migration Nova Scotia
77     entrances(prov, P_NOS) - exits(prov, P_NOS),
78     // Net migration New-Brunswick
79     entrances(prov, P_NEB) - exits(prov, P_NEB),
80     // Net migration Québec
81     entrances(prov, P_QUE) - exits(prov, P_QUE),
82     // Net migration Ontario
83     entrances(prov, P_ONT) - exits(prov, P_ONT),
84     // Net migration Manitoba
85     entrances(prov, P_MAN) - exits(prov, P_MAN),
86     // Net migration Saskatchewan
87     entrances(prov, P_SAS) - exits(prov, P_SAS),
88     // Net migration Alberta
89     entrances(prov, P_ALB) - exits(prov, P_ALB),
90     // Net migration British Columbia
91     entrances(prov, P_BRC) - exits(prov, P_BRC)
92
93   }
94
95   *split(age_int, AGE_GROUP) +
96   *sex +
97 };
98
99 table Person Small_lx // Small lx (multiregional)
100 [age_int >= agex]
101 {
102
103   sex +
104   *{
105     value_in(alive) // Survivors at age x decimals=4
106   }
107   *prov_birth
108
109 };
110
111 table Person Big_Tx // Big Tx (multiregional)
112 [age_int >= agex]
113 {
114
115   {
116     duration() //EN Person-years to live from age x
117     (multiregional)

```

```

117
118     }
119     *sex +
120     *prov +
121     *prov_birth
122 };
123
124 // Custom user table
125
126 // Creates an empty canvas for the multiregional lifetable
127 user_table MultiregionalLifeExpectancy // Life exp. at age x
(multiregional)
128 {
129     {
130         LIFE_EXPECTANCY // Life expectancy decimals=4
131     }
132
133     *SEX + // sex
134     *PROV + // province of residence
135     *PROV // province of birth
136 };
137
138 // This function fills the empty cells of the multiregional
    life table
139 void UserTables()
140 {
141
142     double TValue;
143     double lValue;
144
145     // Loops through all three specified dimensions to fill the
        cells
146     for (int nSex = 0; nSex <= SIZE(SEX); nSex++) {
147
148         for (int nProvN = 0; nProvN < SIZE(PROV); nProvN++) {
149
150             for (int nProvR = 0; nProvR <= SIZE(PROV); nProvR++) {
151
152                 lValue = GetTableValue("Small_lx.Expr0", nSex,
nProvN);
153                 TValue =
154                     GetTableValue("Big_Tx.Expr0", nSex, nProvR,
nProvN);

```

```
155             SetTableValue("MultiregionalLifeExpectancy.  
156                 LIFE_EXPECTANCY",  
157                 TValue / lValue, nSex, nProvR, nProvN);  
158             }  
159  
160         }  
161  
162     }  
163  
164 }
```

### ***Appendix 3.3 Base(Migration).dat***

```
1 parameters  
2 {  
3     // Outmigration hazard  
4     double IntMigHazard[SEX][AGE][PROV] = { (2220) 1, };  
5     // Interprovincial migration, origin-destination matrix  
6     cumrate OrDestMat[PROV][PROV] = {(100) 1,};  
7     // Répartition des naissances selon la province  
8     cumrate ProvBirth[PROV] = {(10) 1,};  
9 };
```

## Chapter 4

# Modelling Fertility

### Aims of This Chapter

- Creating a new actor during a simulation and linking it to its parent actor
- Transferring information between actors using Modgen links
- Making the *Start* function more complex, so that initialization may differ according to the type of actor simulated
- Ending a simulation after a fixed number of years
- Reorganising the code and generating results tables in a separate module

The models developed in the three previous chapters are “cohort” models, in which all the actors in the simulation are born at the same time. This cohort approach is the principle behind life tables. Although this type of modelling is interesting and very useful for analytical purposes, it cannot be used for simulating real populations exposed to the full range of demographic forces such as immigration and fertility.

In this chapter we will see how to integrate fertility into the model, that is to create new actors (births) during a simulation. By adding fertility, we also add a second type of actor to the simulation: the first type is created at the onset of the simulation (the original cohort), while the second type is created (born) during the simulation. Each of these types of actor will need its own preliminary treatment (or initialization); we will see how the *Start* function can be modified to incorporate the particular features of each type of actor.

In a case-based model like the one developed in this book, actors are simulated one after the other and are not linked, though Modgen enables us to create explicit links between actors of a single case, using a type of variable analogous to a C++ pointer. This means that actors born during the simulation of a case can be connected to their parent actor (their mother) through the creation of a Modgen link. The link allows all the characteristics of the parent actor to be accessed from the child actor. For example, the place of residence of a newborn actor can be determined by accessing the mother’s place of residence. In this book we will stick to using the Modgen links in the most basic way, but it is good to be aware that *links* offer a wide

range of possibilities, especially in time-based models where all the actors are simulated at once. For further details on links, see the Modgen user guide.

Inserting births into the model requires that we define a length of time beyond which the simulation cannot extend. This simulation “horizon” turns out to be essential because the simulation could be infinitely prolonged under fertility rates above the replacement level. Even with fertility below replacement, the simulation could run on well beyond what is needed or practical (continuing to simulate great-great-grandchildren over hundreds or thousands of years).

So before we go on to add the fertility module, we have to build in a distinction between different timelines, namely calendar time and age. As in the previous chapters, the example we use in this chapter takes the model from the preceding chapter (Chap. 3) as its starting point.

## 4.1 Adding Calendar Time

In a cohort-based model in which births are not simulated, all the simulated individuals appear (or “are born”) at the same time, at time 0. So the age of the cohort corresponds to the calendar year: year 1 corresponds to age 1, year 2 to age 2, etc. But once fertility is included in the model, simulated children are born randomly throughout the course of the simulation, and the correspondence between time and age no longer holds; all the actors in the model will not be the same age at the same time. So how can we distinguish between an actor who is aged zero at the start of the simulation from another one born during the simulation? Age and time, the two fundamental concepts of demography, will have to be taken into account separately and explicitly.

A continuous time variable, distinct from age, is already present in the model: this is the variable *time* that we have seen in the first chapter. But as with age, we need to measure the passage of time in discrete steps: an additional time variable of the integer type is therefore needed. Let’s call it *year\_int*. It will be automatically incremented by the self-scheduled event we have used to increment *age\_int*: *self\_scheduling\_int*. The only difference is that the Modgen continuous variable *time* is used as an argument rather than the continuous variable *age*.

PersonCore.mpp

```
32 // Year state variable, auto-increment
33 int year_int = self_scheduling_int(time);
```

The COERCE function is not used, because *year\_int* is an integer (*int*) and not a Modgen *range*: integer variables are not bounded, unlike *range* variables. *Year\_int* will therefore be updated automatically by *self\_scheduling\_int(time)* until the simulation ends.

Now that we have a clock for measuring time in a discrete manner, we are able to end the simulation at a precise time. To do this, we have to create a parameter which will define the maximum duration of the simulation. In our example this parameter will be given the name *horizon*:

```

                                                    PersonCore.mpp
17      int horizon; // Maximum duration of simulation

```

Then we have to initialize the new parameter in the *Base(PersonCore).dat*<sup>1</sup> file with the required value (here we will select a value of 100, so that the simulation stops after a century):

```

                                                    Base(PersonCore).dat
int      horizon = 100; //FR Horizon de la simulation

```

With a clock counting absolute time and a parameter establishing the maximum duration of the simulation, all that remains is to create an event which will remove the actors once the maximum duration of the simulation is reached.

To do this, it would be possible to create a *horizon* event with the sole function of withdrawing an actor at the end of the simulation. But there is already an event, the mortality event, which has precisely this function of withdrawing actors from the simulation at death. So we can save the cost of an event by modifying the mortality function to include the maximum duration of the simulation.

In the example from Chap. 3 the mortality event time function was made to withdraw the actor when the maximum age was reached. The same mortality event time function now has to be modified again to include the maximum duration of the simulation.

```

                                                    PersonCore.mpp
51      // The time function of MortalityEvent
52      TIME Person::timeMortalityEvent()
53      {
54          TIME tEventTime = TIME_INFINITE;
55
56          // If max age or horizon is reached, death event occurs
            immediately
57          if (age_int == AgeMax | year_int >= horizon)
58          {
59              tEventTime = WAIT(0);
60          }

```

---

<sup>1</sup>The file code is not shown at the end of the chapter.

```

61     else {
62         // Draw a random waiting time to death from
63         // an exponential distribution based on the
64         // constant hazard MortalityHazard.
65         tEventTime = WAIT(-TIME(log(RandUniform(1)) /
66             MortalityHazard[sex][age_int][prov]));
67     }
68
69     return tEventTime;
70 }

```

The condition in line 57 is modified to manage the two limiting cases, the maximum age and the maximum simulation duration. Modgen will check this condition every time *year\_int* or *age\_int* are updated by the *self\_scheduling\_int* function. If the condition is fulfilled (if the maximum age or the maximum simulation duration has been reached), the event function (death) is immediately executed using the *tEventTime = WAIT(0)* statement. If the condition is not fulfilled, the simulation follows its normal course and a random duration before death is calculated as before. In practical terms, the mortality function brings about exit from the simulation in three ways: through death directly, through reaching the maximum age, and through reaching the time limit of the simulation.

But if an actor reaches the time limit set for the simulation, it should not really be counted as a death. One of the objectives of the model is to estimate the numbers of deaths, so it is not appropriate to simulate a death in a case where all that has happened is that the model's end point has been reached.

So we need to find a way to distinguish an actual death from the mere ending of a simulated case due to the simulation time limit being reached. The tabulation of deaths is based on the logical type state variable *alive*. It actually registers cases where this variable changes from *TRUE* to *FALSE*.

So here is an easy solution to our problem: the value of the *alive* state should be changed to *FALSE* if and only if a death has really taken place, that is, if the absolute time has not reached the time horizon of the simulation. So if the mortality event occurs before the horizon of the simulation, a death is registered and the case is terminated; otherwise, the case is simply terminated.

```

                                                                    PersonCore.mpp
72 // The implement function of MortalityEvent
73 void Person::MortalityEvent()
74 {
75     if (year_int < Horizon) {alive = FALSE;};
76
77     // Remove the actor from the simulation.
78     Finish();
79 }

```

The model now keeps track of both the integer age of the actor and of the absolute calendar time. It allows us to stop the simulation of an actor using the *Finish()* function after a predetermined number of years. We can now move on to add a fertility module.

## 4.2 Creating a Fertility Module

In the previous chapter we saw how to create a new Modgen module and a new parameters file for inter-regional mobility. We will use this procedure again to create a new fertility module and its corresponding parameters file. The complete code for this module can be found in Appendix 4.2 at the end of this chapter. We will analyse each of the components of the fertility module in this section.

As for all Modgen events, fertility is conceptualized in two phases. First we establish the random duration before a female actor (a potential mother) gives birth to a new actor (a child). Next, once the event has occurred, we go on to simulate this new child actor, who may also in due course have one or several children of her own. All these actors make up a single Modgen case.

First, the event function is declared in the *Person* class of the new fertility module (see line 23 below):

```
Fertility.mpp
```

```

17  actor Person                // Individual
18  {
19
20      int last_age_fertile = { 0 }; // Age of actor at last
      birth
21
22      event TimeFertilityEvent, FertilityEvent; // Fertility
      event (birth)
23
24  };

```

In addition to the declaration of the fertility event, a state variable *last\_age\_fertile* (line 20), which stores the age of the mother at the most recent birth, is also created. This variable will be useful in a number of ways, as we will see later on.

Before moving on to the description of the main body of the event functions themselves, we first have to define the Modgen parameters and data structures which will be needed to simulate fertility. Naturally we have to state the age-specific fertility rates which will determine the frequency of births. Because women are generally fertile between the ages of 15 and 49 (fecundity is actually not zero before 15 and after 49, but births outside this age range are rare), there is no point in defining fertility rates for all ages. So we will create a new range corresponding to the age limits of female fertility:

```

Fertility.mpp
8  range AGE_FERTILE {15,49}; // Fertile ages

```

A Modgen link between the child and its mother (*link*) must also be created. This link will become useful a little later on, once the *Start* function is remodelled to take into account the particular characteristics of the newborn actors entering the simulation. This link actually makes it possible to access the state values of the mother, and so enables us to initialize certain characteristics of the child to match the characteristics of its mother (such as place of residence, for example).

```

Fertility.mpp
7  link    Person.lMother;      // Link to the mother

```

As we can see, the syntax used to create a link between actors of the same type is very simple. After declaring the link, the name of the actor is linked to it using a simple full stop. In our example, the name of the actor (the name of the class) is *Person* and that of the link is *lMother*. The letter *l* in front of the name *Mother* is part of the writing convention; it means that this is a single link to another actor, as a child can have only one mother. It is also possible to create multiple links, for example in the case of a reverse link connecting a mother and her children, but we will not deal with these here.

Next we must declare the parameter variable containing the fertility rates:

```

Fertility.mpp
10  parameters
11  {
12
13      double FertilityHazard[AGE_FERTILE][PROV]; // Fertility
        hazard
14
15  };

```

The dimensions of the parameter will be 35 by 10, because it covers the 35 years of age within the *FERTILE\_AGE* range for each of the ten provinces in the model. The parameter is initialised by adding temporary values to the *Base(Fertility).dat* file (which must first be created, as was done in the preceding chapter for the mobility module).

```

double FertilityHazard[AGE_FERTILE][PROV] = {(350) 0,};

```

Now we can move on to the description of the fertility event itself, whose code is shown below:

```

                                                                    Fertility.mpp
26  // Fertility event
27  TIME Person::TimeFertilityEvent()
28  {
29      TIME tEventTime = TIME_INFINITE;
30
31      // A birth can take place if actor is female and fertile
      (between
32      // 15 and 49 years old)
33      // An actor cannot give birth twice in the same year
34
35      if (age_int >= MIN(AGE_FERTILE) && age_int <= MAX(AGE_
      FERTILE) &&
36          sex == S_FEM && last_age_fertile != age_int)
37      {
38
39          tEventTime = WAIT(-TIME(log(RandUniform(6)) /
      (-log(1 -
40              FertilityHazard[RANGE_POS(AGE_FERTILE,
      age_int)][prov]
41              - 0.0000000001))));
42
43      };
44
45      return tEventTime;
46
47  }
48
49  void Person::FertilityEvent()
50  {
51
52      Person      *prChild = { NULL }; // Creates a new
      actor
53      prChild = new Person(); // Instantiation of the
      class Person
54      prChild->Start(time, 1, this); // Starts simulat-
      ing the new actor
55      last_age_fertile = age_int; // Indicates age of
      actor at last birth
56
57  }
```

The structure of the fertility event time function (*TimeFertilityEvent*), as can be seen in the code above, is similar to that of the mortality and mobility events. There

are nevertheless some differences to be noted. Here the calculation of the random time before the event is inserted into a conditional statement (*if*): time lapse before the event is not calculated unless the actor is considered at risk of giving birth. Firstly the actor must be female (*sex == S\_FEM*). Secondly, the age of the actor must correspond to a valid fertile age, in other words it must lie within the *AGE\_FERTILE* range (*age\_int >= MIN(AGE\_FERTILE) && age\_int <= MAX(AGE\_FERTILE)*). The *MIN* and *MAX* functions provide respectively the minimum and maximum values of the *AGE\_FERTILE* range, which are 15 and 49. Finally, an actor cannot give birth twice in the same year: so age at the most recent childbirth must not be equal to the current age of the actor (*last\_age\_fertile != age\_int*). This final condition is needed because fertility is a repeatable event. After a birth, Modgen immediately recalculates the waiting time until the next birth. There is nothing to prevent an actor from giving birth twice in the same year of age; however, fertility rates are generally annual rates, which assume a single birth per woman per year.<sup>2</sup>

The calculation of the waiting time itself has two peculiarities. In the first place, one of the two dimensions of the *FertilityHazard* parameter is accessed by way of the *RANGE\_POS(AGE\_FERTILE, age\_int)* function. Why not simply use the state variable *age\_int* to access the parameter, as we did before in the other modules? To understand this properly we have to go back a step and look at the statement of the fertility parameter. The *FertilityHazard[AGE\_FERTILE][PROV]* statement actually creates a 35 by 10 parameter because there are 35 fertile age years and ten provinces, as we saw just now. At the same time, although the *AGE\_FERTILE* range contains values from 15 to 49, the index of the first dimension of the parameter still requires values lying between 0 and 34 (because there are 35 elements). So for example, in the fertility event time function, the *FertilityHazard[20][5]* parameter does not correspond to the fertility of a woman aged 20 in province number 5, but rather to the fertility of a woman aged 35 (20 plus 15, the minimum age in the range) in province number 5. The fertility of a woman aged 20 is found instead at index 5 in the parameter (*FertilityHazard[5][5]*). The fertility rates lag behind the actual age of the actor. The *RANGE\_POS* function in Modgen makes it possible to shift a variable automatically to take account of the minimum value of a span of whole numbers. The *RANGE\_POS(AGE\_FERTILE, age\_int)* statement reduces the *age\_int* value by 15 units, or the minimum value of the *AGE\_FERTILE* range, and so produces the correspondence between the age in whole numbers of the actor and the index which gives the fertility parameter by fertile years of age.

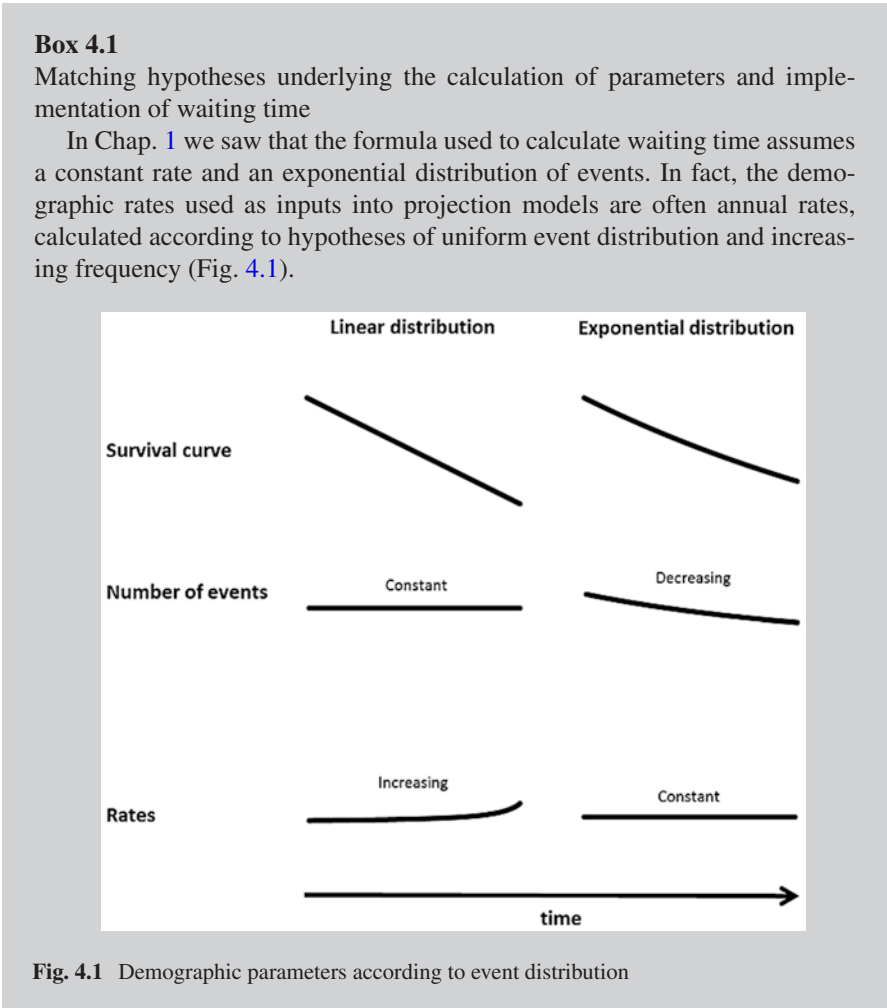
The second peculiarity lies in the formula for calculating the waiting time, which is slightly different from that used for the events in other modules. Instead of dividing the natural log of a random number by the rate, as we did previously, we divide here by  $-\ln(1-\text{rate})$ . Why is there a transformation of the rate? The formula for

---

<sup>2</sup>This argument is also valid for the mobility event we added in Chap. 3. We assumed there that the exit rates were real rates (rather than annual rates or probabilities). In fact the inputs are almost always annual rates, so we have to take the necessary precautions to avoid double counting of repeatable events. Having said this, where rates are low the probability of double counting is very small and the corresponding impact on the results is negligible.

calculating the waiting time which we have used up to now assumes that the rate is constant and that the distribution of events follows an exponential rule (in other words that the events are more frequent at the beginning than at the end of the year, since the population at risk is larger at the beginning). But in general the fertility rates are annual rates which assume a uniform distribution of births throughout the year, so that the rate actually rises slightly because the population at risk gradually diminishes while the number of births per unit of time remains constant (see Box 4.1 below for a detailed explanation).

Here we have corrected the formula in the code so that the number of births generated corresponds to the expected number of births. We could equally well have corrected the rates directly in the parameters file (using  $-\ln(1-\text{rate})$ ), or used another formula to calculate the waiting time. This is a decision for the modeller to make.



(continued)

**Box 4.1** (continued)

In this way, using a parameter calculated on the basis of linear hypotheses in an exponential equation gives different results. To take a concrete example, assume an annual fertility rate of 0.1 children per woman. This rate is calculated by taking the ratio of the number of births in a given year to the mean number of women present during this year (we generally use the mean value of the population measured at the beginning and the end of the period). For a population of 100,000 women, for example, this rate corresponds to 10,000 births per year. Putting this rate into an exponential distribution, we get:

$$\text{Births} = \text{WomenPopulation} (1 - e^{-\text{rate}}) = 100\,000 * (1 - e^{-0.1}) = 9516.$$

An exponential model using an annual rate of 0.1 tends to under-estimate the number of events. So a model which calculates waiting times using an exponential equation but an annual rate based on a linear hypothesis would slightly under-estimate the number of births. Note that these differences will be greater if the rates are higher. Most of the time, however, demographic rates are relatively low, and differences between linear and exponential estimations are small.

To get correct numbers, we can modify the equation determining the waiting time to make it linear. Instead of an equation of the form:

