MODGEN AND THE APPLICATION RISKPATHS FROM THE MODEL DEVELOPER'S VIEW

Martin Spielauer

Statistics Canada – Modeling Division
R.H. Coats Building, 24-O
Ottawa, K1A 0T6

martin.spielauer@statcan.gc.ca

RiskPaths is a simple, competing risk, case-based continuous time microsimulation model. Its main use is as a teaching tool, introducing microsimulation to social scientists and demonstrating how dynamic microsimulation models can be efficiently programmed using the language Modgen.

Modgen is a generic microsimulation programming language developed and maintained at Statistics Canada.

RiskPaths as well as the Modgen programming language and other related documents are available at www.statcan.gc.ca/microsimulation/modgen/modgen-eng.htm

1 Introduction

In this chapter we explore the microsimulation model development package Modgen and the Modgen application RiskPaths from the model developer's point of view. We first introduce the Modgen programming environment, and then discuss basic Modgen language concepts and the RiskPaths code. Modgen requires only moderate programming skills; thus, after some training, it enables social scientists to create their own models without the need for professional programmers. This is possible because Modgen hides underlying mechanisms like event queuing and automatically creates a stand-alone model with a complete visual interface, including scenario management and model documentation (as introduced in the previous chapter). Model developers can therefore concentrate on model specific code: the declaration of parameters, the states defining the simulated actors, and the events changing the

states. High efficiency coding extends also to model output. Modgen includes a powerful language to handle continuous time tabulation. These tabulations are created on-the-fly when simulations are run and the programming to generate them usually requires only a few lines of code per table. Modgen also has a built-in mechanism for estimating the Monte Carlo variation for any cell of any table, without requiring any programming by the model developer.

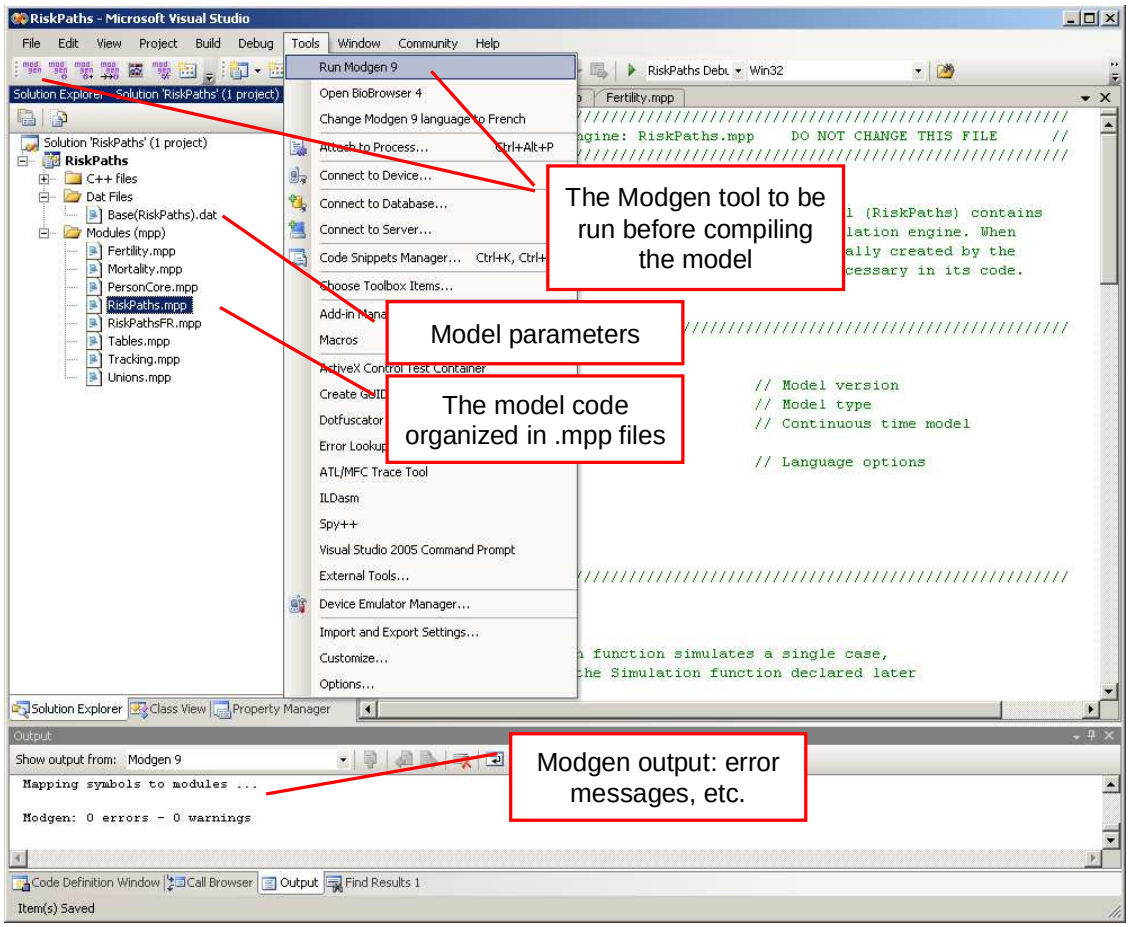
Being a simple model, RiskPaths does not make use of the full range of available Modgen language concepts and capabilities. The discussion in this chapter does not intend to replace existing Modgen documentation, such as the Modgen Developer's Guide, either. But by introducing the main concepts of Modgen programming, we aim to help you get started in Modgen model development and to engage in further exploration.

