# Financial Model-Base Construction for Flexible Model Manipulation of Models and Solvers

Keun-Woo Lee Soon-Young Huh
*Graduate School of Management, Korea Advanced Institute of Science and Technology*
*{kwlee, syhuh}@kgsm.kaist.ac.kr*

## Abstract

*As financial markets are volatile and rapidly changing, preciseness and agility in price evaluation and risk assessment in the portfolios are more important and decision support systems containing diverse financial products and pricing algorithms have been adopted to support those quantitative analysis tasks. To effectively use the financial DSS, the trader should be able to know which algorithms are applicable to a specific product and to match flexibly the product with appropriate algorithms depending on his product-evaluation purposes. This paper proposes model-base construction mechanisms in a financial DSS that facilitates such mix-and-match operations between financial products (considered as models) and pricing algorithms (considered as solvers). As a conceptual framework for the model-base, we use the generic model concept for developing system constructs and procedures. The constructs and procedures of the model-base are represented as self-contained and well-modularized objects that can be applied to a wide variety of problem domains.*

## 1. Introduction

As financial markets are volatile and rapidly change [16,22], it is increasingly important for a trader, who manages a portfolio of financial products, to be able to cope with risks of losses arising from the fluctuations in market prices of the constituting products [6]. Specifically, the trader evaluates each financial product of his portfolio by calculating analysis values such as net present value (NPV) [14] and price movement indicators, and estimates the extent of potential loss in each product using the analysis values. Then, he controls the maximum permissible loss of the portfolio by adjusting the investment amount in each product. For the actual algebraic instructions to calculate the analysis values, financial researchers and practitioners have developed various pricing algorithms and are continually creating new algorithms to improve accuracy of such calculations [14,15,17]. Thus, when a trader calculates the analysis values of a financial product, he adopts a pricing algorithm that is most appropriate to his product-evaluation purposes among the algorithms that can be applied to the product.

However, it is a very difficult task for a trader to select an appropriate pricing algorithm and apply it to a financial product manually since the algorithm is usually based on a complicated mathematical formula. Moreover, since a pricing algorithm can be applied to one or more financial products and a product can be evaluated with one or more algorithms, determining whether an algorithm can be applied to a certain product or not is also not a trivial task. This complexity in applying pricing algorithms call for the creation of financial decision support systems (DSS) to help a trader evaluate a financial product using various pricing algorithms in an easy and timely manner.

In the perspective of traditional DSS areas, the financial product and pricing algorithm can be considered as a model and a solver, respectively. A model is defined as a set of declarative statements describing a real-world problem that the DSS wants to solve, and a solver is defined as a set of algebraic instructions to solve the modeled real-world problem [4,11,13,20]. In the financial DSS, a real-world problem is evaluation of a financial product, and thus we can conceptualize a financial product as a model for its evaluation problem by specifying its properties needed for the evaluation and the analysis values resulted from the evaluation. Also, we can conceptualize a pricing algorithm as a solver since it refers to the properties of a financial product and calculates the analysis values of the product. In this capacity, a financial product and a pricing algorithm are called a **product model** and a **financial solver**, respectively.

To support trader's evaluation of product models effectively, the financial DSS should enable a trader to mix and match product models and financial solvers according to his product-evaluation purposes. For the mix-and-match operation, when a trader evaluates a product model, the financial DSS should be able to intelligently provide financial solvers that can be applied to the model so that the trader can adopt a financial solver easily and at the same time he is prevented from misusing financial solvers

Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)
0-7695-1874-5/03 $17.00 © 2002 IEEE

that are syntactically and semantically incompatible with the product model. After the trader selects a financial solver, complicated tasks such as the application of the selected solver to the product model and calculation of the analysis values should be performed autonomously without the trader's intervention as well. This is very beneficial to non-professional traders, and sometimes to knowledgeable professional traders, because the semantics of product models and financial solvers are complicated and the choice and application of compatible financial solvers for a given product model is not a trivial task.