```
tTimeEvent = WAIT( - TIME( log( RandUniform(1) ) / RATE ) );
```

we could use a linear form

```
tTimeEvent = WAIT( TIME( RandUniform(1) / RATE ) );
```

The rate can also be corrected so that the exponential model gives a number of births to match the linear model. This means resolving the following equation:

$$\text{Births} = \text{Population} * \text{Rate} * t = \text{Population} * (1 - e^{-t * \text{Corrected Rate}})$$

For 1 year ( $t=1$ ),

$$\text{Rate} = 1 - e^{-\text{Corrected Rate}}$$

$$\text{Corrected Rate} = -\ln(1 - \text{rate})$$

(continued)

**Box 4.1** (continued)

This is the solution adopted for the model developed in this book. A rate of 0.1 applied to a population of 100,000 women then gives the required total of 10,000 births. The power and flexibility of Modgen allow the waiting times to be calculated for a wide variety of risk and duration models. We therefore need to be extra careful in order to make sure that the same hypotheses underpin the calculations of inputs and of waiting times.<sup>3</sup>

The aim here is to show that modelling of waiting time in Modgen must always match the hypotheses underlying the rates being used as inputs.

Finally, the time function sends the wait time back to the Modgen events manager, and the event function is implemented as soon as the random duration expires (lines 49–57). The main task of this event is to create a new *Person* object and launch the simulation of this new actor.

The code in line 52 creates an object of type *Person*, or in other words a new actor (in strictly C++ jargon, this is a pointer to an object of type *Person*). The next line (53) creates the actor in concrete terms by allocating a memory span. Next, the code in line 54 launches the simulation of the actor by calling the *Start* function. Notice that the *Start* function now takes on three values as arguments, which was not the case in the previous examples. In the next section we will see how the *Start* function has been modified to take care of births occurring in the course of the simulation. Finally, the *last\_age\_fertile* variable takes the value of the age of the mother at the time of the birth. As we explained earlier, this variable enables us to avoid double events in the same year. Later on, it will also be used to calculate the mean age of childbearing. Now let's look at how to modify the *Start* function to deal with initializing newborn actors.

## 4.3 Modifying the *PersonCore* Module and the *Start* Function

The fertility event creates a new actor whose simulation is initialized by calling the *Start* function. In previous chapters, the *Start* function was only called in the main file (which has the name of the model and contains the *Simulation* and *CaseSimulation* functions). Things were simpler then, because all the actors were born at the same moment at the start of the simulation. The rules for attributing states were the same for all actors.

The appearance of new actors **during** the simulation changes this situation. Now there are actors created at the beginning of the simulation, whose characteristics are attributed according to predefined distributions, and actors created during the simu-

<sup>3</sup>Note here that the examples in Chaps. 1, 2 and 3 assume that the rates used as inputs are derived based on the hypothesis that mortality and mobility are distributed exponentially.

lation, whose characteristics are partially determined by the actors to whom they are linked (through the mother-child link).

Actors born during the simulation are different from actors of the initial cohort in two respects. Firstly, the initial value of the Modgen time variable (*time*, or absolute time) is not the same; for members of the starting cohort it is zero, but for those born in the course of the simulation it is greater than zero. Secondly, the birth province is not attributed in the same way in the two cases; members of the initial cohort have a province randomly attributed to them according to the distribution specified in the parameters, while actors originating from a fertility event are given the province of residence of their mother. So the *Start* function has to be modified to take into account the type of actor being simulated.

To accommodate these specific features, new arguments will have to be added to the *Start* function. To be precise, we will need three new arguments. The first will indicate the exact moment (in absolute time) when an actor enters the simulation. This argument will initialize the *time* variable, which itself determines the derived state *year\_int*. A second argument will specify the type of actor being simulated. This argument enables the *Start* function to distinguish a new birth from a member of the original cohort. The third argument contains a link and enables the *Start* function to access the characteristics of the parent-actor when the initialized actor is a new birth. This argument takes the form of a C++ pointer to the parent-actor.

The *Start* function now takes the following form:

```

                                                    PersonCore.mpp
81  // The start function now takes three arguments:
82  // 1) dTimeStart is the exact time at which a simulation
    starts
83  // 2) ActorType tells if the simulated actor is part of
    the
84  //    starting cohort or born in the model
85  // 3) prMother is a pointer to the actor who gave birth
    to the
86  //    new actor
87  void Person::Start(double dTimeStart, int ActorType,
    Person *prMother)
88  {
89  // Modgen initializes all actor variables
90  // before the code in this function is executed.
91
92  age = 0;
93  // Sets continuous time. time>0 if actor is born during
    simulation
94  time = dTimeStart;

    ...

```

Note the presence of the three arguments in the name of the function (line 87). The argument *dTimeStart* gives the actor's entry time into the simulation: *time* = 0 for members of the initial cohort, and *time* > 0 (and equal to the time of the birth event) for actors born during the simulation. The argument *ActorType* specifies the type of actor, whether a member of the initial cohort or a subsequent birth. Finally, the argument *\*prMother* is a pointer to an object of type *Person* (an actor), the parent-actor.

All actors of the initial cohort enter the simulation at age zero, and the definition of the *age* variable is therefore always the same: *age* = 0 (line 92). The *time* variable has a value equal to the argument *dTimeStart*. The value of the argument is itself defined when the *Start* function is called in the fertility event or in the *CaseSimulation* function, as we will see later.

```

                                                                    PersonCore.mpp
...

96  // Sex is randomly attributed according to parameter
    ProbabilityMale
97      if (RandUniform(2) < ProbabilityMale)
98          {
99              sex = S_MAL;
100         }
101     else
102         {
103             sex = S_FEM;
104         };
...

```

The way sex is assigned is the same for both types of actor (lines 97–104), so there is no need to distinguish between the newly born and members of the initial cohort. On the other hand, attributing a province of residence varies according to the type of actor. So a different sequence of instructions is implemented depending on whether the *Actortype* variable has the value 0 (actor who is a member of the starting cohort) or 1 (subsequent birth, see lines 109–136). In the same way as *dTimeStart*, *ActorType* is defined when the *Start* function is called (line 54, see previous section).

PersonCore.mpp

```

...

106 // ActorType = 0 (actor from the starting cohort)
107 // ActorType = 1 (actor born in the simulation)
108
109 if (ActorType == 0) {
110
111     // The following lines attribute province of birth
112     // and province of residence to the actor
113
114     // A temporary variable to store the province
115     // value
116     int prov_temp = { 0 };
117     // Lookup picks a random province
118     // according to cumrate distribution
119     Lookup_ProvBirth(RandUniform(5), &prov_temp);
120     // Casts the (integer) prov_temp value into the
121     // prov state variable (of PROV type)
122     prov = (PROV)prov_temp;
123     // Province of residence at birth is the province
124     // of birth
125     prov_birth = prov;
126 }
127
128 else {
129
130     if (prMother != NULL) {
131         lMother = prMother;
132         prov = lMother->prov;
133         prov_birth = prov;
134     };
135
136 };
137
...

```

Using a conditional statement on the *ActorType* variable (line 109), we attribute the province of residence. This is done randomly for an actor member of the starting cohort (*ActorType* == 0, lines 109–124); for a subsequent birth, the new actor is given the province of residence of its parent actor (lines 126–136). To do this, we first make sure that the pointer is not empty (i.e. that it does not have NULL as its value, see lines 128–134 – we will see later why the value could be NULL). Next,

we assign the value of the *prMother* pointer (given as an argument) to the Modgen *lMother* link, previously defined in the fertility module (line 130). Then the characteristics of the parent-actor can be transferred using the Modgen link and the *the->* operator. The *prov = lMother -> prov* line then assigns a province of residence (the parent-actor's own province of residence) to the new actor. The province of birth can then be easily defined as the province of residence at the time of the birth (line 132).

Remember that the *Start* function is a member of the *Person* class, and that as such it must be declared inside the class description (the actor *Person*). Since we added three arguments to the *Start* function description, we must make sure that these arguments are also added in the function declaration:

```

                                                    PersonCore.mpp
45    void Start(double dTimeStart , int ActorType, Person
        *prMother);
```

As we can see, it is a relatively simple matter to complexify the *Start* function progressively as the model evolves and as new types of actors and state variables are added. It is easy to adjust the code to include specific features and exceptions in a model. This flexibility is one of the strengths of microsimulation in general and of Modgen in particular.

Before we move on, we will take a look at the two places where the *Start* function is called (i.e. where the simulation is initiated): in the fertility module and in the *CaseSimulation* function.

In the fertility module, the *Start* function is used with the following argument values: *Start(time, 1, this)*. Let's look at each of these in turn. Because a birth occurs during the course of the simulation, absolute time at the moment of birth corresponds to the *time* variable at the moment of the event (first argument, *dTimeStart*). This is the starting time for the new actor. The second argument (*ActorType*), which has the value 1, corresponds to the "Birth" type of actor and tells the *Start* function to create a link to the mother (see the *Start* function code above). Finally, the last argument (*prMother*), takes a rather special value: *this*. *this* is a reserved C++ key word and is a pointer to the object in which it is embedded. In our model, a birth occurs during the lifetime of a female actor. At the time of a birth, the fertility event initialises a new actor by passing a pointer to itself (*this*) to the *Start* function (as a third argument). This will provide the link from the mother to the child.

The *Start* function is also called in the main file (which carries the model name) each time the simulation of a new actor in the starting cohort is launched. In previous chapters, the *Start* function was called without arguments. Now we have to add the three new arguments, otherwise the compiler will return an error message, as it now expects a *Start* function with three arguments. The code for the main file can be found in Appendix 4.4.

ModgenExample.mpp

```
29 poFirstActor->Start( 0,0,NULL );
```

The actors from the initial cohort enter the simulation at time 0 (first argument) and are of type 0 (second argument – remember that a birth is of type 1). Because these actors are not born within the model during the course of the simulation, they cannot have links with a parent-actor. But even when the actor created has no parent, the *Start* function requires a pointer to a *Person* object as a third argument. So the value *NULL* is passed to the *Start* function. *NULL* is a C++ key word (much like *this*), representing a blank pointer. The number, position and type of arguments must always be the same for every call of the same function.

The model is now capable of simulating the evolution of a cohort, taking its fertility characteristics into account. In the next section, we will define the output tables which enable us to visualise the evolution of the cohort and the births taking place as part of the model.

## 4.4 Bringing the Tables Together in a Results Module

Before defining the output tables which will be used to visualise the simulation of fertility, it would be a good idea to reorganise the code so as to separate the sections to do with modelling itself from those to do with result tables. It is not absolutely necessary to do this, but it will prove to be helpful in a number of ways. Firstly, most of the relevant results will concern state variables belonging to a number of different modules, or even to all of the modules taken together. A module to bring together the creation of all the results tables will help to avoid confusions in locating the result tables. Also, centralising the tables in a single file will give us an overview of all the results at once. Finally, taking the tables out of the event modules is a way of making the modelling code less cumbersome and easier to read.

A module for results is created like any other module in Modgen (see Chap. 3). In this chapter the results module has been called *Tables.mpp* (see Appendix 4.3). This module does not require a parameter file, as tables do not require any parameters of their own. All the tables of the previous exercises have been eliminated, and three new tables are inserted in this new results module.

To check that the fertility module is working properly and to validate its outputs, we need a table measuring the number of births and the age-specific fertility rates.

```

Tables.mpp
1  table Person FertilityBirths // Births and age-specific
   fertility rates
2  [age_int    >=    MIN(AGE_FERTILE)    &&    age_int    <=
   MAX(AGE_FERTILE)]
3  {
4      {
5          // Number of births
6          changes(last_age_fertile),
7          // Annual fertility rates decimals=4
8          changes(last_age_fertile) / duration(sex, S_FEM)
9      }
10
11     *split(year_int, YEAR5) + // Year
12     *age_int + // Age
13     *prov + // Province
14 };

```

The *FertilityBirths* table uses the *changes* derived state on *last\_age\_fertile*, counting the number of times this state variable has changed its value, which happens every time there is a birth (line 6). The fertility rate is calculated by taking the ratio between the number of births and the number of person-years. The derived state *duration(sex, S\_FEM)* in line 8 counts the time spent by the actor in the female state (or the number of woman-years).

The dimensions of interest are included next: the year, in five-year groups, to monitor the evolution of fertility; age, to produce age-specific fertility rates; and finally province, to enable inter-provincial comparisons to be made.

Remember that the comments will serve as labels in the table. Also, the expression *decimals = 4* in the comments (line 7) specifies the number of decimal places to be used in displaying the results. This is one of the rare instances where Modgen uses the code found in comments for other purposes than labelling.

Next, it would be interesting to calculate the mean age at which women give birth, or average age at childbirth.

```

Tables.mpp
16  table Person AvAgeCB // Average age at childbirth
17  {
18      {
19          // Average age at childbirth decimals=4
20          value_at_changes(last_age_fertile, age)
21          / changes(last_age_fertile)
22      }
23
24     *prov + //Province
25 };

```

The *AvAgeCB* table calculates the average age at childbirth using the derived state *value\_at\_changes*. This derived state sums the values of the state specified in the second argument (here, the Modgen continuous variable *age*) from every change in value of the first argument (*last\_age\_fertile*). In the example above (at lines 20–21) *value\_at\_changes* sums the values of the *age* variable at each birth. Dividing this sum by the number of births (calculated as in the first table using the derived state *changes*), we get the average age at childbirth. Next we add the province as an extra dimension, which enables us once again to do interprovincial comparisons. Because, in this model, average age at childbirth does not vary with age or year, there is no point in adding these two dimensions to the table.

Finally we can add a “control” table which allows us to do demographic monitoring of the simulated population.

