7. THE DOTNETSIM SIMULATION ENGINE

CHAPTER OVERVIEW

DotNetSim's simulation engine prototypes an event-based simulation executive that runs, through simulated time, the models devised within the DotNetSim modelling environment. Written within the .NET Framework, it reads the modelling data whose collection is described in chapter 6, runs the event-based simulation and stores the results of each replication into an Excel workbook. This chapter stresses the integration of the DotNetSim simulation engine with other components which are

based on other Microsoft applications and executed by instantiating the corresponding classes. Upstream from the DotNetSim simulation engine is the Visio™ Event Graph modelling environment described in chapters 5 and 6. Downstream is an Excel™ application for analysing and reporting the simulation results (see chapter 8). These three coarse-grained components are integrated by proper class instantiation and not merely linked through common files.

The simulation model is run by a purpose-designed, event-based simulation engine. Like the rest of DotNetSim, this has enough functionality to show the main principles of such integration, but should be enhanced for proper use. The experience of developing this application by customising and integrating the components is used to reflect on the value of such .NET integration for discrete event simulation.

7.1. OBJECTIVES OF THE DOTNETSIM SIMULATION ENGINE

The DotNetSim simulation engine consists of a number of .NET components that prototype an event-based simulation executive, which is callable from within the DotNetSim modelling environment. The simulation engine takes the model logic captured within the Microsoft Visio™ Event Graph emulator, though not by simple file transfer. Instead, it instantiates Visio classes and invokes the appropriate methods. The model is then run over simulated time using an event-based simulation algorithm. The number of simulation replications and the length of each run are determined by the parameters stored while defining the model in Visio™.

The simulation results are placed directly in an Excel workbook by instantiating this application's classes – that is, once again, not by using text files readable by Excel™. The results can then be analysed within Excel™ by resorting to its built-in tools for data analysis or to specific tools that could be developed in VBA.

Eventually, the modelling environment becomes active again and the model may be refined if this is appropriate. Fig 7.1 depicts the DotNetSim prototype's architecture, zooming in the simulation engine.

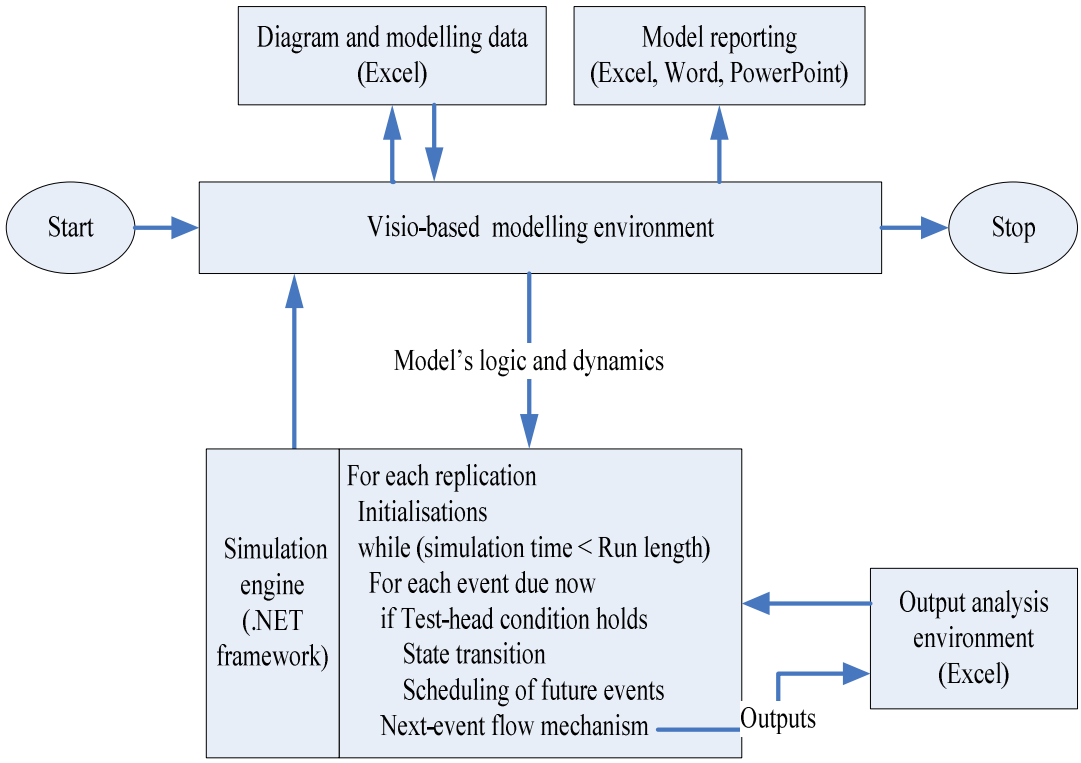

Fig. 7.1: Zooming the simulation engine in the DotNetSim's architecture. The execution of the DotNetSim starts and ends within the Visio-based modelling environment, passing through the stages 'Modelling – Simulation – Output analysis'

The DotNetSim simulation engine gets a reference to the Visio-based DE model and reads its logic and dynamics by invoking the appropriate methods on the diagram's shapes. It runs the model a number of times as determined by simulation parameters, also read from Visio™. In each replication, the attempt is made to execute events scheduled for the current simulation time, i.e. for each event due to occur at this time, if the test-head conditions hold, the system state transits accordingly and the subsequent events are scheduled [107, 17]. A calendar of the time-stamped events is maintained and the next event time flow mechanism is applied to the simulation time (see chapter 2). Eventually, the simulation results are placed in Excel™ by instantiating an Excel-template and invoking the appropriate methods on new

workbook objects. The execution then passes to Excel™. On saving the Excel workbook, the execution returns to the simulation engine and finally back to the Visio-based modelling environment.

This description of the DotNetSim simulation engine aims to highlight the integration of object-oriented components, achieved by the instantiation of the classes of one component from within another component. The inter-packages borders are sufficiently blurred to allow a component developed within one package to manipulate the objects of other components developed within other packages. Fig. 7.2 shows the DotNetSim simulation engine crossing the borders of the .NET Framework to manipulate the objects of Visio™ and the objects of Excel™.

