Software Frameworks in Quantitative Finance

Part I: Fundamental Principles and Applications to Monte Carlo Methods

Daniel J. Duffy and Joerg Kienitz

Abstract

We discuss a number of ongoing efforts when developing customizable software systems and frameworks for problems in Quantitative Finance. In particular, we examine the interplay between architecture, patterns and modern object-oriented and generic programming paradigms. In this way, we give some guidelines on how to develop software components, half-products and frameworks that can be modified and customised by developers. In this article we examine the applications to the Monte Carlo method and this article extends and generalizes the results in Duffy 2005.
In a future article we shall discuss object-oriented frameworks for the Finite Difference Method (FDM) and its application to option pricing models including implementation details in C++.

Background and Motivation

Saying that software is vitally important for applications in Quantitative Finance should come as no surprise to most people. The software industry has grown at an amazing pace since the early 1950's and 1960's. No longer do we need to develop applications using punch cards, paper tape and other devices but since a number of years we have had access to powerful desktop machines and modern software tools. The performance of hardware has been increasing at a phenomenal rate and this situation is likely to continue in the coming years. However, we believe that the underlying software paradigms will need to change if we wish to develop larger, faster and more reliable software systems.
We are witnessing some new and important developments in the related areas of soft- and hardware systems for application development and they will undoubtedly influence the course of software development projects in the coming years. First, not only are 'sequential' desktop computers becoming more powerful but in the not-too-distant future we shall witness the entry of *multi-core computers* in mainstream application development, that is computers with several CPUs on one chip. This means that multi-threaded and shared memory applications are just around the corner as it were. This is a form of *fine-grained parallelism*. Second, *high performance computing* (HPC) methods are also becoming more popular in mainstream developing environments. Once belonging to the realm of rocket scientists and researchers, HPC is now becoming an attractive solution for several numerical problems and applications in Quantitative Finance, for example Monte Carlo simulation, Finite Difference and Finite Volume methods and high-frequency data acquisition and analysis. In particular, the Message Passing Interface (MPI) library can be used to implement *coarse-grained parallel* applications in languages such as Fortran and C++. Finally – and perhaps crucially – software engineering is maturing to the stage where we can design and implement larger systems than in the past. In particular, the slow acceptance of design and system patterns, architectural styles and domain architectures seems to imply that developers are attempting to learn design principles before they start programming.

The advantages for Quantitative Finance are:

- *Performance*: a multi-threaded or parallel program – if properly designed and implemented – runs faster than a single-processor program. For example, a Monte Carlo simulation on a 16-processor machine is typically 10 times faster than the same program on a single processor machine
- *Accuracy*: this requirement has to do with a program giving desired or correct results. A parallel program can be designed in such a way that it produces – or is able to produce – more accurate results than a sequential machine in a given amount of time. A good example is the use of *domain decomposition* techniques for FDM

These advantages come at a cost, however. The definition of 'developer' will need to be deepened and extended. Prediction is difficult (especially the future) but a precondition for success in this new paradigm is the emergence of two major job functions; first, the architect who has a good knowledge of Quantitative Finance and who can translate

the problem domain to a high-level architecture and second – for want of a better word – the HPC developer who can design, implement and fine-tune a HPC solution that satisfies all the given requirements.

Review of the current Monte Carlo Solver

In a previous article (see Duffy 2005) we gave an introduction to a small software system in C++ to price a number of one-factor options, for example plain European options, arithmetic Asian calls and certain kinds of barrier options. We took a number of representative test cases and we presented our results using C++ to Excel interface drivers. We designed this system using the well-known design patterns from Eric Gamma and colleagues (see Gamma 1995). This system is useful as it provides us with a foundation upon which we can build a more general system (the source code for this problem is provided on the CD accompanying Duffy 2006).

In this article we describe our conclusions on the quality of the design in Duffy 2005 and we examine it from a number of viewpoints, namely:

1: Its suitability for n-factor derivatives and more complex derivatives
2: The design patterns and techniques used: are they scalable to larger problems?
3: How we used C++, which paradigms did we use?
4: Implementing the MC Solver in multi-threaded and parallel software environments

We discuss each of these viewpoints in some detail and we shall see that our initial design needs to be modified if we wish to have a system that can evolve as new requirements arise and when modifications must be carried out on the existing system. To this end, we now mention some of the issues that need to be addressed and features that need to be realised in the new version of the software. In the ensuing sections we discuss how to realise these features using a combination of high-level patterns and modern programming paradigms in C++.