```

                                                                    Tables.mpp
27  table Person Demography // Population
28  {
29      {
30          // Population size at the end of period
31          value_at_changes(split(year_int, YEAR5), alive),
32          entrances(alive, FALSE),           // Deaths
33          changes(last_age_fertile),         // Births
34          changes(prov),                     // Exits
35          event(changes(prov))              // Entrances
36      }
37
38      *prov + //Province
39      *split(age_int, AGE_GROUP) + // Age
40      *split(year_int, YEAR5) // Year
41
42  };
41  };

```

The first item in the *Demography* table (line 30) gives the population at the end of the period: the derived state *value\_at\_changes* sums the state variable *alive* when the value of *split(year\_int, YEAR5)* changes, that is every 5 years. The second element counts the deaths (the entries into the *alive = FALSE* state). The two other lines use the derived state *changes* in two different ways in order to count those entering and leaving each of the provinces. Let’s now look at this in more detail.

Remember that the *changes* derived state calculates the number of state changes in a given variable. So *changes(prov)* counts the number of times an actor changes province. If the table also contains a province dimension (*\*prov+* in line 38), *changes(prov)* shows the number of migrations by province of residence. But if a change in the state of a variable has to be tabulated according to this same state variable, which value will be used in the table – the one before or the one after the

change? In our example, will a change of province be recorded under the province of origin or under the province of destination?

By default, Modgen tabulates state values before executing the event function. If an actor resident in Ontario undergoes a migration experience, this will therefore be recorded as a change in the *prov* state for a resident of Ontario. In this way, what is produced is the number of changes of province by province of residence BEFORE the mobility event occurs, or in other words it is those exiting rather than those entering who are counted.

But Modgen also allows us to execute the event function before state values are recorded for tabulation. So for example if a resident of Ontario leaves for Alberta, Modgen can first proceed to implement the event function (changing the value of *prov*) before registering the change using the derived state *changes*. Here the change in province will be accounted for as affecting a resident of Alberta, i.e. AFTER the migration event has taken place. The operator which enables this alternative registration is called *event*<sup>4</sup>: to use it, the derived state is entered as its argument (see line 35). It enables us to obtain the number of changes of state by province of residence after the event has taken place, in this case counting those entering rather than those exiting. The default operator is called *interval()*, so that *changes(prov)* is equivalent to *interval(changes(prov))*.

With these three tables, we will be able to track the different demographic components of our model. The results will also alert us of any errors in design or programming that may have arisen.

## 4.5 Looking at the Results

Now we can precompile, compile and open the microsimulation program (see *Precompiling, compiling and running the microsimulation programme*, chapter 1). We then open the base scenario and add the parameters file *Base(Fertility).dat* which was created along with the fertility module (see Fig. 4.2).

We must next modify the temporary values we inserted for the fertility rates (you can use your own data for these, or take the values provided on the book's website (<http://www.microsimulationandpopulationdynamics.com/>) in the *Base(Fertility).dat* file).

After launching the simulation, we access the results for the number of births and fertility rates by clicking on the table called *Births and age-specific fertility rates*.

The table (Fig. 4.3) gives the births per year of age of the mother, for the ten provinces. By default, the results shown are for the first 5 years: naturally, since the starting cohort has not yet reached the age of 15, no birth is noted here, and the fertility rates cannot be calculated. By selecting different simulation years, it is pos-

---

<sup>4</sup>Not to be confused with the definition of an event in the actor *Person*.

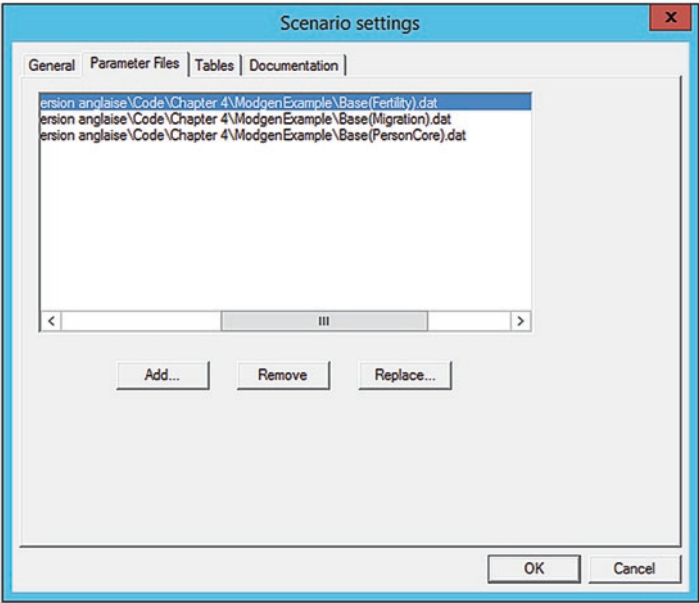


Fig. 4.2 Adding the fertility parameters file

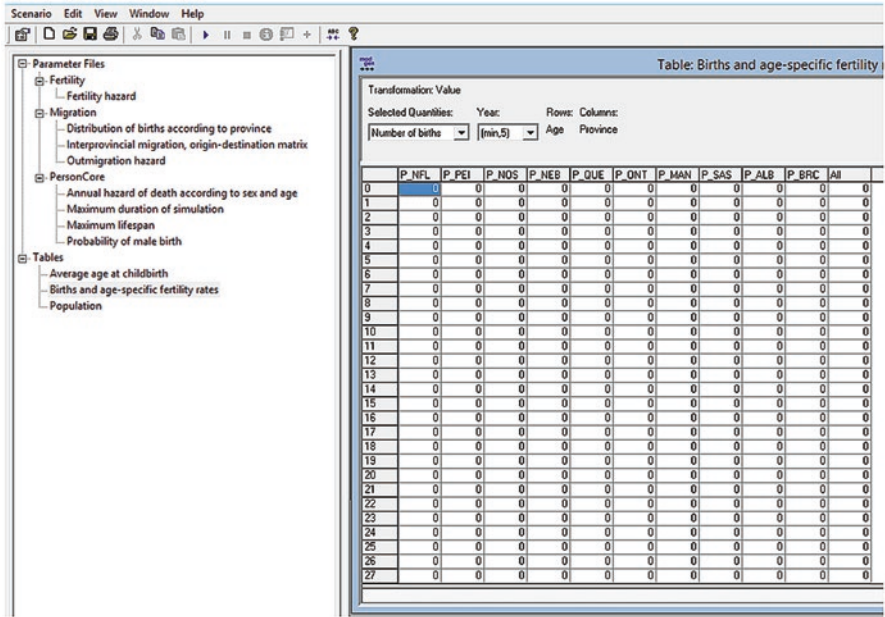


Fig. 4.3 Table of births and age-specific fertility rates

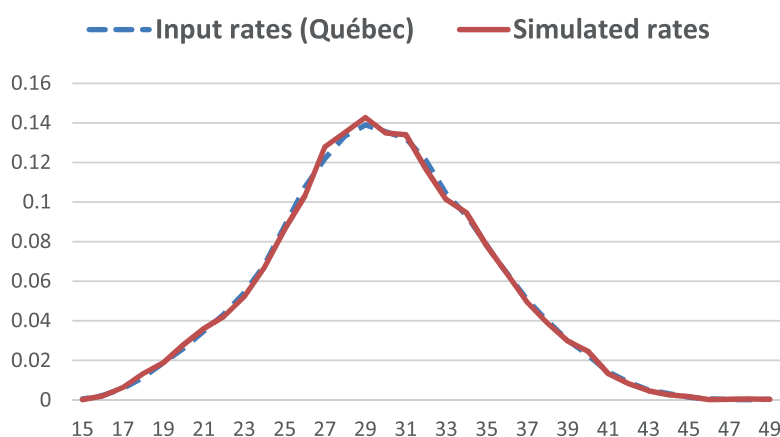
sible to observe the cohort's evolution. Between years 15 and 20 for example, we can observe births to women aged 15–19 (data not shown); between years 20 and 25, births to women aged 20–24, and so on. So the table allows us to follow the children born to women of the initial cohort. But from the 30th year onwards, these children of the initial cohort begin to have children of their own as they get to age 15. Progressively, as the women of the second generation grow older, births occur throughout the entire fertile age range.

In view of this very uneven distribution of births across the years, the observation of age-specific fertility rates will be more robust if we select all years together (*All*) rather than a particular five year period. The simulated fertility rates should correspond to the fertility rates we used as inputs, apart from a small random difference caused by the Monte Carlo error (Fig. 4.4). The difference between inputs and outputs can be reduced to virtually zero by increasing the number of cases in the scenario, thus reducing the Monte Carlo error.

At first sight, trying to reproduce input rates in output tables might seem like a waste of time. In fact it is a useful validation exercise: it allows the developer to check that the model is behaving as planned and that the waiting times have been correctly specified. Here, of course, the aim is mainly educational, as it serves to demonstrate the behaviour of the different Modgen derived states.

The second table shows the average age at childbirth by province. This result is interesting because we would usually have to do a calculation to obtain it. The microsimulation allows us to derive this result without explicit calculations. A researcher interested in fertility could easily improve the model by adding birth-order-specific fertility rates, which would enable the mean age at childbirth to be calculated according to birth order.

Finally we can observe the different demographic events in our simulation by looking at the *Population* table (Fig. 4.5).



**Fig. 4.4** Comparison of simulated fertility rates and input fertility rates for the province of Québec (100,000 cases for the whole of Canada)

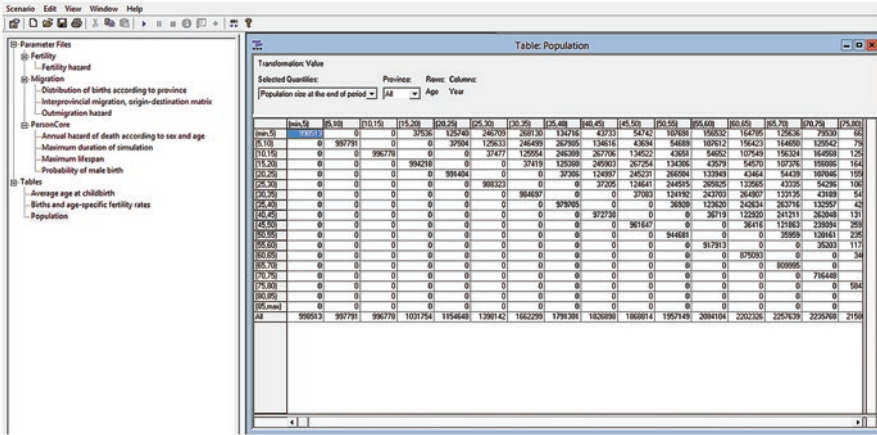


Fig. 4.5 Population, births, deaths, exits and entries, by province

Exploring these results, we see the evolution of the initial cohort on the table diagonal. Note also that the population given by *value\_at\_changes* is the population at the end of the five year period. This can be easily verified by noting that in the first five year period the population equals 998,513, or 1,000,000 (the number of cases specified in the scenario) minus the number of deaths (which can be found by selecting *Deaths* in the drop-down menu *Selected Quantities*).

We can also see that after 30 years of simulation, only the initial cohort can still be clearly distinguished. The second and subsequent generations give birth simultaneously to new actors. It would still be possible to modify the code in the model and the table so as to distinguish the generation of the actors (by adding a state variable *Generation* equal to the generation of the mother+1). For those interested in mobility, the table provides data on entries and exits by province. It is easy to calculate net interprovincial migration by taking the difference between entries and exits for each of the provinces. For Canada as a whole, as expected, net migration is equal to zero, meaning that the sum of exits from all provinces is equal to the sum of all entries into provinces (an actor leaving one province must indeed enter another one in a closed projection).

## 4.6 Summary

Chapter by chapter we have built a microsimulation model to gradually enable us to make a forward projection of the population. This model allows us to simulate three of the main demographic components: mortality, fertility and internal mobility. This model remains with two major gaps. The first is that it is still a model which does not account for international migration, a component whose real demographic importance is becoming more and more evident, especially in Western countries.

The second gap is that the model does not yet enable us to use an observed population as a point of departure with all its different characteristics. Of course we could create a synthetic population analogous to a real population, giving the actors characteristics according to statistical distributions which have been calculated in advance (*cumrates*). But this solution quickly becomes impractical as the number of characteristics increases. For our simple little model, we would have to create a population which includes 100 age groups, two sexes and ten provinces of residence, which generates a total of 2000 categories. The growth of the number of possible combinations is exponential, and we would quickly find ourselves with enormous matrices to contain all the distributions according to the different characteristics included in the model. In a way we would be recreating here some of the problems encountered in multi-state models. The Modgen data importation module is more flexible and much more practical; it allows us to integrate a basic population which is drawn from an external database such as that of the national census. In the next chapter, we will see how to prepare the data file and how to integrate this kind of population into our microsimulation model.

## Appendices

### *Appendix 4.1 PersonCore.mpp*

Code Sections: Header, Parameters, Actor Person and functions

```

1   classification SEX{ S_FEM, S_MAL };
2
3   range AGE{ 0, 110 };
4
5   partition AGE_GROUP{ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50,
6     55, 60, 65,
7     70, 75, 80, 85 };
8
9   partition YEAR5{ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55,
10    60, 65, 70,
11    75, 80, 85, 90, 95, 100 };
12
13  parameters
14  {
15      //EN Annual hazard of death according to sex and age
16      double MortalityHazard[SEX][AGE][PROV];
17      int AgeMax; // Maximum lifespan
18      double ProbabilityMale; // Probability of male birth
19      int horizon; // Maximum duration of simulation

```

```

19  };
20
21  actor Person          //EN Individual
22  {
23
24      //EN Alive
25      logical alive = {TRUE};
26
27      SEX sex;          // Sex state variable
28
29      // Age state variable, auto-increment
30      AGE age_int = COERCE(AGE, self_scheduling_int(age));
31
32      // Year state variable, auto-increment
33      int year_int = self_scheduling_int(time);
34
35
36      event timeMortalityEvent, MortalityEvent;    //EN
      Mortality event
37
38      //LABEL(Person.Start, EN) Starts the actor
39      // The start function now takes three arguments:
40      // 1) dTimeStart is the exact time at which a simulation
      starts
41      // 2) ActorType tells if the simulated actor is part of
      the
42      //      starting cohort or born in the model
43      // 3) prMother is a pointer to the actor who gave birth
      to the
44      //      new actor
45      void Start(double dTimeStart, int ActorType, Person
      *prMother);
46
47      //LABEL(Person.Finish, EN) Finishes the actor
48      void Finish();
49  };
50
51  // The time function of MortalityEvent
52  TIME Person::timeMortalityEvent()
53  {
54      TIME tEventTime = TIME_INFINITE;
55
56      // If max age or horizon is reached, death event occurs
      immediately
57      if (age_int == AgeMax | year_int >= horizon)

```

```
58     {
59         tEventTime = WAIT(0);
60     }
61     else {
62         // Draw a random waiting time to death from
63         // an exponential distribution based on the
64         // constant hazard MortalityHazard.
65         tEventTime = WAIT(-TIME(log(RandUniform(1)) /
66             MortalityHazard[sex][age_int][prov]));
67     }
68
69     return tEventTime;
70 }
71
72 // The implement function of MortalityEvent
73 void Person::MortalityEvent()
74 {
75     if (year_int<Horizon) {alive = FALSE;};
76
77     // Remove the actor from the simulation.
78     Finish();
79 }
80
81 // The start function now takes three arguments:
82 // 1) dTimeStart is the exact time at which a simulation
83 //    starts
84 // 2) ActorType tells if the simulated actor is part of the
85 //    starting cohort or born in the model
86 // 3) prMother is a pointer to the actor who gave birth to
87 //    the
88 //    new actor
89 void Person::Start(double dTimeStart, int ActorType, Person
90 *prMother)
91 {
92     // Modgen initializes all actor variables
93     // before the code in this function is executed.
94
95     age = 0;
96     // Sets continuous time. time>0 if actor is born during
97     // simulation
98     time = dTimeStart;
99
100    // Sex is randomly attributed according to parameter
101    ProbabilityMale
```

```

97     if (RandUniform(2) < ProbabilityMale)
98     {
99         sex = S_MAL;
100    }
101    else
102    {
103        sex = S_FEM;
104    };
105
106    // ActorType = 0 (actor from the starting cohort)
107    // ActorType = 1 (actor born in the simulation)
108
109    if (ActorType == 0) {
110
111        // The following lines attribute province of birth
112        // and province of residence to the actor
113
114        // A temporary variable to store the province value
115        int prov_temp = { 0 };
116        // Lookup picks a random province
117        // according to cumrate distribution
118        Lookup_ProvBirth(RandUniform(5), &prov_temp);
119        // Casts the (integer) prov_temp value into the
120        // prov state variable (of PROV type)
121        prov = (PROV)prov_temp;
122        // Province of residence at birth is the province
123        // of birth
124        prov_birth = prov;
125    }
126    else {
127
128        if (prMother != NULL) {
129
130            lMother = prMother;
131            prov = lMother->prov;
132            prov_birth = prov;
133
134        };
135
136    };
137
138 }
139
140 /*NOTE(Person.Finish, EN)

```

```

141     The Finish function terminates the simulation of an
        actor.
142 */
143 void Person::Finish()
144 {
145     // After the code in this function is executed,
146     // Modgen removes the actor from tables and from the
        simulation.
147     // Modgen also recuperates any memory used by the actor.
148 }

```

## ***Appendix 4.2 Fertility.mpp***

Code Sections: Header, Parameters, Actor Person and functions

```

1  /*
2  This module contains the fertility event and related
    parameters
3  In the fertility event, a Modgen link is created between the
4  parent and the actor
5  */
6
7  link    Person.lMother;    // Link to the mother
8  range AGE_FERTILE{ 15, 49 }; // Fertile ages
9
10 parameters
11 {
12
13     double FertilityHazard[AGE_FERTILE][PROV]; // Fertility
        hazard
14
15 };
16
17 actor Person                // Individual
18 {
19
20     int last_age_fertile = { 0 }; // Age of actor at last
        birth
21
22     event TimeFertilityEvent, FertilityEvent; // Fertility
        event (birth)
23
24 };
25

```

```

26 // Fertility event
27 TIME Person::TimeFertilityEvent()
28 {
29     TIME tEventTime = TIME_INFINITE;
30
31     // A birth can take place if actor is female and fertile
    (between
32     // 15 and 49 years old)
33     // An actor cannot give birth twice in the same year
34
35     if (age_int >= MIN(AGE_FERTILE) && age_int <= MAX(AGE_
    FERTILE) &&
36         sex == S_FEM && last_age_fertile != age_int)
37     {
38
39         tEventTime = WAIT(-TIME(log(RandUniform(6)) / (-log(1 -
40             FertilityHazard[RANGE_POS(AGE_FERTILE, age_
                int)] [prov]
41                 - 0.0000000001))));
42
43     };
44
45     return tEventTime;
46
47 }
48
49 void Person::FertilityEvent()
50 {
51
52     Person      *prChild = { NULL }; // Creates a new actor
53     prChild = new Person(); // Instantiation of the class
    Person
54     prChild->Start(time, 1, this); // Starts simulating the
    new actor
55     last_age_fertile = age_int; // Indicates age of actor at
    last birth
56
57 }

```

**Appendix 4.3 Tables.mpp**