Despite the needs and benefits of the mix-and-match operation, however, most DSS researches do not provide satisfactory discussion for such operations. This is because they have primarily focused on the management science or operations research (MS/OR) models [12,21] that have different characteristics from the product models. Unlike the product models, MS/OR models have a standardized formal structure so that a solver can access and interpret them in a uniform manner. For example, every linear programming (LP) model can be considered as an objective function restricted by several constraints. Thus, a solver such as the simplex method can treat any LP model in a same way by just syntactically interpreting its structure whether it is for a transportation problem or a feed mix problem. On the other hand, product models do not have such a standardized formal structure, and thus a financial solver may have to access syntactically different problem statements for different financial products. For example, Black-Scholes method [14], a representative financial solver, should be able to access the stock price in case of evaluating a stock option but the exchange rate in case of a currency option. That is, a financial solver should be able to access appropriate statements for each product model even though they are syntactically different.

Recognizing these requirements and difficulties, in this paper, we propose model-base construction mechanisms in a financial DSS that facilitates the mix-and-match operation between product models and financial solvers. Specifically, we adopt the generic model concept [13] as a conceptual framework for representing product models and financial solvers systematically. The generic model concept presents an effective modeling scheme to represent management science or operations research (MS/OR) models in a unified manner. Especially, by providing flexible building blocks to conceptualize models and solvers, the modeling scheme accommodates wide applicability to various problem domains including optimization, forecasting and queuing models.

This paper is organized as follows. Section 2 briefly reviews the literature on model-and-solver integration for supporting the mix-and-match operation. Section 3 describes the systematic representations for models and solvers using the generic model concept. Based on the representations, section 4 presents constructs and procedures for a financial model-base that facilitates the mix-and-match operation between product models and financial solvers. The final section discusses the results and contributions of the paper and provides future research directions.

## 2. Literature on model and solver integration

Among the previous model management researches in DSSs, we can reference the following three approaches for the model and solver integration according to the interfacing scheme that defines how a solver is applied to a model and accesses data values from the model.

The first approach has focused on an effective model representation so that a solver can interpret models easily when it solves the modeled real-world problems [8,9,11]. A solver is defined as an executable application, and it accepts a model as input data to solve the modeled real-world problem. Many researches in this approach developed modeling languages such as SML [11] and AMPL [8] to represent a model as a form that can be readable by the solver applications and at the same time to support a human modeler's problem modeling. However, since they primarily focused on MS/OR models which a solver can understand easily due to their standardized formal structure, considering how a solver application find out statements of a model, which are necessary for its problem solving task, was not satisfactorily discussed.

The second approach has also focused on an effective model representation and it incorporated the interfacing scheme between models and solvers into the model representation [13,18,19,25]. Most researches in this approach adopt object-oriented methodology, and define a model as an object which has member operations implementing interfacing schemes with its solvers. Based on the object-oriented mechanisms such as polymorphism and inheritance, these researches have many benefits for reusing or extending the model objects. However, this approach has a disadvantage in system maintenance: whenever a new solver is created or an existing solver is modified, every model that can be solved by the new or modified solver should be also revised to reflect the solver's changes into its member operations.

The last approach has focused on the underlying system architecture for the model and solver integration [5,10,23,24]. Researches in this approach provided detailed system constructs and procedures for system developers to implement DSS applications effectively. However, the constructs and procedures from those researches did not have enough generality to be applied to DSSs in various problem domains since they were designed depending on structures of the models and solvers that the researches dealt with.

In this sense, the three approaches are still

unsatisfactory in supporting the integration of models and solvers. Those limitations, as mentioned earlier, are derived from that the previous researches have primarily focused on MS/OR models that are relatively stable and standardized in comparison with the product models. In the following sections, we develop mechanisms for a financial model-base that supports continually changing but not standardized product models.

## 3. Model and solver representations using generic model concept

This section defines the model and solver representations adopting the generic model concept as a conceptual framework. Based on these representations, we develop detailed structures and mechanisms for product models and financial solvers, which will be discussed in the following sections.

