Max-Planck-Institut für demografische Forschung
Max Planck Institute for Demographic Research
Konrad-Zuse-Strasse 1 · D-18057 Rostock · GERMANY
Tel +49 (0) 3 81 20 81 - 0; Fax +49 (0) 3 81 20 81 - 202;
http://www.demogr.mpg.de

MPIDR WORKING PAPER WP 2006-046
NOVEMBER 2006

The "LifeCourse" Model, a competing risk cohort microsimulation model: source code and basic concepts of the generic microsimulation programming language Modgen

Martin Spielauer (spielauer@demogr.mpg.de)

The “LifeCourse” Model, a competing risk cohort microsimulation model: source code and basic concepts of the generic microsimulation programming language Modgen

Martin Spielauer

November 2006

Abstract

This paper documents the source code of “LifeCourse”, a simple competing risk microsimulation model initially developed alongside a study on fertility decline in Bulgaria and Russia. “LifeCourse” is programmed in the generic microsimulation language Modgen developed at Statistics Canada. In the context of this contribution, the model is introduced step by step as template for other microsimulation applications and as training tool for demographic microsimulation using Modgen.

1. Introduction

The paper documents the source code of a simple cohort microsimulation model “LifeCourse” initially developed for the study of fertility change in Russia and Bulgaria (Spielauer et. al. 2006). The model is programmed in Modgen, a generic microsimulation programming language developed at Statistics Canada[1]. Modgen is known mainly in connection with the large and complex LifePaths and Pohem models (Population Health Model), two Canadian Modgen applications developed at Statistics Canada. The aim of this paper is to introduce Modgen to developers of demographic microsimulation models, based on event-history analysis. The motivation behind the paper is twofold. The first reason is the strength of the Modgen language itself, which unfolds for the development of simple models, too, and makes Modgen an interesting language for learning (and teaching) demographic microsimulation. The second motivation is our belief that the strength of Modgen has been left relatively unexploited by developers of demographic models, and that learning barriers to the underlying microsimulation technology and approach exist, which we aim to reduce with this contribution.

Technically, the Modgen language is a superset of the C++ programming language, providing an efficient environment for the programming of microsimulation models. The functionality covers the handling of model parameters, table definition routines for the presentation of model output,

[1] For a list of links to Statistics Canada online documents concerning the Modgen language and the microsimulation models LifePaths and POHEM please consult the References.

a generic user interface for the microsimulation application, and a broad variety of routines, simplifying microsimulation programming.

Modgen supports both continuous and discrete time models. Continuous time is usually associated with statistical models of durations to an event, following a competing risk approach. Beginning at a fixed starting point, a random process generates the durations to all considered events with the event occurring next to the starting point being executed and all others censured. The whole procedure is repeated at the new starting point until the event 'death' of the simulated actor occurs. Continuous models are technically very convenient, as they allow to add new processes without changing the models of the existing processes as long as the statistical requirements for competing risk models are met (see Galler 1997 for a description of associated problems). The "LifeCourse" model developed in this contribution is a continuous time cohort model, i.e. all actors are born at the same point in time and no new actors are created at birth events. "LifeCourse" is a female-only model and does not include any interactions between the simulated actors.

In this paper, we use Modgen Version 7.31.0.0, developer's edition[2]. Modgen requires Microsoft Visual Studio .NET 2003, which has to be configured for Modgen applications[3]. The applications are programmed in the Modgen language; the Modgen code is then converted into the C++ code by a Modgen tool integrated into the Visual Studio environment before the application is compiled and linked as a C++ application.

Modgen is freely available and well documented by a Developer's Guide. This contribution does not aim to replace existing documentation (which we frequently reference) but should be see as a complement providing a code template for continuous time microsimulation in addition to the discrete time model "Simpex" used to introduce Modgen concepts in the Developers Guide. We do not intend to introduce the full wealth of the Modgen language as Modgen supports a much broader range of modeling approaches than covered by the "LifeCourse" Model.

2. The illustrative "LifeCourse" Model

2.1. Overview

In the following, we describe the Modgen code of "LifeCourse", an illustrative cohort microsimulation model. The model uses a continuous time framework. It simulates a female cohort from birth to their 40th birthday and was initially developed for the study of fertility change in Bulgaria and Russia (Spielauer et. al. 2006). Except for mortality, all behavioral models are based on simple piecewise constant hazard regression models. The models and

[2] Modgen can be downloaded free of charge at http://www.statcan.ca/english/spsd/Modgen.htm.

[3] The installation procedure is described in the Modgen Developer's Guide. (p.: 54ff/141ff) Modgen has to be installed in order to run a Modgen application; for users of existing models, Statistics Canada provides a user edition of Modgen, which can be downloaded at http://www.statcan.ca/english/spsd/Modgen.htm.

parameters estimated from the Bulgarian 2004 Generations and Gender Survey data are presented in the Appendix. We distinguish between the following processes[4]:

- Mortality: parameterized by age-specific death probabilities converted to piecewise constant mortality risks. We allow mortality to be “switched off” until the 40th birthday.
- First and second birth: separate models, with processes starting at age 15 and at first birth respectively. The models control for age and partnership status; for second births we also control for union status at first birth. As we only study the process to second births, there are no higher order births[5].
- First and second union: separate models for first and second union formation and dissolution. The processes start at age 15 and at the last union formation/dissolution event respectively. They control for age and (time spent in) motherhood. Union(s) following the second union status are not modeled, i.e. women continue to have the status “after second union”.

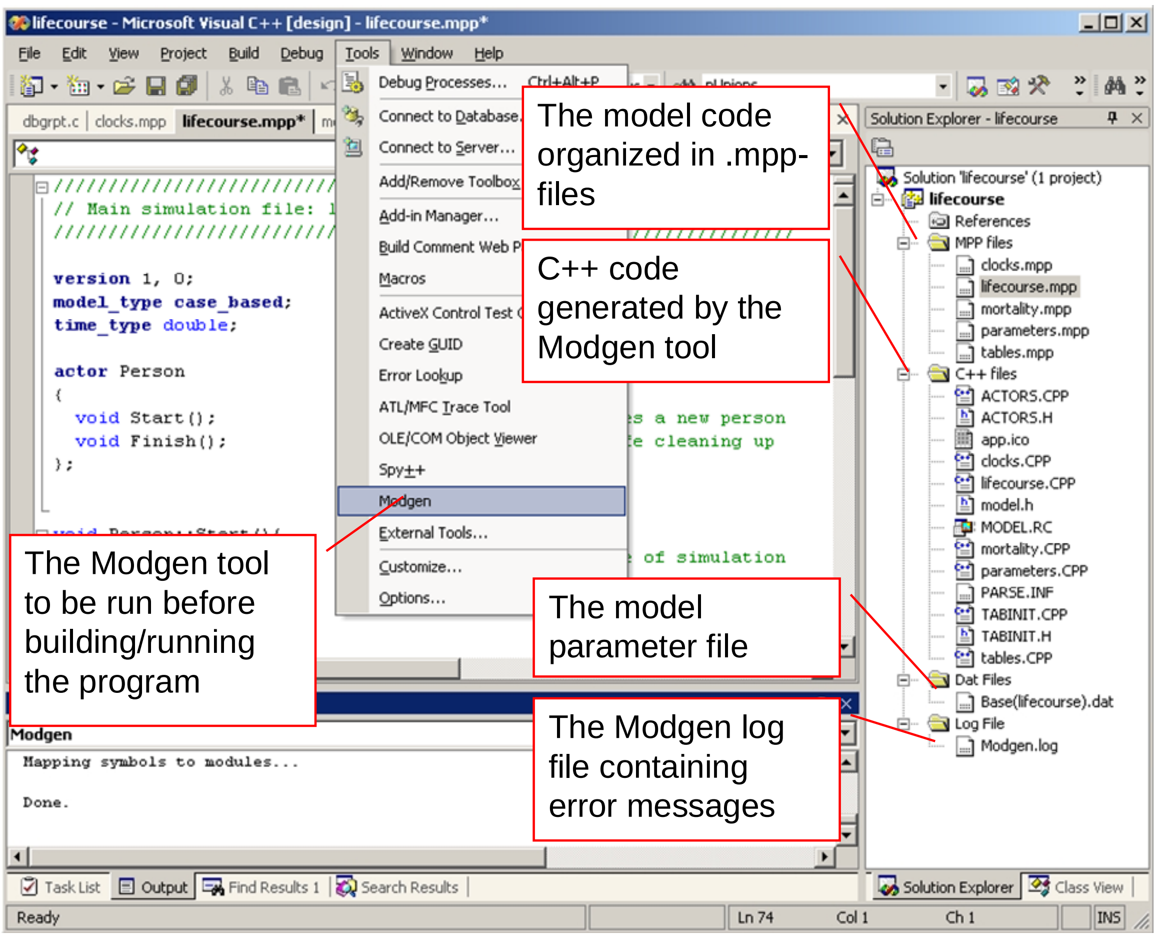
The code development is organized in three steps. We start from a code template which only includes a clock for birthday events and mortality. The resulting simple model is used to introduce basic Modgen concepts common to all models: handling of model parameters, actors, events, table output, and the graphical representation of individual life courses using the BioBrowser tool. In the following steps, we add the behavioral modules for fertility and union formation and dissolution and additional table output, thereby aiming to introduce most Modgen programming aspects relevant for the development of this type of microsimulation models.

2.2. Starting from a simple model template (Step 1)

In this section, we develop and describe a template for (continuous time, case based) cohort simulation models. We recommend always starting one’s own model development from an existing template as it contains the necessary code common to all models of a given type and it implicitly contains all project settings necessary to successfully build Modgen applications within the Visual Studio .NET environment. The screenshot displayed in Figure 1 was taken after opening the Visual Studio Solution ‘lifecourse.sln’ contained in the Step_1 directory.

4 The reduction of the model to first and second births and partnerships is a simplification which is justified in the Bulgarian context, where the first two births account for 97% of all births reported in the population under study, i.e. women of Bulgarian ethnicity born after 1950. Furthermore, all first and second births in the sample occurred before a third partnership.