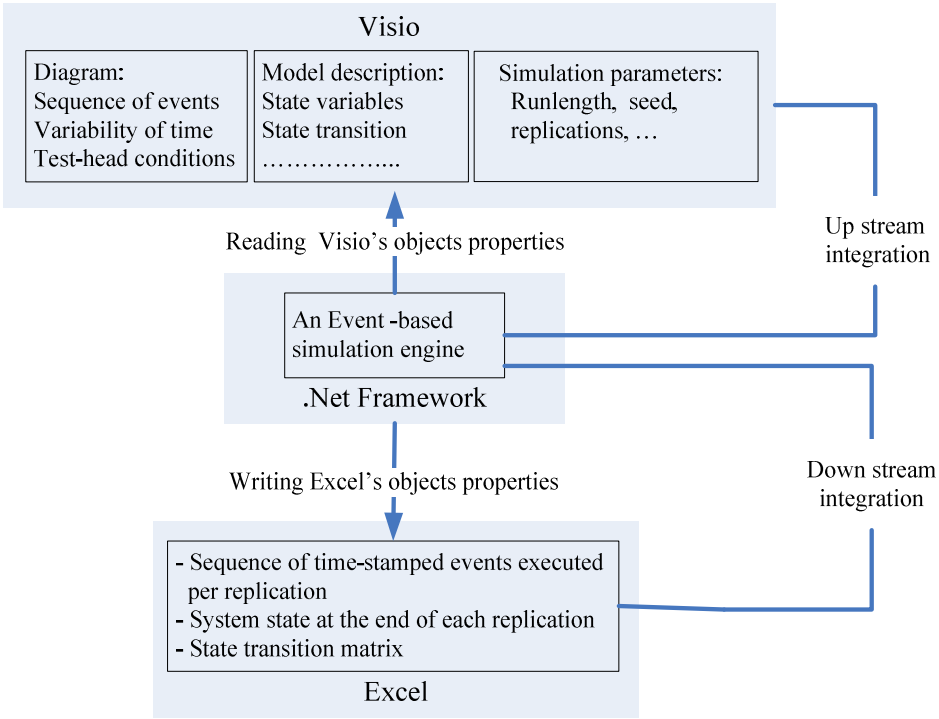

Fig. 7.2: The DotNetSim simulation engine instantiates Visio's classes to read the model's logic and instantiates Excel's classes to write the simulation results

Upstream from the DotNetSim simulation engine is the Visio-based DE model developed within the DotNetSim modelling environment. The DotNetSim simulation engine crosses the upstream border to read directly from the custom properties of the events and the edges diagrammatical data such as the sequence of the events, the

characteristics of the variability of the time that passes through the edges and the test-head conditions. It also reads directly from the Visio-based DE model the modelling data and simulation parameters stored as custom properties of the first page of the Visio drawing document. The modelling data in which the DotNetSim simulation engine is principally interested is the state transition triggered by each event, i.e. the changes that each event causes on the state variables.

The simulation parameters are, for example, the number of replications, the run length and the seed for generating random numbers; other data could also be read by applying the same OOP principles. As soon as the DotNetSim simulation engine gets a reference to the Visio drawing document which stores the DE model, it can manipulate the Visio classes by invoking the appropriate methods.

Downstream from the DotNetSim simulation engine is Excel™. The simulation engine crosses the downstream border to write the simulation results directly on the Excel ranges. It instantiates a workbook based on the SimOutAna Excel template (see chapter 8) and writes directly on the appropriate sheets and ranges of cells the sequence of time-stamped events executed in each replication and the state of the system reached by the end of each replication. It also writes the state transition matrix in this Excel workbook, where each element a_{ij} represents the change triggered by the event i on the state variable j. Other data could also be written by applying OOP principles to the Excel object model.

7.2 DOTNETSIM: THE SIMULATION ENGINE COMPONENT

To run the model captured within Visio™ and, later, to display the simulation results in Excel™, the DotNetSim simulation engine handles the Visio™ and Excel objects as if they were its own, i.e. instantiations of classes defined within the .NET

Framework. To do this, the Visio™ and the Excel PIAS (see chapter 4), the Microsoft.Office. Interop.Visio.dll and the Microsoft.Office. Interop.Excel.dll, have to be installed in the global assembly cache [33]. The .NET applications are then enabled to interoperate with those Microsoft applications just by adding references [51] to the corresponding object libraries.

Thus, references to these libraries were added to those components of the simulation engine (see Fig. 7.3) which read data from Visio™ or write data into Excel™.

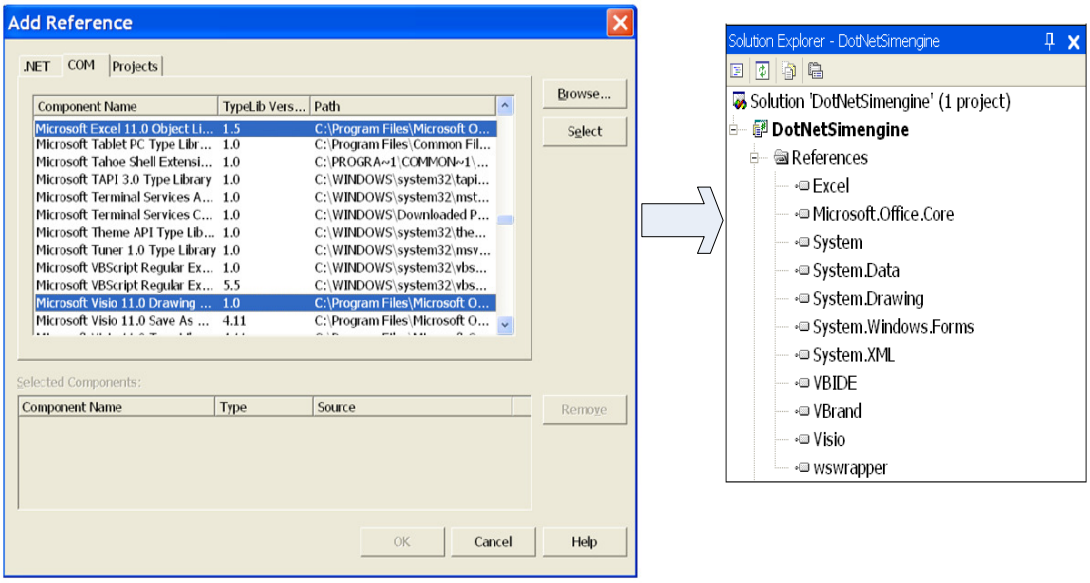

Fig. 7.3: Adding to the DotNetSim simulation engine references to the Microsoft Excel™ 11.0 and Microsoft Visio™ 11.0 libraries

```
using Microsoft.Office.Interop.Visio;
using Visioapp=Microsoft.Office.Interop.Visio.Application;
using Microsoft.Office.Core;
using Microsoft.Office.Interop.Excel;
using Excelapp=Microsoft.Office.Interop.Excel.Application;
using System.Runtime.InteropServices;
```

