3. COMPUTING AND SIMULATION

CHAPTER OVERVIEW

The historical background of simulation software shows that it has rapidly exploited computing advances, reflecting the co-evolution of simulation and computer science. Alongside computing, simulation software has traversed the long journey from machine-oriented to component-oriented programming and now looks forward comprehensive software solutions which are tailored and assembled on demand to meet the requirements of each specific problem or customer at the appropriate time.

The increasing commercial demand for customised solutions has led simulation software to the current huge, monolithic applications with functionalities which are constantly extended by addition of wizards, templates and add-ons in a 'generalising-customising-generalising' development cycle. Current simulation packages have enough functionality to meet the requirements of a large number of problems. This approach, however, may be reaching its limits and other development alternatives could be devised in order to allow easy-to-use, quick-to-develop, quick-to-change and quick-to-run robust software solutions. These alternative approaches could resort to the latest advances in component-based and layered-based paradigms and associated integration mechanisms to promote the composition of simulation software solutions from existing or newly-developed prefabricated components.

Meanwhile, much computing research has already identified routes towards on-demand software, i.e. the 'just in time' assembly of software solutions for specific problems. The idea is for users to receive no more than the required functionality for each specific problem, at the appropriate time, during the required length of time and for them to pay per use. In the long term, it is therefore expected that if simulation software continues to follow computer science advances, it too will move towards the developing idea of on-demand software.

In order to understand current trends and to suggest future developments in simulation software, this chapter reviews some concepts of relevant computing fields, namely programming paradigms, composition of classes, objects and components and integration and interoperability. It also briefly reviews the evolution of simulation software to beat a path towards the future development of simulation software.

3.1. PROGRAMMING PARADIGMS

Since the software development crisis in the early 1970's, several programming paradigms have been introduced to facilitate and standardise the development and the maintenance of programs. Theoretical frameworks or paradigms were successively designed to provide programmers with a set of laws, rules and generalisations that enable programs to display desirable properties. Naturally, there are different paradigms for different programming domains as these also target different sets of desirable properties. Our scope is, however, confined to specific application software, namely simulation software. Table 3.1 lists some of the desirable properties for the simulation software.

Property	Brief description	A program should be:
Dependability	Ability to deliver the intended functionality when it is requested without causing danger or damages, resisting external attacks [116] and handling error conditions.	reliable, available, safe, secure and robust.
Usability	Ability to provide friendly user interfaces which permit different levels of utilisation, detection and recovery from input errors.	easy to learn and operate, adaptable to specific models of work and recoverable from user errors.
Modularity	Ability to be incrementally built by composition of self-contained units – functions, procedures, modules, components, etc.	Modularly-composed and extendible.
Reusability	Ability to run on different configurations, be composed with other programs and to communicate with packages of different vendors	portable, capable of integration, and inter-operable

Table 3.1 List of some properties which a program should display

Successive programming languages were conceived to suit specific paradigms.

3.1.1. STRUCTURED PROGRAMMING

Structured programming [35] was the first formal programming paradigm on which others where built. Modularity and the underlying principle of “divide and conquer” were its most important contributions to programming. Indeed, the techniques for structuring the program’s flow control are still recommended in order to make programs easier to read and to maintain. Structured programs are sets of modules, which are, in turn, sets of procedures. Procedures may invoke themselves or other procedures and functions to manipulate local and global variables. The program is put together by passing data through parameters among procedures. Thus, structured programs are monolithic pieces of code developed from scratch or, at most, from code scavenging.

3.1.2. FUNCTIONAL AND LOGIC PROGRAMMING

Functional and logic programming paradigms [118, 95, 49, 11] and their combination were pushed forward by artificial intelligence in the early 1990s. They are declarative programming paradigms, i.e. they focus on the description of data relationships, either mathematically or logically, as a means to approach the human cognitive processes.

Functional programming derives from Church's λ-Calculus [49, 31, 10, 11], and hence, a functional program is a set of functions which, once applied to their arguments, produce values or other functions. Its execution deals with the evaluation of expressions and their replacement by the corresponding values. This implies referential transparency, i.e. functions cannot produce side effects, so as to guarantee that the application of a function to the same argument always returns the same result. Functional programming originated heavily recursive languages such as Lisp, Miranda and Haskell, which are intuitive but lead to programs that are memory-demanding and slow to run. Logic programming derives from first-order predicate calculus, hence the outputs of a logic program are inferred from the known facts and the relationships between a set of objects. Prolog is the commonest logic programming language and is used mainly for expert systems.

Functional logic programming paradigms combine these two paradigms so that the programming primitives get the power of the logical inference; for example nondeterministic search and the efficiency of functional evaluation strategies such as lazy evaluation. ALF, Babel and Curry are examples of functional logic programming languages.

3.1.3. OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) [12, 67, 95] originated with Simula, an ALGOL-based language developed for discrete simulation. It is based on the notion of objects, their attributes and the functionalities they provide. An object is therefore a data structure composed of data (attributes) and code (methods) which implement the object's functionalities. For example, a queue is an object with attributes such as queue name, location and discipline and methods which enqueue and dequeue elements, determine the length of the queue at each state, reverse the queue, etc. The objects interact by invoking methods of other objects, i.e. by requesting others to execute the functionalities they provide. An enquirer service, for example, might be another object that interacts with the queue by invoking its methods to enqueue and dequeue clients and compute performance indicators.

3.1.3.1. CLASSES, ABSTRACTION AND ENCAPSULATION

OO programs are erroneously considered to be sets of objects. Actually, they are sets of classes of objects. A class is an abstraction of an object which defines the attributes and methods of a family of objects. An object is therefore a class's concretisation. For example, the class Queue is a generalisation of all queues and the object "Queue for Teller A" is an instance of that class. The class's code is reused by each instance.

A class gathers together, in a self-contained unit of code, attribute declarations (properties or field variables) and methods which allow the creation, at runtime, of objects having all that functionality. The class encapsulates all its contents within itself and only exposes the information required for the invocation of its methods.

3.1.3.2. CLASSES, INHERITANCE AND POLYMORPHISM

Classes are hierarchically structured into superclasses and subclasses by resorting to inheritance. Subclasses inherit the attributes and methods of the superclasses from which they are derived, add more attributes and methods and pass all to the subclasses. For example, B's and C's in the Three-Phase simulation worldview are both activities and they thus share attributes such as name and description and methods such as those to set and get the common attributes. Thus, B activity and C activity may be subclasses of the superclass Activity. They also have their own attributes and methods. For example, a C activity has attributes and methods associated with the condition that determines its execution. Inheritance graphs, as shown in Fig. 3.1, are drawn to represent the classes' structure and display the different levels of generalisation.

