4. SIMULATION SOFTWARE: CURRENT EXPERIENCES AND THE DOTNETSIM PROJECT

CHAPTER OVERVIEW

The successive advances in computer science have constantly increased the

expectations of the users and developers of application software, simulation software applications in our case. The early expectations and concerns with the proper processing of the input data were replaced by concerns over the portability, usability, extension and integration of software. Nowadays, correct functioning, portability and ease of use are taken for granted. It is also taken for granted that simulation packages are extensible to meet specific needs and that data can flow in and out of different packages. Current expectations and concerns are diverse, but focus mainly on the ease and speed of customisation. However, it is likely that, some time in the future, developers, builders or end users may want tools that let them select, modify and assemble only the functionality that each simulation solution requires. They may also aim to have the capability to develop the tools for implementing the functionality that they need. Thus, users may expect to be able to select components, customise the functionality which they provide and assemble them, regardless of the environments within they were developed.

To meet these possible expectations, simulation software has, on the one hand, to be fully object, component or layered-oriented so as to allow functionality to be added or removed, according to the specific simulation model. On the other hand, integration-friendly frameworks have to be used in order to enable interoperability across platforms, packages and programming languages.

This chapter discusses some examples of simulation software developers who have already changed their development strategy in order to meet current expectations. The chosen examples are among a small number of software applications which integrate simulation and generic software tools to provide quasi-standard user interfaces or facilitate customisation. The chapter also devises an alternative vision for simulation software which relies on a vertical integration of

prefabricated, self-contained, self-healing components to implement a 'generalise-specialise' development strategy.

The DotNetSim project is then introduced to investigate how the latest Microsoft integration technologies can contribute to the development of simulation software which fits in with the sketched vision for the future. This is done by prototyping software for simulation which links, into a single application, software components written in different programming languages and developed within distinct packages. The technological background is briefly described and the DotNetSim prototype is outlined by presenting its functional structure and its coarse-grained components.

4.1. CURRENT EXPERIENCES IN SIMULATION SOFTWARE:

In the early days of computer simulation there was little concern with ease-of-use or with how simulation software might be integrated with other applications. In many ways, this was no different from other software applications – users were just pleased to have something that worked. However, the increasing portability of applications to run on various computing systems has stimulated the inter-operation and integration of applications. At the simplest level, this might involve the straightforward transfer of data from a simulation program to, say, Excel™, for analysis of output data. At a more complex level, direct exchange of data may be possible between applications as they execute. That is, various approaches to inter-operation have been developed to meet the increasing expectations of the users of simulation software. This has enabled simulation software vendors to extend their products as 'workbench tools' [15], i.e. sets of built-in generic and simulation tools which encapsulate functionality for graphical modelling, model execution, output analysis and the import and export of data to common applications such as Excel™, Access™, SAS™and SAP™. At the

core of these workbench tools, however, are monolithic simulation applications that can function on a standalone basis and were probably designed for that purpose. These function well in well-defined application domains and enjoy significant sales.

There is no doubt that these existing tools have met a market need – as exhibited by continued sales. However, as demands for increasing integration are joined by a wish to see increasing customisation, the approach may be reaching its limit. Though existing packages do allow some degree of customisation, they were, in general, not designed with that idea in mind. Hence, even if customisation is possible, it can be difficult. Nevertheless, there are exceptions which attempt to break this mould and the next section discusses the Promodel Process Simulator™, Micro Saint Sharp™ and the HighMast™ Modeling and Simulation Toolkit as examples of a small number of software applications that currently integrate simulation and generic software tools to provide quasi-standard user interfaces and/or to facilitate customisation.

4.1.1. PROCESS SIMULATOR™

The Process Simulator™ (2002-2005) [96] is a plug-in to Microsoft Visio™ which enables this generic diagram tool to act as a graphical interface to Promodel's discrete event simulation tools. It reuses Visio™'s standard capabilities to derive modelling tools that comply with the specifications of the Promodel™ simulation tools. These Visio modelling tools allow a process flowchart to be drawn and data to be collected on its entities, activities and resources in order to capture the logic and the dynamics of the process. Other modelling data, such as the simulation specifications, are also input within the Visio™ interface. A logic builder provides a programming interface that enables the definition of more complex or detailed processing behaviour. This is done in the user-friendly, but specific, Promodel logic language which resembles C and offers a small set of flow control statements. Also, processes and sub-processes

can be linked hierarchically through Visio's standard hyperlinks, i.e. a compound operation has a hyperlink to the file that models its sub-operations.

These Visio-derived models are then compiled into an appropriate format and run by the Promodel™ simulation engine. Whilst running a model, the Promodel™ runtime environment and its animation capabilities replace the Visio modelling environment. The simulation results are displayed in the Promodel's Output Viewer 3DR in a variety of graphical and tabular formats. These can be saved as text files and opened in generic applications such as Microsoft Excel™. As usual, what-if analysis is then available and different scenarios can be saved.

Fig. 4.1 sketches how a process modelled within Visio™ executes in the Promodel™ simulation engine and returns the results to the Promodel™ Output Viewer 3DR. The Process Simulator™ adds to Visio™ the modelling functionality required by the Promodel™ simulation engine and automates the conversion of the data formats at the input stage.

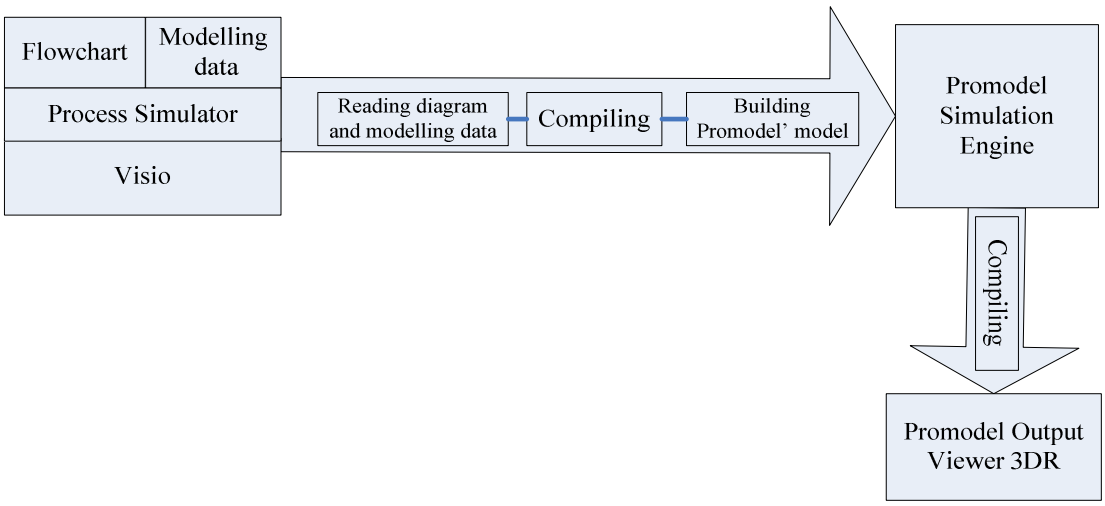

Fig. 4.1: Visio™ acting as an input interface to the Promodel™ simulation application

Diagrams drawn outside the Process Simulator™ can be automatically converted into Process flowcharts. A converter adds to these diagrams the properties required by the Process Simulator™ and assigns them suitable default values. For example, when

converting a network into a Process flowchart, the converter adds activity properties to the nodes and routing properties to the arcs.

This Visio-based graphical interface highlights the trend for Promodel™ developments to integrate its simulation engine with generic software in order to leverage its usability by compliance with human computer interaction "standards". It goes somewhat further than the provision of, say, links to Excel for output analysis that is provided in all commonly used simulation packages.

4.1.2. MICRO SAINT SHARP™

Micro Saint Sharp™, now on version 2.2., is a discrete event simulation application with a large number of tools for modelling and running processes represented by flowcharts. The first release of Micro Saint Sharp™ (2004) was, conceptually, a development of the Micro Saint Windows-based Simulation application. Apart from a new animation mode, Micro Saint Sharp™ uses C# as the basis of its simulation language, which is required for non-trivial applications. Being based on .NET, Micro Saint Sharp™ offers interoperable capabilities with widely-used applications which facilitates its customisation, strengthens its modelling capabilities and its interoperability with other applications [68, 69]. To achieve this, Micro Saint's core application was redesigned and rebuilt by using component-based paradigms and the .NET integration technologies. Additional features, such as the OptQuest™ and animation, constitute components that are plug-able if necessary.