First, we introduce a solver as an independent construct, which was rather hard-wired into the models in the original generic model concept. Hard-wiring a solver into models is not desirable since it mixes the solver's implementation with the models' and makes the models harder to be understood and maintained. Also, it is difficult to add new solvers or vary existing ones when the solver is an integral part of a model. We can avoid these problems by encapsulating solvers as an independent construct.

A model is responsible for maintaining and updating all the problem statements, each of which is either an input or output of the problem. For example, a product model for an option maintains the properties of the option (e.g., exercise price and maturity date) and the analysis values (e.g., NPV and price movement indicators). An end-user of the financial DSS interacts with the product model to evaluate the option. He sets the values of the properties that are the inputs of the evaluation, and obtains the results from the analysis values, which are the outputs. On the other hand, a solver is responsible for calculating the outputs of the problem from the inputs. For the calculation, it implements a solving algorithm such as Black-Scholes method, binomial method, and finite difference method, and maintains its own inputs and outputs for the algorithm. In the option's example, whenever the user requests evaluation of an option, the product model for the option forwards the responsibility of the analysis value calculation to its solver, which is formerly specified by the solver. After finishing the calculation, the solver returns its results to the model, and then the user can obtain the results.

Figure 1 shows the conceptual representations of a model, *Model_A*, and a solver, *Solver_A*, and illustrates their interactions. Conceptually, every model is defined with a set of ports regardless of problem domains and modeling paradigms. A **port** encapsulates a single problem statement of its model, and thus it becomes an external interface of a model. A user can interact with a model through the ports. With respect to the interaction, the ports of a model are classified into two types: input ports (called **inports**) filled by a user and output ports (called **outports**) provided to the user as problem-solving results. For example, an option is represented by specifying a set of ports that stands for its properties and analysis values. The ports for the properties become inports and the ports for the analysis values become outports. In Figure 1, *Model_A* has three inports (*Inport_1*, *Inport_2*, and *Inport_3*) and one outport (*Outport_1*).

On the other hand, a solver consists of a set of ports and a calculation module. The ports of a solver are also classified into inports and outports that contain input and output values of the problem-solving task, respectively. *Solver_A* in Figure 1 has three inports (*Inport_a*, *Inport_b*, and *Inport_c*) and one outport (*Outport_a*). Each of the arrows denotes the movement of port values between *Model_A* and *Solver_A*. Initially, a user fills port values into the *Model_A*'s inports. When *Solver_A* is applied to *Model_A*, the port values in *Model_A*'s inports are delivered to *Solver_A*'s inports. Then, the calculation module of *Solver_A* performs the problem-solving task, and through *Outport_a*, the result value is delivered to *Model_A*'s *Outport_1*. Such a one-to-one relationship between a model's port and a solver's port throughout the movement of the port value is called a **port-mapping**.

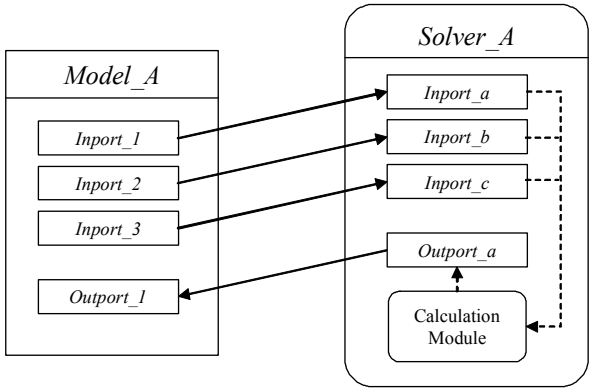



**Figure 1. Model and solver representations using generic model concept.**

## 4. Constructs and procedures for financial model-base

This section defines system constructs and procedures for facilitating the mix-and-match operation between product models and financial solvers in a financial DSS.