Some new Requirements

The software system that we created for the Monte Carlo engine as described in Duffy 2005 was scoped in such a way that we could use it as a stepping-stone for more complex applications. The experience that we gained allowed us to discover a number of different viewpoints that are related to the overall scope of the problem. To this end, we examine the following *dimensions*:

- D1: Financial problem: which kinds of derivatives do we wish to model?
- D2: Numerical methods: approximating the underlyings and their derivatives
- D3: Which software architectural styles and patterns to use?
- D4: Which design features of C++ to use during deployment?
- D5: The hardware platform on which the new Monte Carlo engine will run

These dimensions lead to the specification of a problem in a five-dimensional *design space*. We discuss each of these dimensions in some more detail in the coming sections. In particular, we specify the kinds of derivative products to model using the Monte Carlo method, the kinds of numerical approximations to be used and software-related issues such as the architectural style to be used and whether the software will run on a stand-alone desktop machine or in a cluster or high-performance network.

Derivatives

The Monte Carlo method has been extremely successful as a computational tool in many areas of science, engineering and of course Quantitative Finance. The method is easy to understand, design and implement and it has been used to price a wide range of financial products (see the references Jäckel 2002 and Glasserman 2004 for a discussion).
In general, we are interested in one-factor and n-factor derivatives such as correlation options, fixed income products and other types such as convertible bonds and cross-currency options as well as problems involving stochastic volatility. Furthermore, we need to model problems with early exercise features (the so-called American options). This is problematic in general because it represents an embedded optimal stopping optimization problem. Finally, we may wish to find derivative *sensitivities* and hedge parameters. One way to calculate delta, gamma and other Greeks is to perturb the underlying values, run the Monte Carlo simulation again and then apply finite difference methods to compute the sensitivities (Jäckel 2002). It is intuitively obvious that the computing time will increase if we wish to have this feature in a software system.

Having described the range of problems that Monte Carlo methods can be applied to, we now wish to describe how to represent derivatives using mathematical models. In contrast to finite difference methods – where the behaviour of the derivative quantity is described by a partial differential equation – for a successful application of the Monte Carlo method we first model the underlying variables (such as stock prices, interest rates and volatility) by stochastic differential equations, or SDE for short (see Kloeden 1995). An SDE in this context is a first-order equation containing both a deterministic part and a stochastic part. It describes the path of the underlying variables in a certain interval of time. The basic Monte Carlo simulation technique entails simulating the SDE a number of times up the expiry time, calculating the derivatives price for each 'run' of the simulation and then taking the average of all prices. Finally, we discount this averaged price and this then gives the desired derivatives price.

Numerical Methods

Having described a derivative quantity by an SDE we then need to find its solution. In general, it is not possible to find a solution in closed or analytical form (unless the SDE is

very simple or has a convenient form) and we must then resort to numerical approximations. One of the most popular techniques is the Finite Difference Method (FDM) and we use it to find stable and accurate approximate solutions to SDEs (Kloeden 1997, Duffy 2004). In general we must determine which SDE to approximate and provide it with all necessary parameters before we can commence with a numerical approximation.

The most popular finite difference schemes that approximate the solution of SDEs at the moment of writing seem to be the explicit Euler and Milstein schemes. Each scheme has its advantages and disadvantages; for example, we must take very small time steps if Euler is to produce accurate results while the Milstein method is difficult to extend to multi-factor SDEs. For this reason, it is important to look for robust and accurate schemes that are used in other domains. For example, there are a number of *predictor-corrector* schemes that are stable, accurate and easy to implement (see Kloeden 1997, Duffy 2004). A specific example is the Heun method; the predictor step is explicit Euler while the corrector step is a modified trapezoidal rule. The stability of these schemes is proved by the application of fixed-point theorems.

There are classes of finite difference schemes that have their origins in partial differential equation theory and applications (Duffy 2006A), molecular dynamics and classical mechanics and they can be used as numerical schemes for SDE. Among these are the so-called splitting methods, for example the Verlet scheme and Strang-Marchuk splitting methods.

Once we have chosen a finite difference scheme that approximates an SDE we realise that we must be able to generate random numbers; this is because the Wiener process (and possibly others such as jump and Levy processes) is a component of the SDE. In this case we need efficient random number generators and to this end we use the inversion method and this is generally considered to be preferable to older methods such as Box-Muller and Polar-Marsaglia, for example. In some cases we may wish to improve the efficiency of the numerical schemes by the use of techniques such as Quasi Monte Carlo using low-discrepancy sequences and other techniques as variance reduction methods.