```

1  table Person FertilityBirths // Births and age-specific
   fertility rates
2  [age_int >= MIN(AGE_FERTILE) && age_int <= MAX(AGE_FERTILE)]
3  {
4      {
5          // Number of births
6          changes(last_age_fertile),
7          // Annual fertility rates decimals=4
8          changes(last_age_fertile) / duration(sex, S_FEM)
9      }
10
11     *split(year_int, YEAR5) + // Year
12     *age_int + // Age
13     *prov + // Province
14 };
15
16 table Person AvAgeCB // Average age at childbirth
17 {
18     {
19         // Average age at childbirth decimals=4
20         value_at_changes(last_age_fertile, age)
21         / changes(last_age_fertile)
22     }
23
24     *prov + //Province
25 };
26
27 table Person Demography // Population
28 {
29     {
30         // Population size at the end of period
31         value_at_changes(split(year_int, YEAR5), alive),
32         entrances(alive, FALSE), // Deaths
33         changes(last_age_fertile), // Births
34         changes(prov), // Exits
35         event(changes(prov)) // Entrances
36     }
37
38     *prov + //Province
39     *split(age_int, AGE_GROUP) + // Age
40     *split(year_int, YEAR5) // Year
41
42 };

```

***Appendix 4.4 ModgenExample.mpp***

```

1  //LABEL(ModgenExample, EN) Core simulation functions
2
3  /* NOTE(ModgenExample, EN)
4      This module contains core simulation functions and
5      definitions.
6  */
7
8  // The model version number
9  version 1, 0, 0, 0;
10
11 // The model type
12 model_type case_based;
13
14 // The data type used to represent time
15 time_type double;
16
17 // Supported languages
18 languages {
19     EN // English
20 };
21
22 // The CaseSimulation function simulates a single case,
23 // and is called by the Simulation function declared later
24 // in this module.
25 void CaseSimulation( )
26 {
27     // Initialize the first actor in the case.
28     Person *poFirstActor = new Person();
29     poFirstActor->Start(0, 0, NULL );
30
31     // Continue processing events until there are no more.
32     // Model code is responsible for ending the case by
33     // calling
34     // Finish on all existant actors.
35
36     // The Modgen run-time implements the global
37     // event queue gpoEventQueue.
38     while ( !gpoEventQueue->Empty() )
39     {
40         // The global variables gbCancelled and gbErrors
41         // are maintained by the Modgen run-time.

```

```
41         if ( gbCancelled || gbErrors )
42         {
43             // The user cancelled the simulation,
44             // or run-time errors occurred.
45             // Terminate the case immediately.
46             gpoEventQueue->FinishAllActors();
47         }
48         else
49         {
50             // Age all actors to the time of the next event.
51             gpoEventQueue->WaitUntil( gpoEventQueue->
                NextEvent() );
52
53             // Implement the next event.
54             gpoEventQueue->Implement();
55         }
56     }
57
58     // Note that Modgen handles memory cleanup when
59     // Finish is called on an actor.
60 }
61
62
63 // The Simulation function is called by Modgen to simulate
64 // a set of cases.
65 void Simulation()
66 {
67     // counter for cases simulated
68     long lCase = 0;
69
70     // The Modgen run-time implements CASES (used below),
71     // which supplies the number of cases to simulate in a
72     // particular thread.
73
74     //
75     // The following loop for cases is stopped if
76     // - the simulation is cancelled by the user,
77     //   with partial reports (gbInterrupted)
78     // - the simulation is cancelled by the user,
79     //   with no partial reports (gbCancelled)
80     // - a run-time error occurs (gbErrors)
81     //
82     // The global variables gbInterrupted, gbCancelled and
83     // gbErrors
84     // are maintained by the Modgen run-time.
```

```
81     for ( lCase = 0; lCase < CASES() && !gbInterrupted &&  
      !gbCancelled  
82         && !gbErrors; lCase++ )  
83     {  
84         // Simulate a case.  
85  
86         // Tell the Modgen run-time to prepare to simulate  
      a new case.  
87         StartCase();  
88  
89         // Call the CaseSimulation function defined earlier  
      in this module.  
90         CaseSimulation();  
91  
92         // Tell the Modgen run-time that the case has been  
      completed.  
93         SignalCase();  
94     }  
95 }
```

# Chapter 5

## The Base Population

### Aims of This Chapter

- Building a base population file from an external database
- Adding a class to import data into a Modgen project
- Modifying the main model file so that characteristics of actors can be imported from an external file into the simulation
- Adding a new argument to the *Start* function

This fifth chapter will deal with the integration of a base population into a Modgen microsimulation model, and as such will be more technical than conceptual or demographic. Up to now, the simulated population has been synthetic; all the actors and their characteristics have been generated by the model itself. A synthetic population requires statistical distributions to be created, describing the composition of the population. An example of this is the distribution of births by province which we included in Chap. 3. However, as a starting point for the simulation, it may be more practical to use microdata taken directly from a real population. Rather than generating random characteristics for the actors, such as birth province or sex, we will be using information contained in a database representing an actual population. In this chapter, data from the National Household Survey (NHS) will be used to make a projection of the population of Canada starting from 2011 (the year of the survey). We will start by describing the preparation and the formatting of the data file to be used for the base population. Next, we will see how to use an external class to read the datafile and import its content. Finally, we will modify the main file and the *Start* function so as to use the imported data to initialize actor states.

We will modify the example from Chap. 4, replacing the starting cohort by a real base population (the population of Canada as observed by the 2011 NHS). We will be working on the *ModgenExample.mpp* main file, the *PersonCore.mpp* module and the

results module (*Tables.mpp*). The content of these files can be found in the appendices. As usual, the model's entire code can be downloaded from the website.

## 5.1 Preparing a Microdata File

A base population file must contain the data required to initialize the actors' state values. The file is structured so that each line corresponds to an individual. The values representing each of the states of the individuals (the variables) must be separated by commas (*in .csv format*).

To create a base population file, we first have to obtain a microdata file which is representative of the population to be simulated. In this chapter, we use the public microdata file of the National Household Survey (NHS) of Statistics Canada, which is based on a representative sample of the Canadian population in 2011.

Once we have identified the source of the data, the variables needed for the simulation are selected. The main state variables in the previous chapter were age, sex and province of residence. These must therefore be included in the base population file. In addition to these three variables, we will include the sample weight, each individual in the file being representative of a variable number of individuals in the population as a whole. Ideally we would also add the province of birth, but these data are not available in the published microdata file of the NHS, and so this state variable is removed from the Chap. 4 model.

Any statistical software can be used to prepare the data file. It is important to recode the variables of interest in such a way that their coding scheme corresponds to that of the state variables in the microsimulation model. So make sure that sex is coded 0 for females and 1 for males, and that codes 0–9 are used for the provinces in the same order as in the *PROV* classification. This is preferable to the manipulation of data in the model after the importation.

Once this ordering is done, the whole set is exported into a comma separated data file (.csv). As an example, we have used the *Outsheet* function from the Stata software package<sup>1</sup>:

```
outsheet age sexe pr weight using "Y:\LSD1\BasePop\ ///
\PopBaseBookC5.csv", comma nonames nolabel replace
```

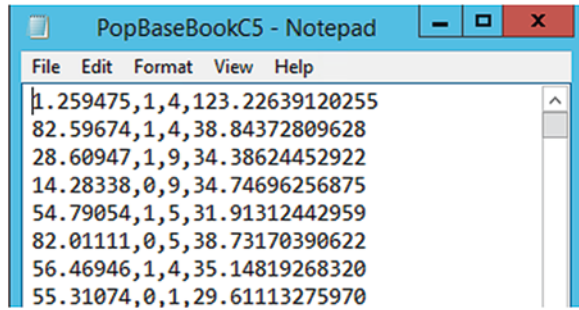
To check that the formatting of the data is correct, open the newly created file using a spreadsheet or word processing program. If the file generated is too big to be opened, tests can be done on smaller sub-samples or databases before exporting the whole file. The contents of the data file should be formatted as in Fig. 5.1.

Age is shown in the first column. Note that it is the exact age that is saved rather than age in completed years. This will be used to initialise the continuous variable

---

<sup>1</sup> Consult the manual of your preferred software for the appropriate export function.

**Fig. 5.1** Format of data in the base population file



*age* in the model. Remember that age in whole years (*age\_int*) is automatically updated by the *self\_scheduling\_int* function.

In the second and third columns, we find the codes for sex and province of residence. Finally, the sample weight is found in the fourth column. Note that the data are separated by commas, and that the decimal points are indicated by full stops.

## 5.2 Importing External Data into Modgen

C++ contains a whole array of functions to open, read and close text files, but to use these tools efficiently, one may require a relatively high level of programming skills. Fortunately, some tools have already been developed by professional programmers to perform this task.

In order to import data into Modgen, we will use a C++ class developed by Steve Gribbles and made freely available through an MIT license.<sup>2</sup>

To use this class, we will first need to copy some files into the project directory. On the book website or on Springer Extras Online, in the Chap. 5 section, download and copy into your project directory the following two files: *microdata\_csv.h* and *custom.h*. It's that simple!

<sup>2</sup>The **MIT License** reads as follows: « Copyright (c) 2013–2015 OpenM++ Contributors. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The new data import class is called *input\_csv*. It contains a number of functions to open, close, read from and write into *csv* files (comma separated files). These will be integrated into the main model module, *ModgenExample.mpp*.

### 5.3 Integrating Import Functions into the Main File

Now that we have a data file and a data import class, we can integrate information from the 2011 Canadian population by modifying the model from Chap. 4.

Since the main simulation loop is located in the *Simulation* function, this is where data importation will take place.

```

ModgenExample.mpp

80 void Simulation()
81 {
82     // Microdata input file
83     input_csv input_file;
84
85     input_file.open("PopBaseBookC5.csv");
86
87     // counter for cases simulated
88     long lCase = 0;
89
90     ...
91
102     for ( lCase = 0; lCase < CASES() && !gbInterrupted
        && !gbCancelled
103         && !gbErrors; lCase++ )
104     {
105
106         // quit the loop if end-of-file is reached
107         if (!input_file.read_record(lCase))
108         {
109             break;
110         }
111
112         // Simulate a case.
113
114         // Tell the Modgen run-time to prepare to simu-
            late a new case.
115         StartCase();
116
117         // Call the CaseSimulation function def. earlier
            in this module.

```

```

118         CaseSimulation(input_file);
119
120         // Tell the Modgen run-time that the case has been
            completed.
121         SignalCase();
122     }
123
124     input_file.close();
125 }
```

First, an object of the class *input\_csv* must be created so that its functions may be used (line 83). The new object is called *input\_file* and is used on line 85 to open the file *PopBaseBookC5.csv* (our base population file). The file is now ready to be read from, line by line.

The read function is implemented inside the simulation loop: with every new case, a new line will be read from the data file. The code on lines 106–110 tries to read the record number *lCase*: if the *input\_file.read\_record* returns the value *FALSE*, that is if the record number exceeds the number of lines available in the file, the simulation loop is stopped (the function *break* is used to exit the loop). Otherwise, a new case is started and the *input\_file* object is passed to the *CaseSimulation* function.

Once all the cases have been simulated, the simulation loop ends and the data file can be closed again using the *input\_file.close()* function (line 124).

Because an argument is now passed to the *CaseSimulation* function, we must modify its declaration to include this argument.

```

ModgenExample.mpp
25     void CaseSimulation(const input_csv& input)
```

All that needs to be understood from this is that the argument *input* must be of type *input\_csv* (the object *input\_file* is of type *input\_csv*). This input argument will contain all the data read from the base population file, as we will see in the next section.

## 5.4 Using the Imported Data and Modifying the *Start* Function

The *CaseSimulation* argument *input* now contains the four elements that were read from a record of the base population file, and is therefore used as a vector.

```

ModgenExample.mpp
25 void CaseSimulation(const input_csv& input)
26 {
27     /* Imported data
28
29     0 - Age
30     1 - Sex
31     2 - Province
32     3 - Weight
33
34     */
35
36     // Gets the number of sub-samples specified in the
        parameters
37     int    nSubSamples = { 0 };
38     nSubSamples = GetSubSamples();
39
40     // Set case weight
41     SetCaseWeight(input[3], input[3] * nSubSamples);
...

```

To access individual elements, you need to write a number (an index) between square brackets added at the end of the variable name. For instance, to access the first element, you need to write *input[0]*; for the second element *input[1]*; etc. In the base population file, data is organised so that the first element is the age, the second is the sex, the third is the province and the fourth and last one is the weight (these are recalled in the comments).

We will start by setting the weight of the case, as this may only be done from the *CaseSimulation* function, just before calling the start function.

The sample weight is found in fourth position in the base population file, so in *input[3]*. The weight of the case is defined using the *SetCaseWeight* function (see line 41). This function takes two arguments. The first is the weight of the case, and the second is the weight of the case when it is part of a sub-sample.

Modgen allows sub-samples to be created in a single simulation, so that the Monte Carlo error may be estimated. The simulation is split into  $n$  sub-simulations, each including a number of cases equal to the total number of cases divided by  $n$ . In order for the result to remain representative of the population, the weight of a case simulated in a sub-sample must be multiplied by the number of sub-samples. To illustrate this, suppose that in a sample of a 100 actors, each has a weight of one unit. A simulation of this population will obviously give a total of 100 members. Now suppose that the simulation is made up of two sub-samples. The simulation of each of these will have a membership of 50 individuals. For these sub-samples to remain representative of the total population, we must multiply their weight by the number of sub-samples (2). Lines 37 and 38 of the code above show how the num-

ber of sub-samples can be obtained using the *GetSubSamples* function. The weight (*input[3]*) and the adjusted weight for the sub-samples (*input[3]\*nSubSamples*) is then passed to the *SetCaseWeight* function (line 41).

Once the case is started, the weight may not be modified again. If an actor experiences a fertility event during the simulation, its weight will be transferred to the actor-child because the latter is part of the same simulated case (remember that one case can contain more than one actor). So in Modgen, weight is really the weight of a case and not that of an actor.

We must now pass the other informations read from the base population file to the *Start* function. The three elements of the input object (*input[0]*, *input[1]*, *input[2]*) are therefore added to the list of arguments of the *Start* function (see line 46 below), in which state values will be initialised.

```

ModgenExample.mpp
43      // Initialize the first actor in the case. Age, sex and
        province
44      // are passed to the start function
45      Person *poFirstActor = new Person();
46      poFirstActor->Start(0, 0, NULL, input[0], input[1],
        input[2]);

```

The *Start* function now has six arguments, whereas in Chap. 4 it included only three. The declaration and the definition of the *Start* function in the *PersonCore* file must therefore be modified. The declaration of the function is modified first (line 46–47).

```

PersonCore.mpp
46      void Start(double dTimeStart, int ActorType, Person
        *prMother, double
47      fileage, double filesex, double fileprovince);

```

The three new arguments are given descriptive names: *fileage* for the age of the actor, *filesex* for the sex and *fileprovince* for the province of residence. These variables will receive values from *input[0]*, *input[1]*, and *input[2]*, respectively. The function definition below must be modified as well.

```

PersonCore.mpp
91      void Person::Start(double dTimeStart, int ActorType,
        Person *prMother,
92      double fileage, double filesex, double fileprovince)

```

Below is the modified *Start* function, now making use of the information from the base population data file.

```

                                                    PersonCore.mpp
91  void Person::Start(double dTimeStart, int ActorType,
    Person *prMother,
92  double fileage, double filesex, double fileprovince)
93  {
94      // Modgen initializes all actor variables
95      // before the code in this function is executed.
96
97      // Sets continuous time. time>0 if actor is born dur-
        ing simulation
98      time = dTimeStart;
99
100     // ActorType = 0 (actor from the base population)
101     // ActorType = 1 (actor born in the simulation)
102
103     if (ActorType == 0) {
104
105         // Continuous age
106         age = fileage;
107         // Sex
108         sex = (SEX)(INT) filesex;
109         // Province
110         prov = (PROV)(INT) fileprovince;
111
112     }
113
114     else {
115
116         // Continuous age is zero at birth
117         age = 0;
118
119         // Sex is rand attributed according to parameter
        ProbabilityMale
120         if (RandUniform(2) < ProbabilityMale)
121         {
122             sex = S_MAL;
123         }
124         else
125         {
126             sex = S_FEM;
127         };
128
129         // Province of residence is the same as the
        mother's
130         if (prMother != NULL) {
131
132             lMother = prMother;

```

```
133             prov = lMother->prov;
134
135         };
136
137     };
138
139 }
```

Comparing the *Start* function here with the one from Chap. 4, we notice that the types of actor have changed (lines 100–101). The 0 type, which previously corresponded to a member of the initial cohort, is now an actor drawn from the base population.

Let's look at how state variables are initialized.

Continuous time is initialized in the same way for both actors born in the model and drawn from the base population. So there is no need to repeat the command for each type of actor, and the initialization takes place before the condition on the type of actor. This does not mean that time is the same for the two types of actor. Members of the base population will start their simulation at time zero, just as the members of the initial cohort used to do. Births continue to start their simulation at a time greater than zero. As before, the value of the initial time is defined before calling the *Start* function and is passed on as an argument (*dTimeStart*).

The condition at line 103 determines whether the actor is part of the base population or not. If this is the case, the first block of code is run (lines 103–112). State values are taken directly from the last three arguments of the *Start* function (*fileage*, *filesex* and *fileprovince*) and assigned to the corresponding state variables (lines 105–110). It is worth emphasizing here that the values passed as arguments are of type *double*, and that they do not necessarily correspond to the types of the state variable used in the model. Initializing age is not a problem, because the *age* variable created by Modgen is a continuous variable of type *double* and because exact age (also type *double*) is used in the base population file. The formats of the two numbers are matched.

By contrast, sex and province are of the type *SEX* and *PROV* respectively, so the arguments will have to be converted to an appropriate format. This is what is known as a *casting*. The value of the variable is first converted to a whole number type (*INT*) and then to the (*SEX*) or (*PROV*) type. The procedure is done in two stages because Modgen does not allow a direct conversion from a *double* type into a Modgen classification. The two castings can however be done on the same line (lines 108–110). Note that birth province was removed from the model because this information is not available in the National Household Survey.

If an actor is not part of the base population, then it must be the result of a birth (*ActorType==1*), and the block of code following the *else* is executed (lines 114–137). Initializing the state variables takes place as in Chap. 4, because initialization of actors born within the model has not changed. Notice that initialization of age and of sex now differs for births and for actors from the base population: for the latter, age and sex are no longer randomly attributed.

Before we move on, we must modify the *Start* function call in the fertility module. Remember that *Start* now takes six arguments instead of three, and that only three were previously included in the function call from the fertility module. Arguments must total six; if not, the compiler will call an error.

