Simulating an Object-Oriented Financial System in a Functional Language*

Chris Clack
Department of Computer Science
University College London
Gower Street
London, U.K.

Lee Braine, Keith Haviland, Owen Smith-Jaynes, Andy Vautier
Accenture†
60 Queen Victoria Street
London, U.K.

December 1998

Abstract. This paper summarises a successful application of *functional programming* within a commercial environment. We report on experience at Accenture's Financial Services Solution Centre in London with simulating an object-oriented financial system in order to assist analysis and design. The work was part of a large IT project for an international investment bank and provides a pragmatic case study.

1 Introduction

Functional languages such as Miranda‡ [Tur85,CMP94], Standard ML [MTH90], Haskell [PHA97] and Clean [Pla95] are used extensively in academia for research and teaching. These languages offer a number of well-known software engineering and formal methods benefits, including rapid development, clear and concise expression of algorithms, and complete type safety. However, despite these benefits, functional languages have experienced only limited acceptance by industry – mainly due to historical problems (such as inefficient compilers and limited input/output) that have been largely extenuated by research in the past decade.

This paper summarises a recent successful application of functional programming within a large IT project (over 100 developers). The project was undertaken by Accenture's Financial Services Solution Centre in London in partnership with an international investment bank and included the construction of a large financial software system using object-oriented and component-based techniques. A functional language was employed in the analysis and design stages to specify complex algorithms precisely and to simulate the high-level behaviour of the entire system. Following successful simulation, the specifications provided validated algorithm designs for subsequent implementation in C++. In this paper, we highlight the benefits which resulted from the functional simulation and also identify aspects that proved difficult to simulate.

2 Related Work

Simulation techniques are used extensively to model complex system behaviour. A variety of languages and tools are applied to problems ranging from analysing the movement of data packets in a network to predicting the value of financial derivatives – see [LK98] for an introduction to simulation modelling and its applications.

Functional programming has been applied to a wide range of problems ([RW95] summarises some recent applications). Illustrative examples of its use in simulation include:

- [PM93] presents a case study of Amoco's use of Miranda to simulate oil reservoirs;
- [GSW93] presents a functional programming solution to a fluid dynamics problem;
- [EL83] describes a high-speed digital simulation using a functional language approach;
- [PW93] and [Poh94] discuss the use of simulation programming in a functional language.

---

* This document is based on the paper "Simulating an Object-Oriented Financial System in a Functional Language", L. Braine and C. Clack, in *Proceedings of the 10th International Workshop on Implementation of Functional Languages (IFL'98)*, pages 487-496, September 1998.

† Accenture was formerly known as Andersen Consulting.

‡ Miranda is a trademark of Research Software Ltd.

Functional languages are also used in the closely-related areas of prototyping and specification; examples include prototyping computer-aided design applications [DB85], specifying image processing primitives [PC94], specifying complex tree transformations [Hec88] and describing telecommunications systems [Dem89].

However, there is little in the literature on commercial applications of functional languages to simulate financial systems. One reason for this is the often-held view that real-world processes can be represented more naturally in simulations using an object-oriented approach, as first advocated by SIMULA [DN66], than using a standard functional approach. Subsequent object-oriented languages, such as Smalltalk [GR83] and Eiffel [Mey91], widened this gap as functional languages offered few object-oriented features – see [BC96] for a brief history of research into object-oriented functional programming.

3 The Application

The subject of simulation activities was a large financial system within an international investment bank. The system contained a number of complex business processes requiring novel optimisation and approximation algorithms in order to perform effectively. Full details are both proprietary and confidential but, for the purposes of this paper, we present a brief overview in this section.

As an illustration of the level of program complexity, one key process involved partitioning a very large collection of data into two sets. A number of business constraints influenced, but not entirely determined, set eligibility for each datum; final eligibility was subsequently determined by the application using a custom optimisation algorithm.

The application contained a number of key data representations:

- Elements were organised into a partially-ordered set of queues, each queue itself a partial order.
- Queues were grouped according to common business criteria and distinct processing rules were applied to each group depending upon reference data.
- Many data elements contained cross-references to elements in other queues. These dependencies, when combined with the business constraints, introduced potential deadlocks into the system and required the construction of data groupings that spanned queues.
- By combining various data representations, the elements can be connected as a graph. Graph-theoretic algorithms (such as Tarjan's algorithm [Tar72] identifying the strongly connected components of a graph) could then be used to manipulate the data.