5 When modeling births, we generally refer to the time of conception (assumed to be nine months prior to the reported births). This allows us to distinguish the period of pregnancy in models for union formation and dissolution.



```
////////////////////////
// Main simulation file: l
////////////////////////

version 1, 0;
model_type case_based;
time_type double;

actor Person
{
   void Start();
   void Finish();
};
```

```
s a new person
e cleaning up
```

```
 of simulation
```

Modgen

```
Mapping symbols to modules...

Done.
```

Figure 1: Modgen integrated into the C++.NET programming environment

2.2.1. Organization of files

The Modgen code is organized into one or (usually) various files. These have the file extension .mpp. As can be seen in the Solution Explorer window (Figure 1), our code template consists of five .mpp files. We grouped them in the “MPP files” folder. When invoking the Modgen tool, which can be accessed from the “Tools” menu after proper installation of Modgen, these files are translated into the C++ code; the Modgen tool creates one .cpp source code file for each .mpp file plus various additional C++ files[6]. We placed the files in the “C++ files” folder.

The model parameters are organized into one or more .dat files. Modgen applications automatically contain a user interface to view and change all model parameters. The parameter files belong to scenarios. The filenames of the parameter files consist of the scenario name plus the parameter file name – in parenthesis –, e.g. Base(lifecourse).dat. The Modgen tool produces a log file “Modgen.log”, which contains error messages. In order to provide fast access to the .dat and the .log files within the development environment, we added them to the LifeCourse solution so that they can be opened by clicking on them in the Solution Explorer window.

[6] ACTORS.CPP, ACTORS.H, app.ico, model.h, model.RC, PARSE.INF, TABINIT.CPP, TABINIT.H. If not starting from a template, these files have to be added to the Visual Studio Project after being created by ModGen.

The model developer has freedom to decide how to organize the Modgen code in different files. For our purpose, we have chosen the following organization:

- The main simulation file: this file has the name of the application, i.e. lifecourse.mpp (such a file is obligatory) and contains the definitions of the model type (e.g.: case based continuous time[7]) and the code of the simulation "engine", which is mostly not model specific within the defined type of models.
- One file for each group of events (behaviors) distinguished in the model. The template contains only two types of events: mortality (and the corresponding file mortality.mpp) and clock events (clock.mpp). Clock events are events whose timing is not determined by the Monte Carlo simulation i.e. their timing is predetermined. The only such event contained in the template model is a birthday event altering the (integer) age of the modeled actor. Other clock events added later are events that change the spell index of the time-varying covariates of the model.
- The parameter definition file (parameters.mpp): This file contains all definitions of parameters, classifications, partitions, and ranges used in the model (see below for explication). For more complex models that have many different behaviors, one may want to organize parameter definitions in a different manner, e.g. by including them in the files of the corresponding behaviors.
- The table definition file (tables.mpp): This file contains the definition of output tables.
- The actual model parameters are contained in one .dat file Base(lifecourse).dat. Again, for more complex models it is advisable to split this file into one file for each group of modeled behaviors.

2.2.2. The lifecourse.mpp main simulation file

This file contains the code essential for the definition of and for running case-based continuous time Modgen applications. The developers of these model types will need to modify only very few parts of the code, which is mostly model-independent.

At the core of any Modgen Simulation are the actors, whose life is simulated. In our application (as in most demographic applications) we have only one type of actors: persons. The characteristics and events changing the characteristics of actors are typically defined in other files that correspond to the various modeled behaviors.

In our application, each person constitutes a case, i.e. the whole life of each person is simulated before the simulation of the next person starts. In more complex demographic applications, birth events would result in the creation of new actors, which then would belong to the parents' case. The number of simulated cases is part of the scenario definition and not set in the model code, i.e. it can be set by the model user.

[7] The second time option is discrete time; alternative model options are time based or cell based. Currently, no Modgen documentation is available for cell- and time based models.

Modgen automatically creates two variables of the specified time_type: time and age. These variables are per default initialized to 0 and handled automatically in the simulation. The initial value can be modified only in the Start() method of an actor. As we want to simulate a birth cohort born in the year 2000, we added the code time=2000 to the Start() method. The Finish()method is applied at the end of life of an actor. Actions to be performed before the actor is destroyed can be added to this method, something that is useful e.g. to generate additional file output.

Before the simulation starts, the Presimulation() method is applied. The method contains a SetMaxTime() command, which is used by Modgen to determine an adequate rounding precision in the handling of continuous time. Omitting this command may result in anomalies at model execution time.

```
///////////////////////////////////////////////////////////////////////////////////////////////
// Main simulation engine: lifecourse.mpp (introduced at STEP 1)                             //
///////////////////////////////////////////////////////////////////////////////////////////////

version 1, 0;                                            // Model version
model_type case_based;                                   // Model type
time_type double;                                        // Continuous time model

actor Person                                             // Individual
{
  void Start();                                          // Initializes a new person
  void Finish();                                         // Destroys the person
};

void Person::Start(){time=2000;}                         // Start time of simulation
void Person::Finish(){}
void PreSimulation(){SetMaxTime(2040);}                  // Maximum value of time variable

void CaseSimulation(){
  Person *prPerson;
  prPerson = new Person();
  prPerson->Start();
  while (!gpoEventQueue->Empty()){
    if (gbCancelled || gbErrors) {gpoEventQueue->FinishAllActors();}
    else {
      gpoEventQueue->WaitUntil(gpoEventQueue->NextEvent());
      gpoEventQueue->Implement();
    }
  }
  DeleteAllPersonActors();
}

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

2.2.3. Adding a clock: clock.mpp

The clock.mpp file defines a birthday event that increases the integer variable current_age. The current_age variable is a characteristic of the actor Person and defined within an "actor Person { … }; code block. It is initialized with 0. In this block, we also find the declaration of the birthday event methods introduced by the Modgen statement event. Each event has two methods, the first determines the timing of the event, the second determines the consequences of the event. For any given point in time, the timeBirthdayEvent method returns the time of the next birthday. To determine the event-time, we make use of the Modgen function WAIT, which adds a specified duration (the age at the next birthday minus the current age) to the current point in time.

When defining clock events, we need to consider the nature of competing risk models, i.e. that the first occurring event censors all other events. This effect is not wanted for clock events, where we allow more than one clock to ring at the same point in time. To assure that no clock event is censured, each clock has to be able to determine whether or not it is time to ring "right now" and – in order to avoid loops - to record the last clock event. Accordingly, we track the time "lastbirthday" in the birthday clock. As the actual day of birth at age 0 is not considered a first birthday, we increase current_age only if the age is above 0. (Alternatively, current_age may be initialized with -1).

```
////////////////////////////////////////////////////////////////////////////////////////////////
// Clock events file: clocks.mpp - STEP 1                                                    //
////////////////////////////////////////////////////////////////////////////////////////////////

actor Person                                          // Individual
{
  int current_age = {0};                              // Current age
  TIME lastbirthday = TIME_INFINITE;                  // Time of last birthday event;
  event timeBirthdayEvent, BirthdayEvent;             // Birthday event
};

////////////////////////////////////////////////////////////////////////////////////////////////
// Birthday event

TIME Person::timeBirthdayEvent(){
  if ((lastbirthday != time) && (age == (int)age) && (age>0)) return time;
  else return WAIT((int)age+1-age);
}

void Person::BirthdayEvent(){current_age++; lastbirthday=time;}
```

2.2.4. Model parameters: parameters.mpp and Base(lifecourse).dat

Our simple model template has only two parameters, the logical parameter "canDie" – a switch to turn on/off mortality before the 40th birthday – and age-specific death probabilities. The parameters are declared within a "parameters {…};" code block. Modgen supports the numeric types int, long, float, and double, Boolean variables ("logical" in the Modgen terminology) and a set of Modgen specific types not used in our application. (see the Developer's Guide p. 24ff).

The dimensionality of the parameters is – as with state variables -defined by classifications, ranges and partitions:

- Classifications define all possible levels of a variable. In our application, we have defined a classification LIFE_STATE with the possible levels ALIVE and NOT_ALIVE (a state variable live_status of the type LIFE_STATE will be defined in the mortality.mpp file).
- Ranges are used to define the size of an array state variable or model parameter. In our application, we have defined a LIFE range with possible values from 0 to 40. Our mortality parameter is accordingly defined as "double ProbMort[LIFE];".
- Partitions define the breakpoints for continuous state variables and are also used for the definition of table output e.g. for different calendar time periods. In our application, we have defined a partition YEARS in order to allow the output of period tables (see below).

One of the powerful features of Modgen is the recognition of comments placed in the programming code as labels used in the user-interface of the Modgen application and in the BioBrowser tool. For instance, the comment "// Age-specific death probabilities" will be used as a label of the corresponding parameter list entry and parameter table. (See Figure 2). The parameters can be grouped for visual selection in the Group View tab of the application. We defined a parameter_group P01_MORTALITY containing our two parameters[8]. Again, the comment "// Mortality parameters" is used as a label in the application. (See Figure 2).

```
////////////////////////////////////////////////////////////////////////////////////////////////
// Parameter definitions: parameters.mpp - STEP 1                                            //
////////////////////////////////////////////////////////////////////////////////////////////////

classification LIFE_STATE {                 // Life status
  ALIVE,                                    // Alive
  NOT_ALIVE                                 // Dead
};

range LIFE {0,40};                          // Simulated age range

parameters {
  logical canDie;                           // Swith to turn mortality before age 40 on/off
  double ProbMort[LIFE];                    // Age-specific death probabilities
};

parameter_group P01_MORTALITY {             // a.) Mortality parameters
      canDie, ProbMort
};

partition YEARS {                           // Single Calendar Years
  2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015,
  2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027, 2028, 2029, 2030,
  2031, 2032, 2033, 2034, 2035, 2036, 2037, 2038, 2039, 2040
};
```

[8] Parameter groups are listed alphabetically in the Modgen application. To control their order, we use the name prefixes P01_, P02_ etc.

The actual parameters are stored in a .dat file, which is loaded at runtime. The syntax is similar as in the .mpp file, except that actual values are assigned to the parameters. In our case, we initialize canDie with the logical value TRUE and ProbMort by a comma-separated list of death probabilities. Modgen recognizes repeaters placed in parenthesis: "(40) .01," is equivalent to 40 times ".01,". Note that Modgen applications automatically include a visual user-interface that allows manipulating model parameters in concise tables. For large and multi-dimensional tables, the actual model parameterization can be simplified and is less error-prone when using the visual interface at runtime; in our case, we initialize mortality with a 1% death probability for each of the first 40 years and then leave it to the user to paste in more realistic values. Comments in .dat files are comments only and not used as labels, i.e. the user can change values but not labels[9].

