# Optimizing Procurement Decisions in Networked Virtual Enterprises

*Amihai Motro, George Mason University, USA*

*Alexander Brodsky, George Mason University, USA*

*Nathan Egge, George Mason University, USA*

*Alessandro D'Atri, Luiss Guido Carli University, Italy*

## ABSTRACT

*A virtual enterprise is an ad hoc coalition of independent business entities who collaborate on the manufacturing of complex products in a networked environment. This collaboration is enabled by the concept of a transaction, a mechanism with which members acquire necessary components from other members. An external procurement request submitted to the enterprise launches a tree-structured series of transactions among its members (similar to supply chains). Each such transaction is associated with a purchase price, but also with a risk of failure. That members have the option to procure components from different co-members, each charging its individual price and posing its specific risk, raises challenging optimization problems related to the fulfillment of business objectives. This paper defines a transaction model for virtual enterprises, with formal concepts such as price, risk, and business objectives. The Decision Guidance Query Language (DGQL) is presented, a language for modeling and solving optimization problems in a database setting, and shows how DGQL can provide intuitive and efficient solutions to the optimization problems raised in the model. The model, the optimization programs, and the experimentation promote strong collaboration and common objectives among its members, and one in which collaboration is limited, with members retaining much of their autonomy and individual objectives.*

*Keywords:* *Collaborative Decision Making, Decision Support, Optimization, Supply Chain, Transaction, Virtual Enterprise*

## INTRODUCTION

A *virtual enterprise* is a coalition of autonomous business entities, usually of small or medium scale, who collaborate on the manufacturing of complex products, often with the intention of competing with large, monolithic enterprises. The members of a virtual enterprise often possess complementary skills and technologies whose combination is deemed necessary for the target product, and the collaboration is often ad

DOI: 10.4018/jdsst.2012070104

hoc, for a specific product only, after which the virtual enterprise might dissolve.

Within this general framework, the level of collaboration among enterprise members, and the extent of sharing of information and strategic decisions can vary substantially, creating virtual enterprises of significantly different styles. In one setup, enterprise members preserve their independence to the greatest degree possible. They share only a minimal amount of information (e.g., the products they are willing to make available to others and the prices they charge), and they optimize their performance according to their own interests and criteria. At the opposite extreme, enterprise members share all their information (e.g., manufacturing processes, sources of supply, costs, and risks), and abide by a global optimization process that instructs them on their production steps. We refer to the former setup as an autonomous enterprise, and to the latter as a coordinated enterprise.

The primary means for enabling collaborations in virtual enterprises are transactions: bilateral exchanges between two enterprise members in which goods are delivered in return for payment. The fulfillment of a target product may thus propagate into a tree-structured set of transactions among the members of the enterprise. Since the same product can often be procured from multiple enterprise members, a target product may be fulfilled with alternative transaction trees. Since each procurement decision is associated with performance parameters such as product price and the risk of non-delivery, members must select their transaction partners judiciously. This presents substantial optimization challenges.

In this paper we explore issues of optimal decision making in virtual enterprises using the Decision Guidance Query Language (DGQL), a language for solving decision optimization problems. A brief overview of the language is provided in the section *The Decision Guidance Query Lang*uage. The section that follows it describes our formal model for virtual enterprise transactions, including concepts such as transaction cost, product price, and procurement risk. Using expected profit as optimization target, the subsequent section presents the DGQL programs for two types of virtual enterprises: autonomous and coordinated. A system that implements (compiles and executes) such DGQL programs is described in the next section. This section also reports on experiments with both autonomous and coordinated virtual enterprises. The final section summarizes our findings and suggests various directions for future work. We begin with a brief review of work related to this research.

## BACKGROUND

To put this work in context, we review briefly of related work in two areas: virtual enterprises and optimization tools.

## VIRTUAL ENTERPRISES

Cooperatives of independent entities that collaborate on the manufacturing of goods have been around for decades. Often the members of such cooperatives reside in the same industrial district. This geographical proximity provides advantages of common culture and mutual trust (Brusco, 1992). The collaborating entities are often of small and medium size, and their strategic approach is to focus on their core business (i.e., excel in a limited section of the "value chain"), and to seek collaborations with neighboring entities to perform the other requisite activities in the value chain.