High-Level Architectures and Frameworks

We have described the kinds of derivatives products that we wish to model and the numerical schemes that we shall use to calculate the price of such products. Once this has been done we must then create a software architecture – consisting of a network of systems and their connectors – that we subsequently design and implement and that satisfies certain requirements.

In one sense the software systems that we build are similar in style to how hardware systems are built. For example, a given system consists of loosely coupled subsystems in which each subsystem has one major responsibility and it provides services to other subsystems. For example, a suitable candidate for a subsystem is one that implements the functionality of a stochastic differential equation. It provides the following services:

- The ability to configure and initialise objects that realise an SDE
- Getting the properties of an SDE
- Updating the parameters of the SDE

There are two major attention areas when designing large systems; first, we find the subsystems and second we specify the system topology that describes the interconnections between the subsystems (for example, the topology could be a chain, a tree or a network). Once we know the subsystems and their relationships we can then start thinking about the *architectural style* to use (Shaw 1996). An architectural style – in general terms – is a description of a family of systems in terms of a pattern of structural organization. It defines a vocabulary of components (computational units) and the connector types between those components. Furthermore, the style also defines a set of constraints on how to combine components and connectors. In short, it states what does and does not constitute legal bindings between components in much the same way that your laptop can be connected to an overhead projector or to a power supply. There are several major architectural styles, each one suitable for a particular kind of application:

- Dataflow systems (for example, pipes and filters and batch jobs)
- Data centred systems (for example, Blackboard and traditional databases)
- Layered systems
- Hierarchical systems (for example, the PAC model, see Duffy 2004A)
- Independent systems that communicate by means of events

A given system is usually a combination of the above styles. At the moment of writing, we have developed a number of systems based on the Presentation-Abstraction-Control (PAC) model (see for example, Duffy 2004A and Duffy 2006). In fact, the initial software system that the authors developed for the Monte Carlo engine was based on this model. We gained much experience on the strengths of the model but we have seen that the Blackboard style – in conjunction with PAC – is useful for software development of a Monte Carlo engine.

We give a short overview of Blackboard style and why we think it is important for applications in Quantitative Finance. More detailed accounts are given in Buschmann 1996 and Jagannathan1989. It is mainly used for situations in which a problem needs to be solved by a group of independent and self-motivating experts (that we call *knowledge sources* or KS for short). The main components are:

- *Blackboard*: this is the data structure from the solution space. This structure is modified by the KSs. Of course, we need synchronization techniques to ensure that the data is correctly updated
- *Knowledge Sources (KS)*: independent modules and systems that contain specific knowledge. They get their data from the Blackboard but they may also have a buffer of (private) internal data
- *Control Structure*: the component that monitors change in the blackboard. It schedules what action to take next and in particular it determines the order in which KSs will operate on the different data in the Blackboard

One of the biggest differences between this style and others is that the final solution (that represents the system goal) is built incrementally. To this end, we need to produce partial solutions as well.
A conceptual example is shown in the figure. It is a re-engineered version of the UML (Unified Modeling Language) diagram for the Monte Carlo engine in Duffy 2005.