```
////////////////////////////////////////////////////////////////////////////////////////////
// Parameter values file: Baseline(lifecourse).dat - STEP 1                              //
////////////////////////////////////////////////////////////////////////////////////////////

parameters {
  logical canDie = FALSE; // Swith to turn mortality before age 40 on/off
  double ProbMort[LIFE] = {(40) .01, 1, };
};
```

2.2.5. Mortality: mortality.mpp

This file defines the mortality event that ends the life of the simulated agent. We declare a state variable of type LIFE_STATE, which is initialized with ALIVE at birth and set to NOT_ALIVE by the death event. The death event also calls the function finish(), the latter which deletes the actor.

The timeDeath method returns a death date. If canDie is set to FALSE, the date of death is always the 40th birthday, otherwise the method determines by the Monte Carlo simulation if and when an actor dies before the next birthday event. If the actor survives, the method returns TIME_INFINITE. Note that the event of birthday automatically censors the process if the time of the event is after the next birthday (as is the case with TIME_INFINITE, a Modgen constant for the very distant future.)

In order to obtain random durations from probabilities, we assume constant mortality hazards within each period (i.e. between birthdays) with the exception of a death probability of 1, which leads to death immediately at the start of the age year. For probabilities less than 1, a random duration can be obtained from a uniform distributed random number by the transformation:

```
randdur=-log(RandUniform(1))/-log(1-ProbMort[(int)age]).
```

The Modgen function RandUniform() returns a uniform distributed random number between 0-1. The function takes an integer argument used to assign a different independent random number

[9] Note that comments placed in .dat files are lost when saving scenarios within the Modgen application; Modgen applications automatically write back the labels defined in .mpp files as comments in the .dat files.

stream to each random number function in the code. When omitted, Modgen automatically writes back a unique index into the .mpp file before translation into C++ code.

```
////////////////////////////////////////////////////////////////////////////////////////////
// Mortality: mortality.mpp - (introduced at STEP 1)                                     //
////////////////////////////////////////////////////////////////////////////////////////////

actor Person                                                // Individual
{
  LIFE_STATE life_status = {ALIVE};                         // Life Status
  event timeDeath, Death;                                   // Death Event
};

TIME Person::timeDeath(){
      TIME event_time = TIME_INFINITE;
      double randdur;
      if (canDie==FALSE){event_time = WAIT((MAX(LIFE)-age));}
      else if ((ProbMort[(int)age]==1) || (age >= MAX( LIFE ))) {event_time = time;}
      else if (ProbMort[(int)age]>0){
        randdur=-log(RandUniform(3))/-log(1-ProbMort[(int)age]);
        if (randdur<((int)age+1-age)) {event_time =WAIT(randdur);}
      }
    return event_time;
}

void Person::Death(){life_status = NOT_ALIVE; Finish();}
```

2.2.6. Defining table output: tables.mpp

Modgen provides a very powerful and flexible cross-tabulation facility to report model results, the description of which far exceeds the scope of this paper (model developers consult the Developer’s Guide p: 34ff). Generally, table definitions start with the “table” keyword and describe the content of table cells and table dimensions. Our template model produces two simple table outputs.

The first contains summary values of our simulation and has no dimensions, i.e. cells apply to the whole population over the whole simulation period. We make use of the Modgen keyword “unit”, which counts the number of actors entering the cell of a table (in our case, the simulation), and the Modgen function duration() which sums up the time actors stay in this cell (in our case, the total years lived by all actors in the simulation). The average age at death of all actors in the simulation is then obtained by dividing duration() by unit. As for parameter definitions, comments placed into the code are used as labels in the application.

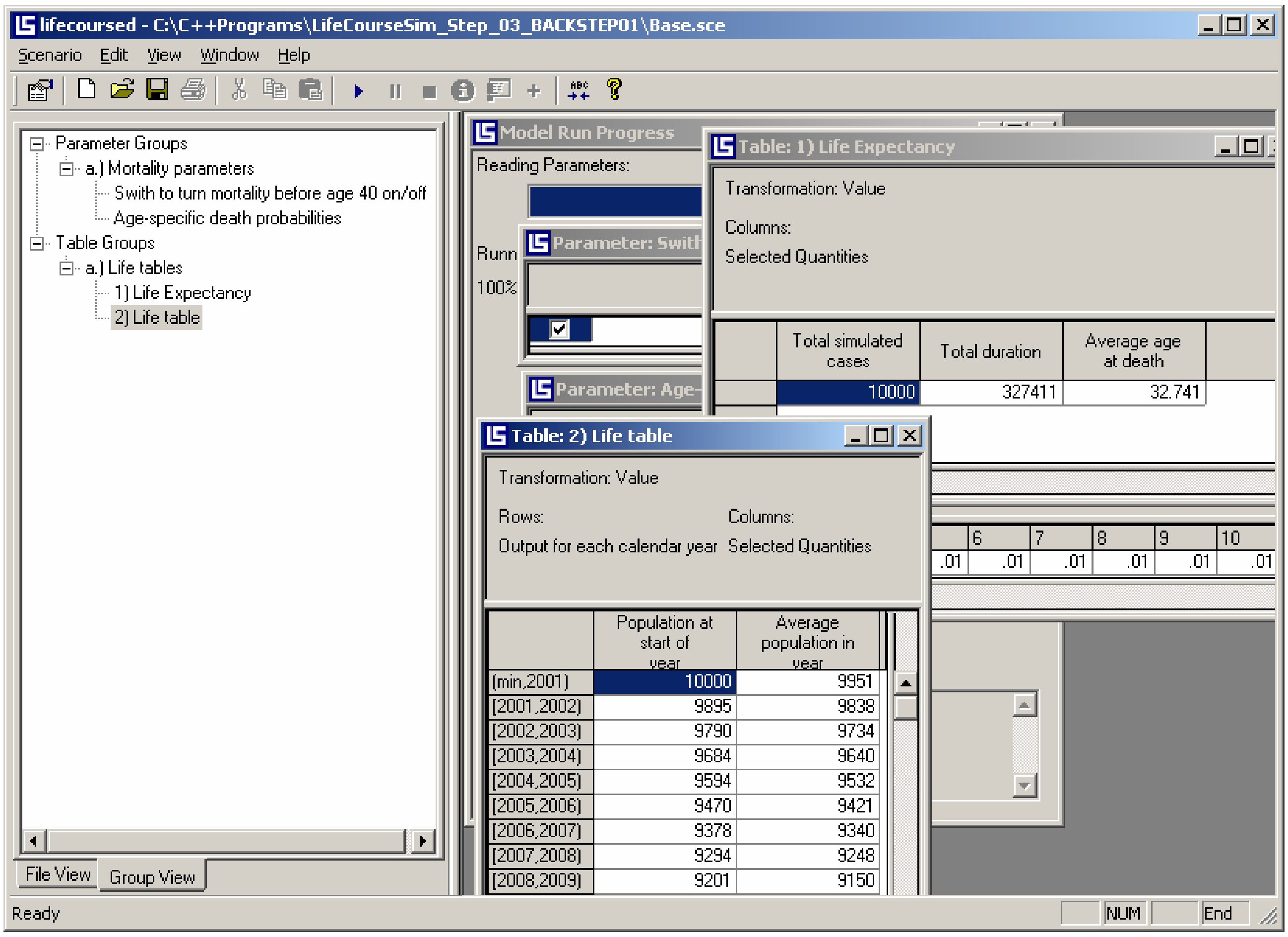
The second table is a period table. Here, we use the partition YEARS defined in parameters.mpp to split up time into calendar years. This is done by the Modgen “split” command. The “unit” keyword now counts the entrances of actors into calendar year cells, i.e. corresponds to actors alive at the beginning of a calendar year. Accordingly, the duration() function returns the total years lived of all actors by calendar year, which is equivalent to the average population alive in a calendar year.

As with the parameters, tables can be grouped for easier navigation. In our template model, we define a table_group TG01_LIFE_TABLES containing both tables.