Fig. 7.4: Directives which instruct the compiler about the namespaces of Visio™ and Excel object models

Also, to shorten the names of the classes, the directives listed in Fig. 7.4 were

included to inform the compiler of their fully qualified names and chosen alias names.

7.2.1. UP-STREAM INTEGRATION: READING DATA FROM VISIO™

The simulation engine is executed by a direct command from within the DotNetSim modelling environment that invokes the corresponding .NET assembly. The simulation engine gets access to the Visio object model by marshalling the object (i.e. preparing the object to cross applications) that points to the active instance of the Visio™ application (see Fig. 7.5).

```
object visioObject = Marshal.GetActiveObject("Visio.Application");
Visioapp visio= visioObject as Visioapp;
visio.Windows.get_ItemEx(1);
```

Fig. 7.5: C# statements that get a reference to the active Visio drawing document

This gives access to the root of the Visio object model, which thus enables the simulation engine components to read the custom properties of the events, the edges of the Event Graph and the associated modelling data, as described in chapter 6. Fig.7.6 lists a C# program that illustrates how a custom property of a Visio shape is

```
using System;
using System.Windows.Forms;
using Microsoft.Office.Interop.Visio;
using Visioapp=Microsoft.Office.Interop.Visio.Application;
using Microsoft.Office.Core;
using System.Runtime.InteropServices;
namespace simengine
{class mainprog
  {public static void Main(string[] args)
    { object visioObject = Marshal.GetActiveObject("Visio.Application");
      Visioapp visio= visioObject as Visioapp;
      visio.Windows.get_ItemEx(1);
      int repl; // number of replications
      repl=visio.Application.ActivePage.PageSheet.get_Cells("Prop.Replic").get_ResultInt("",2);
      MessageBox.Show(repl.ToString());}}}
```

Fig. 7.6: C# program that reads the number of times a DES is to run. This simulation parameter is stored as a custom property of the active Page of the current Visio drawing document

read from within the .NET Framework. First, it gets a reference to the active Visio drawing document as shown in Fig. 7.5, then it invokes the method get_ResultInt("",2) on the custom property of its active page. As an example, it reads the number of times the DE model captured on the active Visio Document is to be run.

In order to take advantage of the object-orientation of C# and so facilitate the implementation of the simulation algorithm, these properties are mostly read into arrays of objects. Thus, classes are created to encapsulate the properties and the methods that mould the behaviour of entities such as the state variables or the scheduling edges. Statevariables, for example, is a class with properties such as name, type, maximum value and current value. It also contains the methods for setting and getting the current values of these properties. Finally, the different kinds of edges are defined as classes that, by inheritance, derive from a generic class Edge and therefore share the common properties and methods (see section 3.1.3.). Fig. 7.7 and Fig. 7.8 show respectively the edges inheritance graph and the C# definition of the classes Edges and Scheduling.

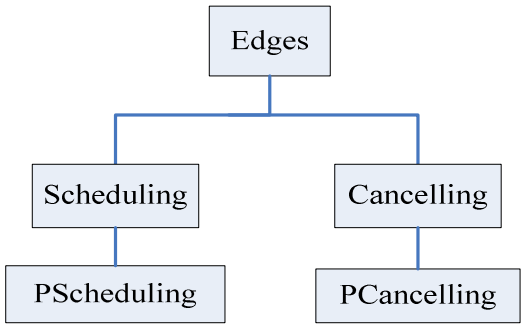

Fig. 7.7: Edges inheritance graph

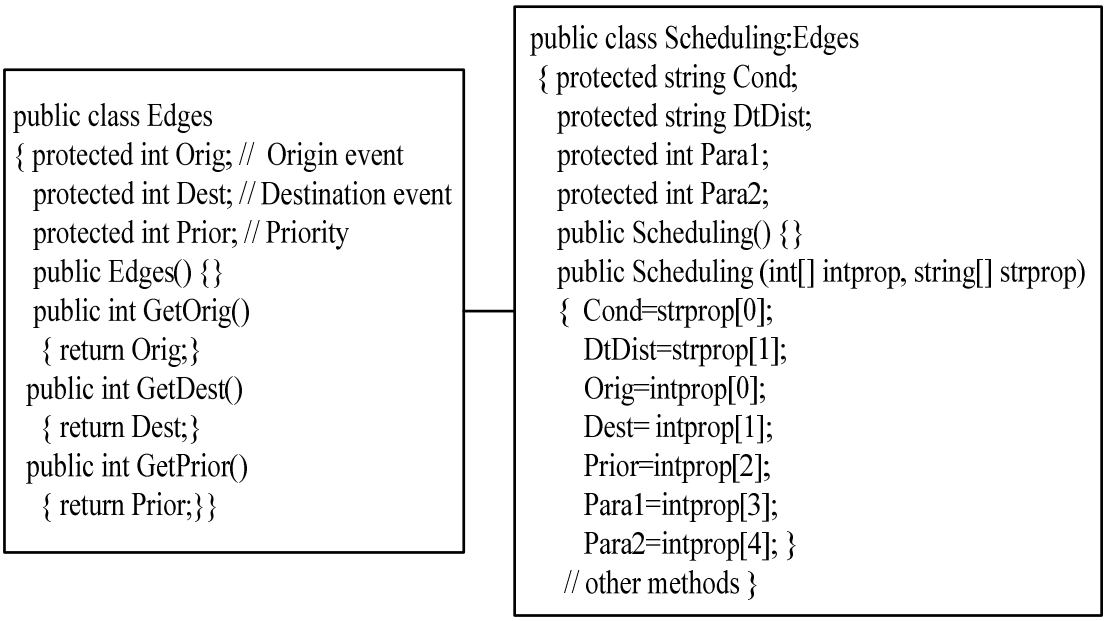

Fig. 7.8: The C# Edges class and Scheduling class

The objects instantiated from these classes are organised in arrays. The following arrays are defined to read the diagrammatical and modelling data from the properties of the Visio shapes:

SV[ ] : One-dimensional array of statevariables objects. Statevariables is a class that defines the properties of the state variables and the methods to set and get their values.

