# Agile model-driven engineering of financial applications

Kevin Lano, Howard Haughton, Sobhan Yassipour-Tehrani, Hessa Alfraihi

Dept. of Informatics King's College London, London, UK Email: { kevin.lano, sobhan.yassipour\_tehrani, hessa.alfraihi }@kcl.ac.uk

*Abstract*—Flexibility and agility are key properties for financial applications development: systems need to be easy to change and evolve to keep up with changing market conditions and opportunities, and changes in the regulatory environment.

In many cases, Excel with VB is the main platform used for the development of applications in the finance industry. This offers advantages of flexibility and user customisation of applications in rapid development cycles. However, this dependence upon a proprietary software package causes problems when applications must be migrated to an alternative platform, or when precise control over functionality is needed. Modeldriven development (MDD) and model transformations have a good potential for improving these aspects of financial development. In this paper, we describe techniques that enable the use of Excel as an MDD front-end for the specification and development of financial applications in a rigorous manner.

# I. INTRODUCTION

Software development in the financial services sector is usually carried out on Excel and VBA-based platforms, with modelling and prototyping performed using Excel [8], [7]. After prototyping, the Excel/VBA solution may be manually re-coded in an implementation platform (Figure 1). This approach has deficiencies with regard to maintainability, software architecture and documentation. Separate spreadsheet-based applications are difficult to integrate into composite systems, and lack the structure and modularity of more advanced programming environments.



Figure 1. Conventional financial application development process

Starting in 2014, the MDD team at King's College were asked to look at alternative model-based approaches for specifying and implementing financial software for multiple platforms. The characteristics of such software represent a challenge for MDD, because: (i) they involve complex mathematical computations, which need to be specified in a clear and comprehensible manner and related to the appropriate finance theory, and (ii) they have high requirements for efficiency and accuracy. In addition, (iii) a high degree of flexibility and agility is typically necessary.

Our solution to these challenges involved using agile MDD and executable modelling as follows:

- Specification of applications in an executable UML dialect, UML-RSDS [9], which supports efficient code generation in Java, C#, C and C++.
- Adoption of an agile process based on the Scrum method [14].
- Using Excel to provide a user-friendly means of defining UML-RSDS specifications, so that end-users do not need to write or understand OCL.

We considered that this approach was suitable to address issues (i), (ii) and (iii), because UML-RSDS supports high-level declarative specification of systems using UML class diagrams and use cases, enabling the system specification to be expressed in the context of relevant financial concepts, and to be structured in a systematic manner. In addition, it supports the execution of specifications by the synthesis of designs and programming language implementations. UML-RSDS supports agility by enabling rapid modification of system specifications: only one application model (integrating class diagrams and use cases) needs to be modified, in contrast to MDD approaches which involve multiple linked models whose consistency needs to be managed at each specification change. The integration of agility and MDD in this case is achieved by making the MDD aspect as lightweight as possible, and by adopting a 'model as code' viewpoint, with agile practices applied to the application models instead of the code.

In Section II we give an overview of the approach. In Sections IV and V we describe an example financial application we have developed. Section VI gives related work, and Section VII gives conclusions.

#### II. INTEGRATING AGILE AND MODEL-DRIVEN DEVELOPMENT

The key principles of agile development include (agilemanifesto.org): (i) satisfy the customer through early and continuous software delivery; (ii) welcome changing requirements; (iii) deliver working software frequently (every 2 weeks to every 2 months); (iv) business people and developers to work together daily; (v) rely on face-to-face communication to convey information; (vi) continuous attention to software quality; (vii) simplicity is essential.

Integration of agile and MDD aims to achieve these principles in the context of model-centered development, instead of traditional code-centered development. Research on integration has produced a number of agile MDD approaches, such as MDD-SLAP [15] and Hybrid MDD [4]. These usually focus on a particular domain, and do not cover all lifecycle stages. In contrast, UML-RSDS can be used as the basis of a general-purpose agile MDD approach, with specialised interfaces and tools for particular domains. We adopt the ideas of three-phase iterations from [15] and parallel tooling teams from [4] (as shown in Figure 2).