```
////////////////////////////////////////////////////////////////////////////////////////////////
// Output tables: tables.mpp - STEP 1                                                        //
////////////////////////////////////////////////////////////////////////////////////////////////

table Person T01_LifeExpectancy                         // 1) Life Expectancy
{
  {
      unit,                                             // Total simulated cases
    duration(),                                         // Total duration
      duration()/unit                                   // Average age at death decimals=3
  }
};

table Person T02_TotalPopulationByYear                  // 2) Life table
{
  split(time,YEARS)                                     // Output for each calendar year
  *
  {
    unit,                                               // Population at start of year
    duration()                                          // Average population in year
  }
};

table_group TG01_LIFE_TABLES {                          // a.) Life tables
  T01_LifeExpectancy, T02_TotalPopulationByYear
};
```

2.2.7. Running the model template

After having successful run the Modgen tool for code conversion and building the C++ program, we can test our template application. The visual interface of all Modgen applications consists of two parts: a selection window containing a list of all model parameters and output tables, and a frame in which all corresponding tables can be displayed. Before running a model, we have to open a scenario by loading a scenario file. The base scenario file provided, Base.sce, contains the initial scenario settings (e.g. the number of simulated cases) and a reference to the Base(lifecourse).dat parameters file used. The scenario settings can be changed using the Senario/Settings command. By saving a scenario under a new name, a copy of all .dat files with the new scenario name (e.g. NewScenario(lifecourse).dat) is created. After running the scenario, all output tables are updated. Note that the values displayed in the output table represent only one of several possible views on results; by left-clicking a table, a properties sheet for the table can be accessed. Among other things, it allows to display distributional information (standard errors and the coefficient of variation) of all simulated values (for a in-depth description of the generic Modgen user-interface, see the Modgen User’s Guide).



lifecoursed - C:\C++Programs\LifeCourseSim_Step_03_BACKSTEP01\Base.sce

- Parameter Groups
  - a.) Mortality parameters
    - Swith to turn mortality before age 40 on/off
    - Age-specific death probabilities
- Table Groups
  - a.) Life tables
    - 1) Life Expectancy
    - 2) Life table

Table: 1) Life Expectancy

Transformation: Value

Columns:
Selected Quantities

	Total simulated cases	Total duration	Average age at death
	10000	327411	32.741

Table: 2) Life table

Transformation: Value

Rows: Output for each calendar year

Columns: Selected Quantities

	Population at start of year	Average population in year
(min,2001)	10000	9951
[2001,2002)	9895	9838
[2002,2003)	9790	9734
[2003,2004)	9684	9640
[2004,2005)	9594	9532
[2005,2006)	9470	9421
[2006,2007)	9378	9340
[2007,2008)	9294	9248
[2008,2009)	9201	9150

6	7	8	9	10
.01	.01	.01	.01	.01	.01

Figure 2: The user-interface of Modgen applications

2.2.8. Displaying individual life-courses by the BioBrowser program

The Modgen Biography Browser "BioBrowser" is a tool for the graphical display of individual life courses. This view on simulation results is especially useful for model debugging. In order to use the tool, the tracking feature has to be switched on in the scenario settings and the list of variables to be tracked has to be declared within a "track actorname {};" code block in an .mpp file, the tables.mpp file being a good location:

```
track Person {current_age, life_status};
```

Modgen tracks all changes of state variables included in the tracking statement for a sample of simulated actors. In our case, we traced both state variables, namely the current age and the life status. Figure 3 displays the life course for a person actor who lives 40 years (see the BioBrowser User's Manual for details)

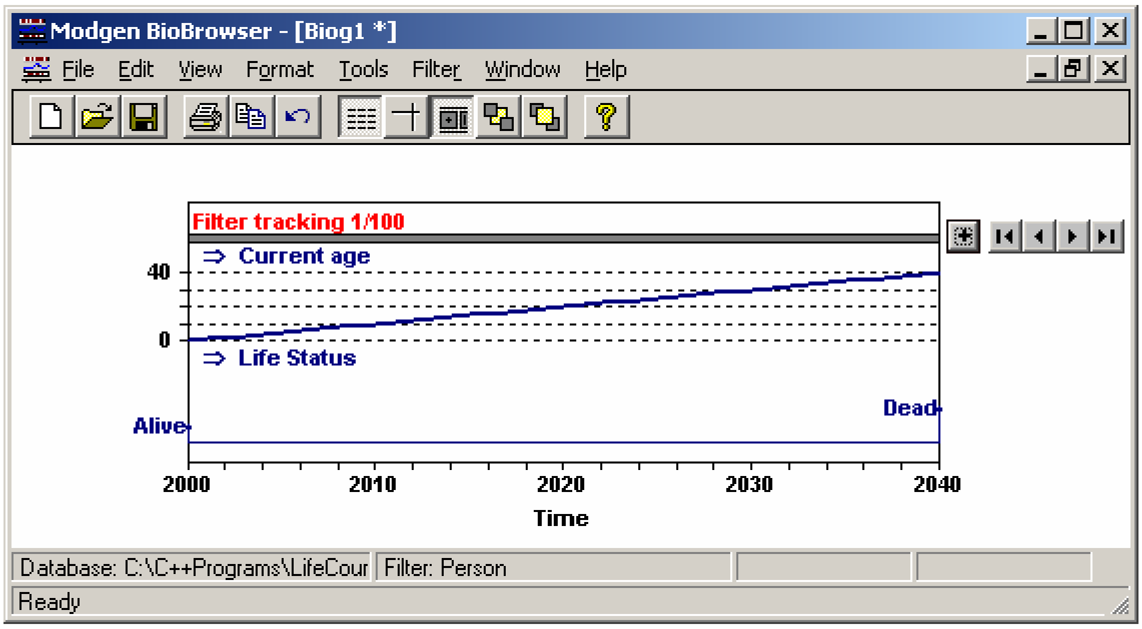


Figure 3: The graphical BioBrowser output of individual life courses

2.3. Extending the model: adding first births (Step 2)

In this, the second step we extend the template by adding the first birth process. First births are modeled by an age baseline hazard for different age groups and relative risks controlling for union status. As we have not included union status yet, for the present we assume a union status of "never having been in union" throughout life.

2.3.1. Adding new parameters

The key parameter concerning first birth is the age-specific baseline hazard of the birth event First_age[AGEINDEX]. Our model distinguishes 11 age periods (0-15, 15-17.5, 17.5-20… 37.5-40); we therefore define a range AGEINDEX {0,10} of possible indexes. We allow the model user to modify these age cutting points, which we therefore also define as model parameters: ageSchedule[AGEINDEX]. The third parameter, First_unionStatus[UNION_STATE], controls for union status. The index UNION_STATE is a classification defining possible levels of union status: never in union, first union </> 3 years, after first union, in second union, and after second union. Other classifications added are PARITY_STATE and FIRST_STATE. The levels of PARITY_STATE denote childlessness, one child and two children; the levels of FIRST_STATE represent union statuses at first birth: first birth in first union, in second union, out of union. The variables of the three classification types introduced are defined and initialized in fertility.mpp; the parity_status is updated at birth event, the first_status at first birth.

```
///////////////////////////////////////////////////////////////////////////////////////////
// Parameter definitions: parameters.mpp - ADDITIONS STEP 2                              //
///////////////////////////////////////////////////////////////////////////////////////////

classification PARITY_STATE {             // Parity status
  CHILDLESS,                              // Childless
  ONE_CHILD,                              // One child
  TWO_CHILDREN                            // Two children
};

classification UNION_STATE {              // Union status
  NEVER_IN_UNION,                         // Never in union
  FIRST_UNION_PERIOD1,                    // First union < 3 years
  FIRST_UNION_PERIOD2,                    // First Union > 3 years
  AFTER_FIRST_UNION,                      // After first union
  SECOND_UNION,                           // Second union
  AFTER_SECOND_UNION                      // After second union
};

classification FIRST_STATE {              // Union status at first birth
  NO_FIRST,                               // No first birth
  FIRST_IN_FIRST_UNION,                   // First birth in first union
  FIRST_IN_SECOND_UNION,                  // First birth in second union
  FIRST_NOT_IN_UNION                      // First birth out of union
};

range AGEINDEX {0,10};                    // Range of age groups indexes

parameters {
  float  ageSchedule[AGEINDEX];           // Lower borders of age groups
  double First_age[AGEINDEX];             // Age baseline hazards first birth
  double First_unionStatus[UNION_STATE];  // Relative risks of union status on first birth
};

partition AGE_GROUP {                     // Age
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
  26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40
};

parameter_group P02_FIRST_BIRTH {         // b.) First birth parameters
      First_age, First_unionStatus
};
```

```
///////////////////////////////////////////////////////////////////////////////////////////
// Parameter values file: Baseline(lifecourse).dat - ADDITIONS STEP 2                    //
///////////////////////////////////////////////////////////////////////////////////////////

parameters {
  float ageSchedule[AGEINDEX] = {
    0, 15, 17.5, 20, 22.5, 25, 27.5, 30, 32.5, 35, 37.5,
  };
  double First_age[AGEINDEX] = {
    0, 0.2869, 0.7591, 0.8458, 0.8167, 0.6727, 0.5105, 0.4882, 0.2562, 0.2597, 0.1542,
  };
  double First_unionStatus[UNION_STATE] = {0.0648, 1.0000, 0.2523, 0.0648, 0.8048, 0.0648,  };
};
```

2.3.2. Adding a second clock

The second clock is used to set the correct index (agespell) of the age baseline hazard. In difference to the birthday clock, which clicks every completed year of life, the second clock needs to look up the parameter ageSchedule to determine the next cutting point in time.

```
////////////////////////////////////////////////////////////////////////////////
// Clock events file: clocks.mpp - ADDITIONS STEP 2                          //
////////////////////////////////////////////////////////////////////////////////

actor Person                                        // Individual
{
  int agespell = {0};                               // Current age group index
  TIME lastagespellchange = TIME_INFINITE;          // Time of last age group index change
  event timeNextAgeSpellEvent, NextAgeSpellEvent;   // Event setting index of age group
};

////////////////////////////////////////////////////////////////////////////////
// Event changing index of age group

TIME Person::timeNextAgeSpellEvent(){
  TIME event_time = TIME_INFINITE;
  int i;
  for (i=MAX(AGEINDEX);i>0;i--){
      if (age < ageSchedule[i]){event_time = WAIT(ageSchedule[i]-age);}
      else if ((lastagespellchange != time) && (age == ageSchedule[i])){event_time = time;}
  }
  return event_time;
}

void Person::NextAgeSpellEvent(){agespell++; lastagespellchange=time;}
```

2.3.3. The event of first birth: fertility.mpp

In order to keep the code for birth events in a separate file, we create a new file, fertility.mpp, as part of the project. This can be done easily by adding a new text-file item of this name to the list of .mpp files via the context menu of the “MPP-files” folder of the solution explorer. Alternatively, the file can be created outside the NET 03 environment and added to the project by choosing “Add existing item…” from the context menu of the “MPP-files” folder. After running the Modgen tool, that creates the corresponding fertility.CPP file, the fertility.CPP file has to be added as an “existing item” to the project (i.e. the C++files list).

The programming of the first birth event is straightforward: we declare a state variable parity_status and initialize it with the level CHILDLESS, which is changed to ONE_CHILD by birth event. If a person is at risk of first birth (i.e. childless) and the hazard rate of the current age interval is greater than 0, a random duration to the next birth event is generated. Note that this event will only occur if not censured by a clock event, ensuring the use of an appropriate age spell index of the piecewise constant hazard rates.

Besides the parity status, other characteristics of the Person actor are updated at the first birth event: a parity variable that counts births (used later for table output), and the variable first_status, which records the union status at first birth (used later as control variable for second births). As union behaviors are not modeled yet, we define a variable union_status and initialize it with NEVER_IN_UNION;