```
Fertility.mpp3
```

```
prChild->Start(time, 1, this, 0,0,0); // Starts simulating
the new actor
```

The values we enter as arguments for *fileage*, *filesex* and *fileprovince* are not very important, as they will not be used in the *Start* function. Here we have simply inserted three zeroes.

## 5.5 Modifying Calendar Time

In a cohort model, the calendar year has little practical meaning, and the simulation can start arbitrarily at time zero. But now that we have added a base population corresponding to a real population observed at a particular time, it is preferable to adjust Modgen time to reflect the appropriate calendar year. Because the National Household Survey was carried out in 2011, we will replace “time zero” with the year 2011 in our simulation. To modify the start time of the model, we first create a parameter to indicate the opening year of the simulation:

```
PersonCore.mpp
```

```
17    int StartYear; // Starting calendar year of simulation
```

The value of this parameter must also be initialized to “2011” in the *Base(PersonCore).dat* file. We could have hard-coded the starting year in the model, but using a parameter has one advantage: the user can change its value if, for example, the base population is changed.

Next we have to modify the code of the main file so that the starting time for each case is set to *StartYear*. Remember that each case is initialized in the *CaseSimulation* function of the main file. This function in turn calls the *Start* function and provides the value of the initial calendar time as a first argument.

---

<sup>3</sup>The entire module code is not reproduced at the end of the chapter.

```

ModgenExample.mpp
46      poFirstActor->Start(StartYear, 0, NULL, input[0],
        input[1], input[2]);

```

Finally, so that the data are tabulated correctly in the results tables, we have to modify the YEAR5 partition in the *PersonCore.mpp* file. In Chap. 4, the partition divided the years into five-year groups starting from year 0. Here we have to start in 2011. Because we are projecting a real population, we will limit ourselves to observing the results over 25 years. A 100 year projection is unrealistic and not very useful. So the last element of the partition is 2036.

```

PersonCore.mpp
8      partition YEAR5{ 2011, 2016, 2021, 2026, 2031, 2036 };

```

## 5.6 Modifying the Scenario and Running the Model

As you can see in Appendix 5.3, the sole output table is identical to the demography table we used in Chap. 4, but now that the base population has been added, the results of the simulation will be quite different. This table contains the demographic data of interest for validating the simulation of the base population – the total population, the deaths, births and migrations by age and year.

The model from Chap. 5 must now be precompiled, compiled and opened (see the section on *Precompiling, compiling and running the microsimulation program*).

The new model does not contain any new parameters files, so none needs to be added to the scenario.

Before launching the simulation, we still have to modify the scenario to indicate the number of cases to be read in the base population file. Since our data file contains 879,464 lines, we enter that number in the *Cases* field of the *scenario settings* (Fig. 5.2).

In the scenario settings, we also have to make sure that the number of simulation threads is equal to 1 (see Fig. 5.2). Modgen makes it possible to create several simulation threads, which can be run in parallel to reduce computing time. But in the example developed in this chapter, adding extra simulation threads may cause problems, in part because more than one thread may try to access the base population file simultaneously. Some C++ functions would allow us to overcome this problem, but their description would be beyond the aims of this book.

Once the number of cases has been specified in the scenario, the horizon of the simulation must be modified to correspond to the calendar year at the expected end of the simulation. It is not very useful to simulate a real population over a hundred years, so a projection of 25 years should be sufficient. The horizon of the simulation will therefore be 2036. Next we start the simulation and open the *Population* table (Fig. 5.3).

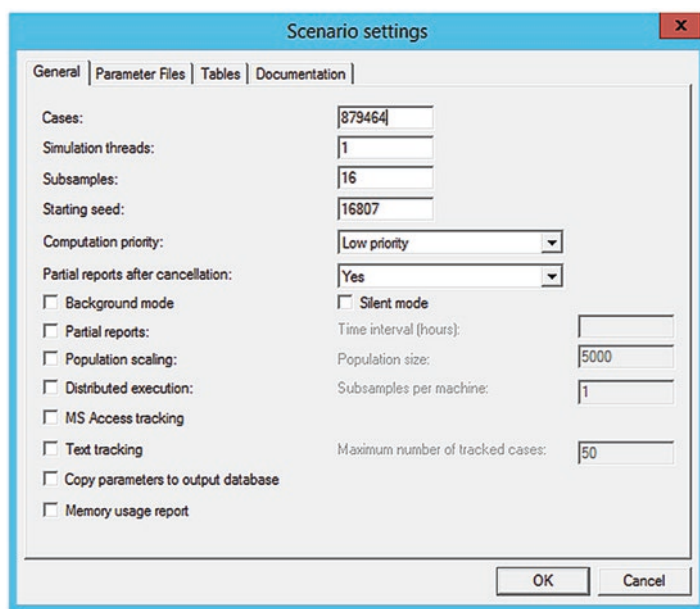


Fig. 5.2 Modifying the number of cases in the scenario

Scenario Edit View Window Help

Parameter Files

- Fertility
  - Fertility hazard
- Migration
  - Interprovincial migration, origin-destination matrix
  - Outmigration hazard
- PersonCore
  - Annual hazard of death according to sex and age
  - End of simulation
  - Maximum lifespan
  - Probability of male birth
  - Starting calendar year of simulation
- Tables
  - Population

Table: Population

Transformation: Value

Selected Quantities: Province: Sex state variable: Rows: Columns:

Population size at the end of period: All: All: Age: Year

	[min,5]	[2011,2015]	[2016,2021]	[2021,2026]	[2026,2031]	[2031,2036]	[2036,max]
[min,5]	0	1963069	1914067	1802893	1650135	1564058	0
(5,10]	0	1888290	1967670	1912296	1801134	1648572	0
(10,15]	0	1801022	1897601	1965518	1911119	1799558	0
(15,20]	0	1906727	1797808	1895184	1963680	1908383	0
(20,25]	0	2223573	1901629	1793439	1880787	1998478	0
(25,30]	0	2339107	2218841	1896144	1789101	1875464	0
(30,35]	0	2353257	2330749	2210507	1889914	1783777	0
(35,40]	0	2309488	2342831	2320976	2202087	1881360	0
(40,45]	0	2251942	2296873	2328891	2309012	2189152	0
(45,50]	0	2366539	2231128	2276369	2306753	2286151	0
(50,55]	0	2671271	2323273	2193681	2244747	2274380	0
(55,60]	0	2622070	2610375	2270348	2185962	2196744	0
(60,65]	0	2262783	2530873	2516460	2186447	2075111	0
(65,70]	0	1927681	2127954	2383843	2368944	2057529	0
(70,75]	0	1388767	1749766	1931372	2168736	2147026	0
(75,80]	0	987626	1188689	1497176	1653858	1853518	0
(80,85]	0	704173	750493	914959	1156776	1285482	0
(85,max]	0	724049	747395	784893	906292	1120220	0
All	0	34687539	34925815	34891930	34542485	33907671	0

Fig. 5.3 Table of results

Figure 5.3 shows the population table for Canada as a whole, by age and year. Tables by province or sex can be obtained by using the corresponding drop down menus.

The first thing to notice is that the population is now of comparable size to the real population of Canada – 34.7 million in 2016 (see the first non-blank column), which more or less corresponds to the expected result.<sup>4</sup> You can explore for yourself the demographic indicators by province and by sex.

Looking more closely at how the numbers evolve over time, we can see that the total population declines, which would seem astonishing given that all the official forecasts predict a rise in the Canadian population over the coming decades. The reason for this difference is of course linked to the fact that our model is closed to immigration, and that demographic growth is not possible in a context of low fertility and population aging without the help of international immigration. So in the sixth and final chapter, we will be looking at how to build in an immigration module.

## 5.7 Summary

The cohort-based simulations of Chaps. 1, 2, 3 and 4 turn out to be useful for analysing and understanding a number of demographic phenomena. The simulation in this chapter gets us closer to real-life projections by importing the characteristics of the Canadian population contained in a microdata file. Projecting a real population may also be done through a synthetic population generated using *cumrates* and real distributions, but using a base population is much more convenient.

Importing data from a microdata file into Modgen allows us to integrate a large amount of information external to the model, although in this chapter we have used only four variables (age, sex, province of residence and weight). This is another example of the greater flexibility Modgen can provide. The same model can be used to project different populations (by varying the base population) according to different parameters (using different scenarios).

In the next chapter we will complete the model by making it open to international migration. This final addition will enable us to make a real projection and to obtain results which are closer to Canada's demographic reality.

---

<sup>4</sup>The population of Canada is estimated to be 35,851 million at 1 July 2015, but the model is not yet counting gains due to net international migration.

## Appendices

### *Appendix 5.1 ModgenExample.mpp*

```

1  //LABEL(ModgenExample, EN) Core simulation functions
2
3  /* NOTE(ModgenExample, EN)
4      This module contains core simulation functions and
       definitions.
5  */
6
7  // The model version number
8  version 1, 0, 0, 0;
9
10 // The model type
11 model_type case_based;
12
13 // The data type used to represent time
14 time_type double;
15
16 // Supported languages
17 languages {
18     EN // English
19 };
20
21 // The CaseSimulation function simulates a single case,
22 // and is called by the Simulation function declared later
23 // in this module.
24
25 void CaseSimulation(const input_csv& input)
26 {
27     /* Imported data
28
29     0 - Age
30     1 - Sex
31     2 - Province
32     3 - Weight
33
34     */
35
36     // Gets the number of sub-samples specified in the
       parameters
37     int    nSubSamples = { 0 };
38     nSubSamples = GetSubSamples();
39

```

```
40      // Set case weight
41      SetCaseWeight(input[3], input[3] * nSubSamples);
42
43      // Initialize the first actor in the case. Age, sex and
         province
44      // are passed to the start function
45      Person *poFirstActor = new Person();
46      poFirstActor->Start(StartYear, 0, NULL, input[0], input[1],
         input[2]);
47
48      // Continue processing events until there are no more.
49      // Model code is responsible for ending the case by
         calling
50      // Finish on all existant actors.
51
52      // The Modgen run-time implements the global
53      // event queue gpoEventQueue.
54      while (!gpoEventQueue->Empty())
55      {
56          // The global variables gbCancelled and gbErrors
57          // are maintained by the Modgen run-time.
58          if ( gbCancelled || gbErrors )
59          {
60              // The user cancelled the simulation,
61              // or run-time errors occurred.
62              // Terminate the case immediately.
63              gpoEventQueue->FinishAllActors();
64          }
65          else
66          {
67              // Age all actors to the time of the next event.
68              gpoEventQueue->WaitUntil( gpoEventQueue->NextEvent
                 () );
69
70              // Implement the next event.
71              gpoEventQueue->Implement();
72          }
73      }
74
75      // Note that Modgen handles memory cleanup when
76      // Finish is called on an actor.
77  }
78
79  // The Simulation function is called by Modgen to simulate a
         set of cases.
```

```
80 void Simulation()
81 {
82     // Microdata input file
83     input_csv input_file;
84
85     input_file.open("PopBaseBookC5.csv");
86
87     // counter for cases simulated
88     long lCase = 0;
89
90     // The Modgen run-time implements CASES (used below),
91     // which supplies the number of cases to simulate in a
92     // particular thread.
93     //
94     // The following loop for cases is stopped if
95     // - the simulation is cancelled by the user,
96     //   with partial reports (gbInterrupted)
97     // - the simulation is cancelled by the user,
98     //   with no partial reports (gbCancelled)
99     // - a run-time error occurs (gbErrors)
100    //
101    // The global variables gbInterrupted, gbCancelled and
102    // gbErrors
103    // are maintained by the Modgen run-time.
104    for ( lCase = 0; lCase < CASES() && !gbInterrupted &&
105          !gbCancelled
106          && !gbErrors; lCase++ )
107    {
108        // quit the loop if end-of-file is reached
109        if (!input_file.read_record(lCase))
110        {
111            break;
112        }
113
114        // Simulate a case.
115
116        // Tell the Modgen run-time to prepare to simulate a
117        // new case.
118        StartCase();
119
120        // Call the CaseSimulation function defined earlier in
121        // this module.
122        CaseSimulation(input_file);
123    }
124 }
```

```

120          // Tell the Modgen run-time that the case has been
           completed.
121      SignalCase();
122  }
123
124      input_file.close();
125  }

```

## ***Appendix 5.2 PersonCore.mpp***

Code Sections: Header (lines 1 to 9), Parameters (lines 10 to 19), Actor *Person* and functions (lines 20 to 149)

```

1  classification SEX{ S_FEM, S_MAL };
2
3  range AGE{ 0, 110 };
4
5  partition AGE_GROUP{ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55,
6  60, 65, 70, 75, 80, 85 };
7
8  partition YEAR5{ 2011, 2016, 2021, 2026, 2031, 2036 };
9
10 parameters
11 {
12     //EN Annual hazard of death according to sex and age
13     double MortalityHazard[SEX][AGE][PROV];
14     int AgeMax;                // Maximum lifespan
15     double ProbabilityMale; // Probability of male birth
16     int horizon; // End of simulation
17     int StartYear; // Starting calendar year of simulation
18 };
19
20 actor Person                //EN Individual
21 {
22
23     //EN Alive
24     logical alive = {TRUE};
25
26     SEX sex;    // Sex state variable
27
28     // Age state variable, auto-increment
29     AGE age_int = COERCE(AGE, self_scheduling_int(age));

```

```

30
31 // Year state variable, auto-increment
32 int year_int = self_scheduling_int(time);
33
34
35 event timeMortalityEvent, MortalityEvent; //EN Mortality
    event
36
37 //LABEL(Person.Start, EN) Starts the actor
38 // The start function now takes six arguments:
39 // 1) dTimeStart is the exact time at which a simulation
    starts
40 // 2) ActorType tells if the simulated actor is part of the
41 //     starting cohort or born in the model
42 // 3) prMother is a pointer to the actor who gave birth to the
43 //     new actor
44 // 4-6) State values extracted from parameter file
45
46 void Start(double dTimeStart, int ActorType, Person
    *prMother, double fileage,
47 double filesex, double fileprovince);
48
49 //LABEL(Person.Finish, EN) Finishes the actor
50 void Finish();
51 };
52
53 // The time function of MortalityEvent
54 TIME Person::timeMortalityEvent()
55 {
56     TIME tEventTime = TIME_INFINITE;
57
58     // If max age is reached, death event occurs immediately
59     if (age_int == AgeMax | year_int >= horizon)
60     {
61         tEventTime = WAIT(0);
62     }
63     else {
64         // Draw a random waiting time to death from
65         // an exponential distribution based on the
66         // constant hazard MortalityHazard.
67         tEventTime = WAIT(-TIME(log(RandUniform(1)) /
68             MortalityHazard[sex][age_int][prov]));
69     }
70

```

```
71     return tEventTime;
72 }
73
74 // The implement function of MortalityEvent
75 void Person::MortalityEvent()
76 {
77     if (year_int < horizon) { alive = FALSE; };
78
79     // Remove the actor from the simulation.
80     Finish();
81 }
82
83 // The start function now takes three arguments:
84 // 1) dTimeStart is the exact time at which a simulation starts
85 // 2) ActorType tells if the simulated actor is part of the
86 //    starting cohort or born in the model
87 // 3) prMother is a pointer to the actor who gave birth to the
88 //    new actor
89 // 4-6) State values extracted from parameter file
90
91 void Person::Start(double dTimeStart, int ActorType, Person
    *prMother, double fileage,
92 double filesex, double fileprovince)
93 {
94     // Modgen initializes all actor variables
95     // before the code in this function is executed.
96
97     // Sets continuous time. time>0 if actor is born during
98     // simulation
99     time = dTimeStart;
100
101     // ActorType = 0 (actor from the base population)
102     // ActorType = 1 (actor born in the simulation)
103
104     if (ActorType == 0) {
105         // Continuous age
106         age = fileage;
107         // Sex
108         sex = (SEX)(INT) filesex;
109         // Province
110         prov = (PROV)(INT) fileprovince;
111     }
112 }
113
```

```
114     else {
115
116         // Continuous age is zero at birth
117         age = 0;
118
119         // Sex is randomly attributed according to parameter
120         ProbabilityMale
121         if (RandUniform(2) < ProbabilityMale)
122         {
123             sex = S_MAL;
124         }
125         else
126         {
127             sex = S_FEM;
128         };
129
130         // Province of residence is the same as the mother's
131         if (prMother != NULL) {
132             lMother = prMother;
133             prov = lMother->prov;
134         }
135     };
136
137 };
138
139 }
140
141 /*NOTE(Person.Finish, EN)
142     The Finish function terminates the simulation of an actor.
143 */
144 void Person::Finish()
145 {
146     // After the code in this function is executed,
147     // Modgen removes the actor from tables and from the
148     // simulation.
149     // Modgen also recuperates any memory used by the actor.
150 }
```

***Appendix 5.3 Tables.mpp***