The above indicates an archetypal intractable problem – as the size of the input data set increases, there is a combinatorial explosion in the number of different allocation options to be explored and, for any useful size of input data set, a program might take millenia to find the optimal solution. Thus, it was essential to simulate different approximation techniques which would: (i) provide a good approximation to the optimal allocation, and (ii) provide an allocation that is guaranteed to obey the business constraints.

Solving the overall problem required simulation of two key modes of processing that correspond closely to: (i) real-time processing, and (ii) batch processing. For real-time processing, standard discrete-event simulation techniques were employed, such as annotating data items with explicit time tags. For batch processing, the approach was to rapidly prototype the algorithms in a functional language, including simulation of some object-oriented aspects and system functionality.

In summary, the goals for this simulation work were to:

1. rapidly develop and evaluate complex algorithms;
2. validate interactions between system components;
3. specify algorithms in an object-oriented style (for subsequent implementation in C++).

4 Simulation Language

We now consider the rationale behind simulating in a functional language to meet the goals identified in the previous section. The first goal favours the use of a functional language, but the last two goals favour an object-oriented language. In particular, the project methodology was object-oriented and employed a component-based approach, requiring modelling of the system's object-oriented algorithms and components.

Simply prototyping the algorithms in the final implementation language, C++, was not a viable option because it would have taken too long to develop underlying components before the high-level algorithms could be simulated. We decided to use a functional programming language rather than an object-oriented rapid application development language, such as Smalltalk, for the following reasons:

- The speed and clarity with which the algorithms could be expressed and validated was the most important consideration. The language features offered by functional programming (e.g. higher-order functions, lazy evaluation and complete type safety) provide benefits of clarity, conciseness and speed of expression in excess of many imperative languages. See [PM93] for illustrative metrics.
- The only parts of the system design that required precise simulation of their object-oriented aspects were the high-level algorithms and their method calls via component interfaces. The internals of all other components could be simulated using a purely functional approach to speed development. Additionally, the extra work involved in coercing a functional language to simulate the object-oriented aspects (e.g. inheritance hierarchy, dynamic despatch and mutable state) could be minimised by applying techniques from recent object-oriented functional programming research (see Section 6.3).
- Accenture's Financial Services Solution Centre in London has rapid application development expertise using functional programming and could utilise its academic links to obtain specialist input as required (e.g. functional programming simulation techniques).
- The execution speed of the simulation was not important, so the relatively slow speed of functional languages was not a disadvantage.

Miranda was the functional language of choice, mainly because it is commercially supported and provides an interpreted environment to assist rapid application development. If an interpreted environment were not as important, a compiled functional language such as Clean could have been used instead.

5 Simulation Environment

Simulation activities were performed by the Simulation Team over a period of 6 months. The software environment was the interpreted functional language Miranda running on the Solaris operating system. Multiple concurrent environments were executed on a SUN Enterprise 4000 server (8 UltraSPARC CPUs with 4GB shared RAM) accessed from networked PCs running Windows NT. Individual environments used between 100MB and 3GB RAM, depending on the complexity of the simulation.

6 Simulation Methods

Different levels of simulation fidelity were required for different parts of the system. For example, in order to provide algorithm specifications for subsequent implementation in C++, it was necessary to simulate those object-oriented algorithms precisely. Additionally, the use of component interfaces had to be simulated precisely in order to validate interactions between system components. However, only the behaviour (not the internal details) of those components had to be simulated, permitting the use of statistical approximation techniques in some cases.

There are important semantic differences between the object-oriented and functional paradigms – [BC96] overviews the main theoretical differences. Some of the obvious mappings are sufficient (e.g.

function signatures for modelling component interfaces), but others are insufficient (e.g. abstract data types for modelling classes). In order to simulate object-oriented designs using a purely functional language, it is necessary to resolve these semantic differences by applying techniques from object-oriented functional programming research.

The remainder of this section discusses the three key simulation methods that were used.