```
///////////////////////////////////////////////////////////////////////////////////////
// Fertility: fertility.mpp - STEP 2  (file introduced at STEP 2)                    //
///////////////////////////////////////////////////////////////////////////////////////

actor Person                                                   // Individual
{
  UNION_STATE union_status = {NEVER_IN_UNION};                 // Union status
  FIRST_STATE first_status = {NO_FIRST};                       // Union status at first birth
  PARITY_STATE parity_status = {CHILDLESS};                    // Parity status
  int parity = {0};                                            // Parity
  TIME lastbirth = TIME_INFINITE;                              // Time of last birth
  event timeFirstBirth, FirstBirth;                            // First birth event
};

///////////////////////////////////////////////////////////////////////////////////////
// First birth

TIME Person::timeFirstBirth(){
  TIME event_time = TIME_INFINITE;
  double randdur, hazard;
  if (parity_status == CHILDLESS) {
    hazard=First_age[agespell] * First_unionStatus[union_status];
    if (hazard>0){
        randdur = -log(RandUniform(1))/hazard;
        event_time = WAIT(randdur);
      }
  }
  return event_time;
}

void Person::FirstBirth(){
  parity++;
  lastbirth=time;
  parity_status=ONE_CHILD;
  if (union_status==FIRST_UNION_PERIOD1 || union_status==FIRST_UNION_PERIOD2)
      first_status=FIRST_IN_FIRST_UNION;
  else if (union_status==SECOND_UNION) first_status=FIRST_IN_SECOND_UNION;
  else first_status=FIRST_NOT_IN_UNION;
}
```

2.3.4. Adding table output

We add two tables in order to report fertility. The first displays the simulated first birth and second birth[10] rates as well as overall fertility by age. It makes use of the Modgen function "entrances", which counts the number of actors entering a specified level of a specified state variable. The function value_out returns the value of a variable when leaving the cell; this expression is used to calculate cumulative rates.

The second table reports cohort measures, namely the average age at first and second birth and the percentage of women either staying childless or of parity 1 or 2. The table definition makes use of the Modgen "value_at_transition" function, which, in our case, returns the age at parity transitions (for a full list of Modgen "derived state expressions" see the Developer's Guide p: 19ff).

[10] As second births are not modeled yet, the tables will only report first births at this step.

```
////////////////////////////////////////////////////////////////////////////////////////
// Output tables: tables.mpp - ADDITIONS STEP 2                                       //
////////////////////////////////////////////////////////////////////////////////////////

table Person T03_FertilityByAge                             // 3) Age-specific fertility
{
  split(age,AGE_GROUP)                                      // Output for each single age year
  *
  {
    parity / duration() ,                                   // Age-specific fertility decimals=4
    entrances(parity_status, ONE_CHILD)/duration(),         // First-birth rate decimals=4
    entrances(parity_status, TWO_CHILDREN)/duration(),      // Second-birth rate decimals=4
      value_out(parity) / duration()                        // Cum average parity decimals=4
  }
};

table Person T04_CohortFertility                            // 4) Cohort fertility statistics
{
  {
    value_at_transitions
        (parity_status,CHILDLESS,ONE_CHILD,age)/
        entrances(parity_status, ONE_CHILD),                // Av. age at 1st birth decimals=2
    value_at_transitions
        (parity_status,ONE_CHILD,TWO_CHILDREN,age)
        /entrances(parity_status, TWO_CHILDREN),            // Av. age at 2nd birth decimals=2
      value_at_changes(parity_status, age )
        /(entrances(parity_status, ONE_CHILD)
        + entrances(parity_status, TWO_CHILDREN)),          // Av. age at birth decimals=2
    1-entrances(parity_status, ONE_CHILD)/unit,             // Childlessness decimals=4
    (entrances(parity_status, ONE_CHILD)
        -entrances(parity_status, TWO_CHILDREN))/unit,      // Percent one child decimals=4
    (entrances(parity_status, TWO_CHILDREN))/unit,          // Percent two children decimals=4
    value_at_entrances(life_status,NOT_ALIVE,parity)/unit // Cohort Fertility Rate decimals=4
  }
};

table_group TG02_FERTILITY_TABLES {                         // b.) Fertility
  T03_FertilityByAge, T04_CohortFertility
};
```

2.4. Finalizing the model (Step 3)

The third step finalizes our illustrative model. The programming mainly consists in a repetition of tasks already explained, i.e. the introduction of all parameters not yet introduced, the definition of additional clocks, second births and union transitions, and the extension of table output.

2.4.1. Finalizing the parameter definitions

The behaviors introduced into the model at the final step are partnership and second birth. Partnership transitions are parameterized by baseline hazards and control for different periods of motherhood. Second conception is modeled accordingly, the process starts at first conception and controls for age, union status, and partnership status at first birth. In Step 3, we introduce three additional clocks (see below). This requires the definition of three additional ranges of time indices and parameters to define the lower borders of the intervals for the three corresponding durations: union duration, duration since last birth, and duration since last union dissolution. We add a classification SECOND_STATE, which is used to record the union status at second birth.

```
/////////////////////////////////////////////////////////////////////////////////////////////
// Parameter definitions: parameters.mpp - ADDITIONS STEP 3                                //
/////////////////////////////////////////////////////////////////////////////////////////////

classification SECOND_STATE {            // Union status at second birth
  NO_SECOND,                             // No second birth
  SECOND_IN_FIRST_UNION,                 // Second birth in first union
  SECOND_IN_SECOND_UNION,                // Second birth in second union
  SECOND_NOT_IN_UNION                    // Second birth out of union
};

range UNIONINDEX {0,5};                  // Range of union duration index
range BIRTHINDEX {0,13};                 // Range of duration spells since last birth
range DISSOINDEX {0,4};                  // Range of duration spells since union dissolution

parameters {
  float  unionSchedule[UNIONINDEX];      // Lower borders of time in union intervals
  float  birthSchedule[BIRTHINDEX];      // Lower borders of time since last birth intervals
  float  dissoSchedule[DISSOINDEX];      // Lower borders of time since last dissolution

  double Secon_age[AGEINDEX];            // Age baseline hazards second birth
  double Secon_unionStatus[UNION_STATE]; // Relative risks of union status on second birth
  double Secon_spellBirth[BIRTHINDEX];   // Relative risk time since last birth on 2nd birth
  double Secon_firstStatus[FIRST_STATE]; // Relative risk union status at first birth on 2nd

  double Form1_age[AGEINDEX];            // Age baseline hazards first union formation
  double Form1_spellBirth[BIRTHINDEX];   // Rel. risk time since last birth on 1st formation
  double Form2_spellDisso[DISSOINDEX];   // Baseline (time since dissolution) of 2nd formation
  double Form2_spellBirth[BIRTHINDEX];   // Rel. risk time since last birth on 2nd formation

  double Diss1_spellUnion[UNIONINDEX];   // Baseline (time since formation) on 1st dissulution
  double Diss1_spellBirth[BIRTHINDEX];   // Rel. risk time since last birth on 1st dissolution
  double Diss2_spellUnion[UNIONINDEX];   // Baseline (time since formation) on 2st dissulution
};

parameter_group P03_SECOND_BIRTH {       // c.) Second Birth Parameters
  Secon_age ,Secon_spellBirth, Secon_unionStatus , Secon_firstStatus
};

parameter_group P04_PART_FORM1 {         // d.) First partnership formation parameters
  Form1_age, Form1_spellBirth
};

parameter_group P05_PART_FORM2 {         // e.) Second partnership formation parameters
  Form2_spellDisso, Form2_spellBirth
};

parameter_group P06_PART_DISS1 {         // f.) First partnership dissolution parameters
  Diss1_spellUnion, Diss1_spellBirth
};

parameter_group P06_PART_DISS2 {         // g.) Second partnership dissolution parameters
  Diss2_spellUnion
};

parameter_group P07_DUR_SPELLS {         // h.) Definition of age and duration spells
  ageSchedule, unionSchedule, birthSchedule, dissoSchedule
};
```

