Concise Papers

Toward the Notion of a Knowledge Repository for Financial Risk Management

Michel Benaroch

Abstract—An approach for designing a knowledge repository for risk management is presented. Since varied representations are used to capture the diverse types of knowledge involved in this domain, the atomic knowledge units stored in the repository are considered to be domain model (*K*) and inference method (*M*) pairs, (*K, M*) pairs, which address subtasks in the domain. Such (*K, M*) pairs are semantically uniform. Conceptually, this allows to view the repository as though it were a shared "database" of (*K, M*) pairs that has two key features. It serves stand-alone systems by enabling them to apply stored (*K, M*) pairs and share the generated results, where the applied pairs can be associated with any subtask that is part of entire application tasks. Additionally, it avoids capturing redundant subtask-specific *K*s to the extent possible by dynamically deriving them from deep principled *K*s it stores.

Index Terms—Financial application, knowledge-based system, knowledge repository, knowledge reuse, knowledge sharing, risk management.

1 Introduction

KNOWLEDGE-BASED technologies became a critical part of the information technology base of many financial firms, some of which are facing two issues. First, due to the ample size of most financial application tasks and their fairly decomposable nature, these firms' portfolio of knowledge-based systems (KBS) grew to include many small systems that solve isolated subtasks using varied technologies (production rules, neural networks, fuzzy logic, etc.). This complicates the integration of systems in the course of solving entire application tasks. Second, most firms' portfolios of KBSs grew to include redundant systems; in large firms, many groups are working apart to solve different tasks that have common subtasks, e.g., prediction, for which solutions often end up being reinvented and reimplemented.

At the heart of attempts to deal with these issues is the desire to build repositories of knowledge that stand-alone systems can share and reuse. The dominant proposals relating to this quest focus on reuse and sharing of "micro" knowledge units—concepts, facts, axioms, etc.—while commonly striving to address the nonuniform semantics of such units by modeling them through their underlying ontologies [8], [14], [15]. However, the problem that financial firms might face with this approach is twofold. First, many existing KBSs are not candidates for rewriting or redesign. More importantly, the search for common ontologies is complicated when the knowledge involved requires using representations that differ on their underlying epistemology, e.g., relational networks and neural networks [9]. Regardless, such proposals are the subject of several ambitious long-term projects that are at a relatively early stage, perhaps except for CYC [10].

• *The author is with the Quantitative Methods Department, School of Management, Syracuse University, Syracuse, NY 13244.*
E-mail: mbenaroc@mailbox.syr.edu.

Manuscript received Apr. 7, 1995.
For information on obtaining reprints of this article, please send e-mail to: transkde@computer.org, and reference IEEECS Log Number K96069.

We propose a less ambitious but more pragmatic approach to the design of a repository that stores "macro" knowledge units—pairs of domain models and matching inference methods—needed to address (sub)tasks in a domain. The semantic uniformity of these macro units allows to conceptualize a repository as being a "database" of such units, and to thus adapt some of the design principles underlying shared databases. This approach was applied in the risk management domain, resulting in a repository that has two key capabilities. First, it permits stand-alone systems to apply stored macro units and share results generated by these units, where the applied units can be associated with any subtasks that these systems might need to solve in order to address an entire application task. Second, the repository avoids the need for redundant domain models by being able to dynamically derive task-specific domain models from "deep" models it stores. Although we have applied this approach only in the risk management domain, its rather general underlying principles are likely to carry over to other domains. Our approach does not annul the long-term need for more basic research on sharing and reuse of micro knowledge units through modeling of their ontologies, but it provides a useful intermediate solution to a problem facing firms that rely extensively on knowledge-based technologies.

2 Goal in Context

Risk management deals with the actions one can take to protect a financial position against devaluative effects of economic changes [17]. A position can include various assets: traded products (stocks, bonds, etc.), liabilities (e.g., loans), finished goods, and the like. Each asset can be sensitive to different economic factors (e.g., interest rate, oil prices). Fig. 1 shows the risk management process, indicating that this process is fairly decomposable and that the specific tasks affiliated with each of its steps depend on the specific risk involved and the subuniverse of traded products that must be considered [17], [19]. Conceptually, the process can be said to involve a set of tasks τ. A task $T \in \tau$ can be comprised of subtasks $\{T_i\}_{i=1}^{m}$, $m > 1$, where the relevance of these subtasks to the process depends on the risky situation tackled.

Many support systems were built to solve tasks in τ; see [3] for a survey of KBSs and [19] for a review of systems that use conventional techniques like time-series analysis and optimization. These systems have two notable traits, provided that any task $T \in \tau$ they solve is said to be associated with a (*K, M*) pair, denoted *T*(*K, M*), where *K* is a domain model (e.g., linear program, rule-base, fuzzy logic model, historical data) and *M* is an inference method (e.g., simplex algorithm, rule chaining, fuzzy reasoning engine, genetic algorithm). First, these systems typically address small isolated subtasks (e.g., rating Treasury bonds, rating corporate bonds, rating municipal bonds). Second, they use different solution approaches with representations that vary on their underlying epistemology. Some apply model-driven approaches, e.g., qualitative reasoning with economic models [6] and rule-based inferencing with experts' heuristics [12]. Others use data-driven approaches, e.g., neural network techniques [13] and genetic algorithms [2] applied with historical data. Fig. 2 shows the general nature of these systems and the (sub)tasks they support, given that a system *S* which solves task *T* is a tuple *S* = (*T*(*K, M*), [*C*], *R*), where *C* is optional control knowledge for ordering the subtasks of *T*, and *R* is the results *T* generates (*R* is called a situation-specific model when it includes intermediate inference results [7]).