Essentially, virtual enterprises (also referred to as virtual organizations or corporations) are modern versions of these cooperatives, from which geographical constraints have been removed. By means of communications and information technology, the entities participating in an alliance need not be confined to a particular location. Virtual enterprises are often characterized as agile, flexible, dynamic, proactive, and unconstrained by predefined structures. The essential principles of virtual enterprises may be summarized thus (Davidow & Malone, 1992; Goldman, Nagel, & Preiss, 1995; Camarinha-Matos, 2003; Barbini & D'Atri,

2005): (1) **Market-driven cooperation**. Virtual enterprises are set-up to exploit specific business opportunities, and are therefore intensely result-oriented. (2) **Complementariness of skills**. The members of each virtual enterprise are chosen to complement each other's competencies. (3) **Dynamic participation**. Members can join or withdraw from an enterprise, according to their own self-interests. (4) **Coalition of peers**. A virtual enterprise is not dominated by individual members; rather, it is a coalition of peers. (5) **Controlled sharing**. Members work together, integrating their processes and sharing their resources; yet, sharing is not boundless, and members may protect certain assets from their peers. (6) **Limited duration**. Virtual enterprises are not intended to be permanent, or even long-term organizations; rather, they are aimed at achieving short or medium term goals.

The interest of the information technology research community in the area of virtual enterprises dates to the mid-1990s, with much of the work focusing on organizational issues, communication processes and information systems support (Mowshowitz, 1997; Monge & DeSanctis, 1999). An overview of current approaches towards the establishment of infrastructures for virtual enterprises is given in Camarinha-Matos and Afsarmanesh (2004).

The concept of transaction has been discussed extensively in economics and related disciplines (business, banking, etc.); and (with a considerably different interpretation) in computer applications such as database systems (Elmagramid, 1992) or workflow systems (Grefen, 2002). In the area of virtual enterprises, the VirtuE model (D'Atri & Motro, 2008, 2010) introduced a concept of transaction that combines elements from both distributed database systems and economics. In other words, it used the structures of computer transactions to implement concepts borrowed from economics.

The work here continues in this vein, with the introduction of a cost model. This cost model, which borrows concepts from transaction cost theory (for example, search and information cost) (Smith, Venkatraman & Dholakia, 1999), enables us to discuss formally concepts such as transaction cost, failure, risk, and revenue in virtual enterprise environments.

## OPTIMIZATION TOOLS

The problem of decision optimization deals with finding values for control variables that maximize or minimize an objective within given constraints. It is used in many applications such as deciding on optimal manufacturing patterns and sourcing of virtual enterprises, or more broadly, deciding on optimal business transactions within supply chains. The state-of-the-art implementation of decision optimization applications involves mathematical and constraint programming (MP and CP), using languages such as AMPL (Fourer, Gay, & Kernighan, 2002) or GAMS (Boisvert, Howe, & Kahaner, 1985).

While software developers find database programming mostly intuitive, they typically do not have the mathematical expertise necessary for MP and CP. In contrast to MP and CP, database management (DBMS) tools are more intuitive and have been adopted in many application domains. In addition, much investment has already been spent on database applications. Clearly, it is desirable to leverage this investment when building decision optimization applications. However, DBMS query languages are not designed for decision optimization as they cannot express decision optimization problems, notably over continuous variables. Indeed, in the continuous variable case, there are potentially infinite possibilities to choose from that cannot be expressed as regular database queries. For the discrete case, when potential choices are from a large space, populating tables with all possible choices and then ranking them with a query can be rather inefficient. Although query languages can handle some limited discrete optimization computations, e.g., find a tuple that has a minimal value over a finite set of discrete choices (Ilyas, Beskales, & Soliman, 2008), even in the cases of expressible rank queries, evaluation algorithms have not typically taken advantage of MP and CP search strate-

gies to achieve potential efficiency and flexible optimization goals. More importantly, the optimization query will look very different from the reporting query, increasing the complexity of system management.