The C#-based logic builder enables users to define the process logic and dynamics using a subset of this generic programming language. User defined functions can then be written within the package to extend its capabilities. Perhaps because C# is so powerful, Micro Saint Sharp™ handles only a restricted subset of the C# programming language with limited OOP features. However, built-in classes, such as

Model and Task, are provided which have their own built-in functions. The user may code methods for responding to events, i.e. beginning and ending effects. New objects with properties, i.e. containers of field variables, may be created by resorting to an object designer. However, proper inheritance is unavailable and polymorphism is limited to variant variables. Fig. 4.2 shows a C# function written within Micro Saint Sharp™ to route customers to a number of different servers.

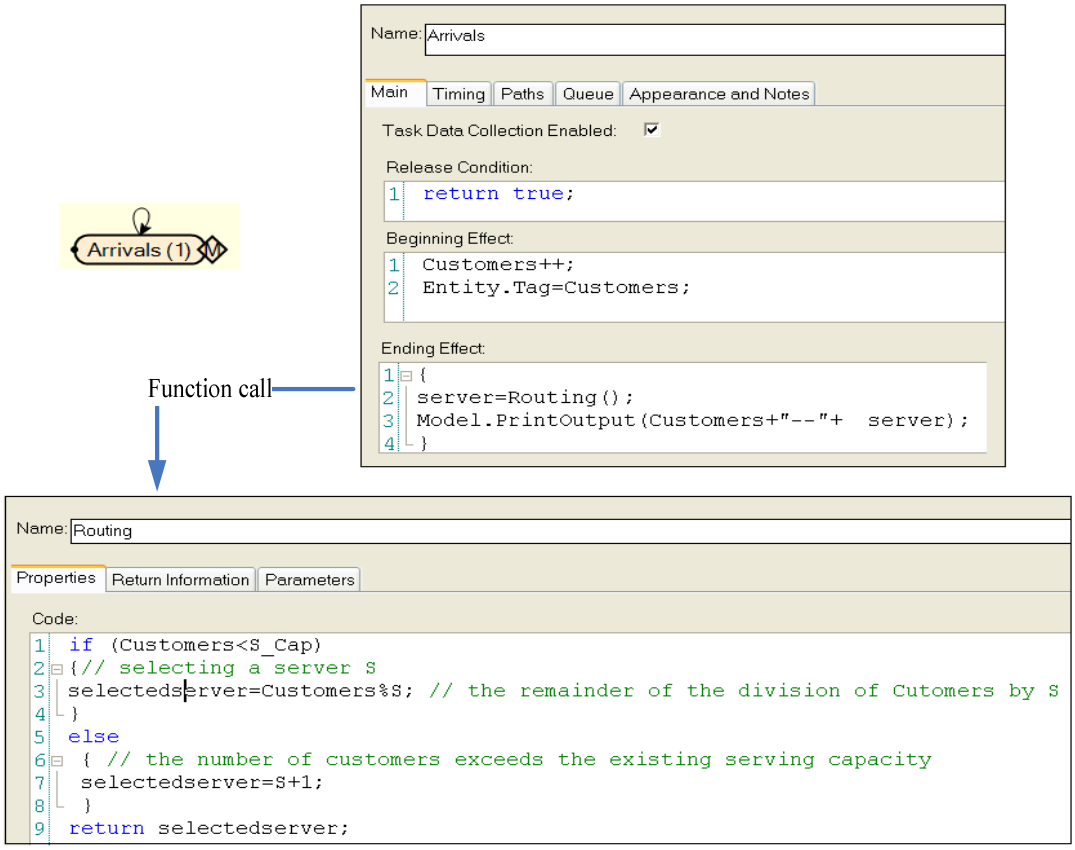

Fig. 4.2: User defined function written in C# for Micro Saint Sharp™

Micro Saint Sharp™ offers communication modules that simplify access to external data by providing built-in connectors for Excel™, Sockets, ADO, Text Files, ConsoleApplications, and Web Sites. Connectors to specific applications can be created in C# by resorting to .NET's networking classes and interfaces. A "Plugin Framework" is also provided to support the installation of user defined add-ins. This framework contains the signatures of the interfaces which the user must implement in

order to attach a DLL file written within the .NET Framework. Data exchange with other applications is also supported through TXT and XLS file formats for input and output. Fig. 4.3 shows how the value of cell "A2" of an Excel Workbook can be assigned to an Entity's attribute.

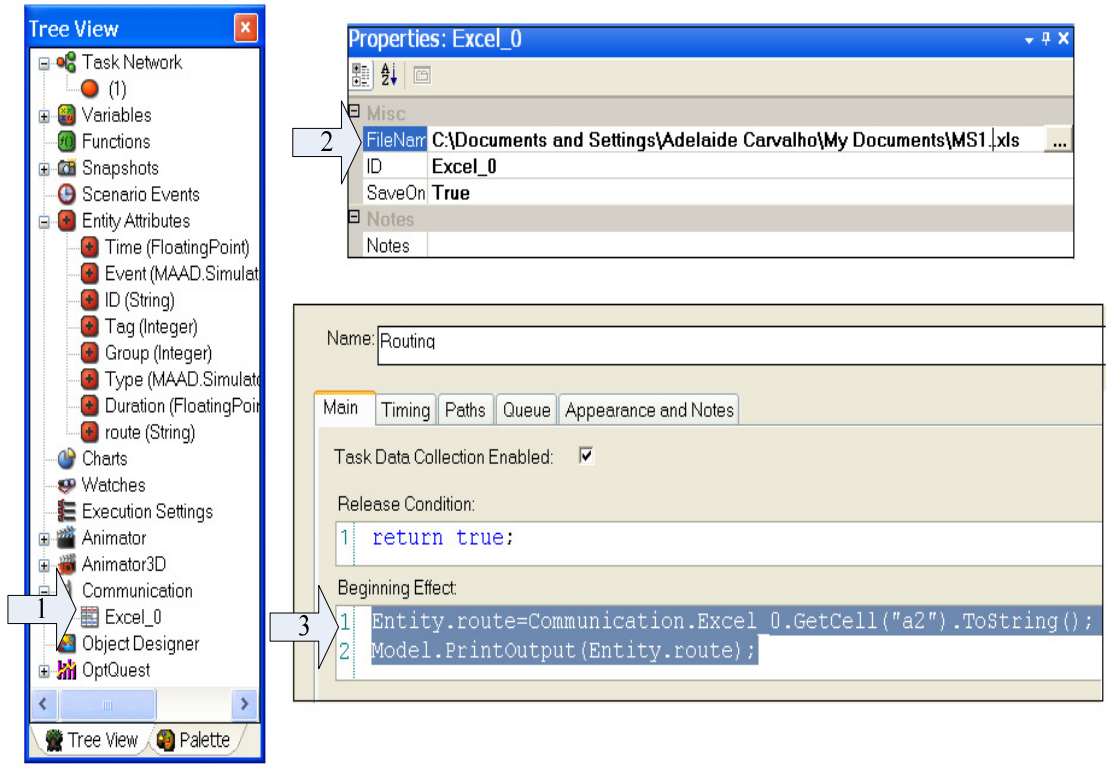

Fig. 4.3: On starting task Routing, the value of cell A2 of the file MSI.XLS is assigned to the attribute route of modelled entity

4.1.3. HIGHMAST™

HighMast™, Highpoint's Modeling And Simulation Toolkit, is a .NET–based modelling and simulation framework [14] which supports the gradual development of software applications for running discrete event and continuous simulation systems. It sits on top of the .NET Framework using a layer-oriented programming paradigm as shown in Fig. 4.4.

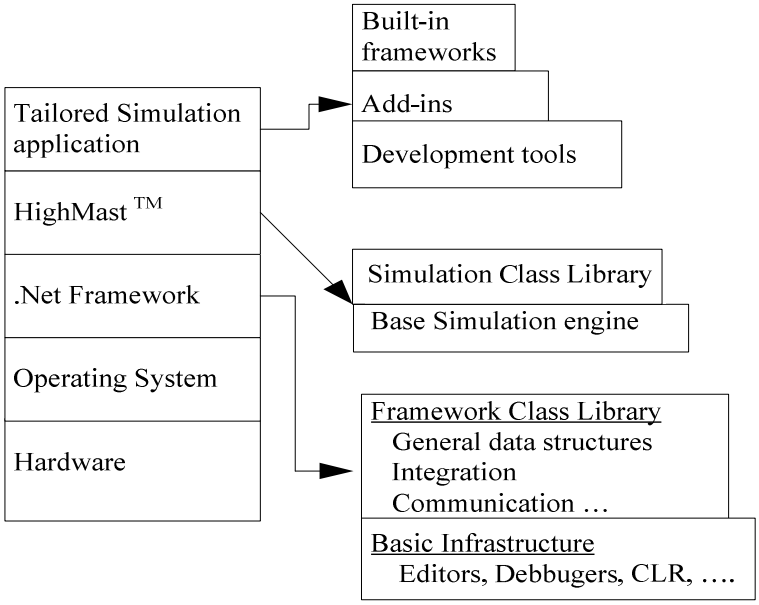

Fig 4.4: HighMast™ infrastructure and composition.