### 4.1. Basic constructs for the model and solver representations

As basic constructs for representing product models and financial solvers, we define the following three classes: *ProductModel*, *FinancialSolver*, and *Port*. First, *Port* class encapsulates the port that is the most primitive base element of product models and financial solvers. To represent their diverse semantics, *Port* class should be able to hold various data formats such as a single value (e.g., the exercise price of a stock option), multiple values (e.g., a time-series of the market price), and a function of other ports (e.g., the coupon amount of a bond is the product of the coupon rate and the face amount). The following attributes are included in *Port* class to store such various forms of port values:

- The *name* is a unique identifier for the port.
- The *description* is a general textual documentation for the meaning of the port.
- The *dimension* is the unit used to measure the port value. It includes not only physical units but also logical conventions in financial domains. For example, the dimension of the interest rate can be '% with semi-annual (SA) compounding'. The '%' is the physical unit and the 'SA' means that the interests are compounded two times per annum, i.e., every six months.
- The *format* is the data type of the port value. It specifies whether the port has a numeric value or a text string and whether it has a single value, a list of values, or a matrix.
- The *value* is the actual value of the port. It can be a mathematical formula if the port represents a function.
- The *iostate* is an interfacing role of the port, whether it is an inport or an outport.

By aggregating *Port* class, *ProductModel* and *FinancialSolver* classes encapsulate product models and financial solvers. Figure 2 shows an object data model for the three classes with an example for a stock option and Black-Scholes method using the class diagram of Unified Modeling Language (UML) [3]. In Figure 2, *ProductModel* and *FinancialSolver* classes have two one-to-many aggregations named *inports* and *outports* with *Port* class to indicate their inports and outports, respectively. Also, *FinancialSolver* class has a member operation named *calculate()* to implement its calculation module.

Although *ProductModel* class has generality in representing diverse product models by aggregating *Port* class, it is required to define more specialized classes for some product models that need product-specific operations. For example, a product model for option products must have operations for exercise management, and a product model for bonds must have operations for redemption processing [6,14]. Such specialized classes are interrelated to one another through the inheritance mechanism such that a more specific class is expanded by incrementally modifying a more general class. At the top of the inheritance hierarchy, *ProductModel* as a meta-model class provides general structural basis characterized by *Port* class for other classes. Figure 3(a) shows an example of the inheritance hierarchy for three kinds of financial products: futures, options, and bonds. On the other hand, *FinancialSolver* class is also specialized for algorithm-