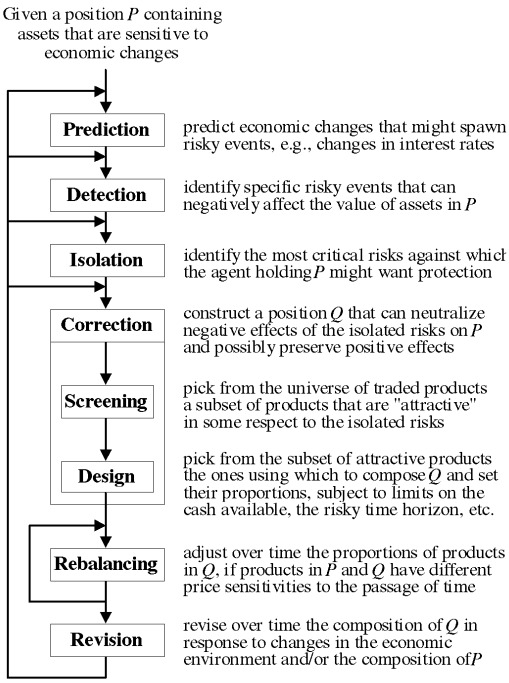

Fig. 1. Risk management process.

These traits raise problematic issues. One issue is the lack of integration of support systems used throughout the risk management process. Currently, risk managers often handle the overall process using ad-hoc experiential tactics, while relying only partially on available support systems [18]. The ability to integrate such systems is critical because, depending on the situation tackled, this process is usually "redefined" in terms of the subtasks it involves and the support systems it calls for. For example, during the revision step, having to consider a different subuniverse of products (e.g., stocks instead of bonds) requires addressing different screening subtasks using different support systems. Another issue is the need to centrally manage the way support systems are developed and used. Most reasonably sized firms have several groups that specialize in different types of financial risks and products. Because these groups address many of the same subtasks, it is not uncommon to find that they build systems which have the same underlying knowledge. This results in duplication of the knowledge coding, validation and maintenance effort.

There are very few publicized attempts to deal with these issues. For example, to deal with the integration issue, the system of Lee et al. [12] links two steps in the risk management process, but only when the subuniverse of products considered includes a small group of specializes stocks; it uses experts' If-Then rules to pick attractive stocks, and then formulates a quadratic programming model for designing a position. In contrast, Schwartz et al. [16] proposed using a blackboard architecture to allow systems to share results. While this approach might provide a more systematic way to deal with the integration issue, it is not geared towards addressing the second problematic issue discussed above.

In light of this state of affairs, our goal is to design a repository of knowledge pertaining to tasks in τ. The repository has to meet three requirements. First, it must allow any support system S to apply knowledge pertaining to (sub)tasks that might need to be solved in order to address any part of the risk management process. S need not know how the repository represents the knowledge; it only has to know what (sub)tasks to address, when, and how to integrate their results. Second, when there exist a deep K' from which several task-specific Ks can be derived dynamically, the repository should store only K'. Last, the repository should permit sharing and integrating results generated by the applied (sub)tasks. This also means allowing S to see these results, in case it uses control knowledge, C, that analyzes these results to decide on the next problem-solving activity it has to carry out.

3 The Knowledge Repository

Designing a shared repository first requires defining the atomic knowledge units to be stored. For reasons we discussed in Section 1, we adopt a macro approach that considers the atomic units to be tasks, or actually their associated (K, M) pairs. The uniform semantics of tasks, or (K, M) pairs, allows to conceptually view a repository as being a "database" of tasks that serves support systems which are in need to apply these tasks and share their results. This leads us to the notion of a *knowledge-base management system* (*KBMS*), denoted $\Phi = \{\rho(\tau), \alpha(\kappa)\}$, where $\rho(\tau)$ is a *shared repository* of the set of tasks τ in a domain and $\alpha(\kappa)$ is the *access manager* with κ being a knowledge dictionary.

3.1 Shared Repository of Tasks—$\rho(\tau)$

Provided that a support system is the tuple $S(T = \{T_i\}_{i=1}^{m}, C, R)$, $m \geq 1$, where C and R are specific to T, $\rho(\tau)$ is set to store only subtasks in $\{T_i(K, M)\}_{i=1}^{m}$ which can be common to other tasks. Tasks in $\rho(\tau)$ are organized mainly around the primary objects on which they apply transformations—*traded products*. Products fall naturally into classes

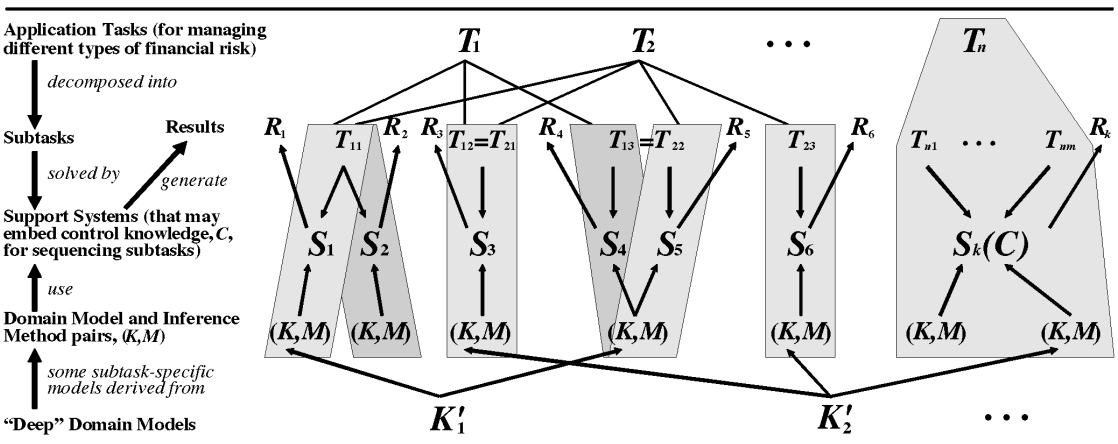

Fig. 2. Nature of existing systems and the risk management tasks they support.