6.1 Real-Time Simulation at the Component Level

At the most abstract level, the application consists of a number of concurrent communicating components. The requirement was to model the real-time behaviour of these components, specifically the timing of data interchange between the components and the operation of the internal algorithms according to the data arrival times.

At this level, there was no requirement to model a complex class hierarchy (for example, components do not exhibit inheritance characteristics). Each component was concisely expressed as a lazily-evaluated "spinning function" – that is, a function which takes one or more infinite streams of data as input and produces one or more infinite streams of data as output (collected into a tuple). An additional accumulating parameter was used to hold the local state for each component.

The input and output streams modelled the independent, buffered, communication channels between the components; each stream was implemented as a lazy list of time-tagged data items. Network starvation (and possible deadlock due to blocking reads of the input streams) was avoided through the use of *hiatons* [Sto85]; whenever a component has nothing to output on a stream, it explicitly outputs an empty data item together with an appropriate time tag.

6.2 Behavioural Simulation of Component Internals

For many components, it was only necessary to model the component behaviour and a straightforward style of functional programming was used. It was observed that the use of functional programming features such as higher-order functions led to a great reduction in code size; this was particularly beneficial when developing complex algorithms as the functional notation provided a very concise and understandable specification. Furthermore, Miranda's interpreted environment and static type inference strongly supported an exploratory programming style; the benefits were *minimal build time* and *minimal run-time debugging*.

The functionality of some parts of this code was determined by the results of statistical approximation techniques. This enabled several parameters, such as event frequency, to be set explicitly and the resultant effects observed by executing the simulation.

6.3 Specifying Object-Oriented Algorithms

The most complex simulation task was to simulate the action of some algorithms at a sufficiently detailed level that the functional code could be used as an object-oriented specification of the algorithm to be subsequently implemented in C++.

This required simulation of several object-oriented features that are not provided by Miranda. In this case, the coding style adopted by the Simulation Team was similar to the style of target code produced by the CLOVER compiler [BC96,BC97a], particularly in the areas of simulating classes with inheritance, overloading, overriding, and dynamic despatch. Full details can be found in the research literature, but we rehearse the essence of the technique here:

- *Classes, inheritance and the meta type system*
  In order to simulate certain aspects of the object-oriented system, it was necessary to create a meta type system. For example, simulating subsumption (the notion of manipulating an object which could be one of several possible subtypes) required inheritance hierarchies to be flattened by creating a new type for each hierarchy that represented the root superclass, and then encoding each subclass as an alternative in an union type.

- *Dynamic method despatch*
  The object-oriented notion of dynamic method despatch conflicts fundamentally with the functional notion of static type safety. However, by using the meta type system mentioned above, efficient method despatchers were constructed that simulated dynamic despatch. These simply pattern-matched on the type constructor to dynamically select the appropriate method.
- *Simulating assignment*
  In order to provide detailed object-oriented specifications, it was necessary to simulate assignment. This included creating multiple single-assignment identifiers in order to represent multiple assignments to a single object. It should be noted, however, that recent functional research can ameliorate such a plethora of identifiers by applying suitable lexical scoping rules, for example the Clean language allows identifiers on the right-hand-side of an expression to be reused on the left-hand-side – they are then internally tagged with a number by the compiler (see [AP97] for further details).

7 Results

During the process of simulation, several algorithms were:

1. developed using Miranda;
2. validated through high-level simulations of the entire system;
3. used as specifications for subsequent implementation in C++.

These activities proved highly successful and the key results are reported below:

- *Rapid development*
  Miranda code was produced far more rapidly than C++ code with similar functionality (this has been estimated as a factor of approximately 5 times). We expect that the productivity gain was mainly due to two key factors: (i) the often-expressed desirable language features of functional programming (higher-order functions, lazy evaluation, automatic memory management, etc.), and (ii) the traditional benefits of an interpreted environment (negligible build time, rapid turnaround cycle, interactive testing of individual functions, etc.).
- *Concise expression*
  Miranda specifications were substantially more concise than C++ code with similar functionality and were therefore much more understandable. As an illustrative example, a key algorithm expressible in 6 pages of Miranda code translated into approximately 25 pages of C++ code. The conciseness of functional programs is often reported in the research literature and as a result of scientific applications; here it has been validated for a financial application.
- *Simulation as executable specification*
  By employing a functional language at the analysis and design stages, complex processes were simulated in advance, allowing designs to be optimised early in the project lifecycle. The functional programs also served as *executable specifications* [Tur85a] – the algorithms were tested on actual data and, once confident of correctness, used as validated specifications for the final designs. Unit testing of the actual system found that fewer errors existed in C++ code that had been first simulated. Considering that this included many of the most complex algorithms in the system, we believe that this result is one of the most important justifications for using simulation work early in the lifecycle of a complex project.
- *Limitations*
  Simulations required extensive computing resources. The most notable was a very large memory space – up to 3GB of heap space in some simulations. The reasons are both application-specific (e.g. the requirement for full and complete traces of events, used later for analyses into the operational behaviour of the algorithms) and functional language-specific (e.g. the maintenance of closures during lazy evaluation). Although execution time of simulations was not important, the most complex took approximately 2 hours.

8 Further Work

There is potential for further application of simulation and specification techniques within this current project and we are also liaising with Accenture's "Centre for Process Simulation and Modeling" on the opportunity for novel simulation techniques in other projects.

We have also identified several ways in which current functional language implementations could be improved in order to support simulation and specification activities for financial applications:

- Our experience indicates that functional programming would benefit from the incorporation of more object-oriented features. Not only do general simulation activities benefit naturally from an object-oriented approach but, with many actual software systems being built using component/object-oriented techniques, simulations can more accurately specify the actual system. However, incorporating object-oriented features into functional programming is not trivial and should be regarded as on-going research.
- On a large project, many specifications are delivered as part of a set of documents created using a standard wordprocessor, such as Word or WordPerfect. This necessitates cutting-and-pasting executed specifications into project documentation; the procedure is potentially error-prone and could be resolved by *executing the wordprocessing documents themselves*. This "literate script" programming style is available in Miranda for the LaTeX typesetting system; we would like to see other languages adopt this style and extend it to Word as well as LaTeX.
- A link with object-oriented design tools, such as Rational Rose, could allow preliminary high-level designs to be generated automatically from validated simulation code. This may require code annotations to guide the generator, but would be particularly useful for creating top-level components and component interface diagrams from simulation classes.
- Many simulation tools permit process execution to be visualised through the use of graphical animation. This allows the behaviour of complex algorithms to be understood by a wider audience (such as at the proposal stage and during training) and can often provide additional insights (such as identifying bottlenecks). Animation could be added to functional simulations by extending state classes to capture event information and output it for post-simulation animation.
- We would like to explore the use of other functional languages in order to benefit from both additional language features and from greater execution speed. However, our experience indicates that for rapid simulation and specification purposes one of the most useful system features is an interpretive environment. We urge functional language implementors to provide both interpreted and compiled modes of execution.

9 Summary and Conclusion

In this paper, we have presented a successful application of functional programming to the simulation and specification of an object-oriented financial system. A variety of simulation methods were employed to model the different parts of the system at appropriate levels of detail. This included simulation of a real-time system at the component level, behavioural simulation of component internals and detailed specification of core object-oriented algorithms.

The results were highly successful and highlighted the key benefits of using a functional language. These included very rapid development (we estimate 5 times faster than prototyping in C++), concise expression (a validated design in 6 pages of Miranda translated to about 20 pages of C++ code) and use as an executable specification which can be tested with real data. Subsequent phases of the project demonstrated that a functional language can serve as a design language for the object-oriented implementation, with the validated Miranda designs resulting in high quality C++ code.

In conclusion, this project has demonstrated the great worth of including simulation activities early in the lifecycle of a complex financial project and, in particular, identified some key benefits obtained by specifying and simulating in a functional language.

Acknowledgements

We wish to acknowledge the contributions made during the course of this work by Tony Wicks (of SearchSpace Ltd.) to the data analysis.

References

[AP97] P. Achten and R. Plasmeijer. Interactive Functional Objects in Clean. In *Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL'97)*, pages 387–406, September 1997.

[BC96] L. Braine and C. Clack. Introducing CLOVER: An Object-Oriented Functional Language. In W. Kluge, editor, *Implementation of Functional Languages, 8th International Workshop (IFL'96), Selected Papers*, Lecture Notes in Computer Science 1268, pages 1–20, Springer-Verlag, September 1996.