2 The Modgen programming environment

When installed on a computer, Modgen integrates itself into the (required) Microsoft Visual Studio C++ environment. The visual components of Modgen are a separate toolbar as well as additional items under the Tools and Help menus of Visual Studio. Modgen also appears as an option in the file dialog box for creating a new project as well as in the dialog box for adding a file to an existing project.

Figure 1 displays a screenshot of the programming interface as it appears after opening the Modgen application 'RiskPaths.sln'. The Modgen toolbar consists of several icons for running Modgen, accessing help, opening the BioBrowser tool, and switching the language (between English and French).

Figure 1: The programming interface



Modgen code is organized into several files, each with the file extension .mpp. As can be seen in the Solution Explorer window (Figure 1), RiskPaths consists of eight .mpp files grouped in the "Models (mpp)" folder. These are the essential files of RiskPaths, i.e. the files containing all Modgen code written by the model developer.

When invoking the Modgen tool (which can be accessed from the toolbar, or from the first item under the "Tools" menu), these .mpp files are translated into C++ code. Thus Modgen acts as a pre-compiler, creating one .cpp source code file for each .mpp file and putting the resulting .cpp files in the "C++ Files" folder. The Modgen tool also adds model-independent C++ code components to the "C++ Files" folder; these additional files[1] should not be changed by the model developer and are essential in order to use the C++ compiler to build the Modgen application.

[1] ACTORS.CPP, ACTORS.H, app.ico, model.h, model.RC, PARSE.INF, TABINIT.CPP, TABINIT.H.

The model parameters are contained in one or more .dat files organized in a folder labelled "Scenarios". These files are loaded at runtime and contain the actual values assigned to the parameters.

When running the Modgen tool, Modgen – like the C++ compiler - produces log output that is displayed in the Output window. Any error messages are also displayed in this window, and clicking on a particular error message leads you directly to the corresponding Modgen code that produced the error.

Two steps are required to create a Modgen application from the Visual Studio environment. First, Modgen has to translate the Modgen code in the .mpp files; this is done when invoking the Modgen tool. Second, the resulting C++ application has to be built and started. This can be done in one step by selecting "Start Debugging" in the "Debug" menu or by clicking the corresponding icon at the toolbar.

3 Basic Modgen Concepts

Actor: An actor is the entity whose life is simulated in a Modgen model. A model's actor is often a person, although this is not a requirement--other models have been developed that use dwellings or occupations as actors. Nevertheless, in RiskPaths, the actor is a person or more specifically, a female (since it is a model for the study of childlessness)

State: States describe the characteristics of a model's actors. Some states can be continuous, such as age, whereas others are categorical, such as gender. For categorical states, the actual categories or levels are defined via Modgen's **classification** command.

Overall, there are two major kinds of states in Modgen—**simple states** and **derived states**, both of which are used by RiskPaths and both of which are declared within an actor declaration. A simple state is a state whose value can be initialized and changed by the code that a model developer creates. Simple states are changed by explicitly declared events. A derived state, on the other hand, is a state whose value is given as an expression which is normally derived from or based on other states. A derived state's values are automatically maintained by Modgen throughout a simulation run. A useful Modgen concept is the self-scheduling derived state. This is a state which changes in a predefined time sequence, such as integer age which will change at each birthday.

Event: In Modgen, simulation takes place through the execution of events. Each event consists of two functions: a time function to determine the time of the next occurrence of the event, and an implementation function to determine the consequences when the event happens. RiskPaths has several events, including a mortality event, union formation and dissolution events, and a first pregnancy event.

Parameter: Parameters are used to give model users a degree of control over the simulations they run. The ability to change different hazards or probabilities that affect various aspects of a simulation allows different scenarios to be explored. Parameters can have many dimensions (such as age, gender, and year) and are stored in .dat data files. In RiskPaths, there is one parameter file, Base(RiskPaths).dat, which stores parameters such as death probabilities by age and risks of first pregnancy by age group. More complex models will usually incorporate more than one .dat file.

Table: Modgen has a powerful cross-tabulation facility built in to report aggregated results in the form of tables. There are two central elements of a table declaration—its captured dimensions (defining when an actor enters and leaves a cell) and its analysis dimension (recording what happens while an actor is in that cell). When running simulations, the tabulations to fill a table are created on the fly, thus removing the need to create and write to large temporary interim files for subsequent reporting. Several examples of table declarations will be shown later in this chapter for RiskPaths.

4 Organization of files

The Modgen code of RiskPaths is organized into eight separate .mpp files, while all RiskPaths parameter values (because RiskPaths is a simple model) are contained in just a single .dat file. In principle, a model developer has complete freedom to decide how to organize the Modgen code in different files, but a modular organization as found in RiskPaths is recommended.