Figure 2. UML-RSDS agile MDD process: iterations

To achieve the agile principles, we adopt the following agile practices with UML-RSDS: (i) short iterations (principles (i) and (iii)); (ii) model refactoring (principle (vi)); (iii) emphasis on model and specification simplicity (principle (vii)); (iv) product and iteration backlogs (principles (i), (ii)); (v) Scrumboards (principles (i), (ii)); (vi) daily standup meetings, continuous integration and testing (principle (vi)); (vii) Customer involvement in modelling (principle (iv)).

The following MDD practices are used: (i) metamodelling; (ii) transformations; (iii) executable modelling.

Within each iteration, phases of requirements analysis, specification, implementation and testing are applied to each task, using exploratory prototyping based on partial specifications, and review of the prototypes/specifications with stakeholders/team members. Figure 2 shows these cycles for the main team on the LHS. Reuse of components, and the contribution of new/updated components to a library, can take place during these iterations. A tool development team applies a similar agile process (on the RHS) to respond to tool support requirements from the main team.

Prototyping and testing of completed specifications is followed by integration with other software elements, and iterative revision of specifications as necessary to pass tests and efficiency requirements.

### III. MDD SPECIFICATION USING EXCEL

Class diagrams and use cases are the core specification notations used in UML-RSDS to specify applications (including model transformations). OCL constraints are used to define the logical postconditions of operations and use cases, and to define the procedural steps (activities) of their designs [9]. Typically, a migration or refinement model transformation involves the mapping of instances a : A of a source entity type A to instances b : B of a target entity type B. Such mappings are formally expressed by postconditions

$$A :: ACond \Rightarrow B \rightarrow exists(b \mid P(self, b))$$

of a use case that defines the transformation. The postcondition asserts that (at termination of the transformation) for every instance *self* of *A* that satisfies *ACond*, there must exist an instance *b* of *B* that satisfies P(self, b).

Many financial applications involve processing of a similar form, where some source data, such as market data, is used as input to derive secondary information and analysis via various functions and procedures. For example, the derivation of the internal rate of return or Macaulay duration for a coupon bond, or the derivation of a price for a derivative security, based on market information.

In Excel, the application data is typically laid out as a table, with columns for each significant attribute of the data (Table *Bond* in Figure 3). Derived data may be placed in cells of additional columns of the table, or in the cells of separate tables (such as *BondYield*). The second situation corresponds directly to rules of migration/refinement model transformations, where A is a source class (eg., *Bond*), and B a target (eg., *BondYield*).



Figure 3. Excel spreadsheet example

The correspondence between Excel and UML is as follows:

- An Excel table defines a UML class with attributes for each column of the table, with the attribute types obtained from the data in an example row of the table. For example, the *Bond* table defines a UML class *Bond* with an integer attribute *Settlement*, double attribute *Price*, etc. Multiple source and target classes can be defined as tables in the same spreadsheet.
- The spreadsheet functionality corresponds to a model transformation (defined as a use case, *sheet*) operating on the class diagram corresponding to the tables.
- When data item T of table B is defined as a function f(A.S) of the data S of another table A, this corresponds to a

postcondition constraint

$$A :: true \Rightarrow B \rightarrow exists(b \mid b.BId = AId \& b.T = f(S))$$

in cases where f expects single data values, and to a constraint

::  

$$AS = f(A \rightarrow collect(S)) \Rightarrow B \rightarrow exists(b \mid b.T = AS)$$

in cases where f expects aggregate data (eg., f is a function such as the SUM, AVERAGE, etc of a column/sequence of individual data items).

In the first case, each instance (table row) of A is mapped to an instance (table row) of B. In the second, a single Binstance is created with its attribute values derived from aggregates of A data.

The dataflow dependencies of the spreadsheet must be acyclic [7], that is, if data item y depends on data x directly or indirectly, there cannot be a dependency in the reverse direction. In the UML representation, this means that the constraints can be ordered in a sequence where all data that is read by one constraint  $C_i$  is directly available or has been produced by earlier constraints  $C_j$  for j < i. This is the usual organisation of use case constraints in UML-RSDS, using the Phased Construction transformation design pattern [10].

An example is shown in Figure 5, which is derived from Figure 3. The use case *sheet* expresses the overall functionality of the spreadsheet application. The derivation of UML from Excel spreadsheets in CSV format is performed automatically by the UML-RSDS tools.