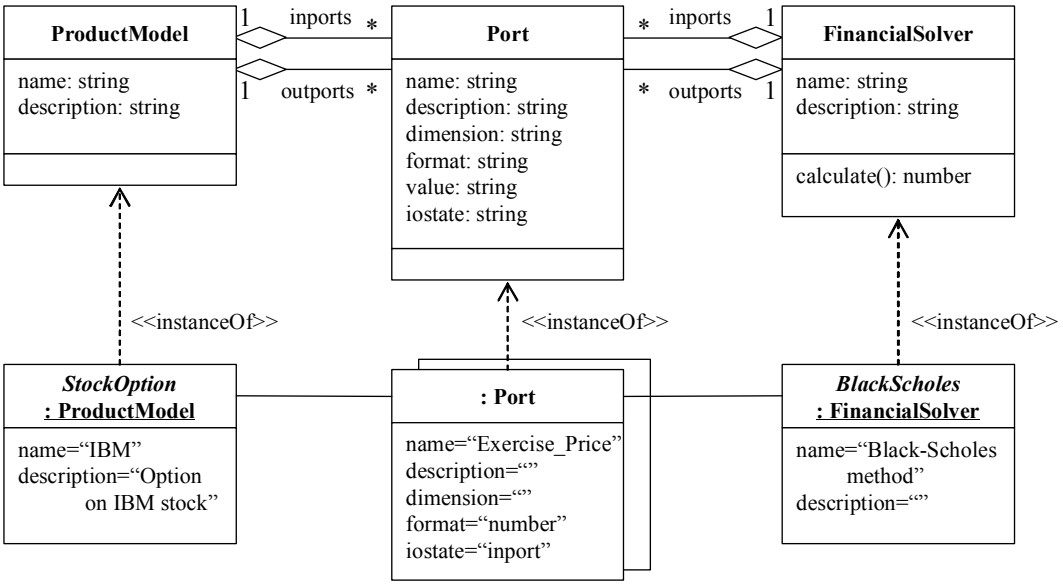



**Figure 2. An object data model for *ProductModel*, *FinancialSolver*, and *Port* classes.**

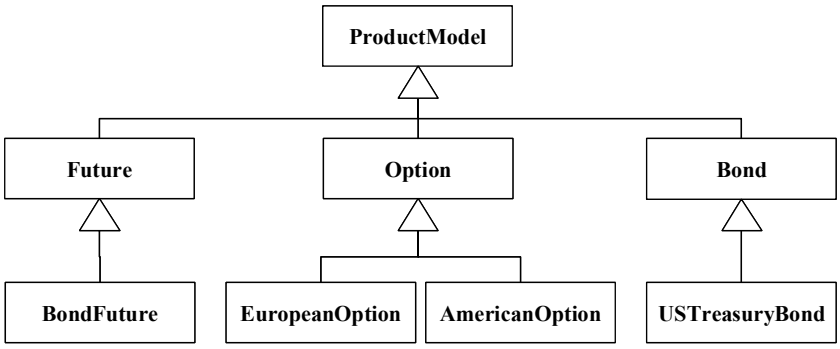
specific operations needed for each pricing algorithm as *ProductModel* class is specialized for product-specific operations. Figure 3(b) shows an example of the inheritance hierarchy for some financial solvers [14,15,17].

These class inheritances lead to effective division of tasks and responsibilities between more general classes and their specialized classes. More general classes take care of generic tasks necessary for a broad range of models and solvers, and their specialized classes focus only on their product-specific or algorithm-specific tasks and inherently have the benefits of the generic tasks as well. Thus, we can create a new class for a product model (financial solver) and make it conform to the model (solver) representation characterized by the ports only by inheriting it from *ProductModel* (*FinancialSolver*) class or its specialized classes.

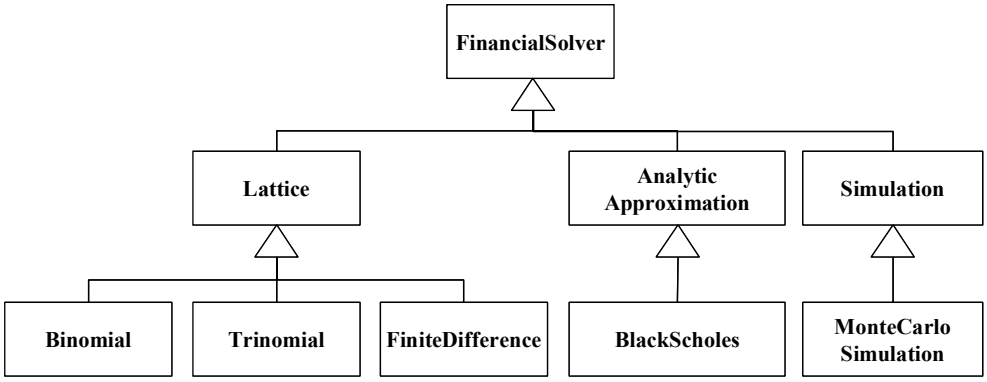
### 4.2. Port-mapping table

To evaluate a product model using a financial solver, the financial DSS should establish port-mappings between them for the solver to be able to get the inport values from appropriate ports of the product model and return the outport values. Such port-mappings are dynamically registered and removed as a new product model or financial solver is added or an existing one is changed. The **port-mapping table** is an information registry for managing such dynamically changing port-mappings effectively. It maintains three pieces of information: a model-solver pair to be matched, a port pair of the model and solver, and their port types. The first two show how to establish a port-mapping for the designated pair of a product model and a financial solver, and the last discriminates whether the port-mapping is for inports or outports. Figure 4 shows the conceptual structure of a port-mapping table with the model and solver shown in Figure 1.