HighMast™ is a source-code based platform that uses the .NET environment's underlying conceptual principles and relies on its integration and interoperability capabilities to provide a platform for the development of tailored standalone and distributed simulation applications. Thus, it relies on object-oriented and component-based paradigms and dynamic binding mechanisms to allow simulation applications to be built in layers of software. Users may build specific simulation software solutions by integrating appropriate software packages, multi-lingual components and web services with a simulation engine and pre-built frameworks. Users can alter prefabricated components, write new ones and assemble the whole package by using general-purpose programming languages. The user's tailored simulation application can, in its turn, be constituted as a web service available for other applications.

HighMast™ itself is coded in C# and consists of a foundation simulation library and a base simulation engine. The foundation simulation library includes classes that implement simulation related features such as distribution functions, event generators, resource management and graph traversing. The base simulation engine consists of a model class and an executive class implemented separately (see Fig. 4.5). The

executive class runs the model by handling synchronous events (event-based simulation) and detachable or batched events (process-based simulation).

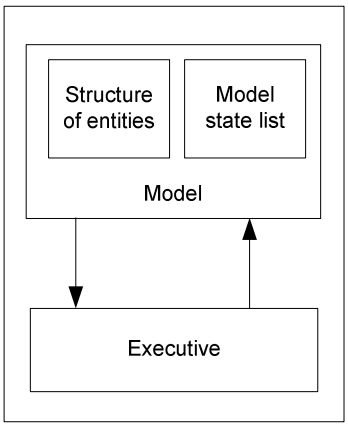

Fig. 4.5: Base Engine

HighMast™ also provides tools and data structures to facilitate the integration of third-party components. Tools such as those for database manipulation, graphics generation and reporting allow the retrieval and display of externally sourced data. Additional tools and micro-frameworks are becoming available as add-ins that can be installed according to the user's needs. Statistics logs, multi-rooted dictionaries of user-specified data structures and modelling expressions, instant snap-shot of the current running state, tree or tabular representations of object hierarchies and handlers of compiled queries at runtime are already available.

HighMast™'s source-code and layered based approach enable the development of frameworks for particular application areas. Such pre-built frameworks are being developed for modular supply chain models, bank teller models and product or service transformation models.

4.2. AN ALTERNATIVE VISION FOR SIMULATION SOFTWARE

These three examples highlight an emerging understanding that the mode of development for much of the current simulation software on the market may not be

well-suited to meeting demands for extensible simulation solutions. Hence the use of Visio™ as a customisable input tool (Promodel™), the .NET Framework (Micro Saint Sharp™ and HighMast™) and an encouragement of a plug and play approach (HighMast™). That is, emerging technologies such as .NET may offer a way out of the impasse presented by monolithic applications.

Furthermore, solutions to specific problems are increasingly demanded in ever shorter time [83]. The current response to demands for customised solutions often implies the development of new features or tools that provide the specific functionality needed at any specific moment. The new features are added to the simulation core applications following the interminable 'generalising-customising-generalising' development cycle (see chapter 3). This has led to huge monolithic applications, shown in Fig. 4.6, that are difficult to maintain, complicated to use and slow to customise. They also still have to be extended by addition of functionalities to respond to the needs of new solutions.

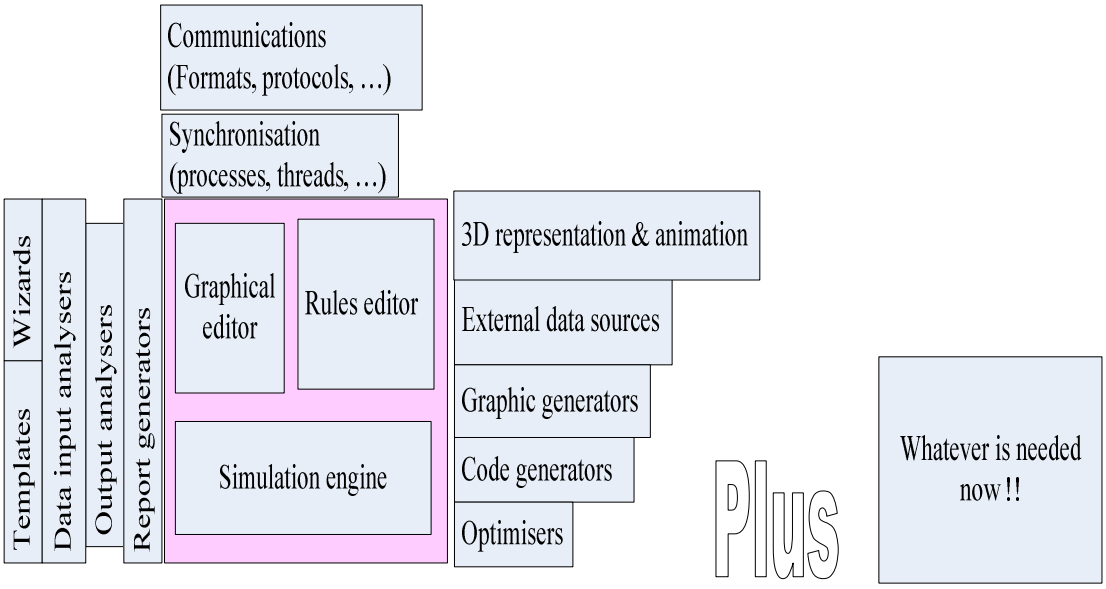

Fig. 4.6: Extension of simulation software to suit customisation of solutions

A different route may pursue the development of generic simulation software tools within frameworks that enable the derivation of progressively specific tools. In this

way, the development of simulation software would be vertically architectured, by deriving frameworks and libraries of simulation functionalities from the base classes following a 'generalise-specialise' development cycle. The different simulation worldviews' executives could, for example, derive from a base class that defines the properties and methods of a general simulation engine. Modelling would also consist of derivation of super models [83] until 'reasonably' detailed behaviour is captured in a sub-model. Simulation solutions would consequently be able to reference the tools needed at the required level of specialisation and would use them by instantiation or by deriving new tools and ultimately new solutions. Solutions and tools would therefore offer different modes of usage from black to white boxes.

The 'generalise-specialise' approach would also apply to the software tools used to develop simulation functionalities. The generic development environments used to produce the base simulation tools would give way to more specialised development tools in order to produce more specific simulation tools.

This vertical development approach would support the composition of simulation software solutions from prefabricated, self-contained, self-healing and light components [118, 82] that could be invoked remotely by users to select, modify and assemble into applications. It might create a route that leads, eventually, to an open market in simulation components, based on the Internet. This, in turn, might support the on-demand simulation software paradigm to build software solutions for each specific DES system. These components could be presented as web simulation services and software solutions would be assembled just-in-time with only the required functionality and could be paid for on a per use basis [118, 39]. That is, the customisation and assembly of components would replace package customisation and the developers and users of software would become mainly solution builders.

This possibility is supported by recent technological achievements and on-going Computer Science research into the development of integration frameworks and programming paradigms, which enable dynamic interoperability across computers, development frameworks, applications, programming languages and technological generations. This, linked to current investigations of dynamic ascertainment of the deployment environments [118], lays the foundations for on-demand software.

Fig. 4.7 is a time line representation of the past and future desirable approaches to develop simulation software solutions. In the interim, many alternative paths, among which may be included the above examples, are being explored to reform or revolutionise the development of simulation software.