Figure 2: RiskPaths file organization

General modules	Filename
Simulation engine	RiskPaths.mpp
Core actor file	PersonCore.mpp
Table definitions	Tables.mpp
Output tracking	Tracking.mpp
French language translations	RiskPathsFR.mpp
Behavioural modules	**Filename**
Mortality	Mortality.mpp
Fertility	Fertility.mpp
Union formations and dissolutions	Unions.mpp
Parameter file	**Filename**
Parameters of Baseline Scenario	Base(RiskPaths).dat

Note that the .mpp code files often contain comments that resemble labels. Such comments are placed beside the declarations of symbols such as states, state levels, parameters, tables and table dimensions. Modgen does in fact interpret these comments as labels and subsequently uses them when tables or parameters are displayed within Modgen's visual interface. These

labels are also used in the model's automatically generated encyclopaedic help file. Code comments that are used as labels begin with a two-character language identifier, e.g. //EN Union status. Many such comments can be seen in the code examples that follow for RiskPaths.

Fore more detailed descriptions of modules, functions and events, notes following the following syntax example can be placed in the code. These notes – besides documenting the code - are used in the automatically generated encyclopaedic help file.

```
/*NOTE(Person.Finish, EN)
      The Finish function terminates the simulation of an actor.
*/
```

4.1 RiskPaths.mpp (the main simulation file)

This file contains the code essential for the definition of the model type (e.g. case-based, continuous time) as well as the simulation engine, i.e. the code that runs the whole simulation. Because RiskPaths is a case-based model, the simulation engine code loops through all cases and processes the event queues of each case. The file also identifies the languages of the model. The code of this file is mostly model independent within a class of models (e.g. continuous time, case-based) and a version of it is provided automatically when using Modgen's built-in wizards to start a new Modgen project.

For the development of our case-based, continuous time cohort RiskPaths model with an actor 'Person' the code provided by the wizard requires very few modifications. The full code of this .mpp file is less than one page in length.

4.2 PersonCore.mpp

The only actor in RiskPaths is a person. In the file PersonCore.mpp, we have organized the code which is part of the actor declaration but not directly related to a specific behaviour. The file contains two age clocks defined as self-scheduling states (integer_age and age_status) and two actor functions, Start() and Finish(), which are performed at the creation of an actor and at her death, respectively.

In the Start() function we initialize the states time and age to 0. Both states are automatically created and maintained by Modgen and can only be changed in the Start() function. Their types depend on the model type; because RiskPaths is a continuous time model, time and age are continuous states.

The Finish() function must be called at the death event of an actor. Its role is to remove the actor from tables and from the simulation, and to recuperate any computer memory used by the actor.

All states and actor functions are declared in an "actor Person { };" block. To allow modularity in the organization of code by different life course domains, there can be multiple actor blocks in a project, typically one for each behavioural file.

The first code section of this module contains three type definitions. We first define a range LIFE.

```
range LIFE        //EN Simulated age range
{
      0,100
};
```

Range is a Modgen type which defines a range of integer values. RiskPaths limits the possible age range of persons to 100 years. This type will be used to declare a derived state containing the age of a person in completed years. The second type definition will be used to divide continuous age into 2.5 year age intervals starting at age 15.

```
partition AGEINT_STATE        //EN 2.5 year age intervals
{
      15, 17.5, 20, 22.5, 25, 27.5, 30, 32.5, 35, 37.5, 40
};
```

The third definition is a classification of union types. In general, if a range, partition, or classification is used in several files, it is good practice to define it in the core actor file.

```
classification UNION_STATE                //EN Union status
{
      US_NEVER_IN_UNION,                  //EN Never in union
      US_FIRST_UNION_PERIOD1,             //EN First union < 3 years
      US_FIRST_UNION_PERIOD2,             //EN First Union > 3 years
      US_AFTER_FIRST_UNION,               //EN After first union
      US_SECOND_UNION,                    //EN Second union
      US_AFTER_SECOND_UNION               //EN After second union
};
```

In the following code segment we declare two derived actor states and two functions. The derived states for time intervals are used to change the values of parameters that vary over time. In our model integer_age is needed because mortality risks are dependent on age in years, whereas age_status comes into play because baseline risks for first conception and first union formation are modelled to change in 2.5 year intervals after the 15th birthday. Both integer_age and age_status have to be maintained over the simulation. The Modgen concept of derived states allows us to have them maintained automatically. Both are derived from the

state age (which is a special state, as it is generated and maintained automatically by Modgen). In order to split up age into the time intervals defined in the AGEINT_STATE partition, we make use of the Modgen function self_scheduling_split. The second derived state, integer_age, could be directly obtained using the Modgen function self_scheduling_int. In order to ensure that its value stays in the possible range of LIFE we convert it to type LIFE, which is done by the Modgen macro COERCE.

```
actor Person
{
	//EN Current age interval
	int age_status = self_scheduling_split(age, AGEINT_STATE);

	//EN Current integer age
	LIFE integer_age = COERCE( LIFE, self_scheduling_int(age) );

	//EN Function starting the life of an actor
	void Start();

	//EN Function finishing the life of an actor
	void Finish();
}
```

The remaining code of this module is the implementation of the Start() and Finish() functions. The Finish() function is left empty as we do not require any actions, other than those automatically performed by Modgen, to take place when an actor dies.

```
void Person::Start()
{
	// Age and time are variables automatically maintained by
	// Modgen. They can be set only in the Start function
	age = 0;
	time = 0;
}

/*NOTE(Person.Finish, EN)
	The Finish function terminates the simulation of an actor.
*/
void Person::Finish()
{
	// After the code in this function (if any) is executed,
	// Modgen removes the actor from tables and from the simulation.
	// Modgen also recuperates any memory used by the actor.
}
```