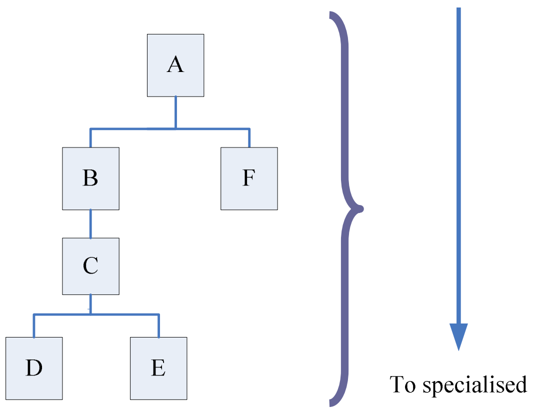

Fig. 3.1: Inheritance graph of a set of classes and subclasses

Some OO languages, C# amongst them, do not allow multiple class inheritance, i.e. a class can only derive from a single superclass. Yet, different classes may, for instance, share sets of methods by implementing interfaces. An interface declares sets of abstract methods by defining their signatures (the method name, the type of the input parameters and the return type). The classes willing to provide these functionalities must implement all those methods with exactly the same signatures. Interfaces also allow for the implementation of common properties, events and indexers and they may extend or combine others and a class may implement multiple interfaces.

Inheritance allows a method of a superclass (or an interface) to be implemented differently in any of its subclasses. At runtime, the implementation of this method in a subclass overrides its implementation in the superclass. Thus, this method appears in many forms depending on the context of its implementation. For example, if the superclass event of an event-based simulation defines a method for updating the state variables, it is likely that this method has different implementations in each event routine. Also, methods may need to refer to collections of different objects whose types are only known at runtime. Polymorphism allows the definition of generic objects that suit different types and at runtime they will assume the type of the given object.

Several programming languages implement OO concepts. Some are fully object-oriented, such as C# [61, 38] and VB.NET [6, 80]. Others, such as earlier versions of Visual Basic and corresponding subsets (VBA) are not fully OO languages as they do not implement inheritance [57] and offer only limited abstraction and polymorphism.

3.1.4. COMPONENT-ORIENTED PROGRAMMING

Component-oriented programming (COP) is the natural successor of OOP as it extends its concepts to provide a framework for constructing programs from prefabricated OO pieces of code (components). A component may be simply defined as a software artefact which is independently deployable, capable of composition by third-parties and with no observable external state [118]. This implicitly assumes that components are self-contained and encapsulated units which, by exposing the functionalities they offer and the needs for their delivery, can be to be plugged into (and unplugged from) other components. Also, they are immutable in the sense that they are abstract definitions and, consequently, two copies provide exactly the same functionality under the same conditions.

3.1.4.1. GRANULARITY AND ABSTRACTION

Components may range from add-ons and plug-ins to applications. The former are said to be fine-grained components as they provide limited functionality and the latter are coarse-grained components that offer extensive functionality. Fine-grained components deal with detailed data structures [95], hence they tend to be specific and dependent on the deployment environment (they may even include OS basic functionality). On the other hand, coarse-grained components deal with large-scale data structures [95] whose high degree of abstraction make them generic and adaptable to several deployment environments.

For example, a component which implements a simulation engine is coarse-grained as it deals with classes of entities, activities and events to offer generic functionality that can be specialised while plugging into a simulation system. An add-on that generates random numbers to be plugged into a spreadsheet is fine-grained as it deals with the objects themselves.

3.1.5. LAYERED-ORIENTED PROGRAMMING

Layered-oriented programming (LOP) is a paradigm that applies mainly to the design of systems architectures, as it concerns the interconnections of software components. It inspired the OSI reference model [120] and was first used to develop onion-structured operating systems [37]. A layered system is incrementally built, layer upon layer [118]. Each layer receives from the layer immediately below well-defined functionality which it increments and passes to the layer immediately above. Each layer may be composed of several components.

Simulation systems might then be layered structured, e.g. by adding layers as shown in Fig. 3.2. This is a pyramid-layered structure [79] in which each layer receives from the layer immediately below the necessary functionality to provide more specific and finer-grained functionality.

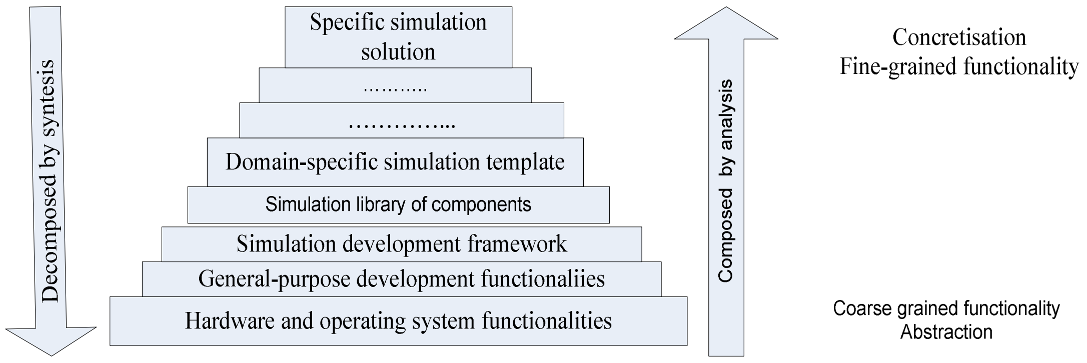
Decomposed by syntesis
Specific simulation solution
……….
…………...
Domain-specific simulation template
Simulation library of components
Simulation development framework
General-purpose development functionaliies
Hardware and operating system functionalities
Composed by analysis
Concretisation
Fine-grained functionality
Coarse grained functionality
Abstraction

Fig.3.2: Pyramid composition of a simulation system

Extensibility is possible by the addition of components and layers. However, this requires a corresponding functional foundation from the next lower layer.

Layered-oriented programming is associated with meta-programs which provide the functionality to interpret, inspect, adapt and extend the structure and behaviour of the underlying software. The metadata (attributes) and computational reflection will allow for the dynamic composition of layers of components. This may lead to on-demand software programming [39, 40] that will promote the dynamic assembly of software solutions by selecting, negotiating and invoking suitable components, such as web services, at runtime.

3.2. CLASS, OBJECT AND COMPONENT COMPOSITION

The composition of classes, objects and components [118] is partly motivated by the reuse of existing functionality, but its principal aim is to promote the extensibility of OO and CO programs by addition of independently-developed functionality. The composition of classes, objects and components must be within an architectural structure, i.e. an abstract view over the system elements and their interrelationships [82] which defines the rules, the standards and the constraints for their composition. This raises several issues that require the clarification of a few basic concepts.

3.2.1. CLASS COMPOSITION

The OOP principle of inheritance enables the composition of classes. Classes and subclasses are linked in a hierarchical structure that allows subclasses to incorporate the functionality implemented further up the inheritance graph, to use and to override it according to the needs detected at compile time [118]. Implementation inheritance, i.e. inheritance of the properties and methods implemented by the superclasses, can, however, generate relationships among classes that do not comply with OOP principles and, hence, unbalance the state and the behaviour of the compound classes. Take the case of multiple inheritance when the inheritance graphs are not disjoint. When, for example, a class (model A) is derived into two subclasses (models B and C) and those two are the parents of a subclass further down (model D), we do not know which functionality is inherited by the model D. Either encapsulation is broken by allowing two submodels, B and C, to see the changes operated by each one on A or, if they act independently, we do not know which state of model reaches model D. This can be aggravated if the intermediate models override the functionality of model A. For this reason, most OO programming languages do not allow multiple implementation inheritance. Some, however, try to solve the inconsistencies by disciplining inheritance setting, for example, an order of inheritance, i.e. defining a precedence list which determines the superclasses that are dominant. Multiple interface inheritance threatens OOP principles, hence, it is not generally offered by the programming languages.