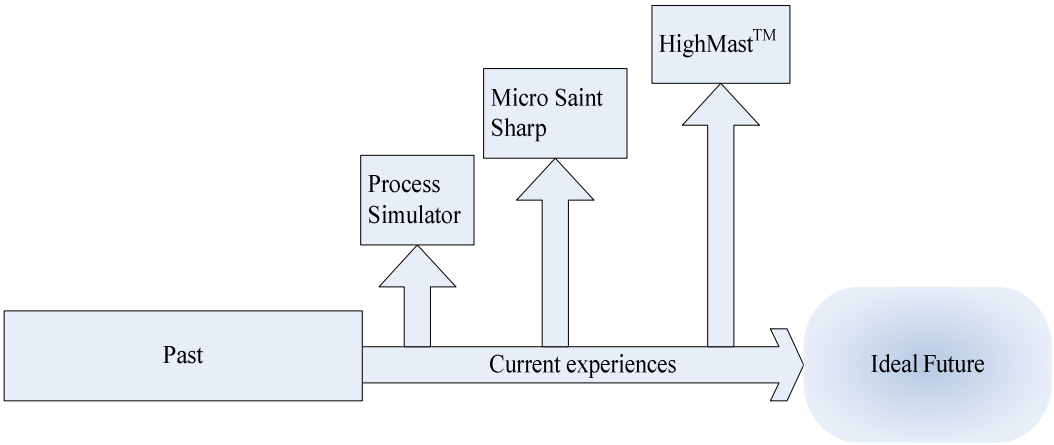

Fig. 4.7: Time line of the past, current and ideal approaches to the development of simulation software

The three examples of the ProModel™ Process Simulator, Micro Saint Sharp™ and HighMast™ illustrate different strategies to raise the usability, capability and

applicability of current simulation applications. All combine generic software tools with specific simulation applications, but at different stages of development. The Promodel Process Simulator™ is the nearest to the development approach pursued to date in most simulation packages. Promodel™ did not re-design its simulation package but added to it a very sensible and friendly user-interface for modelling DES systems. By designing a new user-interface using Microsoft Visio™, which is generic and widely used, it certainly accelerated the users' learning process, eased its use and probably expanded the applicability of the Promodel simulation. Also, the interface can be customised quickly if its implementation reuses the built-in capabilities of Visio™. However, it only currently captures models intended for process-based simulations.

The Process Simulator™ illustrates a very interesting use of Visio™, which constitutes an innovative input interface to an existing simulation package. Nevertheless, the conventional approach to simulation software resumes as soon as the simulator is invoked, requiring the data to be converted and saved in a different format ready for Promodel™. The integration of the Visio modelling environment and the simulation engine consists of a black-boxed compilation into a file, which is later automatically input into the simulation engine. The simulation engine and the output interface are still specific and, therefore, not changeable by the builder or the user of the simulation solutions. The user can, however, open the results files in other applications for further analysis.

Micro Saint Sharp™ moves further towards the ideal scenario, as its core application was re-designed within the .NET Framework so as to take advantage of object-oriented and component-based programming paradigms. Micro Saint Sharp™ bases its approach on these programming paradigms and their implementation in the

powerful C# language, which enables easier customisation and interoperability with other applications. Additional features are loaded only if required and new features can be written in C# and attached to Micro Saint Sharp™ as DLLs. Also, the built-in tools to interoperate with other applications can be extended by resorting to a "Plugin Framework" based on the. NET's networking classes and interfaces. This greatly increases its capability to connect several packages, including those for simulation, as well as its ability to model and run distributed systems. In addition, it provides a C#-based logic builder which allows the user to add more complex logic to the models. This is a very restricted subset of C#, but logic that requires more powerful programming primitives can be externally implemented in C# and invoked as DLLs.

Micro Saint Sharp™ illustrates an interesting use of C# to customise simulation tools and to promote interoperability among applications. Nevertheless, Micro Saint Sharp's user interfaces require refinement, especially the editors for defining more complex logic. Moreover, its applicability is confined to task-based simulation and its extension to other simulation worldviews depends on the unpublished architecture of the core application.

HighMast™ moves even closer towards the ideal scenario. It is itself a development framework that derives from the Microsoft .NET Framework and lies above it; i.e. its vertical architecture promotes the 'generalise-specialise' development cycle. Theoretically, further specialisation can be built on to the underlying framework. Its object and component orientation allow simulation tools to be derived from foundation libraries of tools to provide the functionality which a simulation model requires. This transfers to the user the capability of producing the functionality required by each simulation model requires, by combining and altering the existing source-coded tools.

HighMast™ offers event and process-based simulation executives, but all varieties of simulation worldviews can be implemented. Interoperability across computers, packages and programming languages is available through the Microsoft .NET Framework. In fact, HighMast™ proposes an entirely new approach to the development of simulation software applications. Theoretically, HighMast™ is very close to the ideal scenario, although it does not support the dynamic selection and assembly of components needed for on-demand simulation.

4.3. THE IDEA OF DOTNETSIM

The DotNetSim project aims to investigate the value of the new Microsoft integration philosophy for the progressive development of simulation software along the time line toward the ideal scenario discussed in section 4.2. The idea is to explore how far the integration of the .NET Framework with Microsoft Office applications can encourage DE modelling and simulation software towards fully object-oriented components that cross programming languages, packages and platforms to be linked in a single application and, if appropriate, deployed as web services. This approach is investigated across the entire requirements of a simulation application package including user interfaces, simulation executives and output analysis. This focuses on two main development issues:

(i) Data exchange between distinct software development environments through the instantiation of objects in order to apply the object-oriented programming paradigm, so as to replace the current creation of intermediate files and associated format conversions.

(ii) The integration of powerful simulation engines with widely-used packages in an architecture that supports the straightforward modification of modelling

and output analyses environments.

The DotNetSim project addresses three major fields and their interrelationships: Discrete Event Modelling, Integration Technologies and Discrete Simulation, as shown in Fig. 4.8.

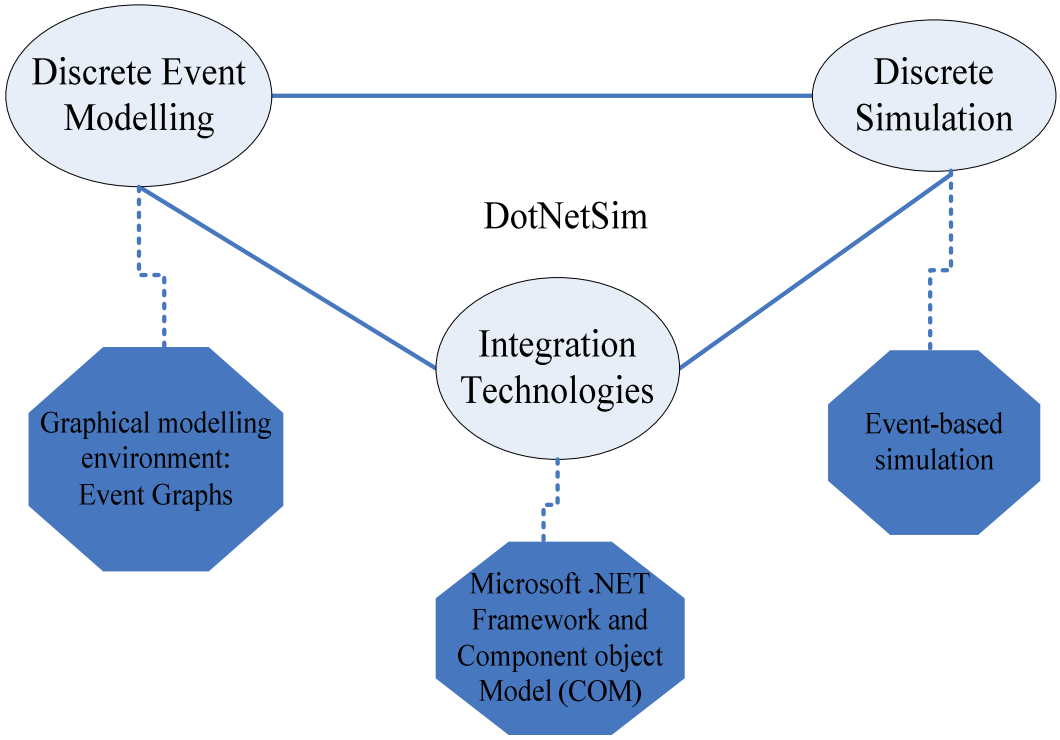

Fig. 4.8: Research aspects of DotNetSim

In DotNetSim, approaches to discrete event modelling and discrete simulation as described in chapter 2 give way to software components which integrate to provide the appropriate functionalities through Microsoft Integration Technologies. The remainder of this thesis discusses the development of the DotNetSim prototype. This integrates a graphical modelling environment developed within Microsoft Visio™ with a simulation engine developed within the Microsoft .NET Framework and an Excel-based output analysis environment. Our study employs an event-based simulation worldview and resorts to the Event Graph methodology [108] to capture the application logic and dynamics of the DE models. The basic concepts of Event Graphs are described in detail in chapter 5. It would obviously be possible to use or to

substitute other worldviews.

4.4. THE DOTNETSIM COMPUTING TECHNOLOGICAL BACKGROUND

The implementation of the DotNetSim prototype is technologically framed by the Microsoft integration paradigms. These materialise in the Microsoft Automation, the Microsoft .NET Framework and their integration which enable cross-application interoperability, among the Microsoft Office applications and the .NET Framework. This enables cross-lingual, cross-application and cross generation interoperability.