With the port-mapping table, we can get the following three advantages. First, the financial DSS application become independent of continually changing port-mappings as they are separated from the application program and maintained in the port-mapping table. Thus, even when a new product model or a financial solver is added or an existing one is changed, the application needs not to be re-implemented or re-compiled to reflect the changes of the port-mappings, which would be required if




(a) An example of the class inheritance hierarchy for product models




(a) An example of the class inheritance hierarchy for financial solvers

**Figure 3. Class inheritance hierarchies for product models and financial solvers.**

the port-mappings were hard-coded in the application. Second, the financial DSS can find out financial solvers that can be applied to a specific product model by identifying solvers that have port-mappings with the product model in the port-mapping table. These financial solvers will be provided for traders who want to evaluate the product model. Last, managing the complex relationships between product models and financial solvers becomes easy and familiar to the system administrator since he can manipulate port-mappings as if he treats data in database tables.

**Port Mapping Table**

| Model | Model Port | Solver | Solver Port | Port Type |
|---|---|---|---|---|
| *Model_A* | *Inport_1* | *Solver_A* | *Inport_a* | Inport |
| *Model_A* | *Inport_2* | *Solver_A* | *Inport_b* | Inport |
| *Model_A* | *Inport_3* | *Solver_A* | *Inport_c* | Inport |
| *Model_A* | *Outport_1* | *Solver_A* | *Outport_a* | Outport |

**Figure 4. An example context of a port-mapping table.**

### 4.3. Application of financial solver to product model

Figure 5 shows an example procedure for application of a financial solver to a product model. The financial DSS has two product models, *M1* and *M2*, and two financial solvers, *S1* and *S2*, and the trader wants to evaluate *M1* using *S1*. As shown in the port-mapping table, *M1* and *S1* have three port-mappings: two for their inports (*MP1-SP1* mapping and *MP2-SP2* mapping) and one for their outports (*MP3-SP3* mapping).

First, when the trader starts the evaluation of *M1* using *S1* ①, *M1* looks up the port-mappings between *M1* and *S1* on their inports in the port-mapping table ②. On receiving the port-mappings from the port-mapping table ③, *M1* sends its inport values to *S1* and sets each of them in the corresponding inport of *S1* according to the received port-mappings ④. After setting all the inports, *M1* requests *S1* to calculate analysis values ⑤. Then, using the calculation module *CM1*, *S1* performs the analysis value calculation, and stores the result in its outport. To return the calculation result to *M1*, *S1* looks up the port-mappings between *M1* and *S1* on their outports in the port-mapping table ⑥. On receiving the port-mapping from the port-mapping table ⑦, *S1* sends the result to *M1* and sets it in the corresponding outport of *M1* according to the received port-mapping ⑧. Finally, *M1* routes the result (i.e., the analysis value of *M1*) to the trader ⑨.

As shown in this example, the solver-application procedure is performed autonomously based on the interfacing information registered in the port-mapping table without any involvement of the trader. A trader can evaluate a financial product just by designating the pricing algorithm to be used, even if he does not have any