Excel functions such as SUM, AVERAGE, DATE, RATE, DURA-TION, PRICE, etc have been formalised using OCL. These definitions are provided in a library class *ExcelLib*. For example:

In our style of Excel specification, logical expressions such as Bond.Settlement are used to denote data items/groups, instead of physical spreadsheet locations/ranges such as B7:B17. Logical specifications of Excel functions and other financial and numerical functions are provided in ExcelLib using OCL, facilitating analysis and formal verification. In the reverse direction, the tables of a spreadsheet can be synthesised from a class diagram: each concrete class maps to a table, with the attributes of the class mapped to table columns.

In Sections IV and V we describe two iterations of an agile MDD case study in finance, for a bond-pricing application. This was developed in collaboration with an industrial expert in financial software construction.

#### IV. CASE STUDY ITERATION 1: INTERNAL RATE OF RETURN

The bond-pricing application iteration 1 computes the internal rate of return (IRR) of a bond. The IRR measures the quality of an investment: the effective rate at which the investment returns value over its term. A *coupon bond* consists of a series of cash flows: an initial payment (-ve cash flow), followed by coupon payments/dividends (+ve cash flows to the investor), and repayment of capital at the end of term (Figure 4). Eg.: a £100 bond is purchased for a price of £105, with an investment term of 10 years, it pays 8% annual interest bi-annually (20 payments of £4), then £100 capital repayment (redemption) at the end of the term. In contrast, a *zero-coupon* bond only pays back accumulated gains and capital at the end of its term.



Figure 4. Finance domain bond classes in ExcelLib library

Using discrete compounding, the IRR is the rate r such that:

 $price = \sum_{i=1}^{flows.size} flows[i].amount/(1+r)^{flows[i].timePoint} + redemption/(1+r)^{flows.last.timePoint}$ 

Time can be measured in days, months, years, etc.: r will be the rate wrt this measure. Generally, r can be estimated by numerical approximation techniques, eg., the secant method or bisection. Excel uses the secant procedure to estimate the IRR, in the function RATE (npays : int, payment : double, price : double, fv : double) : double where npays is the number of coupon payments to be made, payment is the constant amount of each payment, price is the present value of the investment, and fv (optional) is the redemption amount at termination of the investment. The result value is an approximation to the IRR r. In ExcelLib we provide a mathematical specification of RATE based on the secant procedure. From the user's perspective, they can use RATE in the same way as in Excel. Behind the scenes, they are using the ExcelLib library specification written in OCL, and from this, procedural code in Java, C#, C or C++ can be automatically generated. Note that in our version of OCL, computational numeric types are used: 32-bit and 64-bit integers, and IEEE 754 standard doubleprecision floating-point numbers. This reduces the semantic gap between the specification and implementation, improving verifiability.

The operation *defineFlows*1() of Figure 4 is specified as:

```
defineFlows1()
pre: true
post:
Integer.subrange(1,
  (maturity*frequency).floor)->forAll( t |
    $CashFlow->exists( f | f.amount = coupon &
    f.timePoint = (t*1.0)/frequency &
    f : flows ) )
```

This creates all the positive cash flows from coupon payments over the lifetime of the bond. (*maturity* in *\$Bond* is the term of the bond). To derive the IRR of a series of bonds, the user specifies two tables in a spreadsheet, as in Figure 3. The *Bond* table identifies the attributes of the *Bond* class, and their types, whilst the *BondYield* table also defines the mapping of *Bond* to *BondYield*, using the *RATE* function. The function in cell D12 is

```
RATE((Bond.Maturity - Bond.Settlement) *
Bond.Frequency,
Bond.Coupon,
Bond.Price, 100)
```

The UML-RSDS tools convert the spreadsheet to the class diagram of Figure 5, and derive the following postcondition constraint of the use case *sheet*:

The meaning of this postcondition is that for each *Bond* instance, a *BondYield* instance will be produced with its *Yield* value set to the IRR computed using *RATE*. In addition, we also compute the discrete Macaulay duration of the bond (the time to maturity of the equivalent zero-coupon bond):

$$\begin{aligned} \text{duration} &= \\ & (\Sigma_{i=1}^{\text{flows.size}} \text{flows}[i].\text{timePoint}*\text{flows}[i].\text{amount}/\\ & (1+\text{yield})^{\text{flows}[i].\text{timePoint}})/\\ & (\Sigma_{i=1}^{\text{flows.size}} \text{flows}[i].\text{amount}/(1+\text{yield})^{\text{flows}[i].\text{timePoint}}) \end{aligned}$$

This computation can use the previously-computed yield value of each *BondYield* instance. In the spreadsheet, the value of the Duration column (cell E12) is expressed as:

```
DURATION(DATE(Bond.Settlement,1,1),
DATE(Bond.Maturity,1,1),
Bond.Coupon, BondYield@pre.Yield,
Bond.Frequency)
```

The @pre annotation indicates that the Yield derived by the preceding equation should be used. A continuous duration function DURA-TIONC is also available.

# V. CASE STUDY ITERATION 2: YIELD CURVE FITTING

In iteration 2 of the bond-pricing application, the yield and duration derived in iteration 1 are used as input datapoints for a process which attempts to find the best-fitting yield curve for interest rate data, according to some yield curve model such as the Nelson-Siegal or Nelson-Siegal-Svensson models. The adapted Nelson-Siegal-Svensson (NSS) model [5] which we use is defined by







Figure 5. Class diagram derived from Excel spreadsheet

the equation:

$$y(t) = \beta_1 + \beta_2 * (1 - exp(-t/\lambda_1))/(t/\lambda_1) + \beta_3 * ((1 - exp(-t/\lambda_2))/(t/\lambda_2) - exp(-t/\lambda_2))$$

This models how the yield y(t) of a bond varies depending on its duration t.

The yield curve in this model has a long-term rate component  $(\beta_1)$ , a short-term (2nd factor), and a 'hump' (3rd factor). The problem is to estimate the  $\beta_i$  and  $\lambda_j$ , given market data – 'fitting the curve' to this data. Estimation procedures to find the NSS parameters that fit a given set of market bond data include genetic algorithms (GA) and Matlab's fminsearch using a simplex algorithm [5]. We use a combination of genetic algorithms are specified using OCL in ExcelLib.

The adapted Nelson-Siegal-Svensson model is specified in UML-RSDS by the *nelsonsiegalx* and *nsx* functions of the finance library class *\$Bond* (Figure 4). These are then used to evaluate the fitness of a candidate solution (a tuple of values for the parameters  $\beta_1$ ,  $\beta_2$ ,  $\beta_3$ ,  $\lambda_1$ ,  $\lambda_2$ ), based on the sum of squares of differences between the actual and predicted interest rates for the market data.

NSS-estimation is performed by an ExcelLib function ESTIMATENSX(yields: Sequence(double), durations: Sequence(double)) : Sequence(double). This is called on BondYield.Yield, BondYield.Duration (considered as sequence data) to populate a result table *NelsonSiegal*.

ESTIMATENSX initialises a population for a genetic algorithm using the bond interest rate (yield) and duration data, and iterates the genetic algorithm until there is no further improvement in the maximum fitness in the population. The best result is then passed to a numerical optimiser, in our case the simplex algorithm.

Having derived the NSS parameters, these can then be used to calculate the fair price of bonds of different durations, using a formalisation of the Excel PRICE function. This functionality was developed in iteration 3. We compared our formalisation of this bond-pricing procedure with a traditional Excel/VBA solution, in terms of software size, accuracy and efficiency A Java implementation of the UML-RSDS specification was used, applied to a test case of 8 coupon bonds ranging from 1 year to 12 year terms. Code size was reduced from 34K to 22K, accuracy improved from  $2 \times 10^{-4}$  to  $6 \times 10^{-6}$ , however execution time increased from 550ms to 730ms, mainly due to use of a genetic algorithm.

The curve produced from the estimated parameters using our approach is shown in Figure 6.