ST[,] : Bi-dimensional array of Etransit objects. Etransit is a class that defines the methods for setting and getting the state variables' changes triggered by each event.

SC[], PSC[], CA[] and PCA [] : One-dimensional arrays of Scheduling, PScheduling, Cancelling and PCancelling classes that define the properties and methods to set and to get the properties of the edges of the Event Graph.

Other modelling data, e.g. the run length of simulation, the number of replications and the seed for generating random numbers are read into field variables of the class simparameters.

7.2.2. EVENT-BASED SIMULATION: TOP-DOWN AND ALGORITHM

The DotNetSim simulation engine implements an event-based simulation executive that, using a Top-Down approach, can be expressed as follows:

1. Read diagrammatical and modelling data from Visio™ into C# data structures
2. Execute each of the replications specified by the modelling data
   2.1. Initialise the simulation time, state variables and calendar of events
   2.2. Run the replication for the time length specified by the modelling data
      2.2.1. Make a TO DO list for the current simulation time
      2.2.2. Execute the TO DO list
         Evaluate the head-condition
         Perform state transitions
         Output data on the executed events
         Schedule following events
      2.2.3. Delete from the calendar the events executed at this simulation time
      2.2.4. Update simulation time
   2.3. Output state variables

In addition to the data structures described above, are the CALENDAR and the TODO lists:

CALENDAR [n,4]: A bi-dimensional array which records the events that are scheduled for the future. It records two types of events: the events scheduled to occur in the future and the events that were scheduled for the past but did not occur because the head-condition did not hold.

The structure of these records contains the following fields:

- time: time at which the event is to occur
- event-to-execute: the number of the event that is scheduled for this time
- precedent-event: the number of the event that has scheduled the event-to-execute. This field allows the evaluation of the data that passes over the edge, e.g. head-condition and parameters as shown in Fig. 7.9.

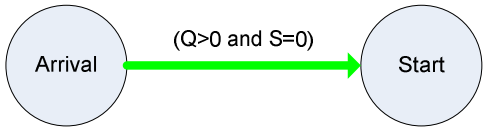

Fig. 7.9: The head condition of the event Start passes over the scheduling edge that connects this event to its scheduler

- executed: indicates whether the event was executed at the current simulation time

TODO [n,4]: A bi-dimensional array which records the events that should or could occur at the current simulation time, either because this is their scheduling time or because their execution was not possible in the past.

The structure of the record contains the fields:

- event-to-execute: the number of the event to be executed
- precedent-event: the number of the event that scheduled the event-to-execute. This field allows the evaluation of the data that passes over the edge, e.g. head-condition and parameters
- calendar-row: The row of the record of the event-to-execute
- time: indicates whether the event is scheduled for now or could not executed in the past.

Detailing further this top-down operation, the simulation executive algorithm can be

expressed in a C#-based pseudo-code as follows:

```
Algorithm Event-based Simulation
{Declare and initialise the data structures as described in
previous section;
Read data from Visio's custom properties into the C# data
structures;
foreach (replication)
   {Initialise CALENDAR, state variables, clock;
   while (clock < Runlength)
      {Make TODO list;
      foreach (TODO record)
         {Truthvalue=Check the head-condition;
          If (Truthvalue==true)
            {Perform state transition;
             Output data on executed event;
             Schedule following events }}
      Delete TODO records from Calendar
      Update simulation time;}
   Output the current values of the state variables}
```

This algorithm calls a number of sub-algorithms, each of which provides a part of the functionality of the simulation executive. The algorithm was implemented as a C# Windows application and the sub algorithms as C# and VB.NET class libraries. All of the latter were developed as components that are added as references to the former. Thus, the simulation engine consists of the following components:

- Simeng: The C# windows application which implements the main algorithm and invokes all other components.
- ReadModel: A set of classes that gets access to the Visio™ application object model and reads into C# data structures the custom properties of the events and

the edges of the current Event Graph. Also, it reads the custom properties of the diagram's first page where other modelling data is stored.

- Calendarising: A class that contains methods to operate the CALENDAR and TODO lists. Thus, it initialises, inserts and deletes records into and from the CALENDAR; it extracts from the CALENDAR the events that are to execute at the current simulation time; and it updates the simulation time to the nearest time of the CALENDAR when there are events scheduled.

- Conditions: A class that contains methods to look for the head-condition of an event; splits the conditional proposition into conditions; evaluates the truth value of each condition and of the whole proposition.

- Transitions: A class that contains the methods to read the state changes of each event and perform the state transition as soon as the events execute.

- FutureEvents: A class which contains the methods to schedule the events that follow the one which has just executed. If the corresponding delay time is not deterministic, it invokes the methods of a VB.NET class to generate the occurrence time for the next events from the uniform distribution and the negative exponential distribution.

- Outputting: A class that contains the methods to write the simulation results, as they are produced, to cells of an Excel workbook.