```
////////////////////////////////////////////////////////////////////////////////////////////
// Parameter values file: Baseline(lifecourse).dat - ADDITIONS STEP 3                     //
////////////////////////////////////////////////////////////////////////////////////////////

parameters {
  float unionSchedule[UNIONINDEX] = {0, 1, 3, 5, 9, 13, };
  float birthSchedule[BIRTHINDEX] = {
    0, 0.75, 1.25, 1.75, 2.75, 3.75, 4.75, 6.75, 8.75, 9.75, 10.75, 12.75, 17.75, 22.75,
  };
  float dissoSchedule[DISSOINDEX] = {0, 2, 6, 10, 15, };
  double Secon_age[AGEINDEX] = {
    0, 1.76290, 1.76470, 1.29019, 1.00000, 0.91534, 0.78281, 0.61662, 0.47991, 0.37239,
    0.20478,
  };
  double Secon_unionStatus[UNION_STATE] = {
    0.2922246, 1, 0.9217742, 0.2922246, 3.481644, 0.2922246,
  };
  double Secon_spellBirth[BIRTHINDEX] = {
    0, 0.067913, 0.145071, 0.200951, (2) 0.231365, 0.239884, 0.148317, (2) 0.110490, 0.072949,
    0.042124, 0.025746, 0.000006,
  };
  double Secon_firstStatus[FIRST_STATE] = {0,1,0.41,1.34};
  double Form1_age[AGEINDEX] = {
    0, 0.030898, 0.134066, 0.167197, 0.165551, 0.147390, 0.108470, 0.080378, 0.033944,
    0.045454, 0.040038,
  };
  double Form1_spellBirth[BIRTHINDEX] = {
    18.38257, (2) 1.705141, (2) 0.4210245, (9) 0.2938546,
  };
  double Form2_spellDisso[DISSOINDEX] = {
    0.1995702, 0.1353028, 0.1099149, 0.0261186, 0.0456905,
  };
  double Form2_spellBirth[BIRTHINDEX] = {3.991086, (13) 0.5807688, };
  double Diss1_spellBirth[BIRTHINDEX] = {
    0.2931838, (4) 0.5274971, (2) 0.6121143, (2) 0.5987179, (5) 0.9019690,
  };
  double Diss1_spellUnion[UNIONINDEX] = {
    0.0096017, (2) 0.0199994, 0.0213172, 0.0150836, 0.0110791,
  };
  double Diss2_spellUnion[UNIONINDEX] = {(2) 0.0370541, (2) 0.012775, (2) 0.0661157, };
};
```

2.4.2. Further clocks

Except for first partnership formation, all processes added at this point start at the occurrence of events, e.g. the process of union dissolution begins at union formation; this requires additional clocks recording (1) the time since first conception, (2) the time since last partnership formation, and (3) the time since last union dissolution. All clocks look up their corresponding parameters determining the next cutting point in time. As for age groups, we allow user to change the time intervals.

```
/////////////////////////////////////////////////////////////////////////////////////////////////
// Clock events file: clocks.mpp - ADDITIONS STEP 3                                             //
/////////////////////////////////////////////////////////////////////////////////////////////////

actor Person                                                // Individual
{
  int birthspell = {0};                                     // Current interval since last birth
  TIME lastbirthspellchange = TIME_INFINITE;                // Time of last interval index change
  event timeNextBirthSpellEvent, NextBirthSpellEvent;       // Event setting index last birth

  int unionspell = {0};                                     // Current interval since union form.
  TIME lastunionspellchange = TIME_INFINITE;                // Time of last interval index change
  event timeNextUnionSpellEvent, NextUnionSpellEvent;       // Event setting index last union form.

  int dissspell = {0};                                      // Current interval since dissolution
  TIME lastdissspellchange = TIME_INFINITE;                 // Time of last interval index change
  event timeNextDissSpellEvent, NextDissSpellEvent;         // Event setting index last dissolution
};

/////////////////////////////////////////////////////////////////////////////////////////////////
// Event changing index of time interval since last birth

TIME Person::timeNextBirthSpellEvent(){
  TIME event_time = TIME_INFINITE;
  TIME spelldurat = time - lastbirth;
  int i;
  for (i=MAX(BIRTHINDEX);i>0;i--){
      if ((spelldurat < birthSchedule[i]) && (spelldurat >=0))
        event_time = WAIT(birthSchedule[i]-spelldurat);
      else if ((spelldurat == birthSchedule[i]) && (lastbirthspellchange != time))
        event_time = time;
  }
  return event_time;
}

void Person::NextBirthSpellEvent(){birthspell++; lastbirthspellchange=time;}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////
// Event that changes index of time interval since last union formation

TIME Person::timeNextUnionSpellEvent(){
  TIME event_time = TIME_INFINITE;
  TIME spelldurat = time - lastunion;
  int i;
  for (i=MAX(UNIONINDEX);i>0;i--){
      if ((spelldurat < unionSchedule[i]) && (spelldurat >=0))
        event_time = WAIT(unionSchedule[i]-spelldurat);
      else if ((spelldurat == unionSchedule[i]) && (lastunionspellchange != time))
        event_time = time;
  }
  return event_time;
}

void Person::NextUnionSpellEvent(){
  unionspell++; lastunionspellchange=time;
  if (unionspell>=2 && union_status==FIRST_UNION_PERIOD1){union_status=FIRST_UNION_PERIOD2;}
}

////////////////////////////////////////////////////////////////////////////////////////////////
// Event that changes index of time interval since last union dissolution

TIME Person::timeNextDissSpellEvent(){
  TIME event_time = TIME_INFINITE;
  TIME spelldurat = time - lastdiss;
  int i;
  for (i=MAX(DISSOINDEX);i>0;i--){
      if ((spelldurat < dissoSchedule[i]) && (spelldurat >=0))
        event_time = WAIT(dissoSchedule[i]-spelldurat);
      else if ((spelldurat == dissoSchedule[i]) && (lastdissspellchange != time))
        event_time = time;
  }
  return event_time;
}

void Person::NextDissSpellEvent(){dissspell++; lastdissspellchange=time;}
```

2.4.3. Second births

The programming of the second birth event differs only slightly from that of first birth. A woman is at risk if she has parity 1. Besides a duration baseline since last birth, the model controls for age, union status at first birth, and the actual union status. The second birth event updates the parity_status to TWO_CHILDREN and increments the variable parity. Furthermore, it records the union status at second birth.

```
////////////////////////////////////////////////////////////////////////////////
// Fertility: fertility.mpp - ADDITIONS STEP 3                                //
////////////////////////////////////////////////////////////////////////////////

actor Person {
  SECOND_STATE second_status = {NO_SECOND};                 // Union status at second birth
  event timeSecondBirth, SecondBirth;                       // Second birth event
};

////////////////////////////////////////////////////////////////////////////////
// Second birth

TIME Person::timeSecondBirth(){
  TIME event_time = TIME_INFINITE;
  double randdur, hazard;
  if (parity_status == ONE_CHILD) {
    hazard=Secon_age[agespell] * Secon_spellBirth[birthspell]
      * Secon_unionStatus[union_status] * Secon_firstStatus[first_status];
    if (hazard>0){
        randdur = -log(RandUniform(2))/hazard;
        event_time = WAIT(randdur);
    }
  }
  return event_time;
}

void Person::SecondBirth(){
  parity++;
  parity_status=TWO_CHILDREN;
  if (union_status==FIRST_UNION_PERIOD1 || union_status==FIRST_UNION_PERIOD2)
      second_status=SECOND_IN_FIRST_UNION;
  else if (union_status==SECOND_UNION) second_status=SECOND_IN_SECOND_UNION;
  else second_status=SECOND_NOT_IN_UNION;
}
```

2.4.4. Unions

The main additions of the third step to our model code development are partnership behaviors: first and second union formation and dissolution. If a women’s union_status is NEVER_IN_UNION, she is at risk of entering a first union[11]. Being in a union also means being at risk of union dissolution. All related events update the union status and record the time of the event. The number of unions is counted by a variable nUnions, which is later used for table output.

If a union formation occurs during pregnancy, the union status at birth is ex-post-corrected from corresponding “birth out of union” to “birth within union” levels.

[11] More precisely, she enters the first period of the first union FIRST_UNION_PERIOD1, from where she is automatically moved to FIRST_UNION_PERIOD2 after three years by the union duration clock if the union was not dissolved in the meantime.

```
////////////////////////////////////////////////////////////////////////////////////////////////
// Union formation & dissolution events: unions.mpp - introduced at STEP 3                   //
////////////////////////////////////////////////////////////////////////////////////////////////

actor Person                                             // Individual
{
  int nUnions = {0};                                     // Union counter
  TIME lastunion = TIME_INFINITE;                        // Time of last union formation
  TIME lastdiss  = TIME_INFINITE;                        // Time of last union dissolution
  event timeUnion1Formation, Union1Formation;            // First union formation event
  event timeUnion1Dissolution, Union1Dissolution;        // First union dissolution event
  event timeUnion2Dissolution, Union2Dissolution;        // Second union dissolution event
  event timeUnion2Formation, Union2Formation;            // Second union formation event
};

////////////////////////////////////////////////////////////////////////////////////////////////
// First union formation

TIME Person::timeUnion1Formation(){
      TIME event_time = TIME_INFINITE;
      double randdur, hazard;
      if (union_status==NEVER_IN_UNION) {

        hazard=Form1_age[agespell];
        if (parity_status==ONE_CHILD){hazard=hazard* Form1_spellBirth[birthspell];}
        if (hazard>0){
              randdur = -log(RandUniform(4))/hazard;
              event_time = WAIT(randdur);
        }
      }
      return event_time;
}
void Person::Union1Formation(){
      nUnions++;
      union_status=FIRST_UNION_PERIOD1;
      lastunion=time;
      unionspell=0;
      // Ex-post correction of union status at birth for union formation in pregnancy
      if (birthspell==0 && parity==1) first_status=FIRST_IN_FIRST_UNION;
      else if (birthspell==0 && parity==2) second_status=SECOND_IN_FIRST_UNION;
}
```

```
////////////////////////////////////////////////////////////////////////////////
// Second union formation

TIME Person::timeUnion2Formation(){
      TIME event_time = TIME_INFINITE;
      double randdur, hazard;
      if (union_status==AFTER_FIRST_UNION) {

        hazard=Form2_spellDisso[dissspell];
        if (parity_status==ONE_CHILD){hazard=hazard* Form2_spellBirth[birthspell];}
        if (hazard>0){
              randdur = -log(RandUniform(7))/hazard;
              event_time = WAIT(randdur);
        }
      }
      return event_time;
}
void Person::Union2Formation(){
      nUnions++;
      union_status=SECOND_UNION;
      lastunion=time;
      unionspell=0;
      // Ex-post correction of union status at birth for union formation in pregnancy
      if (birthspell==0 && parity==1) {first_status=FIRST_IN_SECOND_UNION;}
      else if (birthspell==0 && parity==2) {second_status=SECOND_IN_SECOND_UNION;}
}

////////////////////////////////////////////////////////////////////////////////
// First union dissolution