4.4.1. THE MICROSOFT AUTOMATION

Microsoft Automation [113], formerly called OLE Automation, is a COM-based integration technology which enables one application to expose its objects to be programmatically manipulated from within other applications. Thus, programs written in a programming language, such as Visual Basic for Applications, can enable the application to acquire the functionalities provided by another, provided that both support Automation. Most Microsoft applications, especially those included in the Office suite, support Automation. This allows, for example, Visio™ to get a reference to the Excel™ application, manipulate a range of cells, calculate some functions and return the results to a shape. Fig. 4.9 shows a VBA program of Microsoft Visio™ for instantiating Excel, computing the average of a range of cells and displaying the result in a Visio shape. After setting a reference to the Microsoft Excel 11.0 Object Library within the Visual Basic Editor for Visio™, this program instantiates a new Excel application object and browses the Excel object model to open the C:\Product.xls workbook and set the range variable R to the range A1:A10 of the Sheet1. Then, it computes the average of this range and assigns its value to the property Text of the

shape represented by the shape variable S.

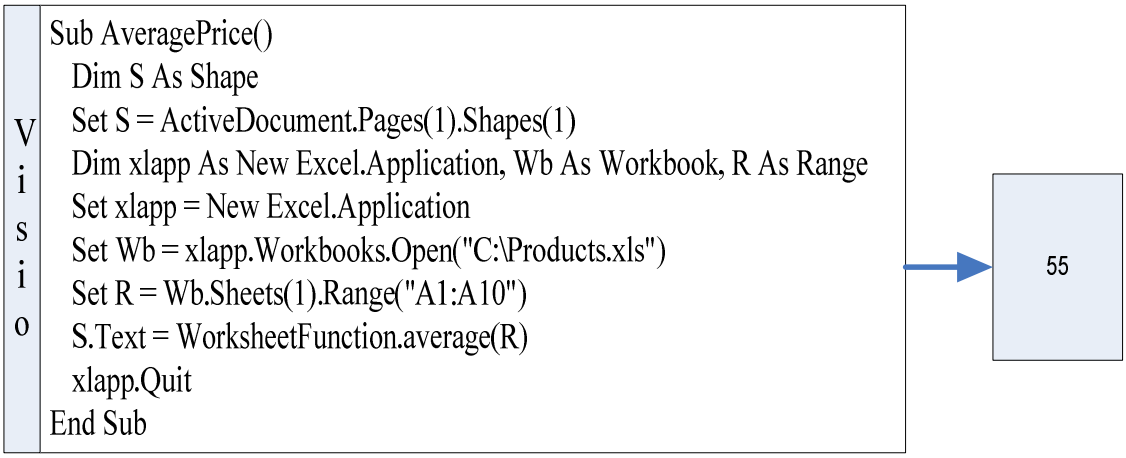

Fig. 4.9: Microsoft VisioTM commanding the calculation of the average of an Excel range. The reference to Excel™ is resolved at design time (early binding)

Automation blurs the boundaries between applications, so that one application can run the functionality provided by another application as if it were its own. The integration of the functionality of other applications may be completely hidden from the end-user. For example, the users of the above Visio drawing need not know that the value displayed in the square is computed within Excel™.

Thus, a native application, Visio™ in the above example, may integrate via Automation the functionality of a remote application, Excel™ in the same example. The remote application exposes its object model in order to enable the native application to programmatically instantiate its objects, get and set the properties and invoke the corresponding methods. This exposure may grant access to the object model at the design (early binding) time which requires reference to a file, for example a type library or object library, which describes the remote application's object model. The VBA program shown above assumes that a reference to the Microsoft Excel™ 11.0 Object Library was previously set [126].

Alternatively or complementarily, the binding to the remote application's object model may be delayed until runtime (late binding). In such cases, an interface

(IDispatch) provides the methods for identifying and locating the exact objects at runtime. Fig. 4.10 redesigns the VBA program shown in Fig. 4.9 to illustrate the reference of the Excel object model for late binding. The types of the generic object variables (xlapp, Wb and R) are resolved at runtime. For example, the CreateObject function looks for the latest version installed on the user's computer at runtime and instantiates the Excel application from it [126].

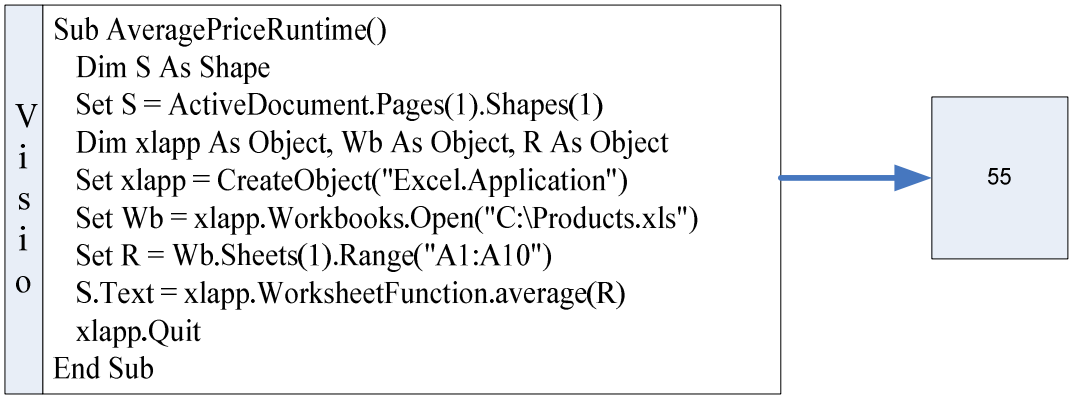

Fig. 4.10: Microsoft Visio™ commanding the calculation of the average of an Excel range. The object reference is resolved at runtime (late binding)

4.4.2. THE MICROSOFT .NET FRAMEWORK

The .NET Framework is the Microsoft response to the increasing web-orientation of the business sector [98] and the steady commercial trend towards on-demand business [39]. These trends lead inevitably to the need for software solutions built by the composition of prefabricated components, which may reside in the Internet. In this model, software solutions must, therefore, combine functionalities or services sourced in heterogeneous hardware architectures, operating systems, programming languages and packages. The high complexity of this process derives in part from the non-existence of standards for exposing, communicating and integrating the required functionality. Yet the integration programming is itself complex, due to the successive levels of abstraction required by the communication of conceptually different objects. This complexity increases when the dynamic integration of components is required

[118] as in the late binding example above.

The following brief description of the .NET Framework is based on the information published by Microsoft (2004, 2005) [73, 75], Platt (2004) [94] and Richter (2002) [98, 99, 100].

.NET Framework is a development platform that is intended to be integration-friendly, i.e. a platform which abstracts from the developer the mechanisms that wire software artefacts across machines, programming languages and technological generations [94, 98]. The integration should be achieved at higher levels than the binary communication of COM and DCOM (Component Object Model and its distributed version) components, so that reusability increases by loose coupling and coarse granularity. Additionally, .NET promotes XML structuring language and SOAP protocol over HTTP connections as standard technologies for exposing and accessing software functionality over the Internet [94, 98, 75].

The .NET Framework is, therefore, a development and runtime platform that constitutes the infrastructure for building distributed and standalone applications which invoke components sourced in heterogeneous contexts. It also allows the development and the deployment of web services, web forms and other software components that can be grouped into libraries and referenced by standalone or distributed applications.

Fig. 4.11 shows the .NET Framework as a software layer that lies above each machine's operating system to achieve a homogeneous runtime platform, allowing the integration and interoperability of components written in a variety of languages for a wide range of electronic devices that may interconnect over the Internet. It primarily targets Windows-based software, although appropriate versions could, in principle, be written for any operating system.