The composition of classes may also result in complex dynamics that threaten its robustness. This raises issues such as the impact of the changes of the base classes over time (e.g. new versions) on their subclasses.

3.2.2. OBJECT COMPOSITION

Message forwarding is the most frequently used technique for compounding objects [118]. Objects – instances of classes – are compoundable at runtime with other objects in such a way that when an object lacks some functionality it sends to other objects a message to run the required functionality. This composition is ruled by a structure of outer and inner objects. Outer objects may forward messages to the inner ones. The outer object runs the missing functionality from within the inner object. Thus, any changes on the inner's object implementation of the functionality reaches the outer object immediately.

The outer and inner objects have independent identities, however. This makes message forwarding different from implementation inheritance among classes. Object composition is dynamic, hence, for example, inner and outer objects can be replaced while the outer is functioning [118]. This can be programmatically added to class inheritance.

3.2.3. COMPONENT COMPOSITION

Components are composed and partitioned so as to add, alter and remove functionality from component-based programs [118]. The composition of components raises several issues regarding the interfaces which they expose, the cohesion and coupling of components, the time they are to be bound to other components and the location where they reside.

3.2.3.1. INTERFACES

Interfaces are the mediating means for two components to connect as they serve the functionality provider (component-server) and its potential clients (component-clients). Generally, there exists an interface for each functionality provided, i.e. an interface defines directly or indirectly [118] the operations that implement the functionality and the conditions the client has to fulfil to invoke those operations (preconditions). The latter are the context dependencies, which comprise the rules of deployment, installation and activation of the component and also the rules of composition. Fig. 3.3 illustrates how a component provides some functionality to a client program. Component C4 provides an interface that specifies the offered functionality and the context dependencies.

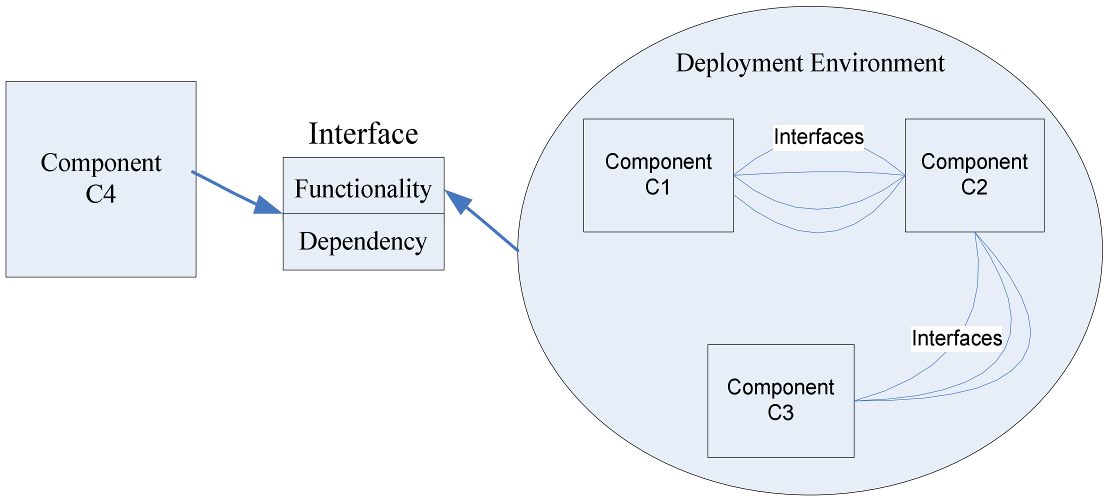

Fig. 3.3: Plugging component C4 into a component–based program

3.2.3.2. COHESION AND COUPLING

The cohesion of a component indicates the closeness of the relationships within it whereas coupling measures the closeness of the relationships between components [116]. Thus, cohesion measures the contribution of each part to the functionality offered by the component. A highly cohesive component is self-contained as it includes all the operations needed to understand and manipulate its functionality. For example, an entity class which provides the necessary methods to understand and operate the entity's attributes is highly cohesive.

Coupling measures the degree of independence among components. Two components are tightly coupled if changes to one component cause changes to the other - the two components depend on each other to implement their functionalities; they are loosely coupled if they are independent – each component functionality is understandable within itself and the impact of changing one component on the other is limited [44] or even zero. Thus, highly cohesive components may be loosely coupled with others. For example, a random number generator which encapsulates all the methods to manipulate the random numbers is loosely coupled with components that implement random sampling algorithms.

3.2.3.3. LOCATION

Component-servers and component-clients may reside in the same machine or be spread across different machines. However, the clients invoke the remote component-servers as if they are residing locally. Further, the servers provide their functionality to remote clients as if they were local. Thus, remote procedure calls (RPC) hide the inter-machine communication that basically consists of marshalling and de-serialising the call parameters [61]. This local-transparent invocation (an extension of the concept of a virtual machine) allows for intercommunication across different platforms, different processes, different application domains and different contexts.

Ideally, a component-oriented program consists of a set of remote or local components, of different degrees of granularity that are highly choesive and loosely coupled at runtime.

3.2.3.4. BINDING TIME

Plugging a component (component-server) into another component (component-client) consists mainly of referencing the component and associating this reference with a particular component. The latter is the binding and it occurs either at design or compile time, as the source code is converted into an executable form (early or static binding), or at runtime, during the program execution (late binding or dynamic binding).

Early binding speeds up the execution, but assumes that the component-client knows, in advance, the particular component it needs and the interfaces whose preconditions it has to fulfil. On the other hand, late binding increases flexibility by deferring as long as possible the decision on the required component. The component-client, during its execution, selects an available component-server and implements the methods of the newly discovered interfaces [95,118].

3.2.4. REUSABILITY

The composition of classes, objects and components enables the reuse of code or prefabricated functionalities in different grades that range from black-box to white-box reusability, passing through several grey tones. All of them encapsulate their implementations but some, the black-box components, let the builder (user or developer) know nothing beyond the interfaces while other, white-box components, allow the builder to change the implementation. Class composition enables white-box reusability while object composition just enables black-box reusability. The grey-box components offer different permissions that let the builder read and change parts of the implementation.

Many different reusability benefits are claimed [82, 102]. The most common are reduction in software development time and cost, an increase in reliability and more flexibility in the solution composition.

However, reuse can generate complexity [102] and, without commercial and financial incentives to upgrade specific components to reusable components, may lead the builder to less acceptable solutions. Plugging black-box components which do not entirely fit a specific problem may be acceptable, however. For example, force-fitting an existing component that alters the structure of a simulation model by not allowing the representation of all the entities' interrelationships is doubtless unacceptable, but using a graphical interface without animation may be acceptable, depending on the purpose of the study.