7.2.3. DOWN-STREAM INTEGRATION: WRITING DATA ONTO EXCEL™

When running, the simulation engine gets access to the Excel object model by instantiating the Excel™ application. A workbook is created, based on the

SimOutAna template described in detail in chapter 8. Fig. 7.10 lists the C# statements to instantiate Excel™ and create a new workbook based on that template.

```
Excelapp excel=new Excelapp();
excel.Visible=true;
excel.Workbooks.Add("C:\\SimEngine\\exceltemplates\\SimOutAna.xlt");
```

Fig. 7.10: C# statements to instantiate Excel™ and create a new workbook based on the SimOutAna template

During each simulation replication and as the events are executed, a record of the executed event and corresponding execution time is written in this workbook. At the end of each replication, the values of the state variables are also written in the workbook. Fig. 7.11 lists the C# statements that write into, an Excel range, the time, in milliseconds, that a replication took to run.

```
int Env=Environment.TickCount;
Worksheet ws = (Worksheet) excel.Worksheets.get_Item("Replications");
Range R=ws.get_Range("A1",Missing.Value);
R.Cells[1,1]=Environment.TickCount-Env ;
```

Fig. 7.11: C# statements which writes in A1 of the worksheet Replications the time that a replication took to run

Eventually, the simulation engine ends by quitting the Excel™ application. The user can then carry on refining the model within the DotNetSim modelling environment.

7.3. COMMENTS ON THE IMPLEMENTATION OF THE SIMULATION ENGINE COMPONENT

The implementation of DotNetSim's simulation engine prototype provides useful insights into the interoperability between components written within distinct packages and in different programming languages. Thus, there are two major areas of interest:

(i) The first is the simulation engine itself, which consists of components written in C# and VB.NET;

(ii) The second splits into two sub-areas:

(ii.1) The input area where the simulation engine reads the modelling data captured within the Microsoft Visio™;

(ii.2) The output area where the simulation engine displays the simulation results within the Microsoft Excel™.

The first area refers to the multilingual interoperability within the .NET Framework and the latter to the interoperability across packages based on different technologies.

7.3.1. CROSSING LINGUAL BARRIERS WITHIN THE SIMULATION ENGINE

The simulation engine consists of C# and VB.NET components that interoperate as if they were written in the same programming language. As explained in chapter 4, .NET components may instantiate and invoke methods of types implemented in other .NET languages as if they are their own, regardless of the language in which they are written in. Thus, for example, the class randoms which generates random numbers from the Uniform and Negative Exponential distributions is written in VB.NET, instantiated by the C# Simeng and its methods invoked by the C# FutureEvent to generate the occurrence times of the next events. Fig. 7.12 shows the class VBrand.Vb with a field variable rnum of type Random, a constructor, and the functions getRandunif and getRandneg which return random numbers.

VBrand.Vb

```
Public Class randoms
 Private rnum As Random
 Public Sub New(ByVal s As Integer)
   rnum = New Random(s)
 End Sub
 Public Function getRandunif(ByVal a As Integer, ByVal b As Integer) As Integer
   getRandunif = rnum.Next(a, b)
 End Function
 Public Function getRandneg(ByVal a As Integer) As Integer
    getRandneg = Math.Round(-a * Math.Log(rnum.NextDouble(), Math.E), 0)
 End Function
End Class
```

Fig. 7.12: randoms is a class written in VB.NET that defines the generation of random numbers given two distribution functions

By including a reference to this class (see Fig. 7.13), the using directive tells the C# Simeng where to find the randoms class, releasing it from requiring a reference to its fully qualified name.

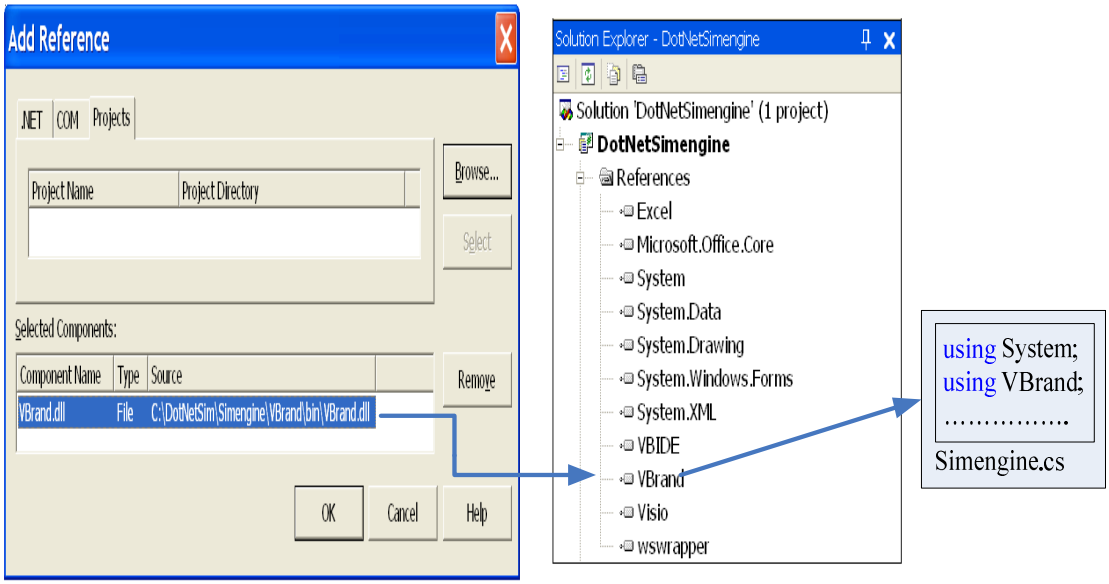

Fig. 7.13: A reference to the VBrand.VB class was added to the simeng.cs

The C# Simeng can now instantiate the randoms class and pass the instance as an argument to the corresponding parameter of the FutureEvents method as shown in Fig. 7.14.