[BC97] L. Braine and C. Clack. Object-Flow. In *Proceedings of the 13th IEEE Symposium on Visual Languages (VL'97)*, pages 422–423, September 1997.

[BC97a] L. Braine and C. Clack. The CLOVER Rewrite Rules: A Translation from OOFP to FP. In *Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL'97)*, pages 467–488, September 1997.

[CMP94] C.Clack, C.Myers, E.Poon. *Programming with Miranda*, Prentice Hall International, ISBN 0-13-192592-X, 1994.

[DB85] A. Duijvestijn and G. Blaauw. Prototyping in Computer-Aided Design Applications Using a Functional Language. In *Proceedings of the First International Conference on Computer-Aided Technologies*, pages 326–333, September 1985.

[Dem89] J. Deman. Description of Telecommunication Systems by Means of a Functional Language. In *Proceedings of the 7th International Conference on Software Engineering for Telecommunications Switching Systems*, pages 91–94, July 1989.

[DN66] O. Dahl and K. Nygaard. SIMULA: an ALGOL-Based Simulation Language. In *Communications of the ACM*, September 1966.

[EL83] M. Ercegovac and S. Lu. A Functional Language Approach in High-Speed Digital Simulation. In *Proceedings of the 1983 Summer Computer Simulation Conference*, pages 383–387, July 1983.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[GSW+93] P. Grant, J. Sharp, M. Webster, and X. Zhang. Functional Programming for a Computational Fluid Dynamics Problem. In *Proceedings of Computational Mechanics in UK*, pages 75–79, 1993.

[Hec88] R. Heckmann. A Functional Language for the Specification of Complex Tree Transformations. In H. Ganzinger, editor, *Second European Symposium on Programming (ESOP'88)*, Lecture Notes in Computer Science 300, pages 175–190, Springer-Verlag, March 1988.

[LK98] A. Law and D. Kelton. *Simulation Modeling and Analysis*, 3rd Edition. McGraw Hill, 1998.

[Mey91] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[PC94] I. Poole and D. Charleston. Formal Specification of Image Processing Primitives in a Functional Language. In *Proceedings of the 12th IAPR International Conference on Pattern Recognition (Conference A: Computer Vision and Image Processing)*, pages 539–542, October 1994.

[PHA+97] J. Peterson, K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, E. Meijer, S. Peyton-Jones, A. Reid, and P. Wadler. *Report on the Programming Language Haskell. A Non-strict, Purely Functional Language*, Version 1.4, April 1997. Available from `http://haskell.org/report/`

[Pla95] M. Plasmeijer. Clean: a Programming Environment Based on Term Graph Rewriting. In *Proceedings of the Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, Electronic Notes in Theoretical Computer Science, pages 233–240, Elsevier, 1995.

[PM93] R. Page and B. Moe. Experience with a Large Scientific Application in a Functional Language. In *Proceedings of the 6th Conference on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 3–11, June 1993.

[Poh94] W. Pohlmann. Simulation Programming in a Purely Functional Language 2 - How to Do It With Some Optimism. In *Proceedings of the 1994 European Simulation Multiconference – Modelling and Simulation 1994*, pages 84–87, June 1994.

[PW93] W. Pohlmann and K. Weiss. Simulation Programming in a Purely Functional Language. In *Proceedings of the 1993 European Simulation Multiconference – Modelling and Simulation 1993*, pages 176–180, June 1993.

[RW95] C. Runciman and D. Wakeling, editors. *Applications of Functional Programming*. UCL Press, 1995.

[Sto85] W. Stoye. The Implementation of Functional Languages Using Custom Hardware. PhD Thesis, Computer Laboratory, University of Cambridge, December 1985.

[Tar72] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, Part 1, Volume 2, pages 146-160, June 1972.

[Tur85] D. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the 2nd Conference on Functional Programming Languages and Computer Architecture (FPCA'85)*, Lecture Notes in Computer Science 201, pages 1–16, Springer-Verlag, 1985.

[Tur85a] D. Turner. Functional Programs as Executable Specifications. In C. Hoare and J. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 29–54, Prentice Hall, 1985.