TABLE 1
LOGICAL CONSTRUCTS

Real-World Concepts	Logical Constructs	
	Relational Database	OO Repository of Tasks -- $\rho(\tau)$
Data model	*collection of tables*	***object types hierarchy***
Entity set	*table*	***object type***: abstract domain entity describing a group of objects having common attributes (or properties), behavior (or operations), relationships to other objects, and semantics.
Entity instance	*row*	***object instance***: concrete object of a specific type. It is unique and has its own identity, OID. Each object type has one instance in $\rho(\tau)$, depicting a (sub)class of products or a capital market. Instances of individual products are rows in the relational database coupled with $\rho(\tau)$.
Entity set attribute	*column*	***attribute***: property shared by all objects of the same type; a field storing a literal or non-literal value that applies collectively to products of that type. (Attributes of individual products, e.g., maturity date, are table columns in the database coupled with $\rho(\tau)$.) ***function***: operation that may be applied to, or by objects of the same type. * ***task-function***: "hybrid" function that models a task in terms of its associated *(K,M)* pair. * ***utility-function***: function that is used mostly to access the database coupled with $\rho(\tau)$, by using off-the-shelf C/C++ functions to apply SQL-like operations. For example, it can issue such SQL-like operations to extract the data sets a certain *(K,M)* pair needs as input (e.g., historical stock prices that are the basis for picking attractive stocks), and store them in pre-specified files in the format that specific *(K,M)* pairs require. * ***monitoring-function***: function that inspects the state of objects vis-a-vis pre-defined types of events. For example, it can capture an event-driven rule of the form: IF (*attribute*) of (*OID*) has changed by (*value-range*) within (*polling-interval*) THEN notify of event occurrence, where *OID* is the monitored object (e.g., stock *s*), and the event is defined by the remaining parameters -- an *attribute* (e.g., price volatility) whose value changed by *value-range* (e.g., +5%) within a *polling-interval* (e.g., one hour). A monitoring-function could be repeatedly executed over a given time *duration* (see section 3.2).
Attribute of instance	*cell*	*attribute* or a *function* in an object instance
Attribute value	*literal atomic value*	literal or non-literal value (e.g., an object)
Relationship	*"common" column*	multiple inheritance relations capturing similarities among object types

and subclasses that bear a high degree of polymorphism. For example, a task for rating bonds can have several versions which are associated with different (*K*, *M*) pairs, each suitable for a certain class of bonds: Corporate bonds, Treasury bonds, etc. Thus, tasks are organized in an ISA taxonomy reflecting many similarities among product classes along dimensions like: sensitivity to similar economic events, maturity ranges, and trading procedures. We hence view $\rho(\tau)$ as being an object-oriented (OO) "database" of tasks. Fig. 3 shows the abstract *object types hierarchy* underlying $\rho(\tau)$, omitting the extensive use of multiple inheritance called for by the many similarities among product classes. An instantiation of this types hierarchy, or $\rho(\tau)$, "sits" on top of a relational database of product instances.

Table 1 lists the logical constructs used in $\rho(\tau)$ and their counterparts in databases. These are the standard OO constructs—object type, object instance, attribute, and function [5]—except for the distinction made between three types of functions, of which task-functions are central to the role of $\rho(\tau)$. A ***task-function*** models the (*K*, *M*) pair associated with a task. It cannot call, or be defined in terms of, other task-functions, to avoid embedding control knowledge, *C*, that might be specific to a certain application task. It is defined to be the triplet:

$$\langle \texttt{Task\_pointer}, \texttt{task\_trigger}, \texttt{task\_result} \rangle,$$

where *task-pointer* points to a reasoning module holding a (*K*, *M*) pair, *task-trigger* is a utility-function that activates *M* on *K* via the task-pointer, and *task-result* is an attribute storing the result (*K*, *M*) generates. A reasoning module can be:

1) An instance of an off-the-shelf or custom-defined C++ object type containing the representational constructs needed to model knowledge units (e.g., rules) comprising a specific kind of *K*, a function implementing the *M* (e.g., rule chaining) called for by that kind of *K*, and general-purpose utility-functions (e.g., for loading *K* from a file);
2) An instance of a C++ object containing an explicit binding relation with an external object-code module of (*K*, *M*) that was produced by a reasoner for which a compiler exists; or
3) an instance OLE object which upon request is sent to a reasoner that runs under Microsoft Windows.

Either one of the above schemes for modeling a (*K*, *M*) pair would work well when *K* is not a deep principled model. For example, if *K* is a neural network model that is used with a screening task which selects attractive stocks, it would not be associated with any other screening task involving products of a different type (e.g., bonds); thus, this *K* can be stored explicitly using the first scheme, or as an object-code module compiling *K* and *M* using the second scheme.