4.3 Behavioural Files

In RiskPaths we distinguish three groups of behaviours: mortality, fertility and union formation/dissolution. Accordingly we have organized the code into three .mpp files: Mortality.mpp, Fertility.mpp and Unions.mpp. Behavioural files are typically arranged in three sections:

- Declaration of parameters
- Declarations of actor states and events
- Implementation of events

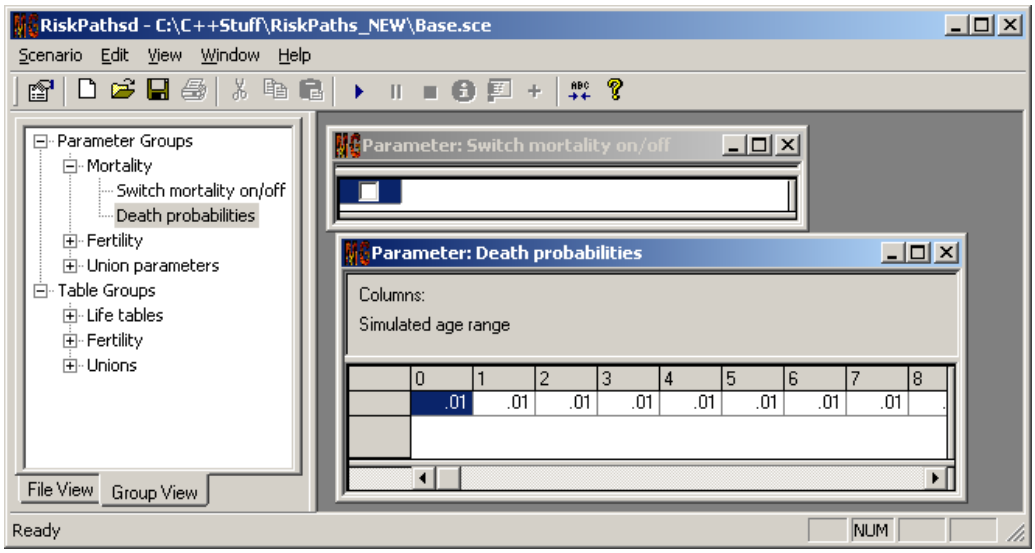
4.3.1 Mortality.mpp

This file defines the mortality event that ends the life of the simulated actor. Mortality.mpp is a typical behavioural module, and we follow a standard organization of the code: type definitions, parameter declarations, actor declarations and event implementations.

Parameter declarations

Mortality is parameterized by death probabilities by age; thus, the probability to survive another year changes at each birthday. We also introduce a parameter which allows us to 'switch off' mortality. When it is used, every actor reaches the maximum age of 100 years (which can be useful for some types of fertility analysis). Figure 3 displays the mortality parameter tables of the RiskPaths application.

Figure 3: Mortality parameters



Parameters are declared within a “parameters {…};” code block. Modgen supports numeric types, such as int, long, float, double, or Boolean (“logical” in the Modgen terminology). The dimensionality of the parameters in the RiskPaths model is defined by classifications and ranges. The following code generates the parameters for RiskPaths, as displayed in Figure 3. For the annual death probabilities we use the range LIFE that was defined in PersonCore.mpp. The following statement (**parameter_group)** groups the two mortality parameters in order to provide an ordered hierarchical selection list in the user interface (again, as displayed in Figure 3).

```
parameters
{
      logical CanDie;               //EN Switch mortality on/off
      double  ProbMort[LIFE];       //EN Death probabilities
};

parameter_group P01_Mortality       //EN Mortality
{
      CanDie, ProbMort
};
```

Actor declarations

Actors are described by states which are changed in events. States can be both continuous (integer or real) or categorical. In the mortality module, the state of interest is whether a person is alive or not, thus making it categorical in nature. The levels of a categorical state are defined with the Modgen **classification** command.

We declare a state life_status of type LIFE_STATE, which is initialized with LS_ALIVE at birth and set to LS_NOT_ALIVE by the death event. It is good practice to initialize all states by assigning initial values. Each initial value, however, must be enclosed in braces, i.e. {}—otherwise, the state is implemented as a derived state.

```
classification LIFE_STATE                       //EN Life status
{
      LS_ALIVE,                                 //EN Alive
      LS_NOT_ALIVE                              //EN Dead
};

actor Person


{
      LIFE_STATE life_status = {LS_ALIVE};      //EN Life Status
      event timeDeathEvent, DeathEvent;         //EN Death Event
};
```

Events are declared in the actor Person {..} block using the keyword **event**. All events consist of a function which returns the time of the next event and a function containing the code describing the consequences of the event.