Figure 6. Estimated NSS curve

#### VI. RELATED WORK

The Kapital system used by JP Morgan has been a highly successful alternative approach in finance [1], enabling financial engineers to quickly prototype and implement new financial products. However, Kapital is based on a specialised platform (Smalltalk). We consider that basing a financial engineering platform on language-independent UML and OCL specifications will provide more flexible and durable support than a language-specific solution. In [11] we describe the risk evaluation of multiple investment instruments [6] using UML-RSDS, this direct MDD approach was technically effective but was considered too difficult for financial engineers to use. As an alternative, we investigated the Excel-based approach described here. An application of agile MDD in finance is given in [13], however this is focussed on the management of data and message formats instead of numerical computations. In [2], spreadsheets are mapped to relational databases in order to optimise the spreadsheets. In [3], spreadsheet evolution is supported via a mapping to a semantic representation. These papers use transformations to support spreadsheet-based development, whilst our work uses spreadsheets to define model transformations. A similar data-flow logical specification of financial applications is described in [7], although all the data derivations are performed within a single table, whilst we separate data into possibly multiple source, target and intermediate tables.

QuantLib (quantlib.org) is a widely-used library of financial analysis functions, written in C++ (interfaces and partially-ported versions also exist for other programming languages). In contrast to QuantLib, we define financial functions in a platform-independent formalism (OCL), from which language-specific implementations can be automatically generated. In future work, we aim to include substantial parts of the QuantLib libraries in our OCL financial library. These could then be used directly in the Excel-style specifications used by our approach.

## VII. CONCLUSIONS

We have described techniques to combine agile MDD with conventional Excel-based financial application development. Excel spreadsheets are interpreted as UML class diagrams operated on by model transformation specifications, and executable implementations of these transformations can then be synthesised in production-quality languages. In future work we will evaluate the usability of the approach with financial engineers, and extend the OCL finance library to provide comprehensive facilities for the use of MDD in the finance domain.

## REFERENCES

- Cincom, JP Morgan derives clear benefits from Cincom Smalltalk, www.cincom.com/pdf/CS040819-1.pdf, accessed October 2016.
- [2] J. Cunha, J. Saraiva, J. Visser, From spreadsheets to relational databases and back, PEPM 2009.
- [3] J. Cunha, J. Fernandes, J. Mendes, H. Pacheco, J. Saraiva, Bidirectional transformations of model-driven spreadsheets, ICMT 2012.
- [4] G. Guta, W. Schreiner, D. Draheim, A lightweight MDSD process applied in small projects, Proceedings 35th Euromicro conference on Software Engineering and Advanced Applications, IEEE, 2009.
- [5] M. Gilli, S. Grosse, E. Schumann, *Calibrating the Nelson-Siegal-Svensson model*, COMISEF working paper WPS-031, 2010.
- [6] O. Hammarlid, Aggregating sectors in the infectious defaults model, Quantitative Finance, vol. 4, no. 1, 2004, pp. 64–69.
- [7] Y. Horry, Financial information description language and visualisation/analysis tools, Computer Languages, Systems and Structures 50 (2017), pp. 31–52.
- [8] A. Kumiega, B. Van Vliet, A software development methodology for research and prototyping in financial markets, Proc. European Spreadsheet Risks Int. Grp., pp. 107–127, 2006.
- [9] K. Lano and S. Kolahdouz-Rahimi, Constraint-based specification of model transformations, Journal of Systems and Software, vol. 88, no. 2, February 2013, pp. 412–436.
- [10] K. Lano, S. Kolahdouz-Rahimi, *Model-transformation Design Patterns*, IEEE Transactions in Software Engineering, vol 40, 2014.
- [11] K. Lano, H. Alfraihi, S. Yassipour-Tehrani, H. Haughton, Improving the Application of Agile Model-based Development: Experiences from Case Studies, ICSEA 2015.
- [12] J. Nelder, R. Mead, A simplex method for function minimisation, Computer Journal, 7, pp. 308–313.
- [13] M. B. Nakicenovic, An Agile Driven Architecture Modernization to a Model-Driven Development Solution, International Journal on Advances in Software, vol 5, nos. 3, 4, 2012, pp. 308–322.
- [14] K. Schwaber, M. Beedble, Agile software development with Scrum, Pearson, 2012.
- [15] Y. Zhang, S. Patel, Agile model-driven development in practice, IEEE Software, vol. 28, no. 2, pp. 84–91, 2011.