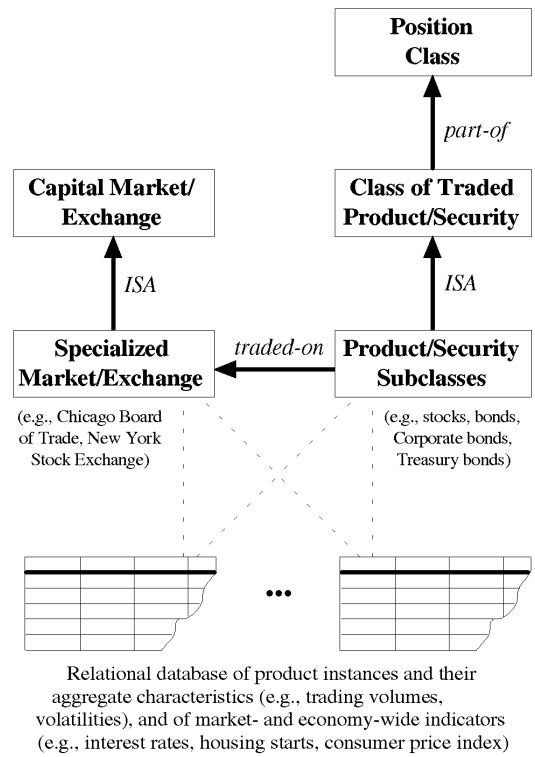

Fig. 3. Abstract object types hierarchy underlying $\rho(\tau)$ and its coupled databases.

TABLE 2
LOGICAL OPERATIONS

		Logical Operations			
		Derive/Modify: change entity attributes	***Select***: find entities that meet specific criteria	***Join***: create new entities from "finer" entities	***Monitor***: detect the occurrence of events
DBMS	***command***	update R set C_i="value"/"expression"	select $C_1,...,C_n$ from R where "search criteria"	join R and R_i where "join criteria"	
	arguments	table R, column C_i	table R, columns $C_1,...,C_n$	tables R and R_i, common column C_i	
	parameters	"value" or "expression"	"search criteria"	"join criteria"	
	action	"value" stored in column C_i	R scanned to identify rows meeting "search criteria"	R and R_i scanned to produce joined rows from R and R_i	
KBMS ***(Φ)***	examples	derive and modify the credit ratings of bonds	identify all products that have a given risk-profile, or offer the highest price appreciation potential	create positions that meet given goals and constraints, or formulate a mathematical optimization program	detect a 5% price change occurring over an hour period, to indicate a need to rebalance/revise positions
	command	⟨**SID, T(parameters), *o*/**o*, output-form**⟩			
	arguments	**SID** of calling S, task- or monitoring-function T, object o (e.g., BOND) or bag† of objects $*o$ (e.g., *BOND), **output-form** requested			
	parameters	T-specific			
	action	T is applied to object o, or iteratively to all subclass objects in $*o$ (one subclass at a time) (if T is a monitoring-function, it could also be applied to a single product instance in the database coupled with $\rho(\tau)$)			

†A *bag* is a set with redundancy, where redundancy is the result of having multiple inheritance relations defined among object types.

When the K in a (K, M) pair has a deep underlying model K', it is better to store only K' using the first scheme, and use special-purpose functions to dynamically derive from K' the Ks different tasks need to reason about domain phenomena from various angles or at different abstraction levels. Those special-purpose functions are reasoner-specific utility-functions defined for the C++ reasoning module requiring K. This reduces the number of Ks stored in $\rho(\tau)$, and lowers the effort needed to maintain them. Examples of how this scheme is used are provided in Section 4.

Overall, the task-function construct we defined indicate that the object types underlying $\rho(\tau)$ are not "flat." It enables objects to hold causal models linking the behavior of products to economic variables, design models for synthesizing positions from products, event models capturing conditions that might lead to the triggering of (K, M) pairs, process models describing the execution procedure for sell/buy orders in various exchanges, and so on.

3.2 Access Manager—$\alpha(\kappa)$

The logical operations a support system S can initiate with respect to tasks in $\rho(\tau)$ are identified in light of the purpose for which tasks are applied in various parts of the risk management process (see Fig. 1). Assuming that results of these tasks can be stored in $\rho(\tau)$ and its coupled database, Table 2 defines these logical operations and summarizes the way they are initiated and executed in $\rho(\tau)$, in part based on their conceptual similarity to logical database operations. To keep $\rho(\tau)$ transparent from system S, requests to initiate logical operations are sent to the access manager, $\alpha(\kappa)$. A request is defined to be the tuple:

$$\langle \texttt{SID, T(parameters), argument output-form} \rangle$$

where *SID* identifies the requesting S, *T*(*parameters*) is the task-function or monitoring-function to be applied and the parameters it requires, *argument* is an object or a bag of objects on which to apply T (e.g., STOCK denotes the class of stocks, and bag *STOCK denotes all subclasses of STOCK), and *output-form* identifies the types of requested output. Note that a request to apply a monitoring-function in fixed intervals for some duration is initially sent by the requesting S, and follow-up requests to reapply the function are sent to $\alpha(\kappa)$ with a time-stamp by the function itself after each time it is executed. With respect to the type of output requested, this can include:

1) A stream of OIDs along with some of their attribute values (stored in task-result attributes), formatted as a prefix tree based on the order of objects in the types hierarchy underlying $\rho(\tau)$.
2) Copies of the object(s) argument storing the SID of S and the results of T. The objects are created in $\rho(\tau)$ to localize effects of T on $\rho(\tau)$, as T might derive results that apply only to the market conditions assumed by S. This feature is often called *versioning* in the context of OO databases [5].
3) New objects in $\rho(\tau)$ and database tables storing composed positions and their instances.