Event implementation

When mortality is activated, the timeDeathEvent function returns a random time based on the mortality parameter for the given year of age. In order to obtain random durations from probabilities, we assume constant mortality hazards within each period, i.e. between birthdays. (The exception is a death probability of 1, which leads to death immediately at the start of the age year). Note that any time later than the next birthday will lead to the birthday event taking precedence over the mortality event; that is, the birthday event will censor the mortality event.

```
TIME Person::timeDeathEvent()
{
	TIME event_time = TIME_INFINITE;
	if (CanDie)
	{
		if (ProbMort[integer_age] >= 1)
		{
			event_time = WAIT(0);
		}
		else
		{
			event_time = WAIT(-log(RandUniform(3)) /
					-log(1 - ProbMort[integer_age]));
		}
	}
	// Death event can not occur after the maximum duration of life
	if (event_time > MAX(LIFE))
	{
		event_time = MAX(LIFE);
	}
	return event_time;
}
```

The event implementation function DeathEvent is straightforward. It sets the life_status to LS_NOT_ALIVE and calls the function Finish(), the latter which deletes the actor.

```
void Person::DeathEvent()
{
	life_status = LS_NOT_ALIVE;
	Finish();
}
```

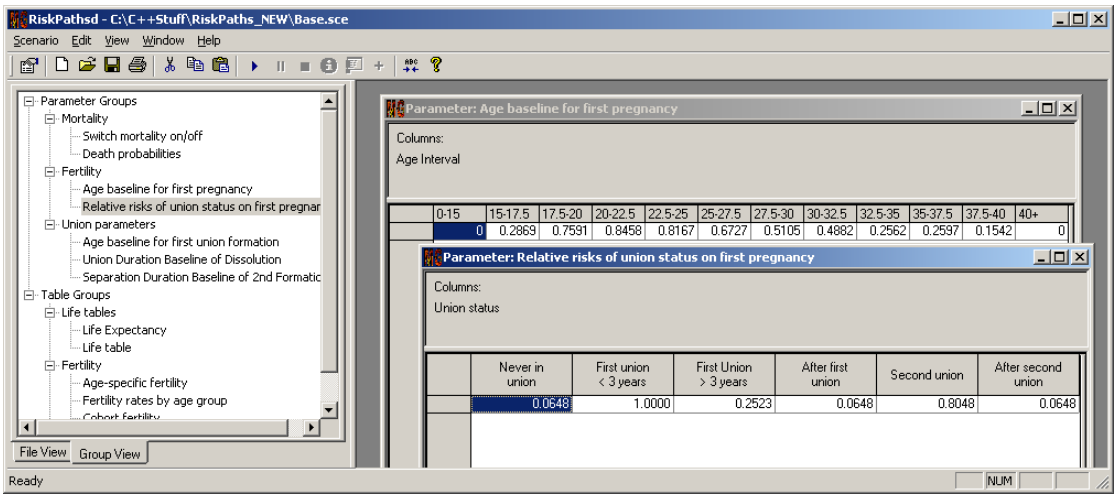
4.3.2 Fertility.mpp

This file defines and implements the first pregnancy event. As we are only interested in the study of childlessness in RiskPaths, no other fertility-related event is simulated. Fertility.mpp is a behavioural module, and again we follow the same standard organization of the code: type definitions, parameter declarations, actor declarations and event implementations.

Parameter declarations

Fertility is parameterized by both a baseline pregnancy risk by 2.5 year age intervals starting at the 15th birthday and a relative risk factor dependent on the union status and duration. We thus define two parameters: AgeBaselinePreg1 and UnionStatusPreg1.

Figure 4: Fertility parameters



Fertility risks use a time partition to define the columns. For the age baseline we use the partition AGEINT_STATE that was defined in PersonCore.mpp. The possible union states for the relative risk factors use the classification UNION_STATE which is declared in PersonCore.mpp as well.

```
parameters
{
      //EN Age baseline for first pregnancy
      double AgeBaselinePreg1[AGEINT_STATE];
      //EN Relative risks of union status on first pregnancy
      double UnionStatusPreg1[UNION_STATE];
};

parameter_group P02_Ferility              //EN Fertility
{
      AgeBaselinePreg1, UnionStatusPreg1
};
```

Actor declarations

The only state of the fertility module is parity_status, which can only have two levels: 'childless' and 'pregnant'. (This is because RiskPaths no longer simulates an actor's fertility events after first conception).

In Fertility.mpp, we only model one event: pregnancy. The corresponding pair of event functions is timeFirstPregEvent and FirstPregEvent.

```
classification PARITY_STATE                //EN Parity status
{
      PS_CHILDLESS,                        //EN Childless
      PS_PREGNANT                          //EN Pregnant
};

actor Person
{
      //EN Parity status derived from the state parity
      PARITY_STATE parity_status = {PS_CHILDLESS};

      //EN First pregnancy event
      event timeFirstPregEvent, FirstPregEvent;
};
```

Event implementation

As is true with all Modgen events, the first pregnancy event is implemented in two parts. The first determines the timing of the event, the second the consequences if the event happens. The timeFirstPregEvent function verifies if the actor is currently at risk and, if so, draws a random duration based on the underlying piecewise proportional constant hazard regression model parameterized by an age baseline and relative risk by union status. Accordingly, the hazard rate is calculated from the two parameters AgeBaselinePreg1 and UnionStatusPreg1. A random duration can be obtained from a uniform distributed random number by the transformation:

randdur=-log(RandUniform(1)) / hazard.

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 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.