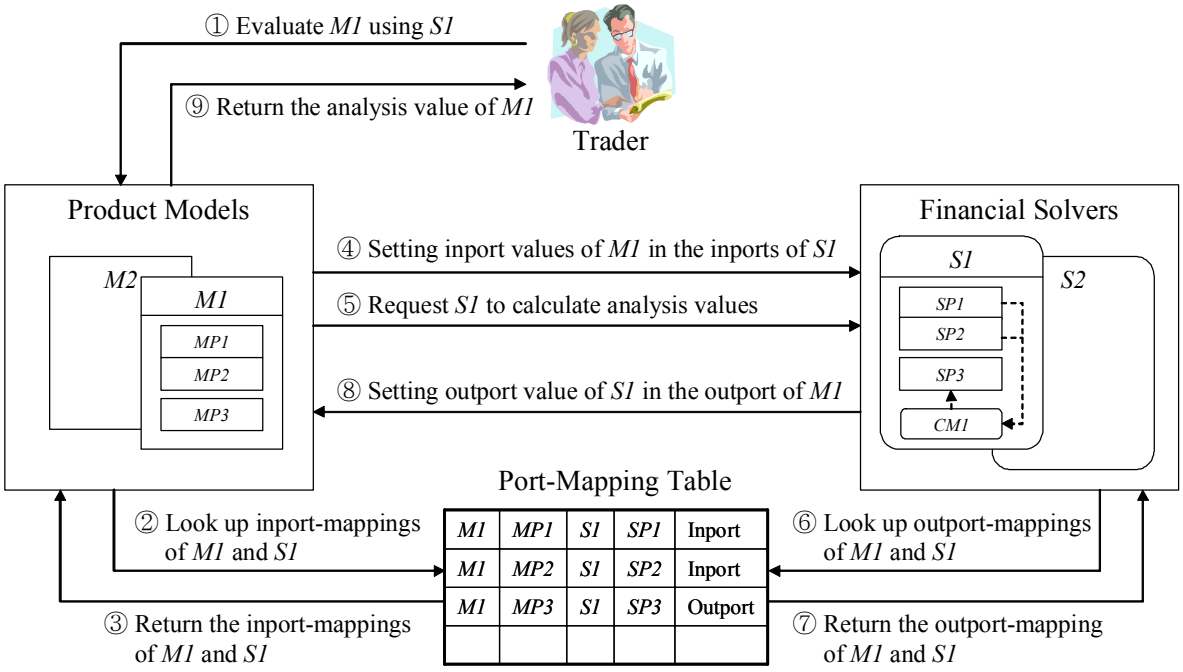



**Figure 5. Application of a financial solver to a product model.**

financial knowledge for the product evaluation.

In addition, the procedure is so general that it can be performed on any kinds of product models and financial solvers. This generality can be achieved since the financial DSS manages detailed interfacing schemes for each product model and financial solver pair in the independent port-mapping table. Accordingly, when a new product model or a financial solver is added, the financial DSS can adapt itself to such changes just by revising the contents of the port-mapping table.

## 5. Conclusions

In this paper, we propose mechanisms for financial model-base construction while facilitating the mix-and-match operations between product models and financial solvers. In developing the mechanisms, we adopt the generic model concept as a single formalism to represent product models and financial solvers since its effective modeling scheme makes it possible to represent the various models and solvers in a unified manner. Based on those representations, detailed interfacing information between a product model and a financial solver is enumerated as a set of port-mappings in the port-mapping table.

By referencing the port-mapping table, a financial DSS can find out which financial solvers can be applied to a certain product model and enable a trader to select one among the possible group of solvers, which is suitable for his product-evaluation purposes. Between the product model and the selected financial solver, data exchange needed for the analysis value calculation is also performed autonomously according to their port-mappings. Thus, the mix-and-match operation including the intelligent solver suggestion and the autonomous solver application can be supported in the financial DSS. Moreover, by separating the interfacing information from the application programs, the financial DSS can be adaptable to the continually changing product models and financial solvers. Those mechanisms proposed in this paper are designed generically so that they can be applied to models and solvers in a wide range of problem domains. In this context, the main contribution of this paper will be the provision of the model-base construction mechanisms, which are adequate for the continually changing financial environment and support the development of an effective DSS.

Future research is to focus on two directions: refinement of the core constructs, and resolution of a port type mismatch. Refinement of the core constructs will include the elaboration of their structures and concrete representation. Resolution of a port type mismatch is geared to match the ports of a model and a solver, which have different data types or domains.

## 6. References


[1] Ba, S., R. Kalakota, and A.B. Whinston, "Using Client-Broker-Server Architecture for Intranet Decision Support," *Decision Support Systems*, Vol.19, No.3, 1997, pp.171-192.

[2] Bhargava, H.K., R. Krishnan, S. Roehrig, M. Casey, D. Kaplan, and R. Müller, "Model Management in Electronic Markets for Decision Technologies: A Software Agent Approach," *Proceedings of the Thirtieth Hawaii International Conference on System Sciences*, Vol.5, 7-10 Jan., 1997, pp.405-415.