Specialized optimization tools (e.g., for optimizing price-revenue, transportation, sourcing or production planning) have been developed for the benefit of end users, or for integration with other systems (e.g., ERP/ERM), so that operations research expertise would not be necessary. However, this approach is not extensible, and does not support general decision-guidance application development.

In this paper, we implement the problem of optimizing virtual enterprise transactions using the Decision Guidance Query Language (DGQL) (Brodsky, Bhot, Chandrashekar, Egge, & Wang, 2009), a language designed for making decision optimization easier, especially in applications where database technology is used heavily. Roughly speaking, DGQL is to SQL what the language CoJava (Brodsky & Nash, 2006) is to the object-oriented programming language Java; i.e., a completely procedural specification is translated into a declarative optimization problem. The language Modelica (Fritzson & Engelson 1998) also allows the specification of constraints using a procedural-like specification, although it is fundamentally an equation specification language. In a nutshell, DGQL provides query-like abstractions for expressing decision optimization problems so that database programmers would be able to use it without prior experience in MP, and more importantly, they would be able to reuse the queries already built into existing applications. An informal description of DGQL is given next.

## THE DECISION GUIDANCE QUERY LANGUAGE

A key observation that motivated the development of DGQL is that database languages are intuitive to use for computing business metrics, e.g., for reporting purposes, while decision optimization in principle is often the "inverse" of the reporting functionality already in place; that is finding operational choices that optimize a business metric, e.g., minimize cost. In addition, code used in reporting functions often contains business logic that is needed in the decision optimization tasks. Based on this observation, a decision optimization problem in DGQL is written as a "regular" database program, i.e., a sequence of relational views and accompanying integrity constraints, together with annotation of which database table column needs to be decided by the system (i.e., variables) and toward what goal (i.e., optimization objective). Here, existing queries in the reporting software can be used directly. Essentially, DGQL allows users to write an optimization problem as if writing a reporting query in a forward manner.

The challenge in the DGQL approach, however, is how to execute the "inverse" decision optimization based on a "forwardly" expressed code. This is done by encoding the DGQL queries as an MP formulation, and by solving the MP problem and then deriving the solution to the DGQL optimization problem. A technical question with two interrelated parts arises: (1) Is it possible to encode a DGQL query as an MP formulation such that (2) the solution can be found efficiently? We answer this question positively and therefore suggest that DGQL has the potential to achieve both easy development and efficiency for decision guidance. In terms of efficiency, the overall performance of intuitive DGQL queries compares squarely with expert-generated MP problems, as will be demonstrated in the experiments that follow.

In this section we informally describe DGQL mainly borrowing from Brodsky et al. (2009). To explain the DGQL approach and its semantics intuitively, we consider a simple example of sourcing optimization, i.e., finding the best suppliers for a given demand of items to be purchased. To begin, we consider an even simpler database reporting application, in which the following are stored in the database (key fields are underlined):

*Demand* = (*Item*, *Requested*)
*Supply* = (*Vendor*, *Item*, *Price*)

*Orders* = (*Vendor*, *Item*, *Quantity*)

*Demand* specifies the quantities that need to be procured for each item, *Supply* specifies the prices vendors charge for the items they provide, and *Orders* lists the items ordered from vendors and their desired quantities. With these tables, a reporting application may use this query to calculate the total cost of all orders (assuming that all items ordered are indeed supplied):

```
create view Total_Cost as
select sum (S.Price x O.Quantity) as
Total
from Supply S, Orders O
where O.Vendor = S.Vendor and O.Item
= S.Item
```

To calculate the total quantity ordered and the total quantity requested for each item in *Demand*, the reporting application may use:

```
create view Requested_vs_Ordered as
select O.Item, sum (O.Quantity) as Or-
dered, D.Requested
from Demand D left outer join Orders O
  on D.Item = O.Item
group by O.Item, D.Requested
```

So far, the reporting application achieved its goals with pure SQL statements. Assume now that instead of having the table *Orders* stored in the database, we would like the system to compute it *optimally*. That is, the cost of the order should be minimal while the demand is satisfied (i.e., for every item, the ordered quantity is at least as the quantity requested in *Demand*). In DGQL, we reuse the existing SQL statements of the reporting application, with some additions. First, we use an **augment** clause to indicate what we would like to find in the *Orders* table:

```
create view Orders as
augment
  select S.Vendor, S.Item from Supply
with Quantity integer >= 0
```

