Computer Aided Finance
Another journey in the quest for the
Holy Grail of financial engineering

Stefan Dirnstorfer

Email: stefan@thetaris.com

Andreas J. Grau

Leopoldstrasse 244, 80807 München, Germany

Email: grau@thetaris.com

Abstract

With a unified theory for pricing any derivative still eluding the financial engineering community, a new approach to product modelling is presented that may help the industry deal with the new trends towards ever-increasing volume and complexity of the products traded. We propose a product description language that is both simple and general. It is suitable for computer processing, enabling tools to automatically derive pricing algorithms. Important features of this language are shown and examples for a wide range of derivatives presented.

Since the dawn of risk neutral derivative pricing a theoretical framework exists that should allow us to compute a price for every financial derivative. The theory requires every source of risk to have an associated financial product that can be traded infinitely often and in infinitesimally small quantities. In the last decades markets have become increasingly "complete" in the mathematical sense due to a large increase in the volume of trades and a larger variety of product types. This also means that increasingly complex derivative products are traded on the market. However, new product types that enter the market are typically sensitive to new risk factors that are slightly different to those risks traded before. Even when all liquidly traded derivatives are considered, multiple models for the risk factors and underlying market parameters can be derived, further complicating the models. Typically this will also inhibit the computation of a unique price. Hence, the single theory that prices every derivative still remains a dream. This paper aims for a new method instead, that allows the convenient pricing of products with respect to all available models.

Looking at the current state of the market, two very clear trends can be observed. Derivatives are traded in increasingly large numbers and with increasingly complex product features. Small investors trade more and more structured products, especially in Germany. According to resources of the German Derivative Organisation (Deutscher Derivate Verband), the open interest of exchange-traded structured products increased from €24bn (2004) to €99bn (2008). At the same time the number of different derivative types that are available at any one time increased by 96% in 2006 and 87% in 2007. By the end of 2007 there were more than 257,329 different derivative securities available for sale on German stock exchanges. Similar numbers could be found for other markets.

The large number of different derivative types coincides with the increasing complexity of these products. Many investors no longer properly understand the risks involved with the instruments they trade. Law suits have been filed by numerous investors who bought OTC constant maturity spread ladder swaps and did not understand the risks. Some of these investors were awarded refunds by German courts who agreed with their arguments that the risks were not communicated properly (Bastian & Benders, 2008).

How do financial engineers currently cope with these trends on the derivative markets? To capture the risks and rewards of a derivative product, computer models are made at various stages of the development process. Initially, the structurer inventing the product has to design a mathematical model of the product to optimize the risk and reward for the bank as well as the customer. Next, the trader needs to use an implementation of the model in order to create fair market prices. Finally, in risk management the product model is re-invented in order to accurately assess the associated risks. The whole process suffers from the mundane task of implementing very similar models with only slight

variations, requiring tedious work that often results in redundant and error-prone code. If the processes of modelling and implementing the models were separated, large parts of this effort could be automated.

Related disciplines have faced similar problems and solved them by implementing a more efficient development process with the aid of computers. Traditional engineering has been revolutionized from a state of manual blueprint processing to a new era where digital data formats contain all necessary details of a modelled object. From this description everyone involved in the process can automatically extract the relevant information, e.g. for analyzing important model properties. In some applications, the complete production process from design to manufacturing is automated to such a degree that the final product can be printed by a 3D printer. This method has become known as CAD (Computer Aided Design).

The vision of creating a similar environment in financial engineering has been around for a long time. The main task in financial engineering is the creation of executable implementations of financial models, which can be facilitated by a domain specific language and accompanying tools. Domain specific languages offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application (Marjan Mernik et al., 2005). Various suggestions have been made for a standardized representation of financial models that would enable automated processing of associated computational tasks. One such approach is the industry standard FpML (ISO 15022), an XML dialect that allows the specification of a wide range of financial securities. It is designed as an exchange data format for the specification of security trades [Fpml 4.0]. However, it is not suitable for the description of new products or the automated derivation of algorithms, since it does not address the internal structure of a security. Various kinds of derivatives are distinguished by different names only.

Other approaches claim to be more generic, but little information is publicly available, since they are packaged as proprietary software products. One example is the specification language MLFi for derivatives, which originates from a functional programming style language [Jones et al., 2000]. Various competing vendors for pricing engines also have created their own proprietary definition languages, which are used primarily to configure their pricing engine. An open standard for product definition has not yet been defined.