```
1 table Person Demography // Population
2 {
3     {
4         // Population size at the end of period
5         value_at_changes(split(year_int, YEAR5), alive),
6         entrances(alive, FALSE),           // Deaths
7         changes(last_age_fertile),         // Births
8         changes(prov),                     // Exits
9         event(changes(prov))               // Entrances
10    }
11
12    *prov + //Province
13    *sex +
14    *split(age_int, AGE_GROUP) + // Age
15    *split(year_int, YEAR5) // Year
16
17 };
```

## Chapter 6

# International Migration

### Aims of This Chapter

- Adding an international migration module to include emigration and immigration
- Modifying the *Start* function and the main file to take account of immigration
- Modifying the results table to include the counting of international migration events

We are now at the final design stage of a basic microsimulation model – the integration of an international migration module. This module will complete the model we have been developing gradually over the first five chapters. Just as with births, immigrants come into the simulation in the course of the projection, at a simulation time greater than zero. And like actors entering the simulation through birth, immigrants are liable to experience all the demographic events in the model (fertility, mortality, mobility etc.). But unlike births, the creation of an immigrant actor is not the result of an event taking place in the simulation of an existing actor. Immigrant actors are simulated as separate Modgen cases.

A number of different methods can be used to integrate immigrant actors into a microsimulation model. The characteristics of immigrant actors could be imputed using *cumrate* type distributions of the kind we used in the first cohort models (remember that sex and birth province were attributed using *cumrate* distributions). Immigrants created in this way would be simulated in an extra simulation loop inserted into the main model file. Another way would be to integrate immigrants directly into the base population file. All we would need to do is to add to the data file the census date for the base population and the date of entry to Canada for immigrants. This variable would be used to initialise the start time of their simulation at various moments in the projection, depending on the case type. This is a

simple option because it needs little programming, but it is not very flexible. Moreover, it requires manipulation of the base population file, which may eventually lead to errors.

We have chosen to use a mixed strategy. A sub-set of immigrants will be selected in the base population for repeated simulation in each year of the projection. So an immigrant in the base population will be simulated for the first time when the case is read in the base population file (and the actor simulation will be started at *time* = 2011), and then re-simulated for each year of the simulation, in 2011, 2012, 2013 etc (*time* > 2011), up to the projection horizon. To do this we will use a conditional statement (*if*) to identify the immigrants to be re-simulated. A second loop will have to be added in addition to the main simulation loop (described in Chap. 5), to generate annual immigration. This will be described in detail in this chapter.

Modeling immigration in this way has many advantages. First, it is relatively easy to do, as immigrants are dynamically selected from the base population. Second, characteristics of immigrants are implicitly defined. If we choose immigrants who have arrived in Canada in the last 5 years (as we will do in this chapter), the immigrants in the projection will have the characteristics of this sub-population. There is no need to create *cumrate* distributions to randomly assign values to the state variables of the immigrants. This is especially useful when the number of characteristics increases, because *cumrate* matrices then become complex and difficult to manipulate. Finally, the modelling of immigration described in this chapter is also flexible – the composition of an immigration cohort can be easily changed by selecting a different sub-population of immigrants – for example by choosing less recent immigrants, or by modifying simulation weights.

Before going ahead, we have to modify the content of the base population to include two other variables which have to do with immigration (which means adding two new columns to the four columns of the base population file in Chap. 5). The first variable to add is immigration status (in column 5), coded 0 for native-born and 1 for immigrants. The second is the age at the time of immigration, saved as a *double* precision variable (added in column 6). For a quick start, you may use the Chap. 6 base population file available on the book website (<http://www.microsimulationandpopulationdynamics.com/>) or Springer Extras Online. Note that for the purpose of this chapter, non-permanent residents have been removed from the NHS database, and so the number of cases in the Chap. 6 file will be smaller than in the Chap. 5 file. So the scenario will need to be modified accordingly.

## 6.1 Creating an International Migration Module

The new international migration module includes new parameters and new state variables, as well as a new emigration event. But no immigration event is there to be found. Why is that? Why doesn't immigration need its own event? This is because immigration is not an event in the Modgen sense, as it does not occur inside an existing actor. No actor from the base population is "at risk" of immigration. Births,

by contrast, are linked to a female actor of fertile age. So while births are linked to an event within a case, each immigrant must constitute a separate case. We will come back to this later.

First, we need to declare new parameters and state variables for the immigration module. We need some kind of classification for immigration status, with two possible states: non immigrant (I\_NON\_IMM) or immigrant (I\_IMM).

```

InternationalMigration.mpp
1  classification IMM {I_NON_IMM, I_IMM};

```

Next, a corresponding state variable of the IMM type is declared in the description of the actor (line 14).

```

InternationalMigration.mpp
14      IMM imm;

```

Because age at immigration is only useful for selecting actors involved in immigration (we will come back to this in the next section), there is no need to create a *range* for this. However we do have to create a state variable which will store the value of the age at immigration, taken from the base population file (the variable is *double*, as age at immigration is stored as a floating-point number in the base population file).

```

InternationalMigration.mpp
15      double age_imm;

```

An additional variable (line 16) will be used to differentiate immigrants already present in the base population, whose simulation begins in 2011 exactly, from future cohorts of immigrants in the projection, whose simulation starts at a time later than 2011. This state variable will be useful for tabulating the intensity of annual immigration in the different scenarios created from the user interface.

```

InternationalMigration.mpp
16      int is_imm = {0};

```

Finally, parameters allowing us to control the intensity of emigration and immigration are added.

```

InternationalMigration.mpp
3  parameters
4  {
5      // Sum of weights of recent immigration in microdata
        file
6      double WeightRecentImmig2011;
7      double ImmigSimul; // Number of immigrants per year
8      double EmigrationHazard[SEX][AGE][PROV]; // Risk of
        emigration
9  };

```

The *WeightRecentImmig2011* parameter gives the total weight of the set of immigrants that will be selected from the base population file to be part of the annual immigration in the projection. In our example, the selected immigrants will be those who have been in Canada for less than 5 years. *ImmigSimul* on the other hand indicates the number of required immigrants per year during the simulation. With these two parameters we can adjust the weight of each actor-immigrant to control immigration intensity: all we have to do is to standardise the immigration weight to unity using *WeightRecentImmig2011*, and then multiply the result by the required number of immigrants per year for the simulation (*ImmigSimul*).

Finally, the annual risk of emigration (the probability of leaving Canada for a foreign country) is given by the *EmigrationHazard* parameter. Like the mortality risk, this varies with sex, age, and province of residence.

A new parameter file called *Base(Immigration).dat* must be created. It will contain the values of the international migration parameters (the code for this file is not reproduced in the appendix). All values for the *EmigrationHazard* parameter are set to zero: they will be modified from the user interface.

```

Base(InternationalMigration).dat
parameters
{
    // Sum of weights, recent immigration in microdata file
    double WeightRecentImmig2011 = 1121807.6;

    // Number of immigrants per year
    double ImmigSimul = 250000;

    // Risk of emigration
    double EmigrationHazard[SEX][AGE][PROV] = {(2020) 0,};
};

```

In the parameters file we can see that the total weight of recent immigration (defined as those who have arrived in the past 5 years) is 1,121,807.6 in the base population.<sup>1</sup> This is to be expected, given that annual immigration to Canada is around 250,000 immigrants per year, and that some immigrants leave Canada soon after arrival. The number of immigrants per year of simulation is set to 250,000, based on recent trends as reported by Citizenship and Immigration Canada. This number may be modified later to create scenarios with higher or lower immigration flows. A file containing all the international migration parameters for Canada (including emigration hazards) is also available on the book website (<http://www.microsimulationandpopulationdynamics.com/>) or on Springer Extras Online.

## 6.2 The Emigration Event

The international migration module must include an emigration event, so that actors may exit the simulation without going through a mortality event. As usual, this event is declared in the *Person* class.

```

                                                    InternationalMigration.mpp
17      event timeEmigrationEvent, EmigrationEvent;
          // Emigration event
18      logical emigration = { FALSE }; // Indicate that an
          actor has emigrated

```

The *emigration* state variable will be used as an indicator: its value will be changed from *FALSE* to *TRUE* if an emigration event occurs. Emigrants will be tabulated by counting the state transitions from *FALSE* to *TRUE* in the *emigration* variable. The code for the emigration event is very similar to the code used for the mortality event, and this is not surprising because both events have the function of removing an actor from the simulation. In the case of a death, the *alive* state changes from *TRUE* to *FALSE*. For an emigration event, the *emigration* state changes from *FALSE* to *TRUE*. The *Finish* function removes the actor from the simulation.

---

<sup>1</sup>This value was calculated in Stata when the base population was exported.

```

InternationalMigration.mpp
22 // Computes waiting time before emigration
23 TIME Person::timeEmigrationEvent()
24 {
25     TIME tTimeEvent = TIME_INFINITE;
26
27     tTimeEvent = WAIT(-TIME(log(RandUniform(7)) /
28         (EmigrationHazard[sex][age_int][prov]+
29         0.0000000001)));
30
31     return tTimeEvent;
32 }
33 // Event function
34 void Person::EmigrationEvent()
35 {
36     // Terminates actor and signal an emigration event
37     emigration = TRUE;
38     Finish();
39 }

```

Since the emigration event is so similar to the mortality event, it does not need further explanation. The integration of immigration on the other hand is not so straightforward, and this will be the subject of the next two sections.

### 6.3 Modifying the *Start* Function

The two state variables added to the immigration module earlier need to be properly initialised in the *Start* function, using the data imported from the base population file. Two new arguments for immigration status and age at immigration must first be added to the function declaration in the *Person* class

```

PersonCore.mpp
46 void Start(double dTimeStart, int ActorType, Person
    *prMother,
47     double fileage, double filesex, double fileprovince,
    double
    fileimmstat, double fileageimm);

```

and in the function description.

```

PersonCore.mpp
83 // The start function now takes 8 arguments:
84 // 1) dTimeStart is the exact time at which a simulation
    starts
85 // 2) ActorType tells if the simulated actor is part of
    the
86 //    starting cohort or born in the model
87 // 3) prMother is a pointer to the actor who gave birth
    to the
88 //    new actor
89 // 4-8) age, sex, province, immigrant status and age at
    immigration
90 void Person::Start(double dTimeStart, int ActorType,
91     Person *prMother, double fileage, double filesex,
92     double fileprovince, double fileimmstat, double
    fileageimm)

```

Calls to the *Start* function must be modified to include the new arguments. Dummy values (zeroes) are added to the call in the fertility module:

```

Fertility.mpp
prChild->Start(time, 1, this, 0,0,0,0,0); // Starts simu-
lating the new actor

```

In the model's main file, state values extracted from the base population file are inserted:

```

ModgenExample.mpp
43 // Initializes the first actor in the case. Age (0), sex
    (1), province (2),
44 // immigrant status (4) and age at immigration (5)
45 // are passed to the start function
46 Person *poFirstActor = new Person();
47 poFirstActor->Start(StartTime, 0, NULL, input[0], input[1],
48     input[2], input[4], input[5]);

```

Notice that the first argument has changed from *StartYear* to *StartTime*: this will be explained in the next section. The description of the *Start* function itself must be modified to make use of the new information passed in the arguments.

```

                                                    PersonCore.mpp
103    // ActorType = 0 (actor from the base population)
104    // ActorType = 1 (actor born in the simulation)
105
106
107    if (ActorType == 0) {
108
109        if (dTimeStart == StartYear) {
110            // Continuous age
111            age = fileage;
112        }
113        else {
114            // Continuous age is age at immigration
115            // if dTimeStart>0
116            age = fileageimm;
117            is_imm = 1;
118        };
119
120        // Sex
121        sex = (SEX) (INT) filesex;
122        // Province
123        prov = (PROV) (INT) fileprovince;
124        imm = (IMM) (INT) fileimmstat; // Immigrant status
125        age_imm = fileageimm; // Age at immigration
126
127
128    }

```

Immigrant status (*imm*) and age at immigration (*age\_imm*) are initialised directly, based on the values taken from the base population file (lines 124 and 125). The Modgen continuous variable *age* is initialised differently depending on whether the actor is taken from the base population or has immigrated during the simulation. To initialise *age*, the function must identify the origin of the actor: if the simulation of the actor starts precisely in 2011, then the actor is taken from the base population file (line 109); in all other cases, the actor is an immigrant who has arrived during the course of the projection (line 113). If an actor is from the base population, the actor's age is the age measured at the time of the survey (line 111). But if the actor is an immigrant, his age is initialised as the age at immigration (line 116). In fact the age of the immigrant upon his arrival in Canada is usually not the same as the age measured at the time of the survey or census. It is preferable for the age of the immigrant upon entry into the simulation to correspond to the real age at the time of immigration to Canada. If the actor is an immigrant, the *is\_imm* variable also takes the value 1, indicating that the actor is an immigrant who has arrived during the

projection and is not an immigrant from the base population. This variable will be used to tabulate the intensity of annual immigration.

Finally the *imm* and *age\_imm* variables are also initialised for actors born within the model, even though age at immigration is not applicable to these actors. Their immigrant status is of course *non immigrant* (*I\_NON\_IMM*, line 153, see Appendix 6.3), and their age at immigration is initialised arbitrarily at 0 (line 154). Any other values would work, as this variable is used only for immigrants.

## 6.4 Modifying the Simulation Loop to Include Immigration

In Chap. 5 we described the Modgen simulation loop, in which each line of the base population file was read and the simulation of each actor was started. The simulation of all these actors began in 2011 the year of the NHS survey used to construct the base population.

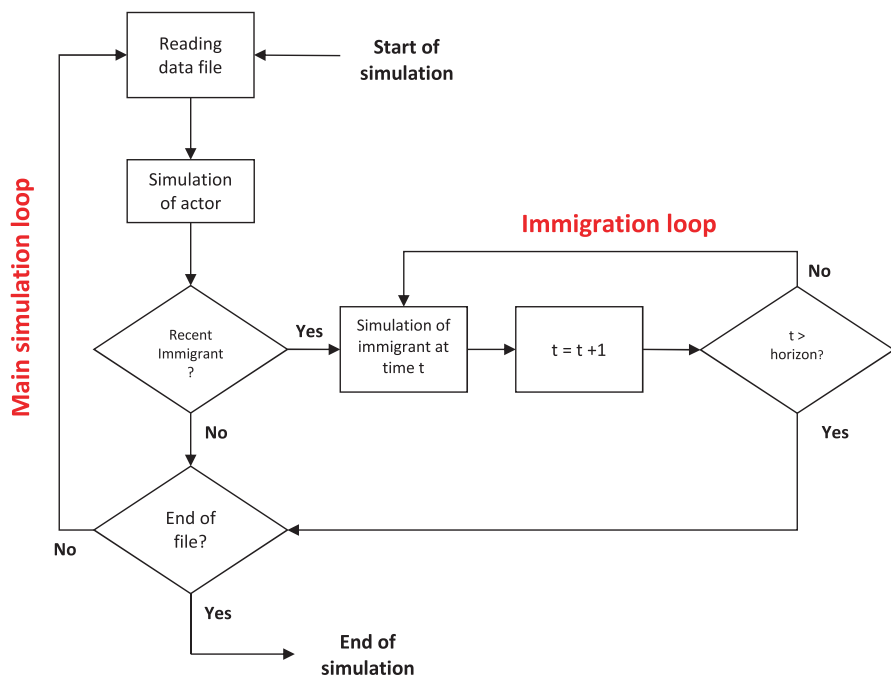
Immigrant actors, just like actors born during the simulation, appear in the simulation at projection times greater than zero. But unlike those who are born in the simulation, immigrants are not tied to any existing actor (or parent), and thus do not appear in the simulation through a regular event. A birth is an actor linked to an event in another actor, and both actors are part of the same case. An immigrant actor constitutes a case in its own right.

If an immigrant is not created by an event, how do we integrate this new type of actor into the projection? The answer, as you may have guessed, is that immigrant actors have to be generated in the main file, in the same way as are regular actors from the base population. As we mentioned earlier in this chapter, the immigrants are taken directly from the base population, based on the duration since immigration (less than 5 years). If a member of the base population satisfies this criterion, the case is replicated for each year of the simulation. Let's see how this is done.

Extra instructions have to be included in the simulation loop to integrate these immigrants into the projection. To be more precise, each time a record in the base population file is read, (1) the actor has to be simulated, then (2) a conditional statement checks whether the actor is a recent immigrant, and (3) if this is the case, the actor is simulated for each year of the simulation (as immigration in the model is annual).

Figure 6.1 below shows a schematic view of how immigration is integrated in the main file.

The diagram shows that integrating immigration implies a second simulation loop, and that this second loop is embedded in the first one (an immigrant has to be simulated for each year of the simulation). Now we can examine how this conceptual diagram is translated into Modgen code within the main file.



**Fig. 6.1** Diagram of microsimulation algorithm (the variable  $t$  represents simulation start time and is reset to 2011 every time a record is read)

ModgenExample.mpp

```

82 void Simulation()
83 {
84
85     // Microdata input file
86     input_csv input_file;
87
88     // Reads the file
89     input_file.open("PopBaseBookC6.csv");
90
91
92     // counter for cases simulated
93     long lCase = 0;
94
95     ...
96
107 for ( lCase = 0; lCase < CASES() && !gbInterrupted &&
!gbCancelled
108     && !gbErrors; lCase++ )
109     {
110

```

```
111         // exits the loop if end-of-file is reached
112         if (!input_file.read_record(lCase))
113         {
114             break;
115         }
116
117         /* Imported data
118
119         0 - Age
120         1 - Sex
121         2 - Province
122         3 - Weight
123         4 - Immigrant status
124         5 - Age at immigration
125
126         */
127
128         // Simulates a case.
129
130         // Tells the Modgen run-time to prepare to simulate a new case.
131         StartCase();
132
133         // Calls the CaseSimulation fcn defined earlier in this module.
134         CaseSimulation(input_file, StartYear);
135
136         // Tells the Modgen run-time that the case has been completed.
137         SignalCase();
138
139
140         // If immigrant is recent (age-age at immigration < 5)
141         // the case is repeated for every year of the simulation
142         if (((input_file[0] - input_file[5])<5) &&
143             input_file[4] == 1)
144         {
145
146             for (int yeartemp = StartYear; yeartemp <
147                 horizon; yeartemp++)
```