This statement indicates that the attribute quantity is not known, but needs to be determined by the system. Second, we add the following SQL integrity constraint, which is self-explanatory:

```
constraint Order_Satisfies_Demand
on Requested_vs_Ordered
check Ordered >= Requested
```

Finally, we issue the **minimize** command:

```
minimize Total_Cost
```

Executing the program that consists of the two original SQL statements and the three additional statements results in an optimal *Orders* table: Vendors and items (extracted from the *Supply* table) are augmented with appropriate quantities so that the requested quantities are satisfied, while the total cost is minimized.

Note that the optimization here may be viewed as the "inverse" of the reporting application: The reporting application calculated the total cost for a given set of orders, whereas the optimization calculated the set of orders for minimal total cost. While in this small example, the optimal solution will designate a single vendor for each item (the vendor with the lowest cost), in general, such optimization problems are non-trivial, as they might impose various constraints (e.g., vendors may have limited quantities on hand, or they should reside in a particular vicinity).

To handle this optimization problem (i.e., find the optimal orders), state-of-the-art tools require modeling the situation in a separate decision optimization system. The DGQL approach is to extend the SQL statements of the reporting application with SQL-like statements for specifying the constraints and the objective − to automatically generate mathematical programming models that will give the optimal solution. The resulting set of statements is called a DGQL query. For the purpose of explanation, the problem we described was particularly simple. The optimization problems discussed or suggested in the following sections are considerably more complex.

## A TRANSACTION MODEL FOR VIRTUAL ENTERPRISES

The model we describe was derived from the virtual enterprise model VirtuE (D'Atri & Motro, 2008), which has been distilled to include only the concepts essential to this work.

### Basic Concepts

A virtual enterprise is a set of *members M* and a set of *products P*. Each member manufactures at least one product from the set *P*. For each product $p \in P$ that member $m \in M$ manufactures, there is at least one *production plan* $s(m, p)$, which is the set of component products that *m* must obtain from other enterprise members to manufacture *p* (i.e., the bill of materials), and for each component, the member from which it will be obtained. A null production plan (i.e., $s(m, p) = \varnothing$) indicates that *m* does not need to import any component products to manufacture *p*.

Each production plan is associated with a price and a risk. The *price* is the amount that the member requests from a client in return for this product. The *risk* is the probability that this member will fail to deliver the requested product. Note that members may charge different prices for the same product, depending on the production plan and the associated risk. For example, a particular plan that reduces risk may entail a higher price, possibly to compensate for higher costs of procurement. Hence, members can be thought as offering different *versions* of the same product. The structure of price and risk is discussed later in this section.

### Transactions

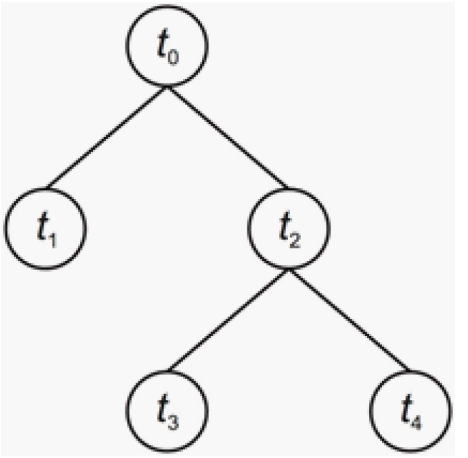
A *transaction* is a bilateral exchange between two parties in which goods are delivered in return of payment. The party initiating the transaction, requesting the goods and providing the payment is the *client*; the party responding to the transaction, providing the goods and receiving the payment is the *supplier*. Transactions are usually divided into distinct steps, and in this paper, we assume they comprise three steps: (1) *order* is the request by the client to the supplier that describes the goods needed; (2) *manufacturing* is the phase in which the supplier prepares the goods; and (3) *fulfillment* is the delivery of the goods by the supplier to the client. Thus, orders initiate transactions, and fulfillments conclude them.