TIME Person::timeUnion1Dissolution(){
      TIME event_time = TIME_INFINITE;
      double randdur, hazard;
      if (union_status==FIRST_UNION_PERIOD1 || union_status==FIRST_UNION_PERIOD2) {
                                                  // at risk
        hazard=Diss1_spellUnion[unionspell];
        if (parity>0) {hazard=hazard*Diss1_spellBirth[birthspell];}
        if (hazard>0){
              randdur = -log(RandUniform(5))/hazard;
              event_time = WAIT(randdur);
        }
      }
      return event_time;
}

void Person::Union1Dissolution(){
  lastdiss=time;
  union_status=AFTER_FIRST_UNION;
}

////////////////////////////////////////////////////////////////////////////////
// Second union dissolution

TIME Person::timeUnion2Dissolution(){
      TIME event_time = TIME_INFINITE;
      double randdur, hazard;
      if (union_status==SECOND_UNION) {
        hazard=Diss2_spellUnion[unionspell];
        if (hazard>0){
              randdur = -log(RandUniform(6))/hazard;
              event_time = WAIT(randdur);
        }
      }
      return event_time;
}
void Person::Union2Dissolution(){union_status=AFTER_SECOND_UNION;}
```

2.4.5. Tables

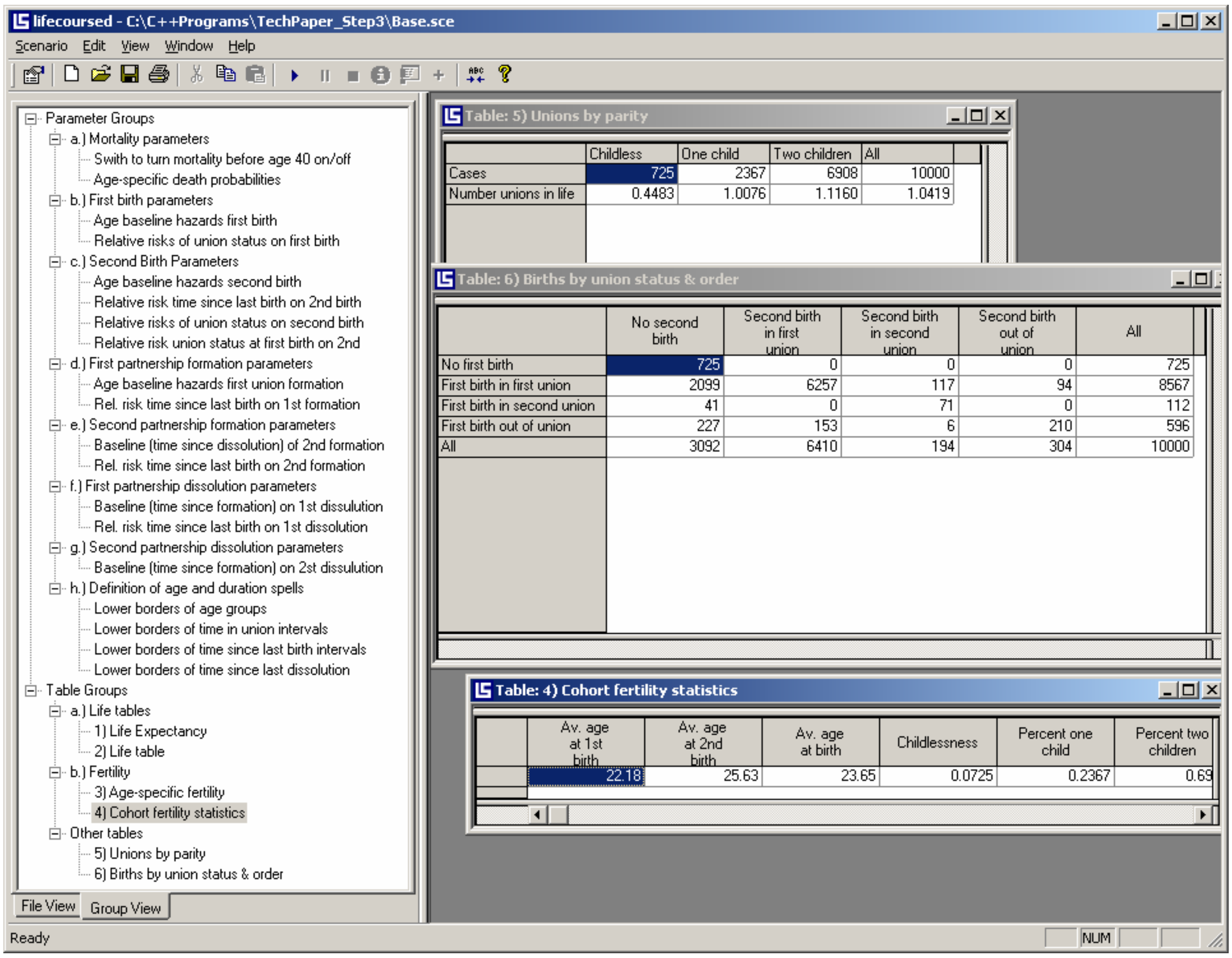
In the final step we add two tables to the application: one for the number of unions by parity, the second for the number of first and second births by union status at birth. Both tables evaluate the final states at the end of the actor's life. This is done by including a filter statement [life_status=NOT_ALIVE] in the table definition. The average number of unions in life is obtained by the Modgen function value_in(nUnions) divided by units; it is calculated separately by parity_status. The second table displays the number of women (unit) by first_status and second_status (i.e. union status at first and second birth).

In order to use the BioBrowser tool, we include a track statement for the automatic tracking of the most important variables for graphical output.

```
////////////////////////////////////////////////////////////////////////////////////////////////
// Output tables: tables.mpp - ADDITIONS STEP 3                                              //
////////////////////////////////////////////////////////////////////////////////////////////////

table Person T05_UnionsByParity [life_status==NOT_ALIVE]  // 5) Unions by parity
{
  {
    unit,                                                 // Cases
      value_in(nUnions)/unit                                // Number unions in life decimals=4
  }
  * parity_status+
};

table Person T06_BirthsByUnion [life_status==NOT_ALIVE]   // 6) Births by union status & order
{
  {
      unit
  }
  * first_status+
  * second_status+
};

track Person
{
   current_age, life_status, parity, agespell, birthspell, unionspell, dissspell,
   nUnions, parity_status, union_status, first_status, second_status
};
```

2.4.6. Running the model

After having successfully converted the mpp-code into c++ code by Modgen and having built the Visual Studio Solution, we can run the final version of our model. The selection window of the resulting application now contains the full list of model parameters and all six tables defined previously.



lifecoursed - C:\C++Programs\TechPaper_Step3\Base.sce

Scenario Edit View Window Help