When the event happens, the state "parity" is increased by 1. (Note that the derived state parity_status is changed to "PS_PREGNANT" automatically).

```
TIME Person::timeFirstPregEvent()
{
      double dHazard = 0;
```

```
      TIME event_time = TIME_INFINITE;
      if (parity_status == PS_CHILDLESS)
      {

            dHazard = AgeBaselinePreg1[age_status]
              * UnionStatusPreg1[union_status];
          if (dHazard > 0)
          {
                event_time = WAIT(-log(RandUniform(1)) / dHazard);
            }
  }
  return event_time;
}

void Person::FirstPregEvent()
{
      parity_status = PS_PREGNANT
}
```

4.3.3 Unions.mpp

The programming of union transitions introduces only minor new concepts in Modgen programming--thus, the following code discussion is mainly limited to union dissolutions. The hazard rates for both first and second union dissolution events are stored in the same parameter table, as they each use the same time intervals of union duration.

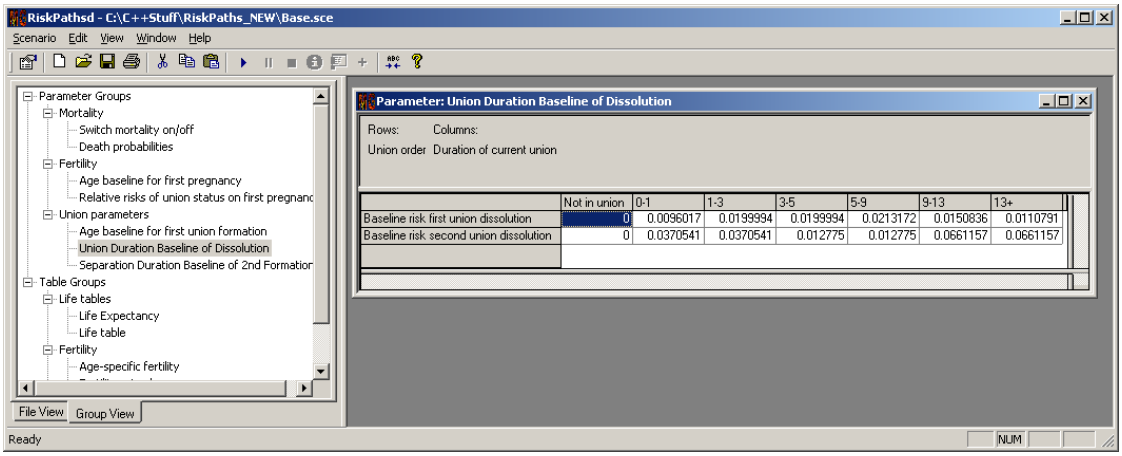
In order to construct a parameter with the dimensions time and union order, we define a time partition and a classification:

```
partition UNION_DURATION            //EN Duration of current union
{
      1, 3, 5, 9, 13
};

classification UNION_ORDER          //EN Union order
{
      UO_FIRST,         //EN First union
      UO_SECOND         //EN Second union
};
```

```
parameters
{
	…
	//EN Union Duration Baseline of Dissolution
	double UnionDurationBaseline[UNION_ORDER][UNION_DUR];
	…
};
```

Figure 5: Union dissolution parameters



In the timeUnion1DissolutionEvent() function, hazard rates for first union dissolution are obtained as:

```
dHazard = UnionDurationBaseline[UO_FIRST][union_duration];
```

Accordingly, timeUnion2DissolutionEvent() references the second row from the parameter:

```
dHazard = UnionDurationBaseline[UO_SECOND][union_duration];
```

As opposed to the processes discussed so far, the union dissolution processes do not start at a predefined time (e.g. the 15th birthday) but at union formation events. The union duration spell is defined as a derived self-scheduling state in the following form:

```
//EN Currently in an union
logical in_union = (union_status == US_FIRST_UNION_PERIOD1
      || union_status == US_FIRST_UNION_PERIOD2
      || union_status == US_SECOND_UNION);

//EN Time interval since union formation
int   union_duration = self_scheduling_split(
            active_spell_duration( in_union, TRUE), UNION_DURATION);
```

With respect to union formation, the implementation of the clock which changes the union duration state union_status from US_FIRST_UNION_PERIOD1 to US_FIRST_UNION_PERIOD2 after three years in a first union deserves some discussion. In

contrast to the self-scheduling derived states used for all other clocks of the model, here – mainly as an illustration of this alternative - we explicitly implement the clock as an event itself. This event occurs after three years in the first union. The clock is set at first union formation. The actor declaration includes a state which records the time of the status change as well as the event declaration.

```
actor Person
{
	…

	//EN Time of union period change
	TIME  union_period2_change = {TIME_INFINITE};

	//EN Union period change event
	event timeUnionPeriod2Event, UnionPeriod2Event;
};
```

The time for the state change is set in the first union formation event. In the code sample, WAIT is a built-in Modgen function that returns the time of the current event, plus a specified time (in our example, three years).

```
void Person::Union1FormationEvent()
{
	unions++;
	union_status = US_FIRST_UNION_PERIOD1;
	union_period2_change = WAIT(3);
}
```

The event implementation is straight forward:

```
TIME Person::timeUnionPeriod2Event()
{
	return union_period2_change;
}

void Person::UnionPeriod2Event()
{
	if (union_status == US_FIRST_UNION_PERIOD1)
	{
		union_status = US_FIRST_UNION_PERIOD2;
	}
	union_period2_change = TIME_INFINITE;
}
```

4.4 Tables.mpp

Modgen provides a very powerful and flexible cross-tabulation facility to report model results. The programming of each output table usually requires only a few lines of code. RiskPaths contains only one table file which contains the declarations of all of its output tables—however, for more detailed models, it is advisable to split up table declarations by behavioural groups.