```

148             StartCase();
149
150             CaseSimulation(input_file,    yeartemp    +
                           RandUniform(5));
151
152             SignalCase();
153         }
154
155     }
156
157 }
158
159     input_file.close(); // Closes file
160 }
```

Let's take a moment to break down this code.

The first thing to notice is that the immigration loop (lines 142–155) is inserted into the main simulation loop (107–157).

Just as in the simulation loop from Chap. 5, a record is first read from the base population (line 112), and then the actor is simulated (lines 131–137). The attentive reader will have noticed that the *CaseSimulation* function now accepts an extra argument (it takes the *StartYear* value at line 134). This is the value of the actor initial simulation time: in this case, it is 2011 (the NHS year).

In the Chap. 5 example, the value of the initial time was integrated directly into the *Start* function call, which was located in the *CaseSimulation* function. All the actors simulated from the base population file therefore entered the simulation at the same time, in 2011 exactly. The situation is different now because the actors generated in the main file can start either at the beginning of 2011 (base population) or later (as immigrants). So from now on we have to specify the starting time of the simulation (*StartTime*) to the *CaseSimulation* function. To do this we add the *StartTime* argument to the *CaseSimulation* function header, a little higher up in the file code

```

ModgenExample.mpp
25     void CaseSimulation(const input_csv& input, double
                           StartTime )
```

and change the argument passed to the *Start* function accordingly:

```

ModgenExample.mpp
47      poFirstActor->Start(StartTime, 0, NULL, input[0],
      input[1],
48      input[2], input[4], input[5]);

```

Note that *StartTime* – the initial time of the simulation, which varies depending on the actor type – should not be confused with *StartYear*, which is the starting time of the base population and the year of the NHS (and therefore a constant).

Now back to the *Simulation* function and the simulation loop. After launching the actor's simulation, we find a condition (lines 142–143) checking whether the actor who has just been simulated is a recent immigrant or not. The condition checks whether the duration of time since the immigration is less than 5 years (*input\_file[0]-input\_file[5]<5* corresponds to age minus age at immigration, or the number of years since immigration), and that the actor is an immigrant. These two conditions are needed because age at immigration for non-immigrants was initialised to zero. Without the second part of the condition, non-immigrants aged less than 5 years old would have been selected as part of recent immigration.

The actor-immigrant is then re-simulated for each year in the projection: this is the immigration loop. In this loop, the value of the *yeartemp* variable moves from *StartYear* (2011 in our case) to *Horizon-1* (2035, for a 25 year projection). The case is launched with the usual Modgen functions – *StartCase*, *SimulationCase* and *SignalCase* – which have been copy-pasted from the main simulation loop. In the first year, instead of starting the simulation at the beginning of 2011 as we did for the base population actors, the simulation starts randomly during the year. The same is true for the succeeding years, so that the arrival of immigrants in the model is spread evenly throughout all the simulation years. Hence the value of *StartTime*, passed as an argument to the *CaseSimulation* function, is *yeartemp* (the immigrant actor's year of arrival) plus a random duration between 0 and 1 year. Once the immigration loop is finished, the simulation continues, and another case is read in the base population file (unless the end of the file has been reached).

## 6.5 Adjusting Weights

You may remember that each actor in the simulation has a given simulation weight specified in the base population file. Setting the weights as in Chap. 5 would yield an annual volume of immigration equal to the sum of weights of all recent immigrants to Canada (those who arrived 5 years or less prior to the NHS). Given the intensity of immigration to Canada, this number would exceed one million immigrants. Since such an annual rate is too high (as it represents the total immigration for a five year period), the weights must be adjusted in order to get the desired number of immigrants for the simulation.

The weights are adjusted in the *CaseSimulation* function, as was done in Chap. 5. But now a conditional statement must be added to differentiate immigration in the

model from other actors in the base population: if the start time of the simulation is larger than 2011 (*StartYear*), then the actor is an immigrant and the weights are adjusted as follows.

As we saw earlier, the total weight of immigrants who arrived in the 5 years prior to the NHS is about 1,121,808. To set the desired volume of annual immigration (in number of immigrants), we adjust the weight using a simple rule of three: we multiply the weight of the actor-immigrant by the the desired volume of annual immigration for the simulation (*ImmigSimul*), and then we divide the result by the weight of recent immigration in the base population (line 35). Since *ImmigSimul* is a parameter, we will be able to control the number of immigrants within the model's user interface.

```

ModgenExample.mpp
33 // Set case weight
34 if (StartTime > StartYear) { // Recent immigrants
35     SetCaseWeight(input[3] * ImmigSimul /
36                   WeightRecentImmig2011,
37                   input[3] * ImmigSimul / WeightRecentImmig2011 *
38                   nSubSamples);
39 }
40 else { // Everyone else
41     SetCaseWeight(input[3], input[3] * nSubSamples);
42 };

```

If the actor is not a recent immigrant, the weights are set as before in Chap. 5.

## 6.6 Generating Results

To generate the results of the simulation, we recycle the table used in Chap. 5 and expand it to derive the numbers of emigrants and immigrants from and to Canada.

```

Tables.mpp
10 entrances(duration_trigger(is_imm,1,0),TRUE)
11 // Immigration
12 entrances(emigration, TRUE) // Emigration

```

The derived state *duration\_trigger(is\_imm,1,0)* is a switch indicating whether an actor is an immigrant or not. It takes the value *TRUE* when an actor has passed a zero duration in the *is\_imm=1* state. Put another way, as soon as an immigrant enters the simulation, the value of the derived state *duration\_trigger* changes to *TRUE* and the derived state *entrances* records this change in state from *FALSE* to *TRUE*. Used on its own, *duration\_trigger*, which takes the value *TRUE* for the whole

of the simulation, would be counted in all the cells of the table (as long as the actor is alive, of course), and would not allow the number of immigrants arriving in a given period to be counted. The same would be true of the derived state *value\_in(is\_imm)*.

So why not use the derived state *entrances* on the *is\_imm* state variable instead? The problem is that state modifications carried out in the *Start* function are not accounted for by the derived states. Derived states only register changes of state occurring inside events. Because the *is\_imm* variable is initialised in the *Start* function (see the previous section), the derived state *entrances* does not “see” this change in state. A combination of the two derived states *duration\_trigger* and *entrances* is needed to achieve the correct count of annual immigration – a trigger to flag the immigrant’s entry into the simulation, with a derived state to detect the change in the value of this trigger. The derived state for emigration is relatively simple by comparison; it counts the number of times an actor enters the state *emigration = TRUE*.

Now that we have modified the table description, we can pre-compile, compile and open the microsimulation program (see the *Precompiling, Compiling and Running the microsimulation program* section). Before the simulation is launched, we have to modify some of the parameters in the base scenario. First, as we did with other modules, we have to add to the scenario the parameters file from the immigration module (*Base(Immigration).dat*). Next, the number of cases must be changed to 869,980 (remember that the number of cases in the base population file is slightly different from the number of cases in the file in Chap. 5).<sup>2</sup>

During the simulation, the progress bar will not be working properly, due to the cases generated by the immigration loop. The movement of the progress bar is actually based on the number of cases specified in the scenario, which controls (and this is very important) the number of iterations in the main simulation loop. In our example, the progress bar reaches 100% once 869,980 cases have been simulated.

But this number assumes that only cases from the base population file are being simulated. Because cases from the immigration loop are counted as well, the total number of cases counted by the progress bar is now equal to the number of cases in the base population, plus the number of recent immigrants times the number of years of simulation. For example, if there are 100 actors in the base population, and 10 of these are recent immigrants, and the length of the simulation is 25 years, then the total number of cases in the simulation will be 100, plus 25 times the 10 immigrant actors, or a total of 350 cases. Since a number of cases equal to 100 was specified in the scenario, the progress bar will go up to 350% by the time the simulation ends. If you want the progress bar to reflect the actual simulation time, you can write

---

<sup>2</sup>You must also make sure that the correct file name is specified in the *ModgenExample.mpp* file. In this chapter, this file is called “PopBaseBookC6.csv”. See *ModgenExample.mpp*, line 89 in Appendix 6.2.

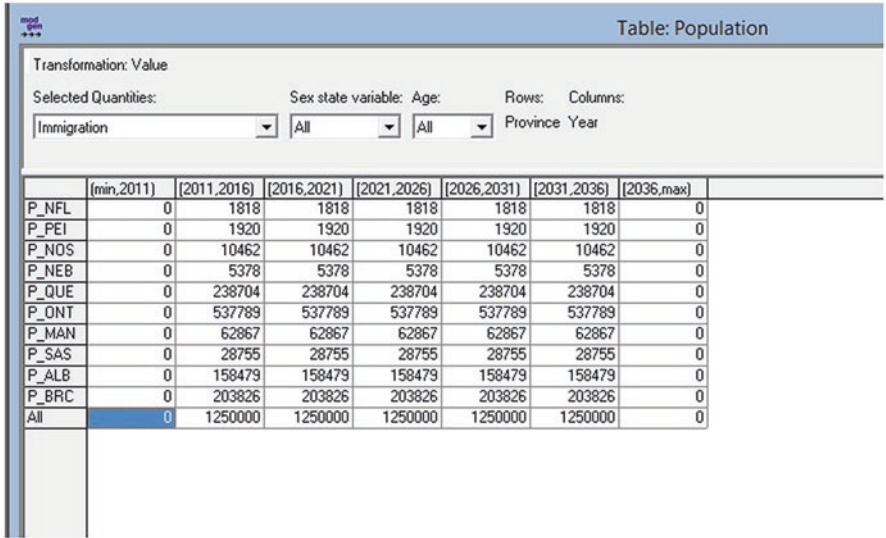
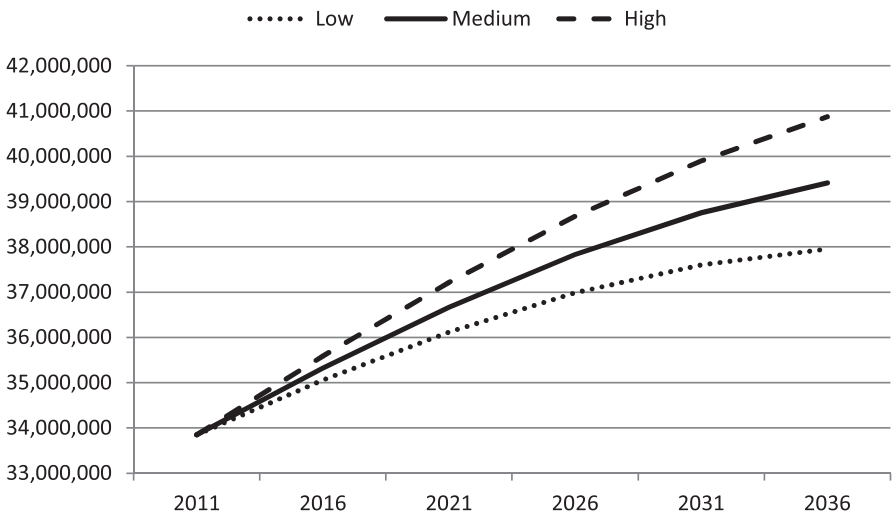


Fig. 6.2 Number of immigrants per five-year period

the actual number of cases in the scenario. But won't this higher number make the main simulation loop run for too many iterations, exceeding the total number of records in the base population file? Actually, specifying a number of cases higher than the number of records in the file does not matter, as the program automatically exits the main simulation loop when the end of the file has been reached.

We then start a simulation and open the *Population* table. Figure 6.2 shows the number of immigrants admitted per period of five years (the *Immigration* item under *Selected Quantities*). As the base scenario specified 250,000 immigrants per year, we can expect to have 1,250,000 immigrants in a five year period, and this is in fact the number that appears at the foot of the table. The table also shows the distribution of immigrants by province, reflecting the observed distribution of recent immigrants who arrived between 2006 and 2011 to Canada (because we chose immigrants who have arrived in the past 5 years).

We can use the Modgen user interface to change the number of immigrants per year and compare the effect of different immigration scenarios on population size. Choosing three scenarios, one low (200,000), one medium (250,000) and one high (300,000) and repeating the simulation using these different volumes of annual immigration, we get the population results illustrated in Fig. 6.3 below (graph drawn using Excel).



**Fig. 6.3** Canadian population according to high (300,000), medium (250,000) and low (200,000) immigration scenarios

Once again, this example shows just how easy it is to use Modgen to create scenarios and generate outputs which can be conveniently exported to external applications for editing.

6.7 Summary

This chapter has put in place the final pieces of the microsimulation model we have been bulding throughout this book. Although the model is still elementary, it is already very useful and can serve as a basis for more complex models. In fact, most national population projections are done with a similar model, although usually of the multi-state type.

## Appendices

### *Appendix 6.1 InternationalMigration.mpp*

Code Sections: Header (lines 1 to 2), Parameters (lines 3 to 9), Actor *Person* and functions (lines 10 to 39)

```

1  classification IMM{ I_NON_IMM, I_IMM };
2
3  parameters
4  {
5      // Sum of weights of recent immigration in microdata file
6      double WeightRecentImmig2011;
7      double ImmigSimul; // Number of immigrants per year
8      double EmigrationHazard[SEX][AGE][PROV]; // Risk of
        emigration
9  };
10
11 actor Person          // Individual
12 {
13
14     IMM imm;
15     double age_imm;
16     int is_imm = { 0 };
17     event timeEmigrationEvent, EmigrationEvent;
        // Emigration event
18     logical emigration = { FALSE }; // Indicate that an actor
        has emigrated
19
20 };
21
22 // Computes waiting time before emigration
23 TIME Person::timeEmigrationEvent()
24 {
25     TIME tTimeEvent = TIME_INFINITE;
26
27     tTimeEvent = WAIT(-TIME(log(RandUniform(7)) /
28         (EmigrationHazard[sex][age_int][prov] +
29         0.0000000001)));
30
31     return tTimeEvent;
32 }
33 // Event function
34 void Person::EmigrationEvent()

```

```
35 {
36     // Terminates actor and signal an emigration event
37     emigration = TRUE;
38     Finish();
39 }
```

## ***Appendix 6.2 ModgenExample.mpp***

```
1 //LABEL(ModgenExample, EN) Core simulation functions
2
3 /* NOTE (ModgenExample, EN)
4     This module contains core simulation functions and
5     definitions.
6 */
7 // The model version number
8 version 1, 0, 0, 0;
9
10 // The model type
11 model_type case_based;
12
13 // The data type used to represent time
14 time_type double;
15
16 // Supported languages
17 languages {
18     EN // English
19 };
20
21 // The CaseSimulation function simulates a single case,
22 // and is called by the Simulation function declared later
23 // in this module.
24
25 void CaseSimulation(const input_csv& input, double StartTime)
26 {
27
28
29     // Gets the number of sub-samples specified in the
30     // parameters
31     int nSubSamples = { 0 };
32     nSubSamples = GetSubSamples();
33
34     // Set case weight
```

```

34     if (StartTime > StartYear) { // Immigrants
35         SetCaseWeight(input[3] * ImmigSimul /
36             WeightRecentImmig2011,
37             input[3] * ImmigSimul / WeightRecentImmig2011 *
38                 nSubSamples);
39     }
40     else { // Everyone else
41         SetCaseWeight(input[3], input[3] * nSubSamples);
42     };
43     // Initializes the first actor in the case. Age, sex,
44     // province,
45     // immigrant status and age at immigration
46     // are passed to the start function
47     Person *poFirstActor = new Person();
48     poFirstActor->Start(StartTime, 0, NULL, input[0],
49         input[1],
50         input[2], input[4], input[5]);
51     // Continue processing events until there are no more.
52     // Model code is responsible for ending the case by
53     // calling
54     // Finish on all existant actors.
55     // The Modgen run-time implements the global
56     // event queue gpoEventQueue.
57     while ( !gpoEventQueue->Empty() )
58     {
59         // The global variables gbCancelled and gbErrors
60         // are maintained by the Modgen run-time.
61         if ( gbCancelled || gbErrors )
62         {
63             // The user cancelled the simulation,
64             // or run-time errors occurred.
65             // Terminate the case immediately.
66             gpoEventQueue->FinishAllActors();
67         }
68         else
69         {
70             // Age all actors to the time of the next event.
71             gpoEventQueue->WaitUntil( gpoEventQueue->
72                 >NextEvent() );
73             // Implement the next event.
74             gpoEventQueue->Implement();

```

```
74     }
75 }
76
77 // Note that Modgen handles memory cleanup when
78 // Finish is called on an actor.
79 }
80
81 // The Simulation function is called by Modgen to simulate
    a set of cases.
82 void Simulation()
83 {
84
85     // Microdata input file
86     input_csv input_file;
87
88     // Reads the file
89     input_file.open("PopBaseBookC6.csv");
90
91
92     // counter for cases simulated
93     long lCase = 0;
94
95     // The Modgen run-time implements CASES (used below),
96     // which supplies the number of cases to simulate in a
        particular thread.
97     //
98     // The following loop for cases is stopped if
99     // - the simulation is cancelled by the user,
100     //     with partial reports (gbInterrupted)
101     // - the simulation is cancelled by the user,
102     //     with no partial reports (gbCancelled)
103     // - a run-time error occurs (gbErrors)
104     //
105     // The global variables gbInterrupted, gbCancelled and
        gbErrors
106     // are maintained by the Modgen run-time.
107     for ( lCase = 0; lCase < CASES() && !gbInterrupted &&
        !gbCancelled
108         && !gbErrors; lCase++ )
109     {
110
111         // exits the loop if end-of-file is reached
112         if (!input_file.read_record(lCase))
113         {
114             break;
115         }
```

```

116
117      /* Imported data
118
119      0 - Age
120      1 - Sex
121      2 - Province
122      3 - Weight
123      4 - Immigrant status
124      5 - Age at immigration
125
126      */
127
128      // Simulates a case.
129
130      // Tells the Modgen run-time to prepare to simulate
131      // a new case.
132      StartCase();
133
134      // Calls the CaseSimulation function defined earlier
135      // in this module.
136      CaseSimulation(input_file, StartYear);
137
138      // Tells the Modgen run-time that the case has been
139      // completed.
140      SignalCase();
141
142      // If immigrant is recent (age-age at immigration <
143      // 5)
144      // the case is repeated for every year of the
145      // simulation
146      if (((input_file[0] - input_file[5])<5) &&
147          input_file[4] == 1)
148      {
149          for (int yeartemp = StartYear; yeartemp <
150              horizon; yeartemp++)
151          {
152              StartCase();
153
154              CaseSimulation(input_file, yeartemp +
155                  RandUniform(5));
156
157              SignalCase();
158          }
159      }

```