Reusability is encouraged by the open source business model, which promotes the free distribution of the software source code so that users may improve and extend the delivered functionality.

3.2.5. INTEGRATION AND INTEROPERABILITY

OOP, COP and LOP support the construction of software by the composition of reusable components. This, allied to the promotion of interoperability across space and time, integration architectures and the open source business model, is gradually leading software engineering from traditional generic packages and in-house software towards the customised construction of software solutions for specific problems. Standards definition [116] is a key factor for the interoperation and integration of reusable components.

Both interoperability and integration refer to the ability of pieces of software to communicate with one another. The former generally means software and hardware communication across different machines and the latter means the communication between different pieces of code separately developed and put together to work as a whole [95]. The distinction between the two terms is fading as programs tend to be sets of components which reside in different machines and were developed for different software platforms. Thus, integration is melding into interoperability, as its scope tends to include the communication of components across hardware platforms, computer networks, applications and programming languages. Also, software components of different technological generations (legacy code) must work together.

There are several technologies which support integration and interoperability, including Microsoft automation and Microsoft .NET Framework. Interoperability and integration are also promoted by communication standards, such as XML, a meta-language for structuring documents, and SOAP, a protocol for invoking remote code by using XML over HTTP connections. Further details on Microsoft automation and Microsoft .NET Framework are presented in chapter 4.

Nowadays, components are developed within integrated development environments (IDE), such as Visual Studio .NET, which offer graphical and menu driven-tools to hide from the user the details of the wiring, the deployment and installation of component-based applications across networks of various electronic devices.

3.3. SIMULATION SOFTWARE FOLLOWS THE TECHNOLOGICAL ADVANCES

The chronological evolution of simulation software shows that it has followed computer science since the advent of computers, reacting promptly to technological innovation. It traversed the long journey from machine-oriented to component-oriented programming, looking forward to the development of platform-independent and comprehensive software. Simulation-oriented programming languages were derived from general-purpose languages to facilitate the implementation of specific data structures, stochastic and time-dependent processes; simulation programs were successively re-written in high-level languages and simulation-oriented languages and specific software applications were generalised by parameterisation and functionality extension.

Currently, the strategy of simulation software developers seems to stress a different approach to customisation. Specific applications developed to solve particular problems are converted into generic simulation packages and these are now tailored to fulfil the needs of specific problems or customers. Numerous add-ons are being plugged into these simulation applications so that generic simulation packages are ready to be customised to a larger number of specific problems. Access to external packages is also added to simulation packages in order to allow data exchange between simulation packages and widely-used generic software, such as database management systems and spreadsheets. Fig. 3.4 lists the major computer advances that

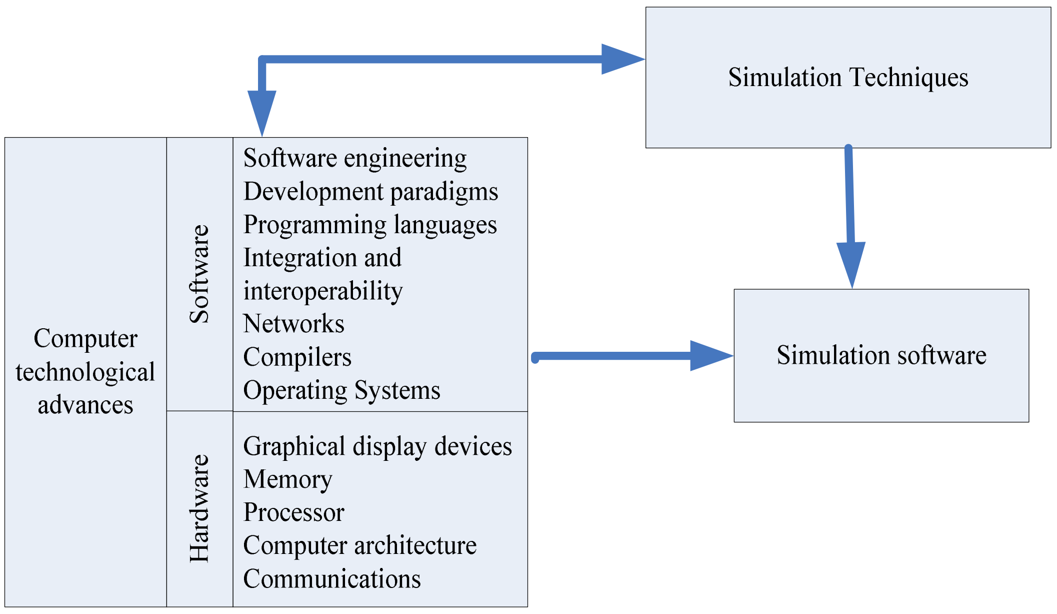

Fig. 3.4: Computer advances that have propelled the development of simulation technology

have contributed and continue to contribute to the development of simulation technology.

It seems reasonable to expect that simulation software will continue to follow the advances of computer science in order to benefit from the latest technology. The idea is to widen its functionality to an increasing range of target problems and to promote easy-to-use, quick-to-develop, quick-to-deploy, quick-to-change and quick-to-run robust software solutions [7]. Also, it is expected that computing technology will progress into integrated environments and programming paradigms that will facilitate the achievement of simulation software goals. This thesis explores some of the implications of simulation software exploiting these developments.

3.3.1. A BRIEF HISTORICAL REVIEW OF SIMULATION SOFTWARE

The size and complexity of the data processing required for discrete event simulation has propelled the development of simulation software tools since the late 1950s. Large data structures subject to iterative or recursive algorithms which trace the state changes of complex systems have always defied full automation [4]. In addition, discrete event systems are frequently stochastic which increases the need for automatic tools to support the study of variability. Fig. 3.5 lists the different categories of software tools that support all the stages of discrete event simulation. These tools are often bundled together and marketed as decision-support packages or as Visual Interactive Modelling Systems [110, 85]. Additional tools are frequently added to the existing simulation packages to increase their functionality by implementing features for specific problems, by creating built-in tools such as templates and wizards [7].