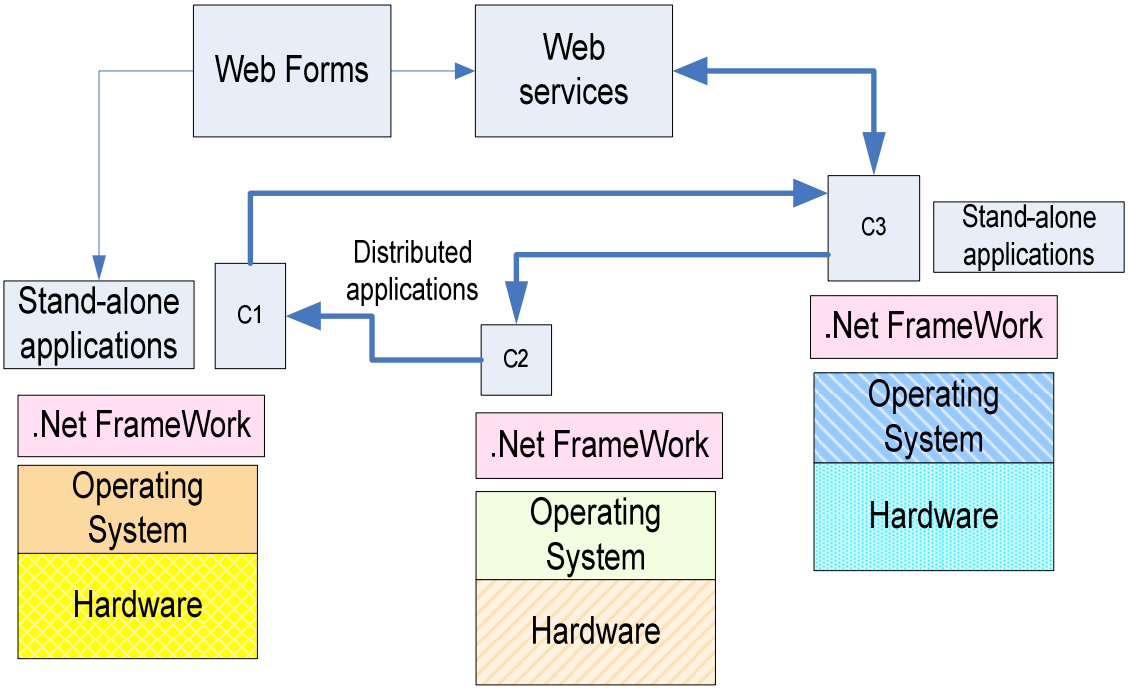

Fig. 4.11 .NET Framework makes the systems and the programming languages "homogeneous", thus allowing cross-boundaries integration and interoperability

The .NET Framework relies on object and component orientation, integration and Internet technologies to promote the development of Internet-friendly, secure, distributed and extensible components which integrate with other components in single applications. The integration is independent of the components' technological family and time generations and operates both at compile and runtime [94, 98]

4.4.3. .NET FRAMEWORK'S ARCHITECTURAL COMPONENTS

.NET applications are sets of components written in a wide range of programming languages, developed on top of a multi-layered library of classes and integrated under the umbrella of the .NET Runtime system. Thus, the .NET Framework basically comprises a development and a runtime platform. The former consists of an IDE, the Visual Studio.NET, and a foundation library of classes, the Framework Class Library, while the latter materialises in the Common Language Runtime.

Visual Studio.NET offers a common development environment to all the .NET

programming languages. It bundles together a set of tools such as editors, project managers, wizards, debuggers, compilers, linkers, etc. that are customised for each programming language and category of applications.

The Framework Class Library (FCL) is a set of built-in classes that are shared by .NET applications irrespective of their programming languages and categories. The FCL is arranged in successive layers, each layer extending the functionality provided by its predecessor. The FCL supports three tiers of functionalities as shown in Fig. 4.12.

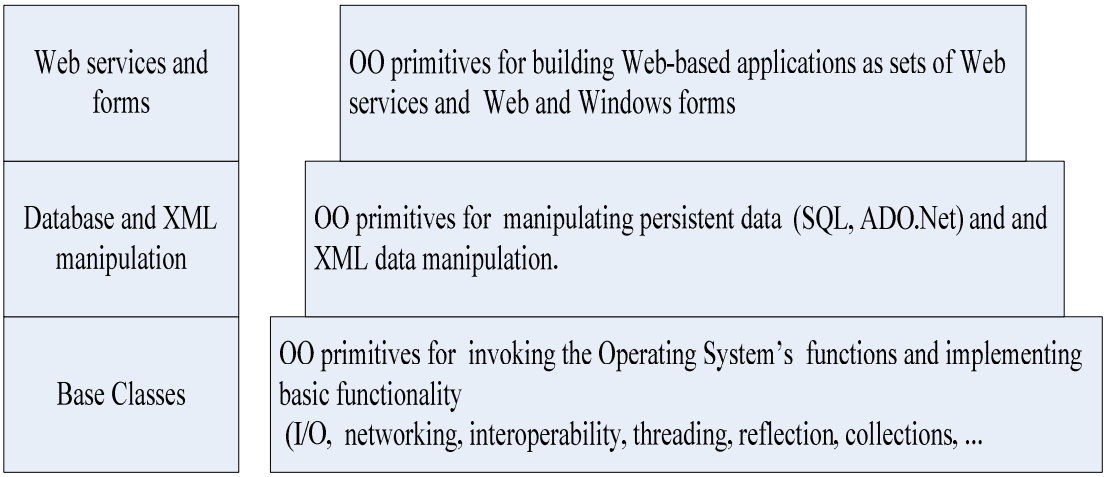

Fig. 4.12: Three foundation tiers of classes to implement Windows and Web applications and web services

The Common Language Runtime (CLR) is the runtime system which executes the managed code of the .NET applications, i.e. it runs the pieces of code that were written and compiled within the .NET Framework. Managed code comprises safe and unsafe code, as both may comply with .NET specifications. Safe code is memory type safe and conforms to the security polices laid out for it, i.e. any access to memory is controlled by the CLR and its execution is restricted to what it can do and where it can run [78]. Unmanaged code, such as legacy code, does not comply with the .NET standards but can also run. Managed and unmanaged code may interoperate in the same .NET application, but the former is compiled and executed on the fly by the

CLR while the latter, already compiled into machine code, runs directly on the operating system as shown in Fig. 4.13.

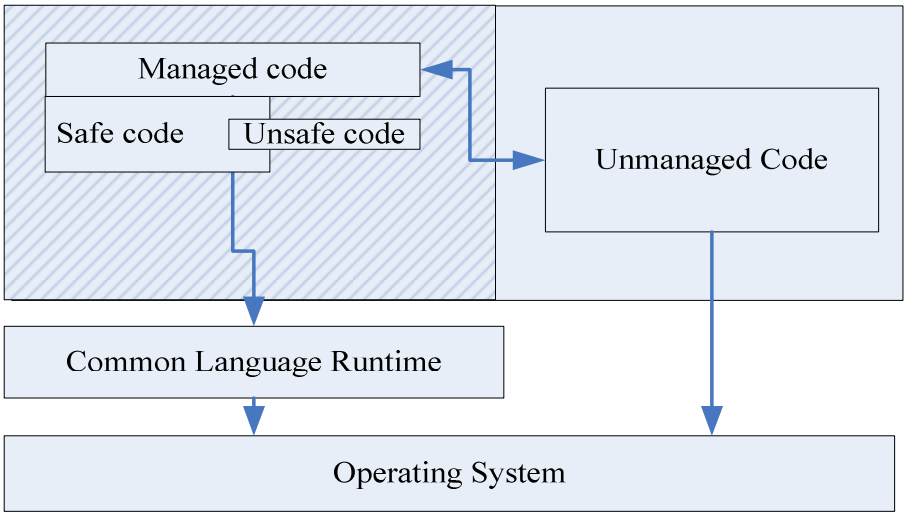

Fig. 4.13: Managed code, safe and unsafe, run on CLR while unmanaged code runs on the operating system

The CLR offers, among others, the following runtime services:

(i) Class loading: The CLR loads the pieces of code when they are needed.

(ii) Version integrity checking: The CLR refuses to run files that have been altered after distribution and it therefore avoids the 'DLL Hell' conflicts [61] that derive from updating shared DLLs.

(iii) Security checking: The CLR checks the compliance with the security policies laid out for the application itself (code-based security) or for the host user (role-based security).

(iv) Type-safe checking: The CLR checks the operations against the data types for safe code.

(v) JIT compilation: The CLR compiles the methods on demand by applying Just in Time compilation mechanisms [78].

(vi) Memory management: The CLR periodically removes unreferenced objects (acting as a garbage collector) and frees the corresponding memory addresses.

4.4.4. MULTIPLATFORM AND MULTIPLE LINGUAL INTEGRATION

The .NET Framework's integration philosophy is based on a two-stage compilation process that pre-compiles the diverse source codes into an intermediate object program, which is compiled at runtime to the machine's native language. Firstly, the source codes are translated into Microsoft Intermediate Language (IL) by language-specific compilers. This is a pre-compilation that checks and analyses the source code and also produces metadata which describe the runtime environment that the code requires for execution. Metadata and the IL code form a .NET assembly. The various .NET assemblies are combined and more metadata is added to allow, amongst other things, integrity and security checking.