The basic syntax for tables is displayed in Figure 6. The two central elements of a table declaration are the captured classificatory dimensions (defining when an actor enters and leaves a cell) and the analysis dimension (recording what happens while an actor is in that cell). Typical classificatory dimensions are age or time intervals (e.g. fertility by age), states (e.g. fertility by union status), or a combination of both. Modgen does not limit the number of dimensions.

The analysis dimension can contain many expressions, which can be states or derived states. Modgen provides a very useful list of special derived state functions which record, for example, the number of occurrences of certain events, the number of changes in states, or the duration in states. Two particularly helpful concepts are the keyword **unit** and the derived state function **duration()** -- **unit** records the number of actors entering a table cell whereas **duration()** records the total time an actor stayed in the cell.

Tables can contain filter criteria for defining if and under which conditions actor characteristics will be recorded. The Modgen table concepts are best understood by concrete examples as given below. As the full wealth of the Modgen table language goes beyond the scope of this chapter, you are also invited to consult the Modgen Developer's Guide.

Figure 6: Table Syntax

```
table actor_name table_name                        //EN table label
[filter_criteria]
{
      dimension_a *                                //EN dimension label
      …
      {
            analysis_dimension_expression_x,       //EN expression label
            …
      }
      * dimension_n                                //EN dimension label
      …
};
```

Table 1: Life expectancy

The first table example contains summary values of our simulation and has no dimensions, i.e. cells apply to the entire population over the entire simulation period. We make use of the Modgen keyword **unit**, which counts the number of actors entering the cell of a table (in our example, the simulation itself), and the Modgen function **duration()** which sums up the time actors stay in this cell (in our example, 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 declarations, comments placed in the code are used as labels in the application. (Note that in the table declaration below, the 'decimals=3' portion of the comment is used to determine the number of decimal places in the table; this part of the comment does not carry through to the label used in the report).

```
table Person T01_LifeExpectancy //EN 1) Life Expectancy
{
 {
 unit,                  // EN Total simulated cases
 duration(),            // EN Total duration
 duration()/unit        // EN Life expectancy decimals=3
 }
};
```

Table 2: Life table

In the second table we record the population by age. For output by age, we use integer_age as table dimension.

```
table Person T02_TotalPopulationByYear    //EN Life table
{
      //EN Age
      integer_age *
      {
            unit,                         //EN Population start of year
            duration()                    //EN Average population in year
      }
};
```

Unit and **duration()** now refer to the number of entrances into - and durations within - one year age intervals. **Unit** thus counts the actors present at the beginning of each year, while **duration()** refers to the average population in the year.

Tables 3 and 4: Age-specific fertility

As well as the keywords **unit** and the derived state function **duration()**, states and a set of other derived state functions can be used in tables. If using a state without a function, Modgen records the change of the state while in a particular cell, i.e. the value of the state when the cell is exited minus the value of the state when the cell was entered.

The expression transitions(parity_status, PS_CHILDLESS, PS_PREGNANT) / duration() records the (age specific) fertility as the number of birth events divided by the average number of women by year of age.

The second expression is used to calculate the true rate, i.e. the number of birth events by exposure time. A woman is under exposure for first pregnancy when childless. We thus divide the number of events by the term 'duration(parity_status, PS_CHILDLESS)'.

The table dimension is age in full years. As fertility is 0 until age 15 and very low after 40, the age periods before 15 and after 40 are not further divided. We thus define a partition AGE_FERTILEYEARS which is used in the self_scheduling_split which defines the table dimension.

```
partition AGE_FERTILEYEARS //EN Fertile age partition
{
      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
};

table Person T03_FertilityByAge     //EN Age-specific fertility
{
      //EN Age
      self_scheduling_split(age,AGE_FERTILEYEARS) *
      {
            //EN First birth rate all women decimals=4
            transitions( parity_status, PS_CHILDLESS, PS_PREGNANT ) /
                  duration() ,
            //EN First birth rate woman at risk decimals=4
            transitions( parity_status, PS_CHILDLESS, PS_PREGNANT ) /
                  duration( parity_status, PS_CHILDLESS )
      }
};
```

Table 4 produces first birth rates by the 2.5 year age groups used for parameterization. We also add an additional dimension, namely the union status; we thus obtain simulated values of the model parameters.

```
table Person T04_FertilityRatesByAgeGroup //EN Fertility rates by age group
[parity_status == PS_CHILDLESS]
{
```

```
      {
            //EN Fertility decimals=4
            transitions(parity_status, PS_CHILDLESS, PS_PREGNANT) /
                  duration()
      }
      * self_scheduling_split(age, AGEINT_STATE)      //EN Age interval
      * union_status                                   //EN Union Status
};
```