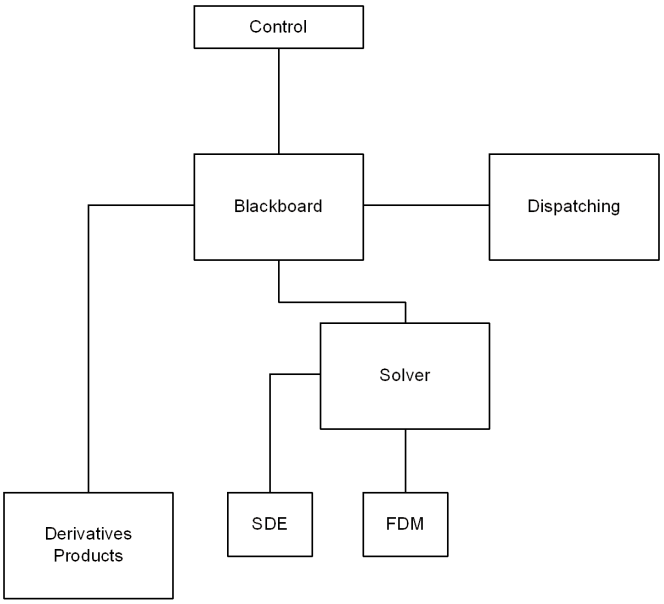

Figure 1 Conceptual Blackboard Model for new Monte Carlo Engine

Detailed Design and the Role of C++

C++ is one of the most popular languages for the development of software systems for Quantitative Finance. It is more than 25 years old and it supports the modular paradigm, object-oriented paradigm (OOP) and the generic paradigm (GP) in the form of C++ template functions and classes. Most developers are familiar with OOP but a smaller percentage of these developers use C++ templates to design and implement efficient, portable and robust software. There are a number of reasons for this situation some of which are that the syntax of C++ templating can be difficult to understand and second the idea of *designing software components* using C++ has not been given full justice in the literature. Using templates to design components is similar to how hardware is designed. For example, think about the *hardware ports* on your laptop; it *provides services* to other hardware entities and it *requires services* from other hardware entities. In a similar vein, we are designing the Monte Carlo engine based on similar principles; it consists of a number of *plug* and *socket* components. For example, an SDE component provides a set of interfaces for defining and accessing the mathematical description of a stochastic differential equation. It requires data and parameters from a GUI screen, database system and real-time data feed systems. Continuing, a component 'FDM' that approximates the solution of a stochastic differential equation requires the services from SDE and provides services such as discrete paths and other statistics to other components, for example an Excel application. In this sense we build large applications using interoperable and 'pluggable' building blocks. This idea is called *policy-based design* using C++ templates (see Alexandrescu 2001).

We conclude this section by emphasizing that the object-oriented and generic programming paradigms can be combined and they should not necessarily be seen as competitors. The relationship is as follows: we employ C++ templates to design and specify template-based components. In other words, we define contracts by specifying the component's provided and required interfaces at the meta level. Then we can instantiate the component's generic parameters by specifying, for example pointers to base classes. In this way we combine reliability and robustness at the meta level with dynamic run-time behaviour at the class level. In short, we are implementing parametric and subtype polymorphism, respectively. We thus draw a distinction between a type (for example, a template class) and a class (for example, a instantiated template class). It is the best of both worlds as it were.

On the Horizon: Multi-threading and High-Performance Computing

Once we have determined how to decompose a system into a set of communicating components we then have to decide how to implement these components. In many cases it may be possible to implement the full system on a single desktop machine using C++, Fortran or some other language. This approach may be acceptable for small problems but when a Monte Carlo simulation takes a few days to compute a solution to a pricing problem we will need to consider alternative hardware and software platforms. To this end, modern desktop computer support multi-threading, shared memory and we can create networks of computers that perform in parallel. In particular, the Message Passing Interface (MPI) library supports C++ and Fortran bindings and allows developers to create parallel Monte Carlo applications. We expand on this topic in a future article.

Summary and Conclusions

We have given an introduction to the issues to be ad-

dressed when designing large and customizable software systems for Quantitative Finance. For these kinds of problems it is necessary to create a high-level Blackboard architecture – consisting of systems known as knowledge sources – that is able to support current and future requirements and which can be customized by developers to suit their needs. We considered the Monte Carlo as one example of where the architecture can be used.
We shall elaborate on these general principles in future editions of *Wilmott* magazine.

References

Alexandrescu, A. 2001 *Modern C++ Design: Generic Programming and Design Patterns Applied* Addison-Wesley
Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad and M. Stal 1996 *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley and Sons, Chichester, UK
Duffy, D. J. 2004 *Financial instrument pricing using C++* John Wiley & Sons Chichester UK
Duffy, D. J. 2004A *Domain Architectures: Models and Architectures for UML Applications*, John Wiley and Sons, Chichester, UK
Duffy, D.J. and J. Kienitz 2005 *Monte Carlo Methods in Quantitative Finance: Generic and Efficient MC Solver in C++* Wilmott Magazine November
Duffy, D. J. 2006 *An Introduction to C++ for financial engineers An Object-Oriented Approach* John Wiley & Sons Chichester UK
Duffy, D. J. 2006*A Finite difference methods in financial engineering A Partial Differential Equation Approach* John Wiley & Sons Chichester UK
Gamma, E., R. Helm, R. Johnson and J. Vlissides 1995 *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley Reading MA
Glasserman, P. 2004 *Monte Carlo Methods in Financial Engineering* Springer New York
Jäckel, P. 2002 *Monte Carlo methods in finance* John Wiley & Sons Chichester UK
Jagannathan, V., R. Didhiaewal and L. S. Baum 1989 *Blackboard Architectures and Applications* Academic Press Inc.
Kloeden, P. and E. Platen. 1995 *Numerical Solution of Stochastic Differential Equations* Springer Berlin
Kloeden, P., E. Platen and H. Schurz 1997 *Numerical Solution of SDE Through Computer Experiments* Springer Berlin
Shaw, M. and D. Garlan 1996 *Software Architectures Perspectives on an emerging Discipline* Prentice Hall

Figure 4 Visualisation in Excel