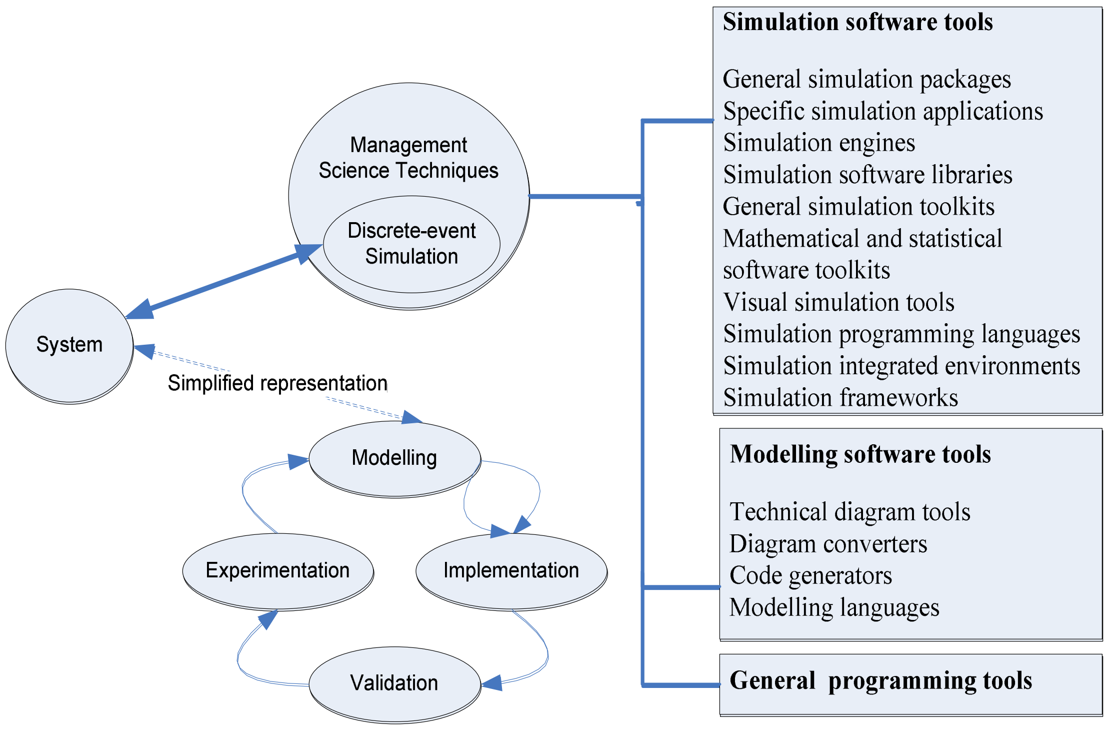
Simulation software tools
General simulation packages
Specific simulation applications
Simulation engines
Simulation software libraries
General simulation toolkits
Mathematical and statistical software toolkits
Visual simulation tools
Simulation programming languages
Simulation integrated environments
Simulation frameworks
Management Science Techniques
Discrete-event Simulation
System
Simplified representation
Modelling
Experimentation
Implementation
Validation
Modelling software tools
Technical diagram tools
Diagram converters
Code generators
Modelling languages
General programming tools

Fig. 3.5: Software tools for the simulation's main stages (partly derived from Pidd, 2004)

3.3.1.1. FROM MANUAL SIMULATION TOWARDS PORTABLE SIMULATION SOFTWARE

The initial informal procedures which supported manual simulation were soon replaced by formal methods, such as Hocus, which were less prone to errors [48]. The new models, described by formal languages, simplified the whole process of modelling and simulating but the execution of the simulation continued to be manual and time-consuming. Thus, mechanical devices were developed to quicken the computation of repetitive and stochastic operations [117].

Shortly after the advent of computers, the mechanical devices were replaced by software routines written in machine code which performed simple computations, namely pseudo-random numbers generators and random sampling. Coding, debugging and portability difficulties [48, 55] were partly overcome by the emergence of the first high-level languages, namely FORTRAN (1954) and ALGOL (1960) into which simulation routines were translated. Simulation programs written in these languages were portable to different machines as soon as FORTRAN and ALGOL compilers were developed for several computers [48]. Their resemblance to human language and powerful primitives accelerated the development of simulation programs. However, FORTRAN and ALGOL were science-oriented languages, with primitives that were well suited for scientific problems, such as the representation of equations, but less efficient for simulation-oriented tasks [48]. This was the driving-force for the creation of simulation programming languages.

3.3.1.2. TOWARDS SIMULATION PROGRAMMING LANGUAGES

Simulation programming languages were gradually derived from existing general-purpose programming languages by extending their functionalities to simulation concepts. New data structures and macro-instructions were added to the base languages to allow the implementation of simulation techniques and time-dependent stochastic processes [48, 55]. Fig. 3.6 shows simulation-related features that were appended to Assembly, FORTRAN and ALGOL.

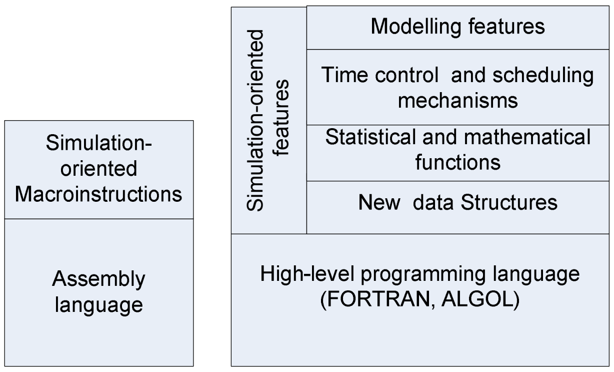
Simulation-oriented Macroinstructions
Assembly language
Simulation-oriented features
Modelling features
Time control and scheduling mechanisms
Statistical and mathematical functions
New data Structures
High-level programming language (FORTRAN, ALGOL)

Fig. 3.6: The first simulation languages were extensions of Assembly, FORTRAN or ALGOL

The first simulation programming languages were pre-processed into the programming language they derived from, i.e. programs were translated into a target language prior to their compilation and execution, as illustrated in Fig. 3.7.

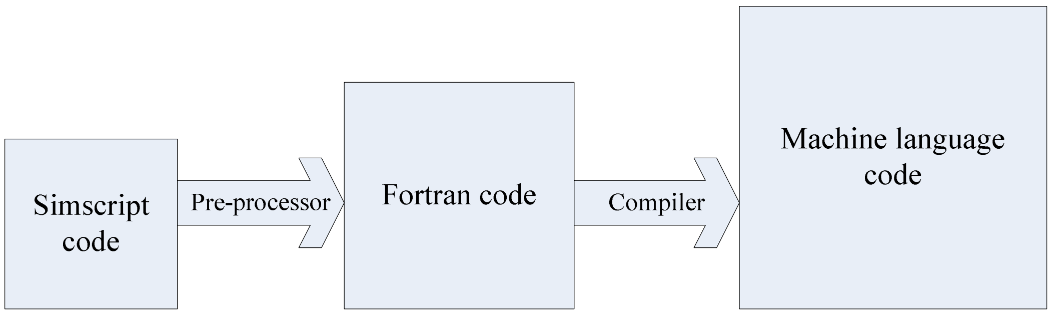

Fig. 3.7: Simscript was originally implemented as a pre-processor for Fortran

Numerous simulation specific languages were developed or revised in the 1960s [85]. All were procedural languages that supported the structured programming paradigm. One of them, Simula, is claimed to be the precursor of the object-oriented programming languages since its basic data structure – the process – contains data and methods for handling the activities [16, 48, 55].

As simulation languages became more powerful by the addition of new primitives, specific compilers were developed and their dependency on general-purpose languages gradually vanished. Simulation-specific languages and general-purpose languages became disjoint alternatives for developing simulation software [48, 55]. As the former were tailored for simulation problems, their data structures, libraries of functions and even syntax were well-suited for implementing the simulation engine and application logic. This approximated modellers to software developers and facilitated the construction, implementation, validation and experimentation of simulation models [55]. Yet, the latter were supported by a wide range of development tools, namely editors and debuggers, and by a larger number of experienced developers.