To execute requested logical operations, $\alpha(\kappa)$ relies on its knowledge dictionary κ. κ stores the definition of the object types hierarchy underlying $\rho(\tau)$; due to the way task-functions are defined, κ need not include details about how stored (K, M) pairs are implemented or represented within instantiated objects in $\rho(\tau)$. The way α uses κ to execute a request depends on the argument sent. If the argument is an object O (e.g., STOCK), and the requested T is a task-function of O, α simply activates T on O. If the argument is a bag $*O$ (e.g., *STOCK), polymorphism requires to apply all variants of T which might exist in objects specializing O (e.g., subclasses of STOCK). So, α uses its own iterative (depth-first and breadth-first) procedures to scan the part of $\rho(\tau)$ corresponding to $*O$, and activate T on every object in $*O$, one at a time. In both cases, when versioning is required, α activates T from within a copy of the object argument. α creates this copy before activating T, unless it already exists due to an earlier request from the same S.

How does α produce the output requested? If S asks for versioning, by α invoking T from within version objects, T stores its results in these objects. When a stream of OIDs formatted as a prefix tree is requested, α creates the stream while iteratively scanning the part of $\rho(\tau)$ corresponding to the bag argument; a stream can include, e.g., SSMs that the activated T constructs and stores in task-result attribute of the argument object(s). If S asks to store positions composed by T, α creates in $\rho(\tau)$ instances of object type POSITION where it stores the composed position specifications.

4 REPOSITORY IN USE: AN ILLUSTRATION

Let us see how the prototype KBMS for risk management, Φ, serves an application system called HEDGER. HEDGER is itself a prototype KBS that aims at addressing relatively varied risk management situations, by applying tasks Φ currently stores. The nature of these situations can be best explained using the example in

TR: My firm plans to issue bonds to raise capital for a project. We believe that interest rates are going to rise, with a chance that they will decline prior to the issuance date. We need to have a ***hedge position*** -- to buy and/or sell products -- that meets our "cap" ***risk-profile***. If interest rates rise, the position should increase in value to offset the extra cost of issuing bonds with a higher yield rate; otherwise, it should decline in value only to a point where the firm can still benefit from issuing bonds with a much lower yield rate.

RM: Buying ***put options*** on bonds can cap the cost of issuing bonds, as they give the right to sell the underlying bonds for some agreed upon price, called strike price, at a fixed expiration date. An increase in interest rates will cause bond prices to decline, allowing you to exercise the option to sell bonds at the higher agreed price. The gain on the sale will offset the extra cost of issuing bonds with a higher yield rate, thus capping the issuance cost. If interest rates decline, bond prices will rise, rendering the put options worthless. But, the reduced cost of issuing bonds with a lower yield rate will offset, and more, the loss on the put options.

TR: This would require a large upfront cash fee to buy the put options. The firm is short on cash.

RM: Let us change the risk-profile slightly -- call it now "floor-cap". We can sell call options with a smaller exercise price to raise enough cash to be able to pay for the put options needed to cap the issuance cost.

TR: But, will it be easy to unwind the position if the firm decides to cancel the rpoject and the bond issuance?

RM: Since your firm's credit spread is stable, we should use exchange-traded options on Treasury bond rates, not over-the-counter options on corporate bond rates. Exchange-traded option are easier to unwind; they are less customized, and thus more liquid and less costly to unwind.

TR: What's the level of protection the firm gets?

RM: Relatively high. Exchange-traded options involve no credit exposure, no repo risk, etc. However, note that they can require a ***mark-to-the-market treatment***, which is an accounting arrangement whereby profits or losses on a position are documented and settled up daily.

TR: What do we do next?

RM: We have established the type of position, or its configuration. We need to "refine" the position -- decide on the type of underlying bonds to use, with what party to trade the options (e.g., Chicago Board of Trade, International Money Market), the options' exact maturity dates and strike prices, how much should you be willing to pay for them, etc.

TR: . . .

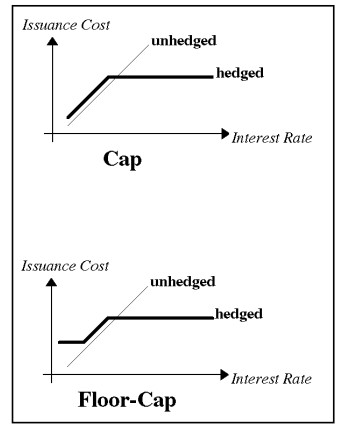

Fig. 4. Sample dialogue scenario between a risk manager (RM) and the treasurer of a firm (TR).

Fig. 4. This example shows that HEDGER focuses on the screening and design parts of risk management (see Fig. 1). It constructs positions while assuming that the risk factors involved are known and that the design constraints and objective functions are given. These constraints require matching a desired risk-profile, matching maturity dates, not exceeding available cash in setting up positions, etc. The objective functions are: maximize liquidity, minimize setup cost, and minimize credit risk, among others. For clarity, we assume that HEDGER applies a sequential four-step solution strategy:

1) *create* alternative positions,
2) *filter out* infeasible alternatives,
3) *rank* the remaining alternatives based on the objective functions, and
4) *pick* the best position upon applying various quantitative analyses.

We will see later that HEDGER uses a different strategy. Following the example in Fig. 4 we next review the major activities HEDGER carries out by applying tasks in Φ, and elaborate only on those tasks that illustrate the principle capabilities Φ offers.