In a virtual enterprise environment, transactions are the mechanism for providing target products to enterprise clients. Such transactions are termed *external* transactions. To fulfill an external transaction, the enterprise member may procure necessary products from other members of the enterprise. These exchanges are termed *internal* transactions. In an internal transaction both the client and the supplier are members of the enterprise. The supplier of an internal transaction may, in turn, initiate other internal transactions. Altogether, the execution of an external transaction is a *distributed* effort of a group of enterprise members.

Assume an external client initiates a transaction *t* that orders product *p* from member *m*. Once this member receives the order, it examines the production plan that corresponds to this offer, and then launches a set of transactions that procure the components. When these transactions have been fulfilled, *m* can manufacture the product and deliver it to its client. The satisfaction of an external transaction is therefore a recursive process. We assume that the process *terminates* with transactions that procure products whose manufacturing does not require importation.

Transactions can be illustrated with tree diagrams, in which nodes represent manufacturing of products by suppliers, and edges indicate initiation of sub-transactions. In a transaction tree, the root node models the manufacturing of the ultimate product (the product ordered by the external client), internal nodes model the manufacturing of component products, and leaf nodes model the manufacturing of import-free products. Each transaction is assumed to have a unique identifier, which is assigned to the manufacturing node. A simple transaction tree is shown Figure 1. In this example, an external client submits a transaction $t_0$ to order product

*Figure 1. A transaction tree*




$p_0$ from member $m_0$. To fulfill this order, $m_0$ submits two sub transactions: $t_1$ orders product $p_1$ from member $m_1$, and $t_2$ orders $p_2$ from $m_2$. $m_1$ fulfills $t_1$ locally; but $m_2$ submits two additional sub-transactions: $t_3$ orders $p_3$ from $m_3$, and $t_4$ orders $p_4$ from $m_4$. Both $m_3$ and $m_4$ fulfill their orders locally.

## Price

As already mentioned, each member associates a price with each of its production plans (product versions). Alternatively, this price may be associated with the transactions that order this product version. Let $t$ be a transaction that orders part $p$ from member $m$, and let $s(m, p)$ be the corresponding production plan. We denote $price(t)$ the amount charged by $m$ for the product $p$. We assume that $price(t)$ is the sum of three components: procurement cost, transaction overhead cost and manufacturing costs. Let $t_1$, . . ., $t_n$ be the sub-transactions of $t$.

1. **Procurement cost:** $price(t_1), \ldots, price(t_n)$. This is the amount paid by $m$ (as a client) to each of its suppliers to obtain the $n$ component products necessary to satisfy the transaction $t$.
2. **Transaction overhead cost:** $overhead(t_1)$, . . ., $overhead(t_n)$. This is the cost associated with the execution of each sub-transaction and borne by $m$ (the client of the sub-transactions).
3. **Manufacturing cost:** $manufacture(t)$. This is the cost of manufacturing the product from its $n$ components (it incorporates the manufacturer's profit).

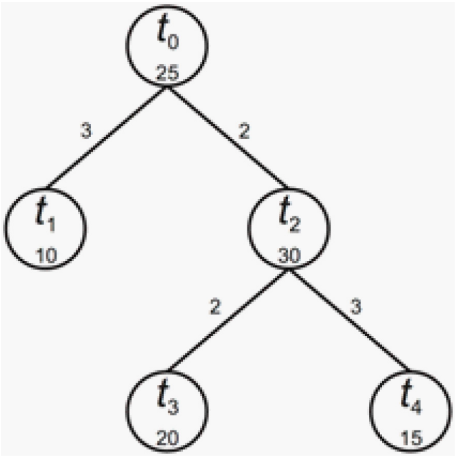
Altogether:

$$price(t) = manufacture(t) + \sum_{i=1}^{n} (price(t_i) + overhead(t_i)) \quad (1)$$

Let $t_0$ denote the root of a transaction initiated by an external client, then $price(t_0)$ is the amount the external client pays for the ultimate product. As price is always a recursive summation of manufacturing costs and transaction overhead costs, it is sufficient to store the latter two. Hence, we label each node of a transaction tree with the associated manufacturing cost, and we label each edge with the associated overhead cost.