As simulation languages continued to be updated with new features, they became more specific-problem oriented and vendor or product dependent; they were, therefore, more accessible to less skilled developers.

3.3.1.3. TOWARDS PACKAGES

As software applications became increasingly portable, due to advances in operating systems and compilers, attempts were made to produce simulation software suited to a wider range of problems. Applications, built for solving specific problems, were generalised by parameterisation and extension of functionalities [62], as shown in Fig. 3.8.

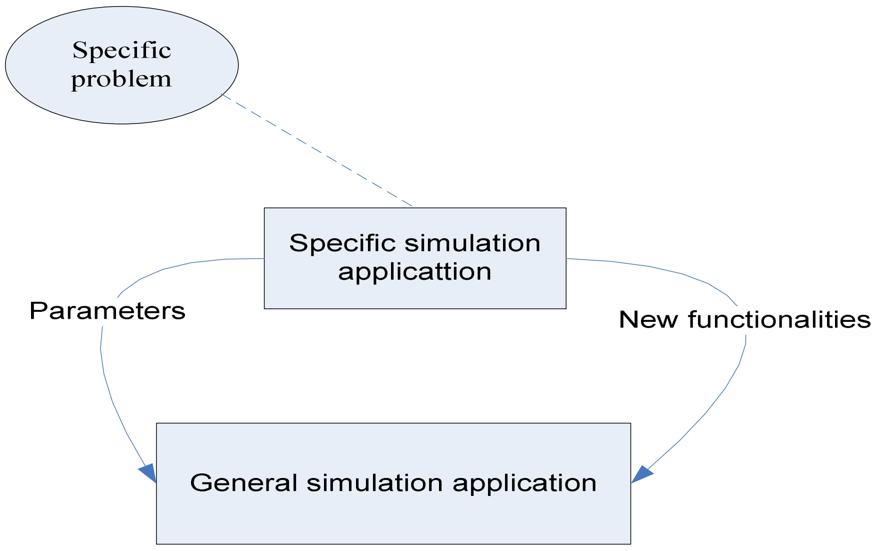
Specific problem
Specific simulation applicattion
Parameters
New functionalities
General simulation application

Fig. 3.8: Generalising a specific problem solution

Simulation tools were bundled together with the corresponding programming language and marketed as packages. At first, they were mere sets of subroutines to automate parts of a simulation system but soon they were structured into single applications [48].

Meanwhile, programming techniques evolved to modular paradigms which suited the attempts to extend the functionalities of the existing packages. This, allied with the spread of graphical displays, led simulation software to the development of packages which aimed to cover all the phases of modelling and simulation. At that point, simulation software progressed into visual interactive modelling systems (VIMS) by adding interactivity and capabilities for graphical model creation and animation.

3.3.1.4. TOWARDS VISUAL INTERACTIVE MODELLING SYSTEMS

Models began to be defined by resorting to menu-driven graphical interfaces and tool bars of prefabricated icons. Graphical user interfaces (GUI) were incorporated in simulation packages to support the emerging Visual Interactive Modelling approach that offered visual and interactive tools as a communication language between the simulation stakeholders. Various simulation packages were extended with visual facilities and interactivity (e.g. FORSS became FORSSIGHT, which later evolved into WITNESS) and became VIMS [85].

VIMS basically create user-friendly modelling and running environments by linking input GUIs, from where the modelling data is collected, to simulation engines, where the model runs, and to output GUIs that display the simulation outcomes. This is done either by specific built-in components or by linkage to appropriate software packages such as graphic generators. Also, VIMS provide the modelling environment with interactivity features so that the simulation run could halt at any point and resume afterwards to allow the user to input data, alter the set of parameters, collect intermediate results, put traces on individual elements of the model or view outputs to evaluate model behaviour [85, 101].

There are currently various VIMS - commercial off-the-shelf VIMS and non-commercial specific applications - that greatly differ in their visual and animation features. Although all of them provide the basic built-in tools for VIM, a few also differ by letting the user extend the suite of prefabricated tools. The users can therefore broaden the VIMS' application scope by resorting to a programming language, though this is often too hermetic.

Other differences derive from the simulation displays they favour. Some VIMS focus on the iconic representation of the system's physical appearance, others stress the graphical display of the system's logical structure and others opt for displaying in charts the performance of the system. VIMS also vary in the output displays that range from the immediate display of the results of single runs to the storage of the results of a series of simulations runs.

The simulation engines of the VIMS, as of any simulation software, differ mainly in the worldview they implement. However, the interactivity which they allow also introduces some variety.

3.3.2. THE CURRENT DEVELOPMENT STRATEGY OF SIMULATION SOFTWARE

The current trend of simulation software is to tailor generic packages to specific problems or needs [5]. A large number of wizards, templates and add-ons have been added to commercial simulation packages to ease their customisation [3, 13, 42, 96].