Clearly there is demand on the market for more comprehensive modelling support in the area of financial products. The current state of the art is dominated by quick and dirty code hacking. Large amounts of redundant code is developed. Aspects of product design, stochastic market model and numerical tweaks are often mixed together. This makes it difficult to accurately document and communicate the model. A definition language for financial products and trading strategies could help in the communication between different departments from research, trading to risk management. After all, it should not be too difficult to come up with such a standard.

1 A New Approach

To address the problems outlined above, we propose a new notation for describing the structure of financial products. The notation is generic, simple and yet is backed by a solid mathematical and computational interpretation. It focuses on financial product features and abstracts these from numerical details. Stochastic models and numeric algorithms already have useable mathematical notations. Having separate notations for all three aspects allows the development of a compiler that generates pricing algorithms automatically. We consider the following features to be crucial to effective financial modelling.

- **Expressiveness**: All features of a financial product are representable in a precise and compact manner.
- **Modularity**: The product structure is separated from model stochastics and numerical details.
- **Transparency**: It is easily comprehensible and allows for concise communication and documentation of financial product details.
- **Integrability**: Easy integration of existing code allows to build upon previous work.
- **Simplicity**: Little learning effort is required for a computer literate.

Expressiveness is important since a description language needs to be able to encompass all types of investments in a modelled portfolio. It is also necessary that all important legal clauses, delivery options and day-count conventions forming the trade can be represented precisely and concisely.

Modularity allows a separation of concerns, such that product structurers can focus on product features and quants can focus on the pricing model. Furthermore, modular code greatly improves maintainability.

The next aspect — transparency — serves several purposes. First of all, the maintainability is again improved. Furthermore, communication between the parties involved in portfolio management can greatly improve. Describing a product in an intuitive and compact way makes it easier to communicate how the product works.

Reuse of stochastic models and numerical procedures is important to allow a smooth transition from the previous modelling process to the new one. Integrability of existing functionality can be guaranteed by a programming paradigm that is very close to standard procedural programming, facilitating variables and procedure calls.

Finally, the language should be simple and easy to learn. Basic programming skills should suffice to learn the language. Widely used primitives, such as loops, conditionals and function evaluations, can build the foundation of such a language.

The suggested solution: ThetaML

The remainder of this article introduces ThetaML, a language that exhibits the features postulated above. The following concepts and language constructs are used.

- ThetaML is an extension to a simple procedural programming language. Hence function calls and fundamental commands to control the program flow are available.
- Commands are written in a chronological order. Every action with any quantitative effect is written according to their real order. Additional commands let time pass (**theta**) or make commands run in parallel (**fork**).

- Conditional stochastic measures on backward variables make it possible to access expected values and risks from a trader's perspective. The backward access operator "!" allows to access variable values as if the program flow was reversed for that particular access. In contrast to other languages this allows referencing of variables that have no value assigned yet.

ThetaML can be used to define financial products, financial pricing problems and dynamic hedging problems in a concise manner. The language can be automatically translated into numerical algorithms. Hence, it is not only a description language, but also specifies a precise numerical problem to be solved. Being completely independent of the numerical scheme, Monte Carlo, PDE solvers or trees could be used to compute the result. Any financial engineer who designs or analyzes a certain financial product in ThetaML can focus on the problem domain without having to worry about the numerics.

2 Introductory Examples: Fixed Income

2.1 Hello world!

It is common to introduce new programming languages by a "Hello World" example. In the case of ThetaML this will be a product definition for an instant cash earning. Cash payments are one of the fundamental actions of any financial model. The framework suggested here represents such a transaction with the most common concept of a procedural programming language: An assignment. Here is the "Hello World" in ThetaML, which returns the cash value of €100:

```
model CashNow
import EUR "Numeraire"
export V    "Value"


    V = 100 EUR
end
```

Consider EUR to be a variable holding the cash value of a €1 payment at that time. The value of the "CashNow" financial product is €100. The script just assigns that value to a variable.

2.2 Zero bond

Heading to the next level we now want to model a zero bond. A bond is basically a set of deferred payments happening at some later time in the future. In this example we import a variable T to represent the time to maturity. The ThetaML command **theta** is followed by the time that passes. All operations in the program are written in chronological order. Hence, the time passing operation is followed by a cash payment. We represent the payment as an assignment to variable V. It might be worth noting that coupon bonds could easily be modelled by increasing the value V using a statement like V=V+coupon.