We illustrate the concept of transaction trees and their costs with the example of Figure 1, shown in Figure 2 with example costs. In Figure 2, node labels denote the transaction identifiers and the manufacturing costs, and edge labels

*Figure 2. A transaction tree with its associated costs*




denote the transaction overhead costs. The derived transaction prices are then: $price(t_1) = 10$, $price(t_3) = 20$, $price(t_4) = 15$, $price(t_2) = 70$, and $price(t_0) = 110$.

## Risk

In addition to price, each member also associates a risk with each of its production plans (product versions). This is the probability that the supplier will fail to fulfill a request for this product version. Like price, risk may also be associated with the transactions that order this product version. Let $t$ be a transaction that orders part $p$ from member $m$, and let $s(m, p)$ be the corresponding production plan. We denote $risk(t)$ the probability that $t$ will fail; that is, the probability that $m$ will not complete the associated production plan.

$risk(t) = P(t \text{ fails}) = P(m \text{ does not complete the production plan for product } p)$

We assume that $risk(t)$ combines two different components (both can be obtained by tracking the performance of the enterprise):
**Manufacturing risk** measures the risk (the probability of failure) associated with the particular member $m$ in the particular manufacturing task:

$mrisk(m) = P(m \text{ fails})$

**Product risk** measures the risk associated with the execution of a particular production plan $s(m, p)$. We interpret this risk as the probability that at least one of the subtransactions to procure components for $p$ fails:

$prisk(p) = P(\text{at least one subtransaction to procure components for } p \text{ fails})$

With this definition, product risk increases with the complexity of the product (its number of components): If a product is changed to require an additional component, product risk will increase.

Altogether:

$risk(t) = P(\{m \text{ fails}\} \cup \{\text{at least one subtransaction to procure components for } p \text{ fails}\})$

Assume that, to procure components for $p$, $m$ initiates subtransactions $t_1, \ldots, t_n$. Then

$$\text{risk(t)} = \text{P}(\{m \text{ fails}\} \cup \{t_1 \text{ fails}\} \cup \ldots \cup \{t_n \text{ fails}\}) \quad (2)$$

For a transaction $t$ that orders an import-free product from supplier $m$ (i.e., $m$ does not

initiate any subtransactions), the risk is simply that *m* fails; that is, *mrisk*(*m*).

It is convenient to view *mrisk*(*m*) as the probability of node failure, and *risk*(*t*) as the probability of edge failure.

If we assume that the $n + 1$ events in Equation 2 are mutually exclusive (i.e., there is at most one failure), then

$$risk(t) = mrisk(m) + \sum_{i=1}^{n} risk(t_i) \quad (3)$$

In general, however, we may not make this assumption, and *risk*(*t*) must be calculated according to De Moivre' *inclusion-exclusion principle* (De Moivre, 1718):

$$P(\bigcup_{i=1}^{n} A_i) = \sum_{i=1}^{n} P(A_i) - \sum_{i<j} P(A_i \cap A_j) + \sum_{i<j<k} P(A_i \cap A_j \cap A_k) - \ldots + (-1)^{n-1} P(\bigcap_{i=1}^{n} A_i)$$

This formula expresses the probability of the union of events with *n* terms that are alternately added and subtracted. (The parity of *n* determines whether the final term is added or subtracted.) The first term totals the probabilities of the *n* events; the second term totals the probabilities of the intersections of two events; the third term totals the probabilities of the intersections of three events, and so on; the final term is the probability of the intersection of all *n* events.

Unless *n* is small, it is normally impossible to calculate the joint probabilities, and the usual approach is to provide lower and upper bounds, such as the Bonferroni inequalities (Bonferroni, 1936). In their simplest form, these inequalities provide the first term as a higher bound and the difference between the first two terms as a lower bound.