The second stage consists of running the .NET assembly within the .NET runtime platform by using a Jitter to compile chunks of the .NET assembly as needed. Fig. 4.14 describes the compilation process of a .NET application.

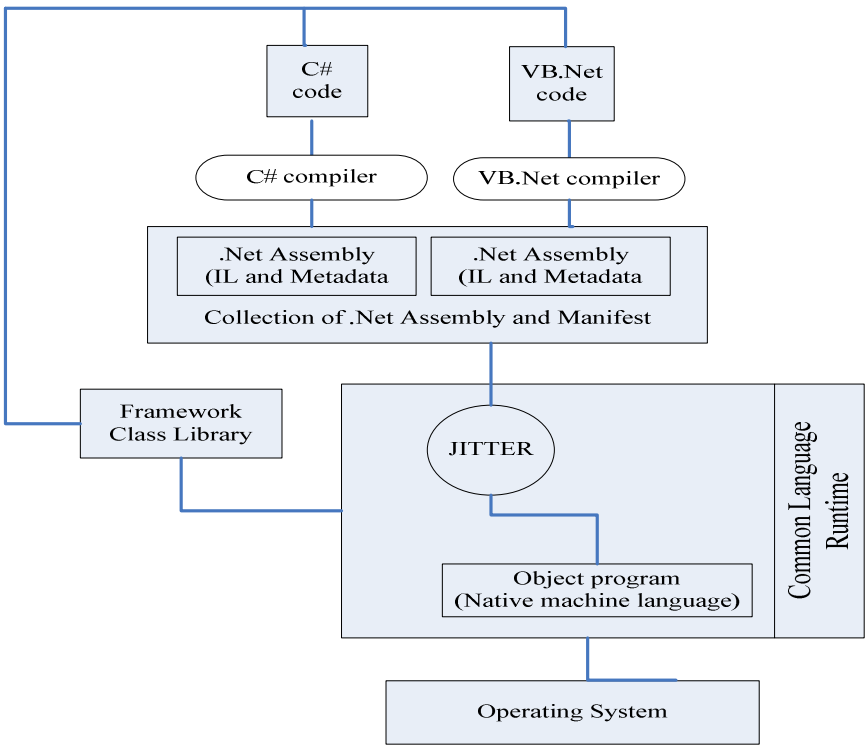

Fig. 4.14: Two-staged compilation of a .NET application which compounds a C#-component and a VB.NET component

Several programming languages, those of Microsoft or a third-party, are compliant with the .NET Framework, i.e. they display a minimum set of features and follow a minimum set of rules that allow their compilation into IL code and subsequent integration. .NET languages must adhere to the Common Language Specifications (CLS) and must ensure that they expose only the compliant features. Non-compliant features, such as case sensitivity and pointers, must be abandoned or camouflaged by mapping processes [78].

The Venn diagram in Fig. 4.15 represents the CLS as the intersection feature set of all .NET Languages. The size of this set is balanced between the enrichment that comes from a large range of programming primitives and the impoverishment that results from excluding languages that are unable to provide all of them [78, 61]

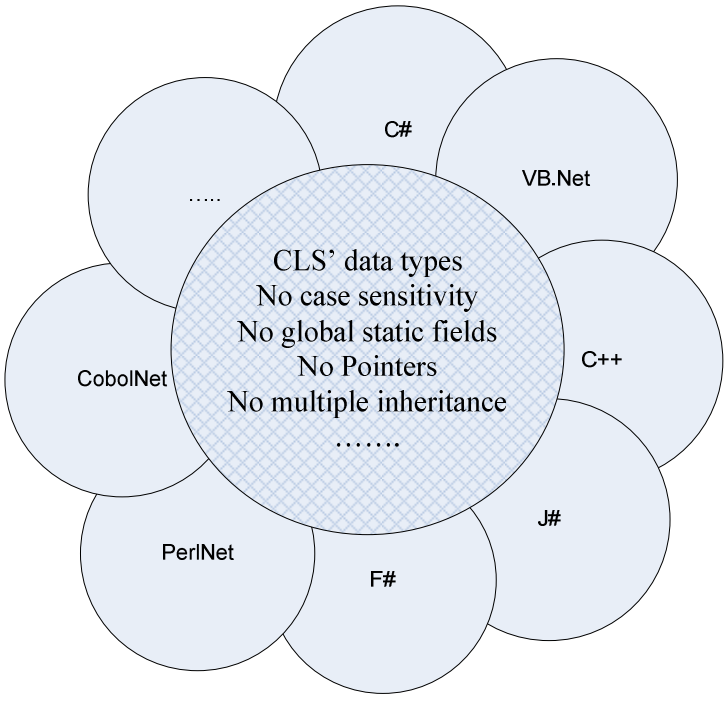

Fig. 4.15: Venn representation of .NET Common Language Specification

Beyond the .NET CLS, .NET languages must conform to the .NET Common Type System (CTS), i.e. the set of admissible types, their declaration, reference and invocation during runtime. Compliance with CLS and CTS grants true language integration by permitting abstraction, inheritance and polymorphism across .NET

languages. .NET languages may, therefore, instantiate and invoke methods of types implemented in a different .NET language as if they are their own. Thus, for example:

(i) a .NET application may instantiate objects from classes written in different .NET languages classes

(ii) a class may be derived from a super class implemented in a different .NET language

(iii) methods written in one language may be invoked and overridden by those written in other .NET languages.

4.4.5. INTEGRATION OF MICROSOFT OFFICE WITH THE .NET FRAMEWORK

The bi-directional integration of Microsoft Office and .NET applications is achieved by wrapping COM and .NET components into each other's formats, so that one can call any other as if both were technologically equal. Microsoft Office is largely built on the COM technology [95, 47], which sets a binary standard for application integration. Applications acquire functionality and integrate by implementing the interfaces exposed by the COM components. Objects, interfaces and types included are described in standalone files or embedded in DLL and Exe files [38, 71]. On the other hand, .NET components are self-described within assemblies and integrate at language-level so as to run on CLR.

Integrating an Office COM within .NET applications is achieved by generating an assembly which stores the type library as .NET metadata and resorts to a proxy to call the component [45, 46, 47]. Fig. 4.16 shows how a COM component is pulled up to be called as if it were managed code.

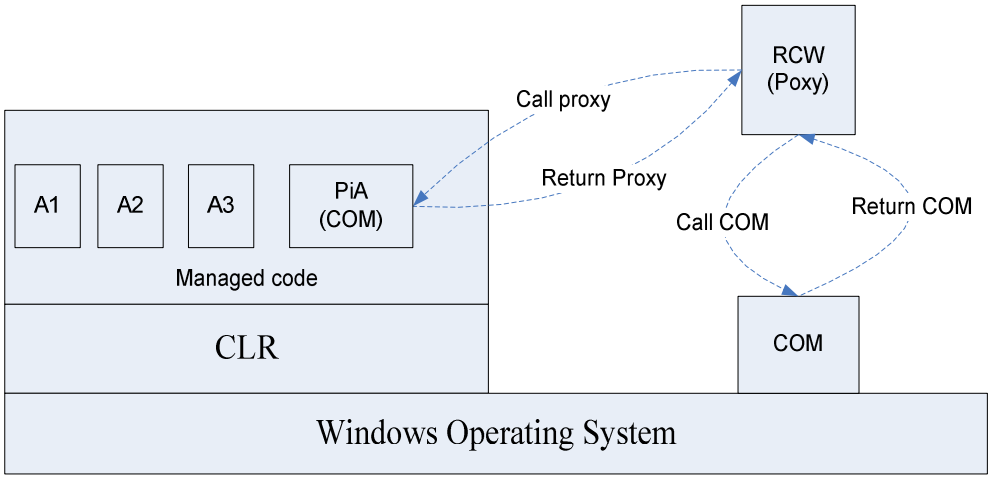

Fig. 4.16: The .NET application calls the COM component as if it were managed code. A proxy, the Runtime Callable Wrapper (RCW), intercedes as a bridge between the two technologies

Microsoft has generated several of these assemblies to describe the commonest Office types. These PIAs (Primary Interop Assemblies) [73] allow .NET applications to bind Office components at compile time and the CLR to marshal them across contexts, processes or machines. A PIA, such as Microsoft.Office.Interop.Visio.dll allows Visio classes to be instantiated from a C# program as if they were its .NET types.

The inverse process that calls a .NET component from a COM consists essentially of registering the corresponding assembly in the Windows registry, so that the COM client can locate it and provide the implementation of standard COM interfaces. The CLR creates a COM Callable Wrapper (CCW) for each .NET object that mediates, as a proxy, the invocation of a .NET component by unmanaged code [46].