4.1 HEDGER's Interaction with Φ

HEDGER first creates position configurations that meet the goal risk-profile; a configuration is a blend of products, without specification of unit proportions, maturity dates, etc. HEDGER does so by sending Φ a few requests to apply tasks, starting with one that identifies product classes which are sensitive to the given risk factor, *r*, or interest rate:

Request 1: ⟨*HEDGER*, *Find_Sensitive_Products*("*r*"),
*PRODUCT, versioning / prefix-bag⟩.

In the (*K*, *M*) pair associated with this task, *K* is a formal *valuation model*, $f(x_1, ..., x_n)$, that captures the price sensitivity of products of some type to the economic variables $x_1, ..., x_n$ [17], and *M* is a simple method that checks if *r* is one of the variables in the model. As the argument is a bag, *PRODUCT, and the required output calls for versioning, the access manager, α, hierarchically scans $\rho(\tau)$, creating a version copy for each class object and applying the task on that object. The task marks in version objects each product class that is sensitive to *r* as "feasible." Also, α returns the OIDs of feasible product classes formatted as a prefix tree:

(PRODUCT (BOND (T-BOND CORPORATE-BOND ...))
(STOCK (UTILITY-STOCK (...))(OIL-STOCK (...)) ...)
(OPTION (...) (...) ...)).

HEDGER asks for this because, when many classes are sensitive to the risk factor, it would need to identify the most sensitive classes by asking Φ to apply other tasks that involve, e.g., statistical techniques and neural network models.

HEDGER next uses the identified feasible product classes to configure positions that provide the goal risk-profile. As seen in Fig. 4, the risk-profile of a position is derived by qualitatively simulating the way the value of the position (e.g., buy PUT options) changes as a function of how the value of the products involved (e.g., PUT options) changes, contingent on the behavior of the risk factor (e.g., *r* is increasing). *Qualitative simulation* (QSIM) is a technique that can emulates this analysis [4]. HEDGER hence sends Φ the request:

Request 2: ⟨*HEDGER*, *Derive_Risk_Profile*("[*r inc* (r_{now}, ∞)]"),
**PRODUCT / "feasible," versioning⟩.*

The *K* associated with this task is a set of qualitative constraint equations that are dynamically derived from a deep *K'*, the valuation model of some class of products, $f(x_1, ..., x_n)$, and *M* simply implements QSIM. The corresponding task-function has a pointer to an instance of a C++ reasoner that uses its own utility-functions to read a file storing $f(x_1, ..., x_n)$ as a prefix tree and to derive from the tree qualitative constraint equations, before activating *M*. α applies this task iteratively on subclasses of *PRODUCT marked "feasible," generating two risk-profiles of each product class, for a buy and a sell action, and storing them in version objects. HEDGER then launches another simple task in $\rho(\tau)$ that marks as "feasible" only the product classes whose risk-profile, for a buy or a sell action, matches the goal risk-profile. A feasible class now corresponds to a *generic position* configuration involving products from a single class.

For complex goal risk-profiles, HEDGER configures *compound positions*—combinations of generic positions. Going back to the example in Fig. 4, suppose that all generic positions providing the cap risk-profile are found infeasible because their setup cost exceeds the amount of cash available. HEDGER uses its own heuristics to relax constraints, e.g., the heuristic "resolve cash shortage

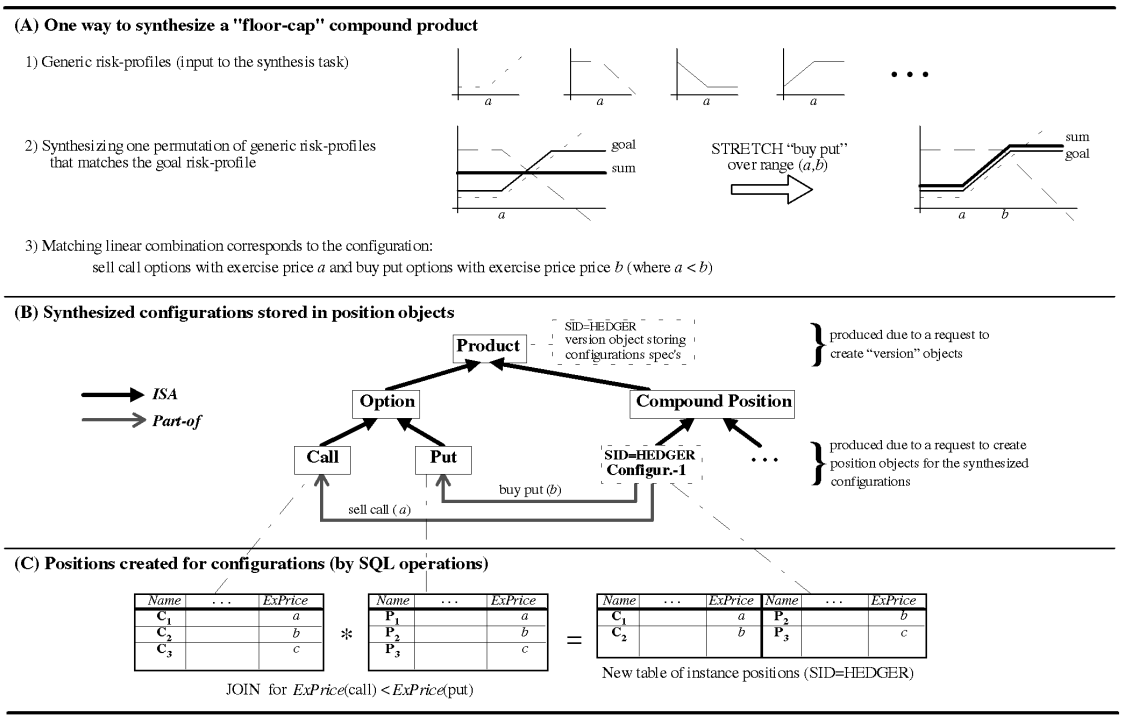

Fig. 5. Creation of compound risk management positions.

by selling products" leads it to redefine the goal risk-profile to be a floor-cap (see Fig. 4). HEDGER now creates compound configurations that meet the floor-cap risk-profile by sending Φ the request:

Request 3: ⟨*HEDGER*, *Synthesize_Risk_Profile*(*goal risk-profile*), *PRODUCT*, *versioning* / *create-position-objects* / *prefix-tree*⟩.