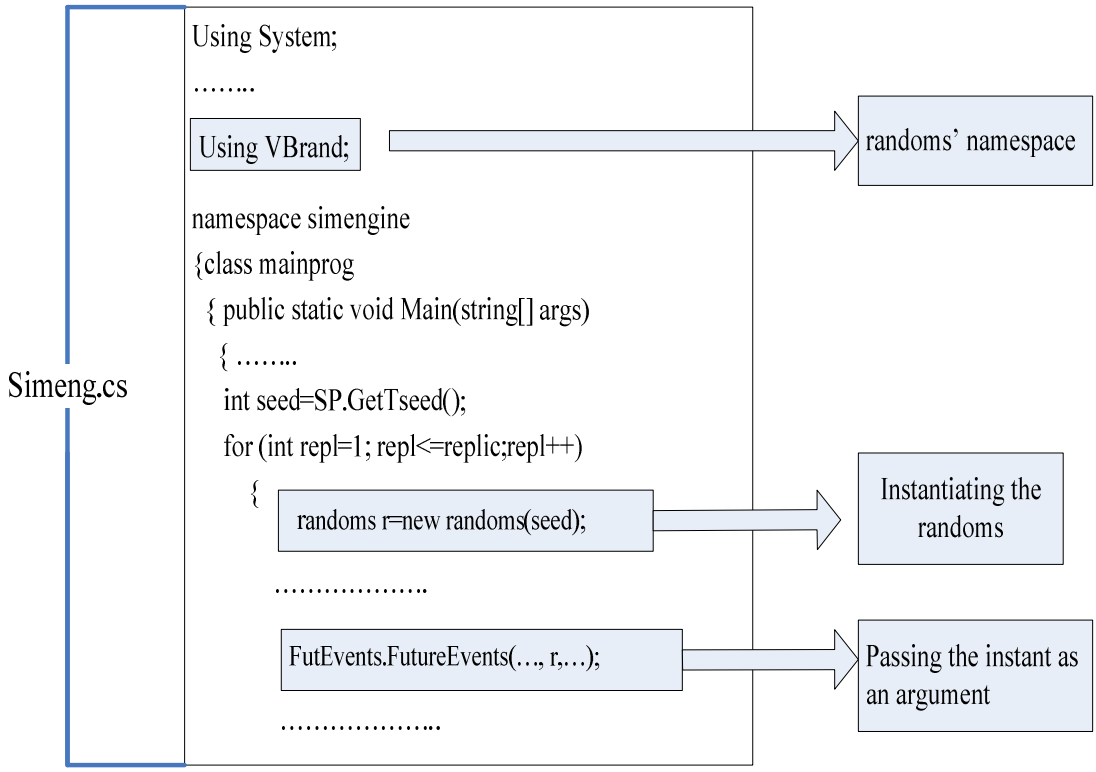

Fig. 7.14: The randoms class is instantiated in C# as if it were also written in C#

The FutureEvents method, in its turn, invokes the randoms class methods on this instance to generate the occurrence time of the next event, as Fig. 7.15 illustrates.

```
nexttime=(int) r.getRandneg(SC[i].GetPara1());
```

Fig. 7.15: The method getRandneg is invoked on the instance r of the randoms class

This demonstrates that the language barriers among .NET programming languages were, in fact, pulled down at the logical level. Within the .NET Framework, the selection of components to assemble into a single application depends more on the functionalities they provide and less on the programming language. The implementation language is only important because of the programming expertise of the developer. This widens the spectrum of prefabricated simulation components that can be selected, modified and assembled to produce simulation tools or solutions.

7.3.2. CROSSING PACKAGES BARRIERS BETWEEN MICROSOFT .NET AND MICROSOFT OFFICE

The simulation engine crosses the .NET Framework to read from and write data to Microsoft Office applications. It needs only an object that points to the active or a new instance of these applications in order to be able to manipulate the objects they expose. This is straightforward but the simulation engine uses the C# language to handle the objects that we are used to manipulating in VBA. As the two languages are syntactically and semantically different, they invoke the properties and the methods on the same objects differently with the same results. The major difference is that the C# subset which deals with those objects is restricted. For example, it only provides constructors and methods for the whole set of parameters while the corresponding VBA primitives allow optional parameters.

Fig. 7.16 illustrates how we have initially dealt with this by assigning to each optional parameter the Missing.Value which is provided by the System.Reflection library to replace at runtime the missing parameter by its default value.

```
using Missing = System.Reflection.Missing;
using Excelapp=Microsoft.Office.Interop.Excel.Application;
……….
Excelapp excel=new Excelapp();
String Exname=”filename.xls”;
excel.ActiveWorkbook.SaveAs(@Exname,Missing.Value, Missing.Value,
            Missing.Value, Missing.Value, Missing.Value,
            Microsoft.Office.Interop.Excel.XlSaveAsAccessMode.xlExclusive,
            Missing.Value, Missing.Value, Missing.Value, Missing.Value,
            Missing.Value);
```

Fig. 7.16: The method SaveAs requires the enumeration of all its 11 parameters

As this is difficult to write and to debug, a C# class library, named WSWRAP, was written to wrap this method and other similar methods into others whose invocation only requires the known parameters. For example, the SaveAs method was wrapped into the savebook method as listed in Fig. 7.17.

```
using System;
using Excel = Microsoft.Office.Interop.Excel;
using System.Runtime.InteropServices;
using Missing = System.Reflection.Missing;
namespace WSWRAP
{ public class WExcel
  { public static void savebook(string excelname, Excel.Workbook workbook )
      {object[] o=new object[12];
      o[0]=(object) excelname;
      for (int i=1; i<=11; i++)
        {o[i] = Missing.Value;}
     workbook.SaveAs(o[0],o[1],o[2],o[3],o[4],o[5],Microsoft.Office.Interop.
     Excel.XlSaveAsAccessMode.xlExclusive,o[7],o[8],o[9],o[10],o[11]);}}
```

Fig. 7.17: The Savebook method wraps the SaveAs so as to require only two parameters

As WSWRAP also contains a wrapper for the VBA's InputBox statement, saving the active workbook within the simulation engine becomes as simple as Fig. 7.18 shows:

```
string Exname= WExcel.inputbox(excel,"Name for this Workbook")+".xls";
WExcel.savebook(Exname, excel.ActiveWorkbook);}
```

Fig. 7.18: Saving the active workbook from within a C# application which references the WSWRAP

The invocation of these methods on Excel™ or Visio™ objects can be improved by resorting to tools such as the *ExcelXmlWriter* [1], which is a lightweight wrapper of the Excel's object model based on the Xml workbook schema.

Once the syntactic and semantic differences between the two languages were overcome, reading data from Visio™ or writing data to Excel™ from within the DotNetSim simulation engine became as straightforward as it would be if we were addressing object models in the same packages.