```

154
155         }
156
157     }
158
159     input_file.close(); // Closes file
160 }

```

### ***Appendix 6.3 PersonCore.mpp***

Code Sections: Header (lines 1 to 9), Parameters (lines 10 to 19), Actor *Person* and functions (lines 20 to 172)

```

1  classification SEX{ S_FEM, S_MAL };
2
3  range AGE{ 0, 110 };
4
5  partition AGE_GROUP{ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50,
6  55, 60, 65, 70, 75, 80, 85 };
7
8  partition YEAR5{ 2011, 2016, 2021, 2026, 2031, 2036 };
9
10 parameters
11 {
12     //EN Annual hazard of death according to sex and age
13     double MortalityHazard[SEX][AGE][PROV];
14     int AgeMax; // Maximum lifespan
15     double ProbabilityMale; // Probability of male birth
16     int horizon; // End of simulation
17     int StartYear; // Starting calendar year of simulation
18 };
19
20 actor Person //EN Individual
21 {
22
23     //EN Alive
24     logical alive = {TRUE};
25
26     SEX sex; // Sex state variable
27
28     // Age state variable, auto-increment
29     AGE age_int = COERCE(AGE, self_scheduling_int(age));
30
31     // Year state variable, auto-increment

```

```

32     int year_int = self_scheduling_int(time);
33
34
35     event timeMortalityEvent, MortalityEvent;    //EN
    Mortality event
36
37     //LABEL(Person.Start, EN) Starts the actor
38     // The start function now takes six arguments:
39     // 1) dTimeStart is the exact time at which a simulation
        starts
40     // 2) ActorType tells if the simulated actor is part of the
41     //     starting cohort or born in the model
42     // 3) prMother is a pointer to the actor who gave birth
        to the
43     //     new actor
44     // 4-8) State values extracted from parameter file
45
46     void Start(double dTimeStart, int ActorType, Person
        *prMother, double fileage,
47         double filesex, double fileprovince, double
        fileimmstat, double fileageimm);
48
49     //LABEL(Person.Finish, EN) Finishes the actor
50     void Finish();
51 };
52
53 // The time function of MortalityEvent
54 TIME Person::timeMortalityEvent()
55 {
56     TIME tEventTime = TIME_INFINITE;
57
58     // If max age is reached, death event occurs immediately
59     if (age_int == AgeMax | year_int >= horizon)
60     {
61         tEventTime = WAIT(0);
62     }
63     else {
64         // Draw a random waiting time to death from
65         // an exponential distribution based on the
66         // constant hazard MortalityHazard.
67         tEventTime = WAIT(-TIME(log(RandUniform(1)) /
68             MortalityHazard[sex][age_int][prov]));
69     }
70
71     return tEventTime;
72 }

```

```
73
74 // The implement function of MortalityEvent
75 void Person::MortalityEvent()
76 {
77     if (year_int < horizon) { alive = FALSE; };
78
79     // Remove the actor from the simulation.
80     Finish();
81 }
82
83 // The start function now takes 8 arguments:
84 // 1) dTimeStart is the exact time at which a simulation
      starts
85 // 2) ActorType tells if the simulated actor is part of the
86 //     starting cohort or born in the model
87 // 3) prMother is a pointer to the actor who gave birth to
      the
88 //     new actor
89 // 4-8) age, sex, province, immigrant status and age at
      immigration
90 void Person::Start(double dTimeStart, int ActorType,
91     Person *prMother, double fileage, double filesex,
92     double fileprovince, double fileimmstat, double fileageimm)
93 {
94     // Modgen initializes all actor variables
95     // before the code in this function is executed.
96
97
98     // Sets continuous time. time>0
99     // if actor is born during simulation
100     time = dTimeStart;
101
102
103     // ActorType = 0 (actor from the base population)
104     // ActorType = 1 (actor born in the simulation)
105
106
107     if (ActorType == 0) {
108
109         if (dTimeStart == StartYear) {
110             // Continuous age
111             age = fileage;
112         }
113         else {
114             // Continuous age is age at immigration
```

```

115             // if dTimeStart>0
116             age = fileageimm;
117             is_imm = 1;
118         };
119
120         // Sex
121         sex = (SEX)(INT) filesex;
122         // Province
123         prov = (PROV)(INT) fileprovince;
124         imm = (IMM)(INT) fileimmstat; // Immigrant status
125         age_imm = fileageimm; // Age at immigration
126
127
128     }
129
130     else {
131
132         // Continuous age is zero at birth
133         age = 0;
134
135
136         // Sex is randomly attributed according to parameter
137         // ProbabilityMale
138         if (RandUniform(2) < ProbabilityMale)
139         {
140             sex = S_MAL;
141         }
142         else
143         {
144             sex = S_FEM;
145         }
146
147         // Province of residence is the same as the mother's
148         if (prMother != NULL) {
149
150             lMother = prMother;
151             prov = lMother->prov;
152
153             // The actor is not an immigrant
154             imm = I_NON_IMM;
155             age_imm = 0;
156
157         };

```

```

158
159     };
160
161
162 }
163
164 /*NOTE(Person.Finish, EN)
165     The Finish function terminates the simulation of an
        actor.
166 */
167 void Person::Finish()
168 {
169     // After the code in this function is executed,
170     // Modgen removes the actor from tables and from the
        simulation.
171     // Modgen also recuperates any memory used by the
        actor.
172 }

```

### ***Appendix 6.4 Tables.mpp***

```

1  table Person Demography // Population
2  {
3      {
4          // Population size at the end of period
5          value_at_changes(split(year_int, YEAR5), alive),
6          entrances(alive, FALSE),           // Deaths
7          changes(last_age_fertile),         // Births
8          changes(prov),                     // Exits
9          event(changes(prov)),              // Entrances
10         entrances(duration_trigger(is_imm, 1, 0), TRUE),
            // Immigration
11         entrances(emigration, TRUE)        // Emigration
12     }
13 }
14
15     *prov + //Province
16     *sex +
17     *split(age_int, AGE_GROUP) + // Age
18     *split(year_int, YEAR5) // Year
19
20 }

```

# Conclusion

In the course of this book, we have developed a dynamic population microsimulation model in successive stages. We have progressively increased the number of state variables and events associated with actors in a simulation. The added characteristics enabled us to differentiate the risks of undergoing one or another of the events in the model. The final model enabled us to make projections of a national population by age, sex and region of residence, as well as to calculate the components of a classic multi-regional life table.

The first chapter paid particular attention to explaining the structure of a Modgen program and to exploring the interface used to generate scenarios. At this stage, the model generated by the Modgen Wizard was taken as is, and a single event, death, was modelled using a constant risk. In the following chapter we added a *classification* to attribute a sex to the actors, and also a *range* to measure their ages in completed years. These two new characteristics of the *Person* actor, contained in the two state variables (*sex* and *age\_int*) allowed us to specify more accurately the risk of death. We were able to construct an abridged life table and appreciate how easy it is to add new dimensions to the model.

Adding an internal migration event in Chap. 3 demonstrated the other advantages of Modgen for programming microsimulation models. We saw the ease with which a new event module could be added, a useful feature given the advantages of modular programming. We also introduced and explained a new type of parameter (*cumrate*) and a function (*Lookup*) specific to Modgen, which together enable us to change the value of a state variable from a probability distribution matrix. In the mobility example, this matrix was solely a function of the province of origin, but *cumrate* and *Lookup* may be used with an arbitrary number of dimensions.

Another of Modgen's major advantage is its ability to automatically manage competing risks. Unlike discrete-time models, Modgen's continuous-time modelling allows us to add a new event without having to re-estimate the parameters of the

other model events, and without the problem of having to explicitly decide in what order the events are to take place.

In the following chapters we saw how a cohort model could be transformed to become a full projection model allowing for the addition of new actors to the population through fertility and immigration. The fertility module enabled new actors to be created and linked to their mother allowing for the transfer of information between actors using pointers and the Modgen *link* instruction.

Gradually complexifying a simple model is a useful didactical approach for a tutorial course and it has enabled us to show that it is relatively simple to complexify or modify an existing model in Modgen; however this is not the best way to proceed in reality. Microsimulation models should be thought through as much as possible before you start programming. Before generating a new model from the Modgen Wizard, the designer of a microsimulation should have a clear conceptual outlook of the final model and should define each of the state variables and their sub-categories using *classification*, *range*, or other types of variables. He or she should also have a clear idea of the equations which will define the waiting time of the different events in the model, as well as the consequences of each of these events for the whole set of characteristics of the simulated actor.

The full model presented in chapter six could easily be developed further by adding relevant dimensions. Let's take the level of education, for instance, which is a characteristic of analytical interest in itself, as well as an important determinant of the other events we want to model, such as fertility, migration or death. To add the level of education in the model, we would create a module determining the age at which each actor reaches each educational milestone. This would involve a new *classification* describing the educational level of the actor, as well as the corresponding state variables. The modeling of education itself could be done in several ways. We could first determine the highest educational attainment of an actor, and then determine the timing of the event, as well as the timing of intermediate educational achievements. We could also calculate transition rates between the different educational levels (this would be the preferred solution for multistate models). In any case, the result of this modeling would be implemented in the event function and its associated time function. Education could then be added as an additional dimension in other demographic events, such as fertility or mortality.

The model developed in this book is a dynamic case-based microsimulation model in which each actor is simulated separately up to the time of death or the end of the projection. As we saw in the chapter on the fertility module, it is possible to create links between actors of the same case, such as between children and their mother. However, this type of model does not allow for links or interactions between different cases.

Modgen can also be used to create microsimulation models with interactions between the simulated actors. To do this, the model builder has to choose a time-based rather than a case-based model when creating the microsimulation program. The code described in this book can be used without major changes to define a similar corresponding time-based model. In such a model, all the actors are simulated simultaneously, thus enabling interactions between simulated actors. Time-based

models are useful if one wants, for example, to simulate the evolution of a contagious disease in a population, or to create families by explicitly modelling the formation of unions between two actors in the simulation. To do this, Modgen provides tools to create actor sets, so that simulated actors can be grouped and accessed dynamically according to some of their characteristics. To model union formation, for example, we could create actor sets by sex, marital status and other characteristics associated with marriage practices and partner selection – age group, level of education, place of residence, etc. In this way we could create a union-formation event which would match unmarried candidates from the different mate pools according to pre-determined probabilities based on mate preferences. Union types having a generally low probability of formation, for instance, could still have good chances of happening in a marriage market where suitable candidates are rare.

So the Modgen language has a lot to offer and allows for many different types of microsimulation models to be created. In this book we have developed a relatively simple dynamic microsimulation model, but one which is sufficiently general to serve as the basis for a set of models with different objectives. This concluding section has shown the different development paths the user could follow, using the model created here, to meet some specific research needs.

## Appendix: Coding Standards

The coding convention we propose is a set of simple generic guidelines for making a Modgen microsimulation program easier to write and to read. A model builder should follow these guidelines as closely as possible, but they are not hard and fast rules; when you build a model you will naturally adapt them to your particular set of problems while respecting their essential spirit.

- A microsimulation model should be divided as far as possible into thematic or functional modules. Results tables are grouped together in a separate module.
- The order of sections should be the same for each module. In order of appearance, starting from the top, this should be: classifications and ranges; partitions; parameters; the *Person* class and its functions, if possible in their order of appearance in the class.
- As many comments as possible should be added next to the code. Clearly written and detailed comments are of immense value, not just for other users and developers but as reminders for oneself.
- No numbers should ever be inserted into the code; instead, state variables or parameters should be created.
- Give meaningful names to states, state variables, parameters, classifications, ranges and other types of variable. Names of classifications and ranges are written in CAPITALS. Classification modalities are also in upper case letters. They start with the first letter of the classification name, followed by an underline and the name of the modality.
- The first letter of each word in the name of a parameter is capitalised, as in `ParameterNameExample`.
- State variable names are written in lower case letters. This makes it easy to distinguish, for example, the province state variable from the PROVINCE classification.
- Names of links begin with the letter l (for link). So a link to the mother has the form *lMother*.

# Glossary

**Actor** Generally corresponds to an instance of the *Person* object. Analog to an individual in a real population.

**Case** A case is a unit of simulation in Modgen. A case starts by launching the simulation of an actor. Events taking place during the simulation of this case may in turn create other actors within the same case (for example when there is a birth).

**Derived State** A derived state is information generated by Modgen on the state of an actor at a given moment. For example, a derived state can detect a change in state or compile the value of a state variable at a particular moment (when there is a change in the state of another variable, for example).

**Event** An actor's state variables are modified in events. Each event function is paired with its own time function, which determines the exact time at which the event takes place.

**Main file** This is the file which drives a Modgen simulation and usually bears the model name. It contains the *Simulation* function (containing the loop that generates the cases) and the *CaseSimulation* function.

**User interface** This is the graphic interface of a Modgen compiled program.

**Precompilation/Compilation** The action of transforming the model code into an independently executable program (which does not need additional software to be executed). In Modgen, compilation takes place in two phases. In the first, Modgen code is transformed into C++ code by the Modgen compiler: this is pre-compilation. In the second phase, the Visual Studio compiler transforms the C++ code into an executable program. In Modgen 12, the two phases occur simultaneously when the project is compiled from the Visual Studio build menu.

**Waiting time** Waiting time is the duration before the occurrence of an event. Modgen automatically orders and manages waiting time for all the events. It recalculates this time whenever there is a change in a state variable on which waiting times depend. For example if mortality risk varies with age, Modgen recalculates the time before death on each birthday.

# References

- Bélanger, A., & Caron Malenfant, E. (2005). *Population projections of visible minority groups, Canada, provinces and regions, 2001–2017*. s.l. Ottawa: Statistics Canada, Demography Division.
- Bélanger, A., & Larrivée, D. (1992). New approach for constructing Canadian working life tables, 1986–1987. *Statistical Journal of the United Nations Economic Commission for Europe*, 9(1, IOS Press), 27–49.
- Bélanger, A., Martel, L., Berthelot, J.-M., & Wilkins, R. (2002). Gender differences in disability-free life expectancy for selected risk factors and chronic conditions in Canada. *Journal of Women & Aging*, 14(1–2), 61–83.
- Orcutt, G. H. (1957). A new type of socio-economic system. *The Review of Economics and Statistics*, 39(2, JSTOR), 116–123.
- Rogers, A. (1975). *Introduction to multiregional mathematical demography*. New York: Wiley.
- Rogers, A., Rogers, R. G., & Bélanger, A. (1990). Longer life but worse health? Measurement and dynamics. *The Gerontologist*, 30(5, Oxford University Press), 640–649.
- Rowland, D. T. (2003). *Demographic methods and concepts*. Oxford: Oxford University Press, 546 pages.
- Spielauer, M. (2009). *Modgen and the application RiskPaths from the model developer's view*. Ottawa: Statistics Canada.
- Van Imhoff, E., & Post, W. (1998). Microsimulation methods for population projection. *Population: An English Selection*, 10(JSTOR), 97–138.
- Willekens, F. J. (1980). Multistate analysis: Tables of working life. *Environment and Planning A*, 12(5, SAGE), 563–588.
- Willekens, F. J., Shah, I., Shah, J. M., & Ramachandran, P. (1982). Multi-state analysis of marital status life tables: Theory and application. *Population Studies*, 36(1, Taylor & Francis), 129–144.
- Wolf, D. A. (2001). The role of microsimulation in longitudinal data analysis. *Canadian Studies in Population*, 28(2), 313–339.
- Zeng, Yi., Morgan, S. P., Wang, Z., Gu, D., & Yang, C. (2012). A multistate life table analysis of union regimes in the United States: Trends and racial differentials, 1970–2002. *Population research and policy review*, 31(2, Springer), 207–234.

# Index

## A

Actor, 3  
Adding an Event, 33–35

## B

Base.scex, 5  
Birthday event, 33

## C

Calendar time, 100–103, 140–141  
Case, 3  
Case-based model, 4  
CaseSimulation(), 7  
Casting, 139  
Classification, 30–31  
Clock, 101  
Comments, 8  
Competing risks, 40  
Compilation, 16  
Cumrate, 63, 71–74

## D

Decimals, 42  
Derived state, 12, 13, 17, 30, 41–49, 51, 55,  
80–83, 85, 110, 115–117, 119, 166, 167  
Duration, 12, 44, 82

## E

Entrances, 43, 48, 80  
Event, 3

Event function, 10  
Exits, 80

## F

Fertility, 103–109  
Finish, 8, 10

## G

GetSubSamples, 137  
GetTableValue, 85

## I

Importing external data, 133–134  
input\_csv, 135  
Interprovincial migration, 66–71

## L

Life table, 1, 41  
Link, 104, 113  
Lookup, 71–74

## M

MAX, 106  
max\_value\_out, 12  
Microdata file, 132–133  
MIN, 106  
min\_value\_out, 12  
Model wizard, 2–6, 9  
ModgenExample.mpp, 6–8

module, 66–71  
    add, 66  
Modules folder, 5  
Monte-Carlo error, 16

**N**  
NULL, 114

**P**  
Parameters, 9  
    adding dimensions, 37  
    hypothesis, 107  
    modifying, 78  
Parameters file, 39  
    add, 66  
    scenario, 86  
PersonCore.mpp, 8–16  
Pointer, 7  
Precompilation, 16

**R**  
RandUniform, 15, 37  
Range, 32  
RANGE\_POS, 106  
Risk recalculation, 40

**S**  
Scenario, 50, 51  
Scenarios folder, 5  
self\_scheduling\_int, 64–65, 100  
SetCaseWeight, 136  
SetTableValue, 86

Simulation(), 7  
SIZE, 85  
Solution Explorer, 4  
Start, 8, 10, 36, 137  
    arguments, 110  
State, 3  
Sub-samples, 136

**T**  
Table, 11, 41  
    customizing, 83–86  
    decimals, 53  
    derived states, 41, 43–49  
    dimensions, 41  
    properties, 53  
*this* (argument values), 113  
Time function, 10, 11  
TIME\_INFINITE, 11  
Time-based model, 4

**U**  
User interface, 17, 50–54  
user\_table, 83  
UserTables, 84

**V**  
value\_at\_changes, 116, 120  
value\_in, 12, 43  
Visual Studio project, 2

**W**  
WAIT, 11, 15