Visual Studio.NET provides a wide range of tools to abstract the "plumbing work" behind the bi-directional integration of COM and .NET components.

4.5. THE DOTNETSIM PROTOTYPE

With this technological background, DotNetSim prototypes software for discrete event

modelling and simulation. Its functional structure, depicted at the highest general level by Fig. 4.17, highlights how different packages interoperate to build and report a model which is input into a simulation engine that, with further work, may be deployed as a web service.

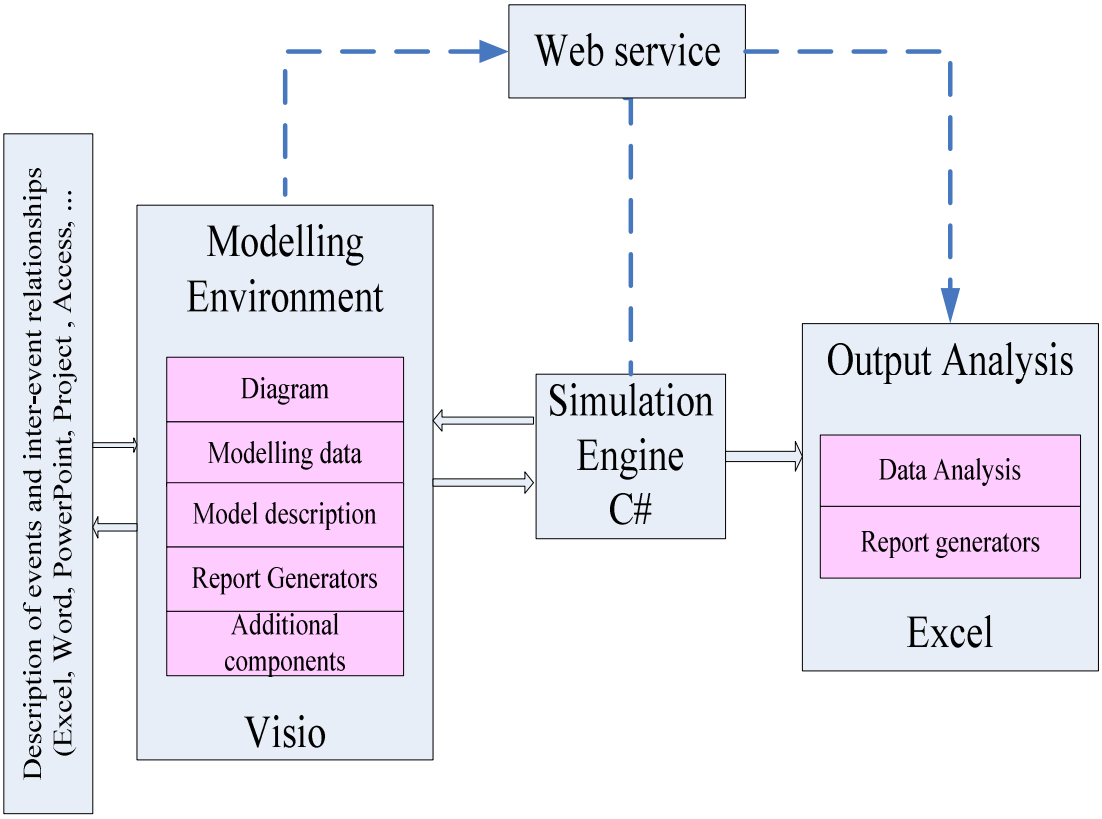

Fig. 4.17: Overview of the DotNetSim functional structure

The prototype graphical modelling environment emulates Schruben's Event Graph methodology for simulation modelling, whose basic concepts are described in chapter 5. However, other methodologies such as activity cycle diagrams, Petri nets or control flow graphs could be substituted. If required, the same approach could be used to develop a graphical modelling environment which suits particular application domains such as manufacturing.

The prototype graphical modelling environment is based on Microsoft Visio™ 2003. Event Graphs are drawn by the user or generated automatically from Excel-

based lists of attributes given the stencil modelling notation. Modelling data is stored in relational tables associated with the events and their interrelationships. The model's data is reported in Word™, Excel and PowerPoint™ documents. This can be extended so that the logic and dynamics of the models and the data reporting can be generated or displayed within a wider range of Microsoft applications.

Specially developed VBA components link together different Microsoft applications to bi-directionally exchange data in order to create the stencil's modelling notation and to capture the models' application logic and the model's dynamics.

The simulation engine consists of a number of C# and Visual Basic.NET components that implement an event-based simulation executive. It reads the model's logic by instantiating the Visio modelling environment, runs the event-based simulation and returns the simulation results to Excel™ for analysis. With further work, it can eventually be deployed as a web service to which the model's logic is remotely input.

The output analysis component is an Excel template which is instantiated by the simulation engine to place the simulation results as they are produced. It implements a set of VBA components to analyse and report the simulation results.

The components of the DotNetSim prototype are described in detail in the next chapters.

4.5.1. DOTNETSIM AND OTHER DEVELOPMENTS

Alternative approaches to the current monolithic DES applications, which tend to grow immeasurably to meet the demand for new features, are being explored by some major developers. The DotNetSim prototype follows this exploratory search and is among those which look into the use of generic software tools to develop modelling and simulation components that can integrate in a single application.

In a similar manner as the examples described earlier in this chapter - the Promodel Process Simulator™, the Micro Saint Sharp™ and the HighMast™ Toolkit - DotNetSim continues the stepped path toward the suggested ideal scenario for the simulation software. It does not go beyond the integration of widely-used software to prototype software for discrete event modelling and simulation based on the event-based simulation worldview. However, this is expected to constitute merely the launching pad for developing libraries of modelling and simulation components within generic software environments and for following the 'generalise-specialise' development cycle. Software solutions can then be assembled to suit specific DE simulations.

Thus, like the Promodel Process Simulator™, DotNetSim develops a DE modelling component on the top of Microsoft Visio™ generic programming features. Both take advantage of the almost 'standard' framework for usability which underlies widely-used Microsoft packages to develop easy-to-learn, easy-to-use and easy-to-customise graphical user interfaces. However, while the Process Simulator™ is a front-end that integrates from downstream with the Promodel's core application, the DotNetSim modelling environment is a self-contained component that can be integrated with any .NET-based simulation engine. The DotNetSim modelling environment can be integrated with components, developed within any Microsoft application that supports Automation, so as to provide the data to capture the DE model.

Like the Micro Saint Sharp core application, the DotNetSim simulation engine is written within the .NET Framework. Both resort to .NET programming languages, mainly C#, to ease customisation and interoperability between components developed within different programming environments by adopting fully object and component

orientation. The difference is that the DotNetSim simulation engine is just a software component that integrates from upstream with the modelling environment and from downstream with the output analysis components. It can be replaced by other simulation engines even if based on a different simulation worldview. On the other hand, the Micro Saint Sharp core application is confined to task-based simulation and its extension or replacement by other worldviews depends on its unpublished architecture.

Furthermore, the DotNetSim simulation engine relies on the .NET capabilities to load and develop new components, including those that involve interoperability with other packages. Hence, data exchange between components developed within different packages is based on the object-oriented programming paradigm which enables the objects of one application to be fully manipulated by another. Micro Saint Sharp™, instead, offers built-in tools and frameworks, to plug and unplug components, which limits the C# capabilities that can be accessed from within it. DLLs that implement additional features can be attached in Micro Saint Sharp™ and, naturally, in DotNetSim simulation engine.

DotNetSim, like HighMast™, aims for a vertical architecture in which the simulation specific components lie on top of generic components. However, the scope of DotNetSim is limited to the development of the three coarse-grained simulation components (described above), on top of widely-used Microsoft packages. These source-code based components are derived from the built-in capabilities of generic Microsoft packages and integrate by instantiating the object models of one component from within another component. However, Microsoft's built-in features are not open source.

New features can be loaded or developed for both HighMast™ and DotNetSim

from within the .NET Framework. Thus, the development of libraries of simulation components can sustain the assembling of simulation solutions.

4.6. THE CHAPTER IN CONTEXT

This chapter describes the idea of the DotNetSim project to explore an alternative approach to the development of DES software. The need to change the development strategy stems from the current expectations and concerns of software users and is supported by some examples of simulation software developers who have already initiated new strategies. A vision for the future of the DES software development is devised, using the latest computing advances and ongoing research towards on-demand software. The DotNetSim project was conceived to indicate a way leading to this vision by using the Microsoft .NET integration Framework.

Chapter 5 introduces the DotNetSim prototype and its graphical modelling environment, which emulates the Event Graph paradigm [108].