As a consequence of using the **theta** command, the evaluation of EUR occurs at a later time T. We remember that EUR is the present value of a

€1 payment and thus time dependent. More formally, we define the EUR variable as a sequence $(\text{€}_t)_{t\in\mathbb{R}^+}$. At each time step t the value of this numeraire is a stochastic value $\text{€}_t(\omega)\in\mathbb{R}$. Following the terminology of Monte Carlo pricing we will also refer to $\omega \in \Omega$ as a scenario and to €_t as paths of the discount factor EUR.

The script below assigns the present value of a future payment to variable V. This value is usually stochastic and is not yet the bond price. The mathematical result can be written as $V(\omega) = 100\text{€}_T(\omega)$, which is the discounted present value of a payment of 100 EUR at time T.

```
model Bond
import EUR "Numeraire"
import T   "Maturity"
export V   "Value"


    theta T
    V = 100 EUR
end
```

2.3 Pricing a zero bond

Next we naturally want to evaluate the price of such a bond. We evaluate that price in a risk-neutral setting as the expected value conditioned on our knowledge at time zero. Consider a probability space $(\Omega, \mathcal{F}, \mathcal{P})$. In the next model we now assign values to the two variables V and P, operating in this space. While the value of V is measureable at time T, its expected value is deterministic at time zero. At time zero, the exact discount factor EUR might not be known, since it depends on a stochastic model. Thus, the exact value of V is not known at time zero, but we can perform the computation of the expected value of V. The expected value E(V) is measureable at time zero and our model assigns this value to P which is then exported as the price of the bond. A precise definition of such relationships can be found in (Dirnstorfer, 2006).

Since P depends on a future value of V, a backward reference is required. ThetaML can access future definitions by adding an exclamation mark, in the example E(V!). The "!" operator references the definition of V, which is assigned later in the code. This means the actual value of V! depends on the later value of the discount factor EUR. Adding the line P = E(V!) extends the previous model "Bond" to the computation of the bond price at the initial time.

```
model BondPrice
import EUR "Numeraire"
import T   "Maturity"
export P   "Price"


    P = E(V!)


    theta T
    V = 100 EUR
end
```

2.4 Pricing a coupon bond

An important concept in ThetaML is the notation in chronological order. Sometimes it is necessary to express simultaneous actions. This is enabled by the **fork** statement, which creates parallelized code blocks. Each block advances in model time independent of the other blocks. Unlike in multi-threaded computer programming there are no race conditions. In case of two statements happening at the same model time, the original code order implies the interpretation order.

In the next example the two scripts show two representations of a bond that pays €5 coupons after 6 months and after 1 year. Finally, a face value of €100 is payed in addition. Again, the bond price at the initial time is computed by the line P = E(V!). This example shows how different actions at different times can be represented: The left script presents a parallel and the right script a serial execution.

```
model coupon_bond1              model coupon_bond2
import EUR                      import EUR
export P                        export P
    P = E(V!)                       P = E(V!)
    fork
        theta 1                     theta 6/12
        V = V! + 5 EUR              V = V! + 5 EUR
    end
    fork                            theta 0.5
        theta 6/12                  V = V! + 5 EUR
        V = V! + 5 EUR
    end                             V = 100 EUR
    theta 1                     end
    V = 100 EUR
end
```

This section concludes the introduction of ThetaML features. Two commands and one access operator enhance a normal programming language. The commands are **theta**, which defines the model time, and **E**, the expected value of future variables. The operator "!" is used to define the access of a variable's future value.

3 Derivatives

Having laid out the basics of the ThetaML language we now move on to more interesting financial products. The additional programming concepts introduced make the definition of arbitrarily complex products simple.

3.1 European option

In this setting we import a new stochastic variable for the stock price S, which again is a stochastic process $(S_t)_{t\in\mathbb{R}^+}$. The option price is again determined at time zero, with respect to a risk neutral probability space $(\Omega, \mathcal{F}, \mathcal{P})$. We model the payoff $\max(100 - S, 0)$ to be paid in € and multiply the discount factor EUR accordingly. The following example is a put option with a strike of 100. Different payoff functions "could effectively be modelled using the same language features".

```
model EuropeanOption
import EUR  "Numeraire"
import S    "Stock price"
import T    "Maturity"
export P    "Price"

    P = E(V!)
    theta T
    V = max(100 − S, 0) ∗ EUR
end
```

3.2 Bermudan option

Modelling early exercises is certainly one of the biggest challenges for descriptive languages. The ThetaML approach allows this with the few concepts introduced above.