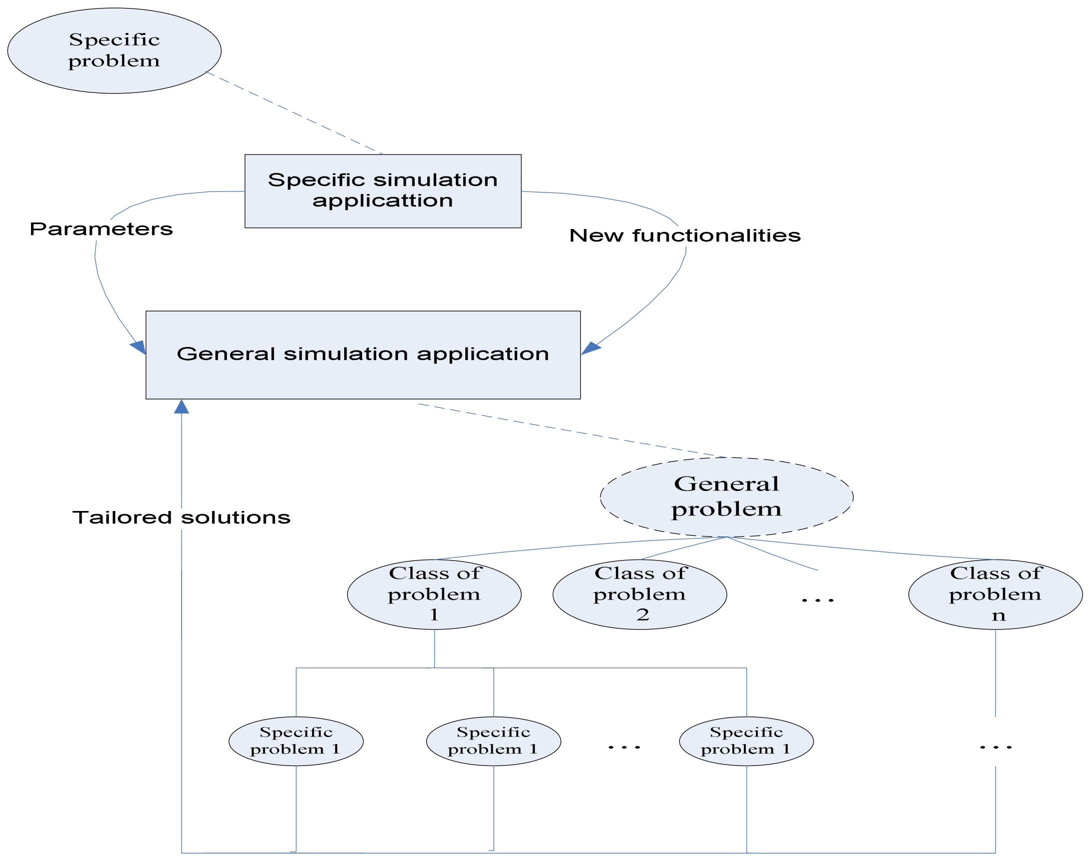

Fig. 3.9: Tailoring generic simulation software to a specific problem solution

Generic pre-prepared structures are therefore available for tailoring specific applications. These may later be generalised and constituted into new templates and add-ons. The trend appears to follow the circular path illustrated in Fig. 3.9. A specific problem leads to a specific application that later is generalised into a package which, in its turn, is tailored to fulfil the requirements of a specific problem or customer.

Thus, some simulation packages have become huge applications with functionalities that are constantly being extended by the addition of new modules which sustain the generalising-customising-generalising development cycle. Current packages aim to cover a broad spectrum of techniques ranging from visual 3D modelling to optimisation, data mining, input and output data analysis and statistical experimentation techniques. Also, graphic and report generators are being improved to enhance the comprehensiveness of the applications [48].

Other important extensions address the linkage with external packages with which the simulation software exchange data, namely importing and exporting data from and to existing databases, technical drawing software, spreadsheets or statistical packages. In 2004, four major vendors of simulation software (Simul8 [114], Lanner [59], Arena [3] and Promodel [96]) started marketing modules that link Microsoft Visio^{TM}, a technical diagram tool, to their core simulation applications. Thus, simulation packages are becoming large suites of tools which provide the means for linking modelling and simulation features to external packages where data are based. (See Fig. 3.10). With such packages, users can easily build new models based on the prefabricated templates by resorting to the built-in tools. However, models whose logic demands special tools are difficult to build as they require expertise in the package's programming interface and the underlying simulation programming language.

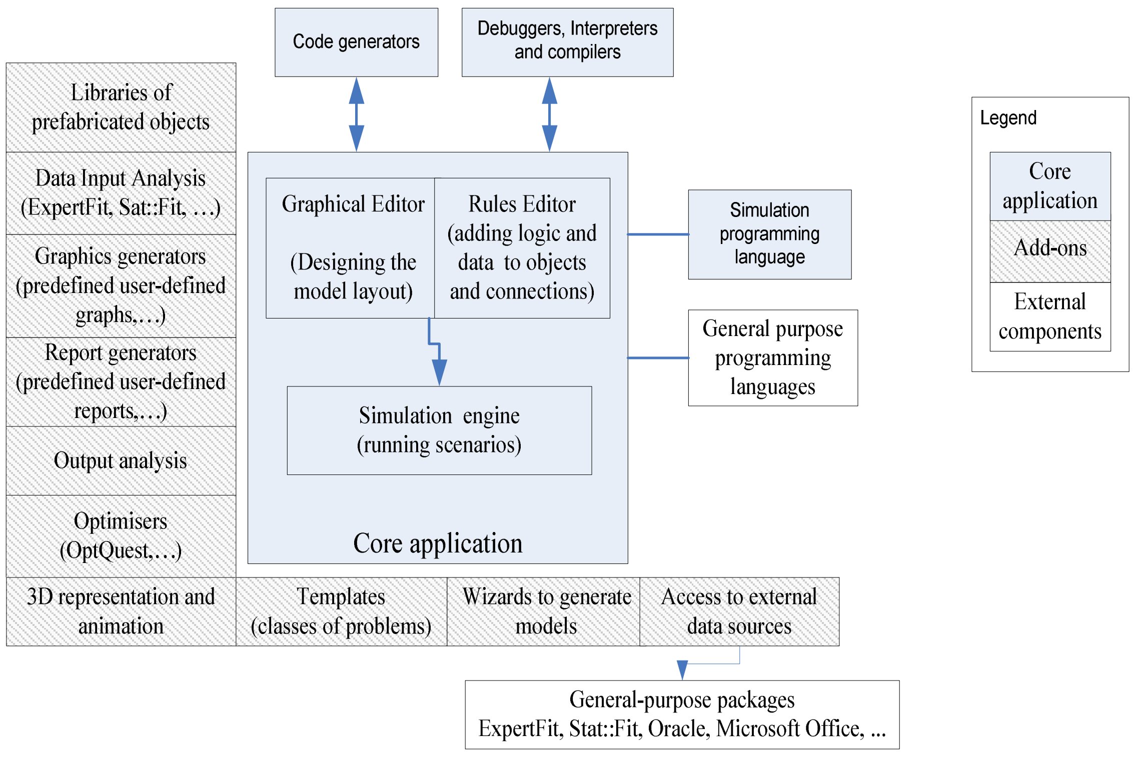
Code generators
Debuggers, Interpreters and compilers
Libraries of prefabricated objects
Data Input Analysis (ExpertFit, Sat::Fit, …)
Graphics generators (predefined user-defined graphs,…)
Report generators (predefined user-defined reports,…)
Output analysis
Optimisers (OptQuest,…)
3D representation and animation
Templates (classes of problems)
Wizards to generate models
Access to external data sources
Graphical Editor (Designing the model layout)
Rules Editor (adding logic and data to objects and connections)
Simulation engine (running scenarios)
Core application
Simulation programming language
General purpose programming languages
General-purpose packages ExpertFit, Stat::Fit, Oracle, Microsoft Office, ...
Legend
Core application
Add-ons
External components

Fig. 3.10: A typical simulation package currently consists of a core application and a wide range of add-ons

Current commercial simulation software has steadily achieved its long-term goals as it is user-friendly, visual-oriented, interactive and robust. The next immediate goals to fulfil are undoubtedly customisation and interconnection of different systems. Both goals imply component reusability, integration of widely-used generic software, web-based interoperability among different systems, source-code based components and web-based software deployment.

3.3.3. FUTURE DEVELOPMENT STRATEGY OF SIMULATION SOFTWARE

The past and the current evolution of simulation software lead us to expect that simulation software will continue to rely on the latest advances in computer science to face up to the increasing size and complexity of computer simulations. Progress achieved in areas such as programming paradigms, integration of hardware and software platforms, networks and communications, along with the continuous advance of Internet technologies is crucial to the future of simulation software [8, 13, 52, 54, 83].

Fig. 3.11 summarises the parallel evolution of computing and simulation software. Computing is now evolving towards the on-demand software or instant assembly of software, in which software builders ‘rent and pay per use’ the only required functionality at runtime [118, 39, 40]. Simulation software might similarly evolve in this direction towards just-in-time assembly of simulation solutions.