7.3.2.1. SOLVING SHORTAGES OF COMPUTER RESOURCES

Due to reasons that were not investigated, the manipulation of Excel objects from within C# programs generates shortages of computer resources, namely the main memory. Thus, for example, writing data on a number of Excel cells from within a C# program throws an exception due to a lack of runtime computer resources. The maximum number of cells which a single object reference can handle depends obviously on the computer system. To extend this upper limit, we have experimented with two implementations:

I. The object variables are regularly reset, e.g. after writing 250 rows.

   The C# code shown in Fig. 7.19 illustrates this implementation, resetting the worksheet and the range variables after every group of 250 rows has been written. The worksheet variable is always reset to the first sheet and the range variable is successively offset 250 rows below. This example writes data into 5000x20 cells of the first worksheet.

```
…….
namespace simengine
{class writingExcel
  {public static void Main(string[] args)
    { const int trows=5000, tcols=20, rowsgroup=250;
      int s=1,tr=0,ro=1, writtenrows=0;
      ……..
      Excelapp excel= outputlog.OpenExcel(); // open excel for output
      Worksheet ws=(Worksheet) excel.Worksheets.get_Item(s);
      Range R= ws.get_Range("A1",Missing.Value);
      try
        { while (writtenrows<trows)
          { if (ro>rowsgroup)
            { ws=(Worksheet) excel.Worksheets.get_Item(1);
              tr++;
              R= ws.get_Range("A1",Missing.Value).get_Offset(tr*rowsgroup,0);
              ro=1;}
              for (int j=1; j<tcols; j++)
                {a[ro,j]=….;
                  R.Cells[ro,j]=a[ro,j];}
              writtenrows++;
              ro++;}}
      catch {}; }}}
```

Fig. 7.19: Resetting the Worksheet and Range variables after writing each group of 250x20 cells

II. The data is written on a number of worksheets

The C# code in Fig. 7.20 illustrates this implementation, setting the worksheet variable to a new sheet after every group of 250 rows has been

written. The range variable is set to the same range in each sheet.

```
……..
namespace simengine
{class WrittingExcel
  {public static void Main(string[] args)
    {const int trows=5000, tcols=20, rowspersheet=250;
      int s=1, ro=1, writtenrows=0; string wsname;
      ……
      Excelapp excel= outputlog.OpenExcel(); // open excel for output
      Worksheet ws=(Worksheet) excel.Worksheets.get_Item(s);
      Range  R= ws.get_Range("A1",Missing.Value);
      try
      {while (writtenrows<trows)
        {if (ro>rowspersheet)
          { s++;
            ws=(Worksheet) excel.Worksheets.Add(Missing.Value,Missing.Value,1,Missing.Value);
            wsname="Sheet"+s.ToString(); ws.Name=wsname;
            ws=(Worksheet) excel.Worksheets.get_Item(wsname);
            R= ws.get_Range("A1",Missing.Value);
            ro=1;}
        for (int j=1; j<tcols; j++)
          { a[ro,j]=….;
            R.Cells[ro,j]=a[ro,j];}
        writtenrows++;
        ro++;}}
      catch {}; }}}
```

Fig. 7.20: Each group of 250x20 cells is written in a separate worksheet

These experiments indicated that, using the same computer resources, the second implementation allows more data to be written in Excel™ from within a C# program. However, the former was implemented in the DotNetSim simulation engine as this prototype does not aim to deal with long simulation runs, thus large numbers of Excel

cells are not to be written from within C#. Were the DotNetSim approach to be implemented in software to be used by others, it would clearly be important to take a proper approach to this problem rather than resorting to the workarounds described here.

Additionally, in simulations with long run lengths, the time which is taken to write a huge number of Excel™ cells from within C# may also be a relevant constraint. Other options to output the records of the executed events could be implemented instead. For example, volatile data structures could be used to store these records which could be partially output into Excel cells.

7.3.3. COMMENTS ON FURTHER ISSUES

The implementation of the DotNetSim simulation engine shows that the integration of components written within the Microsoft Office applications, with components of the Microsoft applications with the .NET Framework, is generally effective and straightforward. The C#-based simulation engine instantiates the object models of Visio™ and Excel™ and reads and writes their properties and invokes methods on them as if they were its own.

This prototype implements an event-based simulation executive but other simulation worldviews can be substituted. .NET components can run other simulation executives and integrate them upstream and downstream with modelling and output analysis components developed within Microsoft Office applications.

However, this implementation has also highlighted that the integration is only straightforward when dealing with the built-in object models of the Microsoft applications. By contrast, the intra-packages borders are especially problematic when we deal with the user-defined classes or other dynamic data structures. For example,

passing an array from Excel™ to Visio™ is possible, as shown in Fig. 7.21, but reading the same data from an Excel range is simpler because this is accessed from within Visio™.