In the next example we consider a Bermudan style option that is exerciseable every day, considering 250 working days per year. Therefore we use a **loop** statement that cycles over a sequence of daily actions as many times as there are days to maturity time. The first operation is again a **theta** that advances the model time by one day. Then a conditional execution decides whether the option should be exercised. This decision is based on the expected value E(V!). The E function inside the loop is now evaluated at a time greater than zero and conditioned on the knowlegde at that time $\mathbb{E}(V|\mathcal{F}_t)$, whereas time t is the total time passed, or the sum of all arguments to **theta** previous to this E function.

```
model BermudanOption
import EUR  "Numeraire"
import S    "Stock price"
import T    "Maturity"
export P    "Price"

    P = E(V!)
    loop T∗250
        theta 1/250
        if E(V!) > max(100 − S, 0) ∗ EUR
            V = max(100 − S, 0) ∗ EUR
        end
    end
    V = max(100 − S, 0) ∗ EUR
end
```

For an option exerciseable in continuous time (e.g. an American option) we need to introduce another mathematical concept into ThetaML. The command **dt** represents the length of a sufficiently small time step. The actual value of **dt** could be evaluated by a numeric limit to zero or analytic concepts. The limit must of course exist, otherwise the ThetaML is invalid. The next code fragment shows a continuous action that takes T time.

```
loop T/dt
    theta dt
    ...
end
```

3.3 Compound option

Finally, we show a possible combination of previously defined options. Many structuring approaches focus on a combination of individual products in the way that assets are combined in a portfolio. Such a combination is also possible in ThetaML by adding or substracting payments. Additionally this approach enables a new type of structuring. One product can be reused as an underlying of another product, naturally creating options on options and so on.

The next example creates a compound option. The price of a Bermudan option is imported as Pb from what was defined as price P in the original model. This is conducted by the **call** command, which allows reusing the American option model defined previously. The price is already discounted so we only need to discount the €10, build the payoff function and create an expected value. This is an European option on a Bermudan option. Note that the commands are written strictly in their real chronological order. First, we determine the option's price P, then the maturity time T1 is awaited, then a decision on buying an inner option is made and finally, the inner option realizes.

```
model CompoundOption
import EUR "Numeraire"
import S   "Stock price"
import T1  "Maturity of outer option"
import T2  "Maturity of inner option"
export P   "Price"

    P = E(V!)
    theta T1

    V = max(Pb! — 10 EUR, 0)

    call BermudanOption
        import Pb from P
        export T2 to T
        export EUR, S
end
```

You can look at ThetaML as a computer programming language that has convenient features to define numerical problems associated with derivative pricing and financial analysis. However, ThetaML is also a suitable domain specific language for the definition of financial products in all their quantitative implications for the holder and the issuer.

3.4 Contracts and pricing

In order to serve as a trade specification, a ThetaML script needs a few conventions to be precise. For instance, in the case where a ThetaML script defines a financial product that is traded between two parties, the use of the **E** operator is restricted: Only two types of uses are permitted in financial trade definitions. One refers to a long embedded option, exerciseable by the holder. The other refers to a short embedded option, exerciseable by the issuer. Respectively they are written as

$$V = \max (E(V!), ...)$$

and

$$V = \min (E(V!), ...).$$

There are other equivalent expressions constructed as **if-else-end** statements, or inserting other names and formulas for the price variable V.

Note that, for exchanging precise product definitions, the parties involved must agree on the meaning of the specific variables: Input parameters like "Stock price" and "Maturity time" are sufficient for product pricing and risk management purposes. But, a trade specification needs to capture more details such as the parties involved and whether the settlement type is physical or cash delivery. Given an agreement on the precise specifications of the input and output data can be reached, ThetaML has the potential to serve as a better trade definition standard (e.g. extending FpML).

4 Numerical Evaluation

In this section we focus on the numerical evaluation of ThetaML code. It is possible to define a mathematically precise interpretation of ThetaML, since ThetaML is equivalent to the operator notation defined by Dirnstorfer [Dirnstorfer, 2004]. For the purpose of this document we will explore the steps necessary for actually deriving an algorithm to evaluate ThetaML. We will not consider efficiency, but purely focus on operation in principle. In the end we want to show that ThetaML really specifies a unique financial problem and provide an intuition of what a numerical solver for these general problems looks like.

One possibility for solving pricing problems defined in ThetaML could be solving the governing partial differential equation. Depending on the type of stochastic processes, PDE solvers can be a good choice since they can be very precise and quick, especially for low-dimensional models. Examples for the process of the underlying where corresponding PDEs are known include geometric Brownian motion (GBM) and GBM with Heston volatility. If the underlying follows a GBM with jumps, then an integro partial differential equation (IPDE) is the equivalent problem, which can be solved efficiently. Note that the financial product itself might still be too complex for a PDE solver, i.e. the dimension of the state space could be too large. An example for this would be a moving window Asian American option [Dirnstorfer et al., 2006].