The (*K*, *M*) pair associated with this task is as follows. *K* contains the risk-profiles of generic positions; it is dynamically formed using a utility-function that scans the product classes hierarchy and retrieves the results request 2 stored in version objects. The idea behind the *M* used to configure compound positions is: their risk-profile is the algebraic sum of risk-profiles of their comprising generic positions, e.g., Fig. 5A shows one way to synthesize a floor-cap as a linear combination of two generic risk-profiles. *Qualitative synthesis* (QSYN) is a technique that can synthesize any goal risk-profile in the same fashion [4]. It uses a goal-driven process to search the space of permutations of elementary piecewise linear functions it is given as input, narrowing down the space by heuristically stretching and / or steepening edges in permutations likely to lead to a match with the goal risk-profile. Thus, *M* is simply an implementation of QSYN. This (*K*, *M*) pair is applied only on object PRODUCT. Based on the requested output, α produces a version object storing the synthesized configurations and new position objects corresponding to these configurations (see Fig. 5B), and a prefix-tree with the OID of these position objects. HEDGER then asks Φ to apply another task that issues SQL-like join operations to produce database tables containing individual positions, based on synthesized configurations (see Fig. 5C). Note that, when the number of configurations produced is large, HEDGER might ask Φ to apply tasks that filter out infeasible configurations, before asking to create individual positions.

HEDGER *filters* out infeasible configurations and / or individual positions by applying other constraints. For example, when accounting constraints require avoiding a mark-to-the-market treatment (see Fig. 4), HEDGER sends the request:

Request 4: ⟨*HEDGER*, *Filter_By_Accounting_Treatment*("≠," "*m-to-m*"), **COMPOUND* / "*feasible*," *versioning*⟩.

This task is applied only on position configurations (subclasses of *COMPOUND created by request 3, see Fig. 5B), since all individual positions corresponding to a configuration require the same accounting treatment. For each configuration, the task simply checks if the product classes it involves meet the required accounting treatment. If not, it marks that configuration as "infeasible." Another example involves the maturity-date constraint, which requires all positions to mature around a given date. Here, the corresponding task-function filters out individual positions by issuing SQL-like commands on database tables (Fig. 5C), and these mark as "infeasible" positions which violate this constraint. Another example involves a cash constraint, which entails filtering out positions whose setup requires more cash than is available. This constraint is enforced using two tasks. One task involves a *K* that is dynamically derived from a principled *K'* which models the trading process associated with specific types of products. For example, the *K'* modeling the sale of a call option [17] embodies formal transactions reflecting the actors, actions, timing requirements, etc. involved in a trade:

```
(t0, Deliver, buyer, exchange, option-price)
(t0, Deliver, exchange, buyer, ownership)
(t0, Deliver, exchange, seller, option-price)
(t0, Deliver, seller, exchange, initial-margin)
 ...
```

Thus, the first task dynamically extracts the subset of transactions reflecting cash requirements (e.g., initial margin) for the specific product classes involved in each configuration. The second task then uses these transactions to compute the setup cost of positions, upon applying trading regulations its *K* captures as If-Then rules. For example, when a compound position comprises two offsetting generic positions that are traded on the same exchange, the required initial margin can be the difference of the initial margins for the two generic positions.

After applying feasibility constraints, HEDGER identifies a relatively small number of "superior" positions by asking Φ to apply tasks in $\rho(\tau)$ that *rank* the remaining feasible positions along

objective functions. Since these tasks do not involve any capabilities of Φ that we have not seen so far, we will not elaborate on these tasks.

Finally, HEDGER has to *pick* the "best" position among the superior positions identified. We can consider the role of HEDGER up to this point as being one of an intelligent assistant that, for the most part, does a lot of screening of alternatives to identify for the risk manager a small set of the most promising positions. From this set, a risk manager has to pick the best position based on a details quantitative analysis. Respectively, Φ is now being expanded to support the necessary quantitative analysis activities, by adding other tasks to $\rho(\tau)$. One of these tasks involves optimizing positions using Constraint Logic Programming techniques [11]. Another task involves the automatic formulation of a mathematical optimization program based on the superior positions found. Unlike common cases where optimization is used, that model should be quite tractable because the screening HEDGER performs vastly lowers the number of alternative positions and constraints involved. Other tasks pertain to the use of what-if analysis for quantifying the protection level positions provide and assessing tradeoffs between positions in response to changes in position parameters. The idea is to use valuation models to do a symbolic curve analysis of risk-profiles in ranges where position parameters can plausibly fluctuate, and notify of parameter ranges where one position is superior to the others.