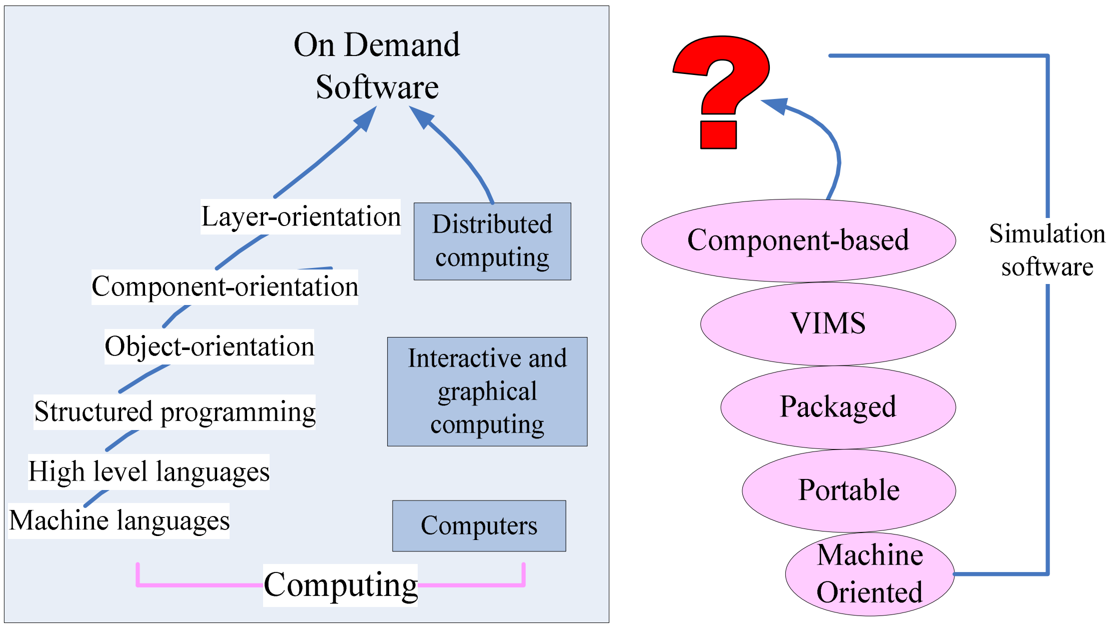

Fig. 3.11: Simulation software has rapidly reacted to technological innovation

The likely irreversible commercial demand of customised solutions will shortly force the current simulation packages to be restructured, so that each user acquires exactly the functionality a solution needs - gets only the required functionality - or, even better, the user acquires the capability to produce the functionality a solution needs. Hence it is likely that these packages will have to be redesigned to fully comply with object-oriented, component-oriented and layered-oriented programming paradigms [7] in order to respond promptly to customisation demand; or their market will disappear.

In the longer term, simulation software might evolve towards on-demand simulation software in an open market of components. Fig. 3.12 sketches the assembly of a specific simulation application in which those who develop and publish simulation software functionalities negotiate with those who select and assemble the functionalities per simulation solution. Ongoing computing research focuses on investigating how the selection of the functionality, its negotiation, delivery and binding can be done at runtime [39].

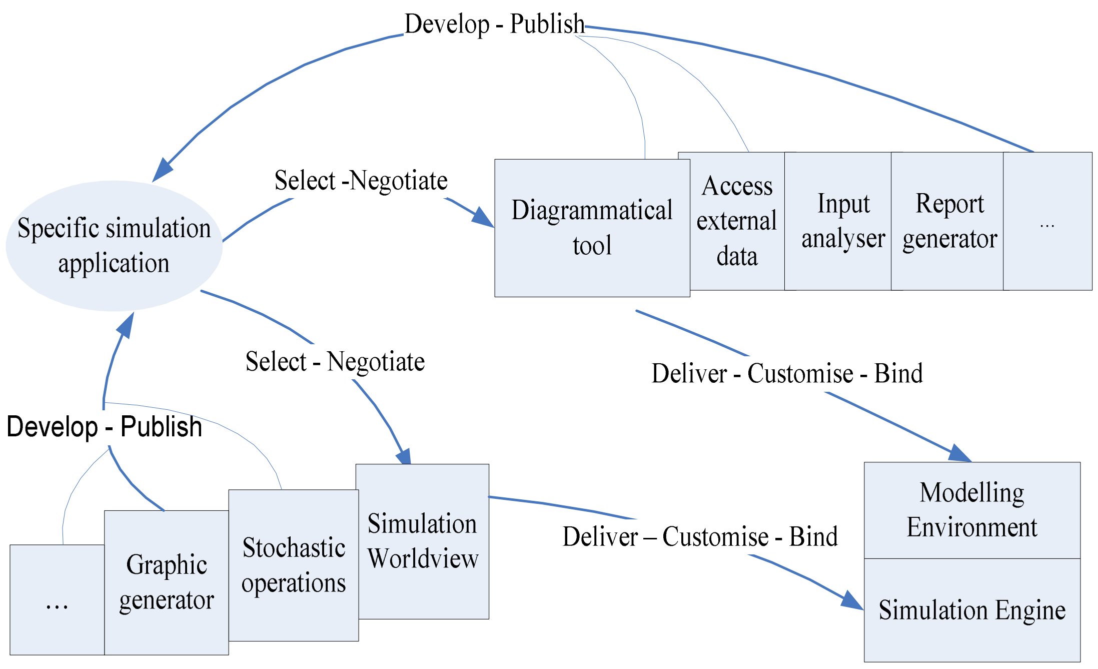

Fig. 3.12: Assembly of a specific simulation application by negotiating each of its features. The development cycle 'develop–publish-delivery' of components meets the assembly cycle 'select-customise–binding-running' through negotiation

The desirable scenario would be a development framework that not only allowed users to build their own solutions but also extended the capabilities for interconnecting different interdependent simulation packages. A simulation application could be a system of tailored systems. Every system could, if necessary, be based on different software applications and might run under different hardware/Operating System configurations. Simulation software would, therefore, provide multi-platformed and multi-packaged interoperability through "standard" data exchange protocols and formats (E.g. XML, HTTP, SOAP). Ideally, such simulation software systems would consist of open source components deployed as services to be consumed on demand. Users would be able to assemble their own solutions by selecting and adjusting the most appropriate components.

The alternative vision for simulation software described in chapter 4 derives from this overview of the future development strategy of simulation software.

3.4. THE CHAPTER IN CONTEXT

Chapter 2 and chapter 3 explain the conceptual knowledge base of modelling, simulation and computer science on which the current trends and possible future developments of simulation software is based. Simulation software has benefited from the co-evolution of these three disciplines and is expected to exploit their new advances to devise development strategies to respond to the on-demand paradigm imposed by the present mass customisation economy.

Chapter 4 introduces the DotNetSim project to explore an alternative approach to the development of simulation software solutions, which is based on the Microsoft integration philosophy and component-based paradigms embodied in the .NET Framework.