Table 5: Cohort fertility

Table 5 calculates two cohort measures of fertility -- average age at first conception and childlessness. To obtain the age at pregnancy we use the Modgen derived state function value_at_transitions(parity_status, PS_CHILDLESS ,PS_PREGNANT, age) which returns the value of one state (age) at a specific transition of another state, namely when parity_status changes from PS_CHILDLESS to PS_PREGNANT.

```
table Person T05_CohortFertility          //EN Cohort fertility
{
      {
      //EN Av. age at 1st pregnancy decimals=2
      value_at_transitions(parity_status,PS_CHILDLESS,PS_PREGNANT,age)/
            transitions(parity_status, PS_CHILDLESS, PS_PREGNANT),

      //EN Childlessness decimals=4
      1 - transitions(parity_status, PS_CHILDLESS, PS_PREGNANT) / unit,

      //EN Percent one child decimals=4
            transitions(parity_status, PS_CHILDLESS, PS_PREGNANT) / unit

      }
};
```

Table 6: Pregnancies by union status and order

In table 6 we use an example of a filter which triggers a person exactly at the entrance of a state, in our case at the occurrence of pregnancy. We are interested in the union status at first conception. Note that this filter also excludes women who stay childless.

```
table Person T06_BirthsByUnion //EN Pregnancies by union status & order
[trigger_entrances(parity_status, PS_PREGNANT)]
{
      {
            unit        //EN Number of pregnancies
      }
      *union_status+    //EN Union Status at pregnancy
};
```

Table 7: First union formation risks

Like table 4 this table reproduces a parameter table. While such an output table does not contain any information (for a sufficiently large sample size it will come close to the original model parameters) it is useful for model validation and to assess Monte Carlo variability.

```
table Person T07_FirstUnionFormation      //EN First union formation
[parity_status == PS_CHILDLESS]
{
      //EN Age group
      self_scheduling_split(age, AGEINT_STATE) *
      {
            //EN First union formation risk decimals=4
            entrances(union_status, US_FIRST_UNION_PERIOD1)
            / duration(union_status, US_NEVER_IN_UNION)

      }
};
```

4.4.1 Grouping of table output

Like parameters, output tables can also be grouped for a more meaningful presentation of results. In the application RiskPaths, we distinguish three groups of tables: life tables, fertility tables, and tables for union status.

```
table_group TG01_Life_Tables        //EN Life tables
{
 T01_LifeExpectancy, T02_TotalPopulationByYear
};

table_group TG02_Birth_Tables       //EN Fertility
{
      T03_FertilityByAge, T04_FertilityRatesByAgeGroup, T05_CohortFertility
};

table_group TG03_Union_Tables       //EN Unions
{
      T06_BirthsByUnion, T07_FirstUnionFormation
};
```

4.5 Tracking.mpp

The track{} code block defines the list of states to be recorded longitudinally for visual BioBrowser output. This command is frequently placed in table files. In our model, however, we have decided to code a separate Tracking.mpp file, since we also track risk patterns calculated as derived states.

```
track Person
{
      integer_age,
      life_status,
      age_status,
      union_duration,
      dissolution_duration,
      unions,
      parity_status,
      union_status,
      preg_hazard,
      formation_hazard,
      dissolution_hazard
};
```

The file also includes the declaration of three derived states. We have used the derived state concept to calculate the three main hazard rates (pregnancy, union formation, and union dissolution) for BioBrowser output. They are for illustrative purposes only, as all hazard rates, broken down by union order, are calculated in the event functions.

The declaration of the derived states preg_hazard, formation_hazard, and dissolution_hazard are also good syntax examples of how derived states can be built from simple states by if-else constructs.

```
actor Person
{
      //EN Pregnancy hazard
      double preg_hazard = (parity_status == PS_CHILDLESS) ?
            AgeBaselinePreg1[age_status] *
            UnionStatusPreg1[union_status] : 0;

      //EN Union formation hazard
      double formation_hazard = (union_status != US_NEVER_IN_UNION
            && union_status != US_AFTER_FIRST_UNION) ? 0 :
            ((union_status == US_NEVER_IN_UNION) ?
            AgeBaselineForm1[age_status] :
            SeparationDurationBaseline[dissolution_duration] );

      //EN Union dissolution hazard
      double dissolution_hazard = (union_status != US_FIRST_UNION_PERIOD1
&&
            union_status != US_FIRST_UNION_PERIOD2 &&
            union_status != US_SECOND_UNION) ? 0 :
            ((union_status == US_SECOND_UNION) ?
            UnionDurationBaseline[UO_SECOND][union_duration] :
            UnionDurationBaseline[UO_FIRST][union_duration]);
};
```

4.6 Language translation file RiskPathsFR.mpp

This .mpp file will only exist for models that are defined in Modgen to be multilingual (which for RiskPaths implies English and French). Even for a bilingual model, however, one of English or French is still deemed to be the first or primary language of the model. English was chosen as the primary language when RiskPaths was originally developed, and so the RiskPathsFR.mpp file essentially contains translations for the model's labels and notes in the other language, i.e. French. (If the original primary language of RiskPaths had been French, this translation file would have been called RiskPathsEN.mpp and it would have contained English translations of the labels and notes for the model.)

Normally, all notes and labels are entered as code comments in the source .mpp files, using the primary language of the model, as has been illustrated several times in the previous examples. The corresponding translations are subsequently placed in this separate .mpp file.