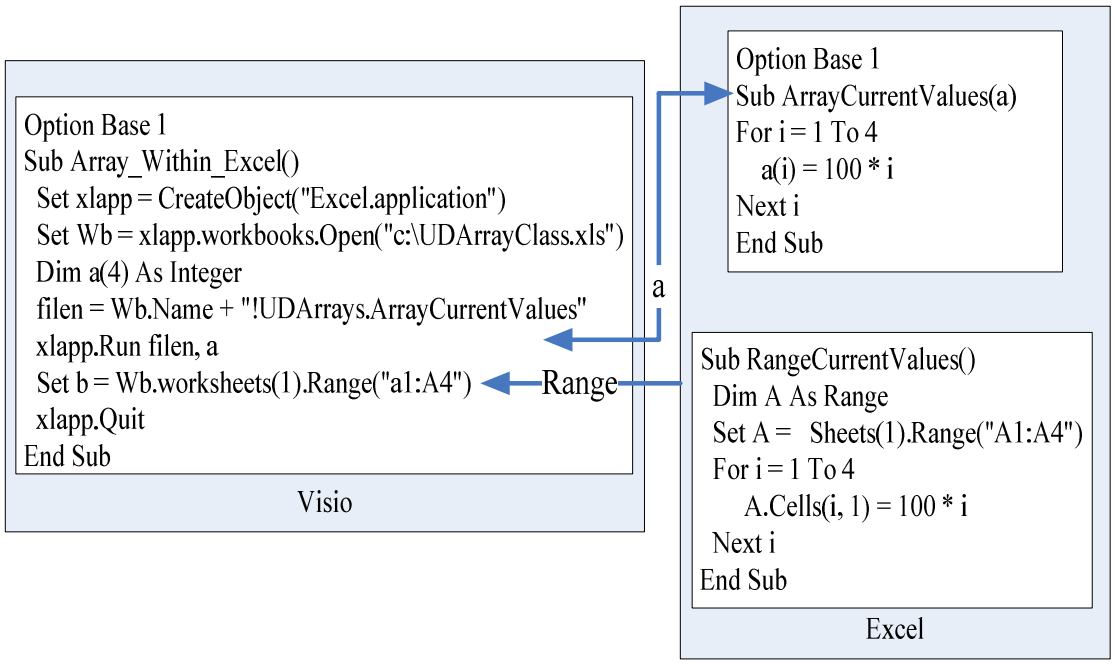

Fig. 7.21: Passing an array from Excel™ to Visio™ and reading the same data from an Excel range

Also, setting the properties of an Excel user-defined class from within Visio™ is possible but far from the object-oriented paradigm, as illustrated in Fig. 7.22.

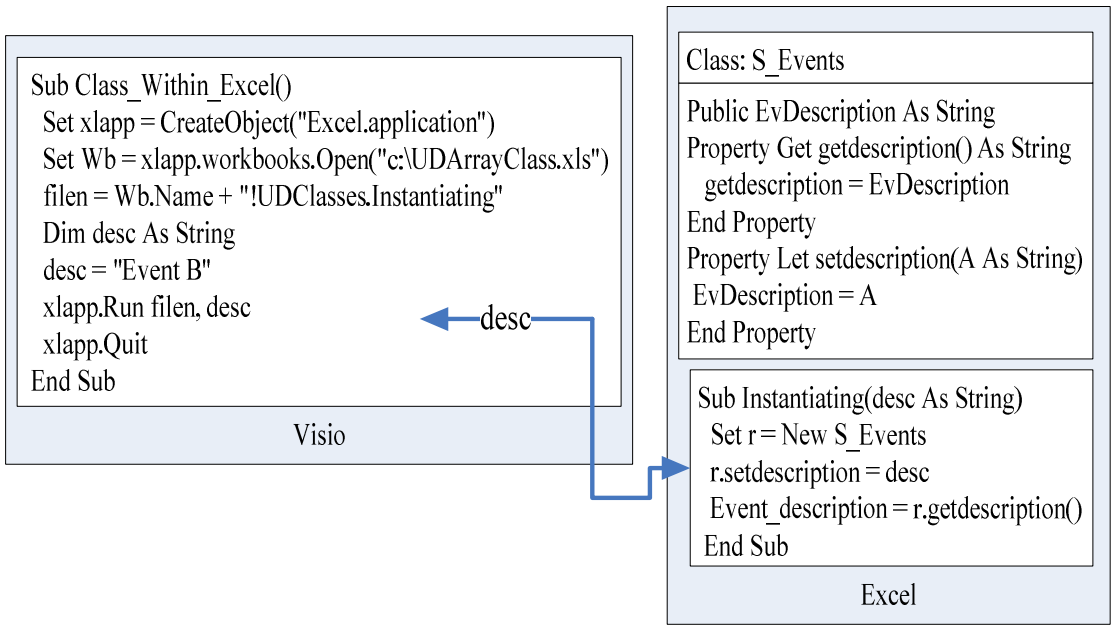

Fig. 7.22: Setting the description of an instance of the class S_Events class defined within Excel™

This difficulty increases when a .NET application, such as the simulation engine, needs to manipulate a dynamic data structure defined within an application of the Microsoft Office. Fig. 7.23 illustrates how a C# component reads a variable defined in Excel™. Reading an Excel range by a C# program is simpler because it consists of applying a method on a built-in object.

These are some examples of the difficulties that may arise while integrating components written in .NET languages with others written in Visual Basic for Applications. The integration primitives provided do not entirely bridge the gap between the former, which are fully object-oriented, and the latter, which is a procedural language that implements a powerful object model. Thus, wrappers have to be developed to interface the manipulation of some data structures. These are not required between .NET languages.

```
using System;
using System.Windows.Forms;
using Microsoft.Office.Core;
using Microsoft.Office.Interop.Excel;
using Excelapp=Microsoft.Office.Interop.Excel.Application;
using System.Runtime.InteropServices;
using Missing = System.Reflection.Missing;
using WSWRAP;
namespace simengine
{class mainprog
  {public static void Main(string[] args)
    {Excelapp excel=new Excelapp();
     WExcel.openbook(excel, "c:\\UDArrayClass.xls");
     Worksheet ws=(Worksheet) excel.Worksheets.get_Item(1);
```

Reading a range from Excel	Range R=ws.get_Range("A1:A4",Missing.Value); for (int i=1; i<=4; i++) {Range R1=(Range) R.Cells[i,1]; ……..

Reading a variable from Excel	object a
	a= WExcel.runproc (excel,"UDArrayClass.xls!CandExcel.VarFromExcel");
	int b= Convert.ToInt16(a)+100;
	……..

Fig. 7.23: Reading a Range from within the simulation engine

The DotNetSim prototype suggests that extensions to its implementation would benefit from the creation of libraries of wrappers which assign object-oriented behaviour to the fine-grained components that support procedural intra-packages interoperability. Simulation software developers would therefore start at a higher level of abstraction, and the solution builders and the users could swap entirely to the object-oriented paradigm in order to select, modify and assemble the components.

7.4 THE CHAPTER IN CONTEXT

This chapter describes and comments on the simulation engine of DotNetSim, which prototypes the implementation of an event-based simulation executive that runs, over time, the models devised within the DotNetSim modelling environment. This .NET-based component OO integrates with the other coarse-grained components of the DotNetSim prototype to read the application logic of the DE model captured within the Visio-based modelling environment and to place the simulation results on the Excel-based output analysis environment.

The next chapter discusses the implementation of the DotNetSim output analysis environment, which prototypes an Excel-based component for analysing and reporting the results of running multiple replications of a DE model over time.