4.2 HEDGER's Internals

Having said that an application system *S* is the tuple

$$S(T = \{T_i\}_{i=1}^{m}, C, R),$$

it can be said that in HEDGER's case all the subtasks of *T* are currently stored in Φ. As to *R*, the results subtasks generate (e.g., SSMs of synthesized position configurations), these are stored in version objects, enabling HEDGER to send to Φ inquiries that would help it to determine, e.g., what feasibility constraint(s) to relax when all positions are found infeasible. We saw that some of these results are sent back to HEDGER for control purposes, e.g., in request 1. This brings us to the control knowledge, *C*, HEDGER uses to sequence subtasks it applies. So far we pretty much ignored *C* by assuming that HEDGER follows a sequential solution strategy (propose, filter, rank, and pick). In effect, this strategy would not work well. For example, sometimes the propose step could produce a vast number of individual positions based on the synthesized configurations. HEDGER avoids producing all these positions by skipping momentarily to the filter step to apply certain constraints directly on configurations. In fact, depending on the constraints involved, HEDGER might start applying feasibility constraints to filter out product classes before even trying to configure positions. As this example indicates, HEDGER uses normative principles and task-specific heuristics to control the order of requests it sends to Φ. For example, one normative principle is qualitative abstraction, whereby subtasks that reason about qualitative or aggregate position attributes are given preference. HEDGER captures control principles and heuristics using If-Then rules, and uses them with an agenda mechanism to prioritize the subtasks it needs to apply.

5 Conclusion

We presented a pragmatic approach to the design of knowledge repositories that support reuse and sharing of macro knowledge units, (*K, M*) pairs, pertaining to tasks in a domain. As we used this approach to develop a repository for risk management two critical issues that require further research surfaced. First, our current design assumes that (*K, M*) pairs are self-contained in the sense that they do not require any user input during their execution. The need for such input requires supporting an ongoing communication between an applied (*K, M*) pair and the support system that applied it. The difficulty here is one of managing the "traffic" between the repository and the multiple systems its aims to serve concurrently. The second issue pertains to the ability to expand the degree to which such a repository supports knowledge reuse and sharing. Recall that our approach does not address the long-term need for more fundamental research on reuse and sharing of "micro" knowledge units through modeling of their ontologies. The main question in case of a repository that already organizes knowledge around the primary ontological domain entities (e.g., products, markets) is this: can we incrementally add ontological knowledge to the repository so as to gradually support reuse and sharing of micro knowledge units as well? The ability to move in this direction would allow the capabilities of a repository of the kind we discuss to evolve over time, in line with research progress made on the reuse of micro knowledge units.

References

[1] E. Baecher, and S. Goodman, *The Goldman Sachs Guide to Hedging Corporate Debt Issuance*. Financial Strategies Group, Goldman Sachs, 1988.

[2] R. Bauer, and G. Liepins, "Genetic Algorithms and Computerized Trading Strategies," *Expert Systems in Finance*, D. O'Leary and P. Watkins, eds. New York: Elsevier Science Publishers, 1992.

[3] M. Benaroch, and V. Dhar, "On the Scope of Reasoning with Financial Knowledge: An Analysis for the Investment Domain," *J. Organizational Computing*, to appear.

[4] M. Benaroch, and V. Dhar, "Controlling the Complexity of Investment Decisions Using Qualitative Reasoning Techniques," *Decision Support Systems*, vol. 15, pp. 115-131, Dec. 1995.

[5] A. Björnerstedt, and C. Hultén, "Version Control in an Object-Oriented Architecture," *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky, eds. New York: ACM Press, 1989.

[6] R. Brendsen, and H. Daniels, "Causal Reasoning and Explanation in Dynamic Economic Systems," *J. Economic Dynamics and Control*, vol. 18, pp. 251-271, June 1994.

[7] W. Clancey, "Model Construction Operators," *Artificial Intelligence*, vol. 53, pp. 1-115, Jan. 1992.

[8] P. Gray, "Knowledge Reuse through Networks of Large KBS," *Directions in Databases*, D. Bowers, ed. Berlin: Springer-Verlag, 1994.

[9] T. Gruber, "The Role of Standard Knowledge Representation for Sharing Knowledge-Based Technology," Technical Report KSL-90-53, Knowledge Systems Laboratory, Stanford Univ., Stanford, Calif., 1990.

[10] R. Guha, and D. Lenat, "Enabling Agents to Work Together," *Comm. ACM*, vol. 37, Apr. 1994.

[11] C. Lassez, K. McAloon, and R. Yap, "Constraint Logic Programming and Option Trading," *IEEE Expert*, vol. 2, 1987.

[12] J. Lee, H. Kim, and S. Chu, "Intelligent Stock Portfolio Management System," *Expert Systems*, vol. 6, pp. 74-86, 1989.

[13] W. Messier, and J. Hensen, "Inducing Rules for Expert System Development: An Example Using Default and Bankruptcy Data," *Management Science*, vol. 34, pp. 1-13, Dec. 1988.

[14] M. Musen, "Dimensions of Knowledge Sharing and Reuse," *Computer and Biomedical Research*, vol. 25, pp. 435-467, July 1992.

[15] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and R. Swartout, "Enabling Technology for Knowledge Sharing," *AI Magazine*, vol. 12, pp. 36-56, July 1991.

[16] D. Schwartz, L. Sterling, and E. Mayland, "FLiPSide Blackboard: A Financial Logic Programming System for Distributed Expertise," *Proc. First Int'l Conf. AI Application on Wall Street*, 1991.

[17] C. Smith, C. Smithson, and D. Wilford, *Managing Financial Risk*. HarperBusiness Publishing, 1990.

[18] E. Sorensen, "Constructing Portfolios with Equity Valuation," *Portfolio and Investment Management*, F. Fabozzi, ed. Chicago: Probus Publishing Company, 1989.

[19] S. Zenios, *Financial Optimization*. Boston: Cambridge Univ. Press, 1993.