```
Parameter Groups
  a.) Mortality parameters
      Swith to turn mortality before age 40 on/off
      Age-specific death probabilities
  b.) First birth parameters
      Age baseline hazards first birth
      Relative risks of union status on first birth
  c.) Second Birth Parameters
      Age baseline hazards second birth
      Relative risk time since last birth on 2nd birth
      Relative risks of union status on second birth
      Relative risk union status at first birth on 2nd
  d.) First partnership formation parameters
      Age baseline hazards first union formation
      Rel. risk time since last birth on 1st formation
  e.) Second partnership formation parameters
      Baseline (time since dissolution) of 2nd formation
      Rel. risk time since last birth on 2nd formation
  f.) First partnership dissolution parameters
      Baseline (time since formation) on 1st dissulution
      Rel. risk time since last birth on 1st dissolution
  g.) Second partnership dissolution parameters
      Baseline (time since formation) on 2st dissulution
  h.) Definition of age and duration spells
      Lower borders of age groups
      Lower borders of time in union intervals
      Lower borders of time since last birth intervals
      Lower borders of time since last dissolution
Table Groups
  a.) Life tables
      1) Life Expectancy
      2) Life table
  b.) Fertility
      3) Age-specific fertility
      4) Cohort fertility statistics
  Other tables
      5) Unions by parity
      6) Births by union status & order
```

File View Group View

Table: 5) Unions by parity

	Childless	One child	Two children	All
Cases	725	2367	6908	10000
Number unions in life	0.4483	1.0076	1.1160	1.0419

Table: 6) Births by union status & order

	No second birth	Second birth in first union	Second birth in second union	Second birth out of union	All
No first birth	725	0	0	0	725
First birth in first union	2099	6257	117	94	8567
First birth in second union	41	0	71	0	112
First birth out of union	227	153	6	210	596
All	3092	6410	194	304	10000

Table: 4) Cohort fertility statistics

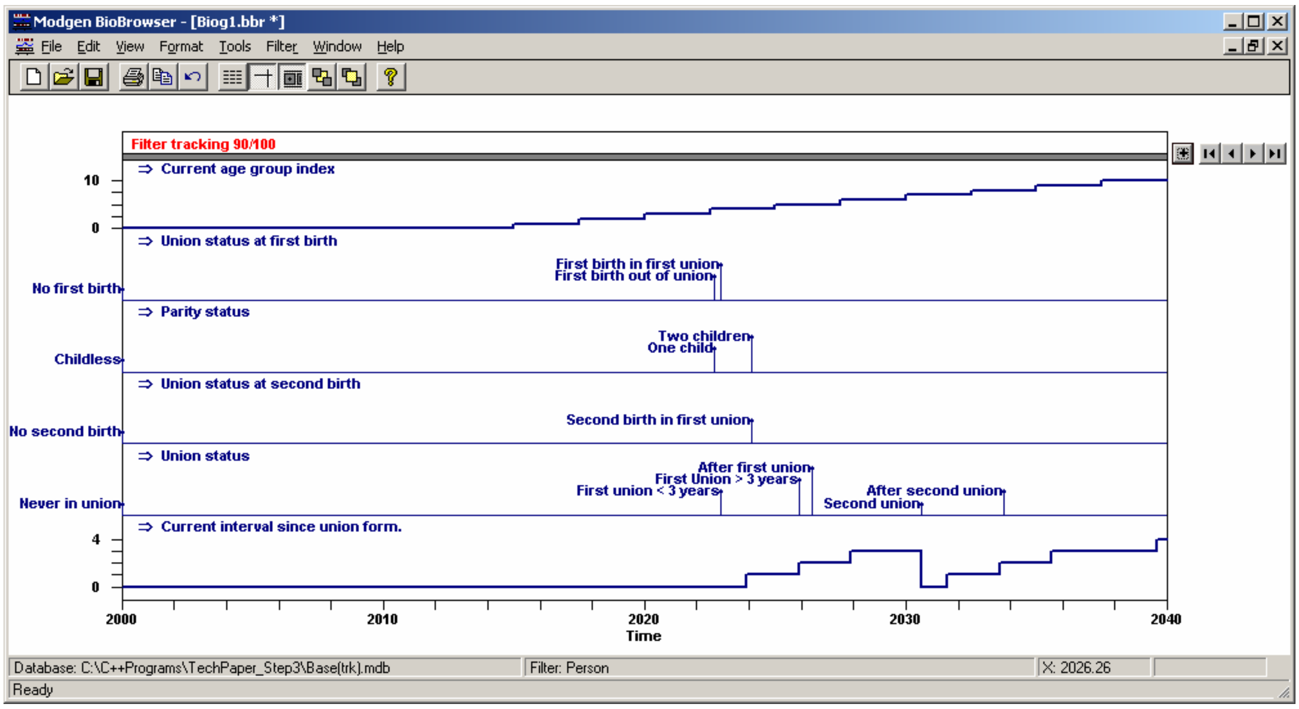
	Av. age at 1st birth	Av. age at 2nd birth	Av. age at birth	Childlessness	Percent one child	Percent two children
	22.18	25.63	23.65	0.0725	0.2367	0.69

Ready NUM

Figure 4: The Modgen User-Interface of the final LifeCourse model

2.4.7. BioBrowser output of individual life courses

Figure 5 below displays the life course of a simulated actor experiencing a first birth before first union formation, first union formation during pregnancy, second birth within the first union, and a union dissolution followed by a second union. We recommend the use of the BioBrowser tool for debugging whenever new events and/or status variables are added to the model in order to verify proper status changes in all status variables.



Database: C:\C++Programs\TechPaper_Step3\Base(trk).mdb

Filter: Person

X: 2026.26

Ready

Figure 5: BioBrowser output, final model

2.5. Using the model

The "LifeCourse" model presented in this contribution was initially developed for the study of fertility decline in Bulgaria and Russia (Spielauer et. al 2006) and served as a template of "sister models" currently developed for the study of the effect of union dissolution on fertility in France. In the presented context of demographic analysis, microsimulation mainly serves as a means of synthesizing different event-history models in order to assess the effect of individual parameters not only on the risks of single events but also on the whole system under study. For instance, from the Bulgarian hazard regression model for first births we know that between 1989 and 1999 first conception risks have dropped by 49% in the first three years of a first partnership. Using microsimulation, we get the additional information, that this change in parameters alone leads to a 48% higher proportion of childlessness (an increase from 7.25% to 10.72%), that it increases the average age at first birth by 9.6 months and decreases overall cohort fertility by 5.1%. While such "what-if" type studies are typical applications of microsimulation, the full field of applications is much broader, ranging from theoretical studies on how good statistical methods and models replicate original data to the main practical application of microsimulation: detailed socio-demographic and economic projections.

Acknowledgements

I am thankful to Susann Backer for careful language editing of this contribution.

References

The Modgen programming language as well as the BioBrowser Tool were developed at Statistics Canada and can be downloaded at http://www.statcan.ca/english/spsd/Modgen.htm. Also all documentation, i.e. the Modgen Users' Guide, the Modgen Developer's Guide, and the BioBrowser User's Guide can be downloaded from this site.

- Statistics Canada, Modgen Version 7.0.31 User's Guide, http://www.statcan.ca/english/spsd/ModgenUser_EN.pdf
- Statistics Canada, Modgen Version 7.0.31 Developer's Guide, http://www.statcan.ca/english/spsd/ModgenDev_EN.pdf
- Statistics Canada, BioBrowser: The ModGen Biography Browser Version 3.1 User's Guide, http://www.statcan.ca/english/spsd/BioBrowser_E.pdf

The Modgen applications LifePaths and POHEM developed at Statistics Canada are documented at http://www.statcan.ca/english/spsd/

- Statistics Canada, LifePaths Overview V1.1, http://www.statcan.ca/english/spsd/LifePathsOverview_E.pdf

Galler, Hans Peter (1997) Discrete-Time and Continuous-Time Approaches to Dynamic Microsimulation Reconsidered. Technical Paper 13. National Centre for Social and Economic Modeling. University of Canberra, http://www.natsem.canberra.edu.au/publications/papers/tps/tp13/tp13.pdf

Spielauer, Martin, Dora Kostova, and Elena Koytcheva (2006) First and second births in first and second unions: a decomposition of the fertility decline in Bulgaria and Russia by means of microsimulation. Forthcoming, check http://www.demogr.mpg.de/

Appendix: The simulated model estimated from Bulgarian GGS data

The illustrative Modgen application developed in this paper is based on simple piecewise constant event history models for first and second conception and first and second union formation and dissolution estimated from 2004 Bulgarian Generations and Gender Survey data for women born after 1950. The model was developed to study the individual contribution of the different processes to the observed drop in fertility in Bulgaria and Russia (Spielauer et. al. 2006). The Base(lifecourse).dat file provided contains the parameters for the period before the Bulgarian economic and political transition in 1989 (bold numbers in the tables below). For this analysis, we exclude non-Bulgarian ethnicities and censor at age 40 and at second birth.

The parameters for **first conception** consist of an age baseline and an interaction variable of calendar time and union status.

FIRST CONCEPTION

15-17.5	**0.29*****
17.5-20	**0.76*****
20-22.5	**0.85*****
22.5-25	**0.82*****
25-27.5	**0.67*****
27.5-30	**0.51*****
30-32.5	**0.49*****
32.5-35	**0.26*****
35-37.5	**0.26*****
37.5-40	**0.15*****

	< 1998 =BASE	1989-1993	1994-1998	1999+
Not in union	**0.06*****	0.07***	0.05***	0.03***
First union <3 years	***1***	*0.94*	0.74***	0.49***
First union >3 years	**0.25*****	0.29***	0.23***	0.27***
Second Union	**0.80**	0.21***	0.40**	0.23***

The model for **second conception** uses time since first birth as baseline hazard and, in addition to the interaction between calendar time and union status, it controls for union status at first birth and age.

SECOND CONCEPTION

<0.5 years after first birth	**0.07*** **
0.5-1 years after first birth	**0.15*** **
1-2 years after first birth	**0.20*** **
2-4 years after first birth	**0.23*** **
4-6 years after first birth	**0.24*** **
6-8 years after first birth	**0.15*** **
8-10 years after first birth	**0.11*** **
10-12 years after first birth	**0.07*** **
12-17 years after first birth	**0.04*** **
17-22 years after first birth	**0.03*** **
22+ years after first birth	**0.00**
First Birth not in Union	**1.34** **
First Birth in first union	*1*
First Birth in Second Union	**0.41** **

15-17.5	**1.76***
17.5-20	**1.76*** **
20-22.5	**1.29*** **
22.5-25	**1**
25-27.5	***0.92***
27.5-30	**0.78*** **
30-32.5	**0.62*** **
32.5-35	**0.48*** **
35-37.5	**0.37*** **
37.5-40	**0.20*** **

	< 1998 =BASE	1989-1993	1994-1998	1999+
Not in union	**0.29*** **	0.19***	0.09***	0.12***
First union <3 years	*1*	*0.78***	0.44***	0.32***
First union >3 years	**0.92**	0.64***	0.44***	0.45***
Second Union	**3.48*** **	1.64	1.58	1.20

The process for **first union formation** starts at age 15 (age baseline). The model controls for different episodes of motherhood and calendar time.

FIRST UNION FORMATION

15-17.5	**0.03*** **
17.5-20	**0.13*** **
20-22.5	**0.17*** **
22.5-25	**0.17*** **
25-27.5	**0.15*** **
27.5-30	**0.11*** **
30-32.5	**0.08*** **
32.5-35	**0.03*** **
35-37.5	**0.05*** **
37.5-40	**0.04*** **

Before first pregnancy	*1*
In first pregnancy	**18.38*** **
first year after birth	**1.71*** **
year 2-3 after birth	**0.42*** **
year 4+ after birth	**0.29*** **
Before 1989 = BASE	*1*
1989-1993	0.94
1994-1998	0.69***
1999+	0.56***

The process for **first union dissolution** starts at first union formation. The model controls for different episodes of motherhood and calendar time.

FIRST UNION DISSOLUTION

1st year of union	**0.01*****
union duration 1-5	**0.02*****
union duration 5-9	**0.02*****
union duration 9-13	**0.02*****
union duration >13	**0.01*****
before first pregnancy	***1***
during first pregnancy	**0.29*****
first 3 years after birth	**0.53*****
3-6 years after first birth	**0.61****
6-9 years after first birth	**0.60****
9+ years after first birth;	**0.90**

Before 1989 = BASE	***1***
1989-1993	1.08
1994-1998	1.48**
1999+	1.26

The process for **second union formation** starts at first union dissolution. The model controls for different episodes of motherhood and calendar time.

SECOND UNION FORMATION

< 2 years after dissolution	**0.20*****
2-6 years after dissolution	**0.14*****
6-10 years after dissolution	**0.11*****
10-15 years after dissolution	**0.03*****
15+ years after dissolution	**0.05*****
Before first pregnancy	***1***
In first pregnancy	**3.99*****
Mother	**0.58*****

Before 1989 = BASE	***1***
1989-1993	0.82
1994-1998	0.85
1999+	0.73

Second union dissolution is modeled as the process starting at second union formation and has no additional controls.

SECOND UNION DISSOLUTION

<3 years union duration	**0.04*****
3-9 years union duration	**0.01*****
9+ years union duration	**0.07*****