[3] Booch, G., J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, Massachusetts, 1999.

[4] Eck, R.D., A. Philippakis, and R. Ramirez, "Solver Representation for Model Management Systems," *Proceedings of the Twenty-Third Annual Hawaii International Conference on Systems Sciences*, Vol.3, 2-5 Jan., 1990, pp.474-483.

[5] Eggenschwiler, T., E. Gamma, "ET++SwapsManager: Using Object Technology in the Financial Engineering Domain," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, B.C. Canada, 18-22 October 1992, pp.166-177.

[6] Elton, E.J. and M.J. Gruber, *Modern Portfolio Theory and Investment Analysis*, John Wiley & Sons, Inc., New York, 1995.

[7] Engebretson, K.J., "The Use of Derivatives," *Issues of Interest*, Morgan Stanley Dean Witter & Co., Fall 1994, http://www.morganstanley.com/institutional/investmentmanagement/.

[8] Fourer, F., D.M. Gay, B.W. Kernighan, "A Modeling Language for Mathematical Programming," *Management Science*, Vol.36, No.5, 1990, pp.519-554.

[9] Fourer, R., "Database Structures for Mathematical Programming Models," *Decision Support Systems*, Vol.20, No.4, 1997, pp.317-344.

[10] Gagliardi, M., C. Spera, "BLOOMS: A Prototype Modeling Language with Object Oriented Features," *Decision Support Systems*, Vol.19, No.1, 1997, pp.1-21.

[11] Geoffrion, A.M., "The Formal Aspects of Structured Modeling," *Operations Research*, Vol.37, No.1, 1989, pp.30-51.

[12] Hiller, F.S. and G.J. Lieberman, *Introduction to Operations Research*, McGraw-Hill, Singapore, 1990.

[13] Huh, S.-Y., "Modelbase Construction with Object-Oriented Constructs," *Decision Science*, Vol.24, No.2, 1993, pp.409-434.

[14] Hull, J.C., *Options, Futures, and Other Derivatives*, Prentice Hall, New Jersey, 1997.

[15] Jarrow, R.A. and S.M. Turnbull, "A Unified Approach for Pricing Contingent Claims on Multiple Term Structures," *Review of Quantitative Finance and Accounting*, Vol.10, No.1, 1998, pp.5-19.

[16] Kempf, A., "Trading System and Market Integration," *Journal of Financial Intermediation*, Vol.7, No.3, 1998, pp.220-239.

[17] Kwok, Y.-K., "Accuracy and Reliability Considerations of Option Pricing Algorithms," *The Journal of Futures Markets*, Vol.21, No.10, 2001, pp.875-903.

[18] Lenard, M.L., "An Object-Oriented Approach to Model Management," *Decision Support Systems*, Vol.9, No.1, 1993, pp.67-73.
[19] Ma, J., "Type and Inheritance Theory for Model Management," *Decision Support Systems*, Vol.19, No.1, 1997, pp.53-60.
[20] Mayer, M.K., "Future Trends in Model Management Systems: Parallel and Distributed Extensions," *Decision Support Systems*, Vol.22, No.4, 1998, pp.325-335.
[21] Pindyck, R.S. and D.L. Rubinfeld, *Econometric Models and Economic Forecasts*, McGraw-Hill, Singapore, 1991.
[22] Raff, D.M.G., "Risk Management in an Age of Change," *working paper*, The Wharton School, University of Pennsylvania, 30 Jun., 2000.
[23] Rizzoli, A.E., J.R. Davis, and D.J. Abel, "Model and Data Integration and Re-use in Environmental Decision Support Systems," *Decision Support Systems*, Vol.24, No.2, 1998, pp.127-144.
[24] Zhang, J.Q. and E.J. Sternbach, "Financial Software Design Patterns," *Journal of Object-Oriented Programming*, Vol.8, No.9, 1996, pp.6-12.
[25] Zhuge, H., "Inheritance Rules for Flexible Model Retrieval," *Decision Support Systems*, Vol.22, No.4, 1998, pp.379-390.