Monte Carlo methods are much more general and are also suited for high-dimensional tasks. The gain in flexibility is paid for by higher computational workloads compared with specialized algorithms like PDE solvers. But for many cases specialized algorithms do not exist and Monte Carlo remains the only valid method.

In a model without backward references (e.g. V!), the internal variables of the ThetaML model can be computed by simple sequential execution. In

models with backward references, the assignments defined in the ThetaML model have to be rearranged to a computable order. Then it is again straightforward to execute the rearranged code, except for the expectation function E().

In fact, the expectation E(Y) is the conditional expected value based on the filtration $\mathcal{F}_t$, i.e. the scenario values X of all previous time-steps

$$E(Y|X) := \mathbb{E}\,(Y|\mathcal{F}_t).$$

That means X contains all information required for computing $\mathbb{E}\,(Y|\mathcal{F}_t)$. For example pricing a vanilla American put option, $\mathcal{F}_t$ may contain solely the current asset price (S_t).

For constructing a numerical estimator of $E(Y|\mathcal{F}_t)$, a smoothing method is required which serves as an estimator of this conditional expectation. This kind of problem is well known in statistics, such that we refer the reader to Härdle for more details [Härdle, 1992] of the solution. In option pricing, (Carrière, 1996) introduced the nonparametric regression to the pricing of early-exerciseable options. Later, (Longstaff and Schwartz, 2001) simplified his algorithm and obtained some lower-bounds on the option price by considering out-of-sample values.

Consider n realizations of Monte Carlo samples Y_i, X_i for $i = 1, \ldots, n$ at time t. Then the conditional expectation can be approximated by $E(Y|X) \approx f(x)$, where

$$f(x) = \sum c_i b_i(x)$$

and the coefficients c_i of basis function b_i can be obtained by solving the least-squares minimization

$$\min_c \sum (f(X_i) - Y_i)^2 .$$

Useful basis functions b_i are e.g. polynomials or splines.

Conclusion

With ThetaML we have introduced a language that is powerful enough to represent financial products in the form of a simple computer program. The power of this language lies in the fact that the description of the

product, the stochastic model and the numerical implementation are separated. Each individual component is much easier to understand than a complex numerical algorithm containing all three aspects in one piece of low level computer code.

The above examples already give an idea of how financial product features are represented in the proposed language. Essentially, ThetaML is a programming language that has the unique features of backward variable access and implicit functions for conditional stochastic values. These features enable new applications in financial engineering, such as the definition of trading strategies on structured products or the definition of investments and real options.

In a follow-up paper we will discuss the use of ThetaML to construct risk-optimal dynamic trading strategies. We will show how to use any of the previously defined products to hedge any other financial product.

REFERENCES

N. Bastian and R. Benders. Sturz von der Zinsleiter. *Handelsblatt*, 01.04.2008.

J. F. Carrière. Valuation of the early-exercise price for options using simulations and non-parametric regression. *Insurance: Mathematics and Economics*, 19:19–30, 1996.

S. Dirnstorfer, A. J. Grau, and R. Zagst. Moving window Asian options: Sparse grids and Least-Squares Monte Carlo. Working paper, grau@thetaris.com, December 2006.

S. Dirnstorfer. On the representation of trading strategies and financial portfolios. In *Intelligent Finance - Convergence of Financial Mathematics with Technical and Fundamental Analysis, Proc. 1st Int. Workshop on Intelligent Finance (IWIF-I)*, ISBN 187685118X, pages 131–143, Melbourne, Australia, 2004.

S. Dirnstorfer. *Mulitscale calculus with applications in quantitative finance*. PhD thesis, TU-München, Germany, Fakultät für Informatik, 2006.

Arie Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35:26 – 36, 2000.

Fpml 4.0 specification. Technical report, www.fpml.com, 2004.

W. Härdle. *Applied Nonparametric Regression*. Cambridge University Press, 1992.

S. P. Jones, J.-M. Eber, and J. Seward. Composing contracts: an adventure in financial engineering. In *Proc. 5th Int. Conf. on Functional Programming*, pages 280–292, 2000.

F. A. Longstaff and E. S. Schwartz. Valuing American options by simulation – a simple least-squares approach. *The Review of Financial Studies*, 14(1):113–147, 2001.

Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

W