If we assume that the $n$+1 events are independent (i.e., the failure of a node and the failure of each incoming edge are unrelated), then a simplified inclusion-exclusion formula may be used that is derived from probabilities of simple events only:

$$P(\bigcup_{i=1}^{n} A_i) = \sum_{i=1}^{n} P(A_i) - \sum_{i<j} P(A_i) \cdot P(A_j) + \sum_{i<j<k} P(A_i) P(A_j) P(A_k) - \ldots + (-1)^{n-1} \prod_{i=1}^{n} P(A_i)$$

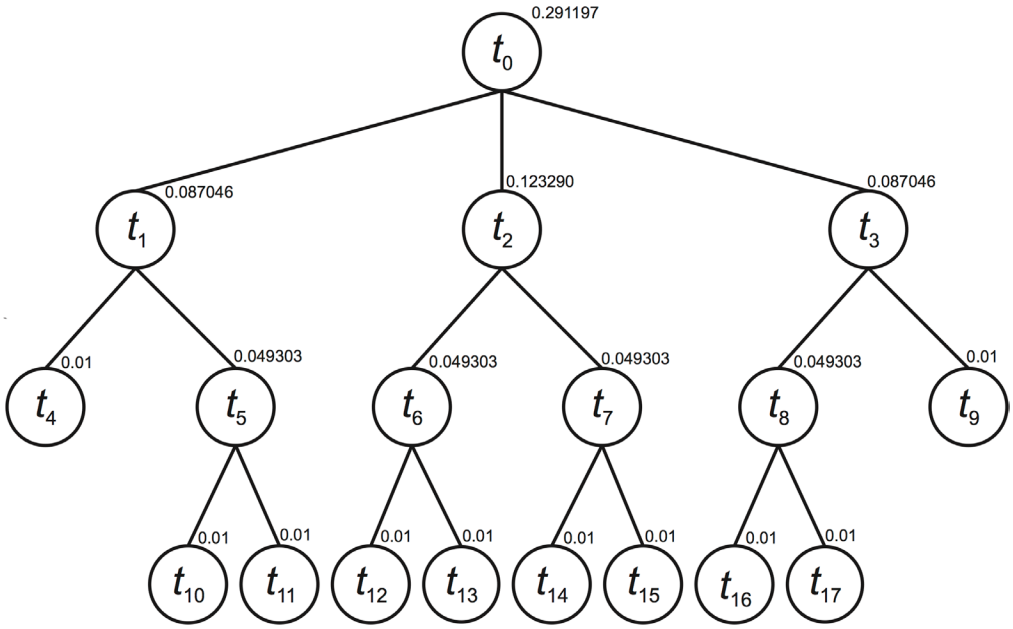
With this assumption of independence, *risk*(*t*) may be calculated from *mrisk* values rather simply, in a propagation process from the leaves of the transaction tree to its root. As an example, consider the previous transaction tree and assume that the risks of member failure are 0.01 for manufacturing of import-free products, and 0.03 for the others. (In general, it should not be that manufacturers are divided into these two classifications.) Then, in three phases of propagation (Figure 3), we derive that the risk of the external transaction is $risk(t_0) = 0.29197$. As the example demonstrates, the risk increases with the number of component products (in this example, 10), and the number of intermediate suppliers (in this example, 8).

## Optimization

The prices and risks of products bear obvious structural similarities: (1) Both are defined in recursive processes that terminate at import-free products; (2) both have components that are determined by the manufacturing member: In the case of price — transaction overhead cost and manufacturing cost, and in the case of risk — manufacturer risk; and (3) both have components that are determined by this manufacturer's suppliers: In the case of price — procurement cost, and in the case of risk — product risk.

Roughly, the behavior of each member follows this paradigm. First, the member considers which products it could offer. Next, it examines various possible bills of materials for each product. For each bill of materials, it then investigates possible suppliers. Finally, it determines which products to offer and the associated prices and risks. In making these

*Figure 3. The propagation of risk*




decisions, it considers information made available by other members regarding the products they offer, and the associated prices and risks. It then adheres to its own business goals and selects those production plans that optimize these goals.

Optimization goals may vary. We assume optimization goals that are based on prices and risks. Thus, members can choose to maximize expected revenue, or to minimize expected procurement costs, or to minimize risk. In the following we focus on the *maximization of expected profit* (an exact definition is provided later).

Thus, decisions are done ahead of orders. When receiving an order for a particular product (at a particular price and risk), the member simply executes the associated production plan.

Obviously, members must reconsider their decisions periodically, to reflect changes made by other members, or to incorporate changes in their own operation.

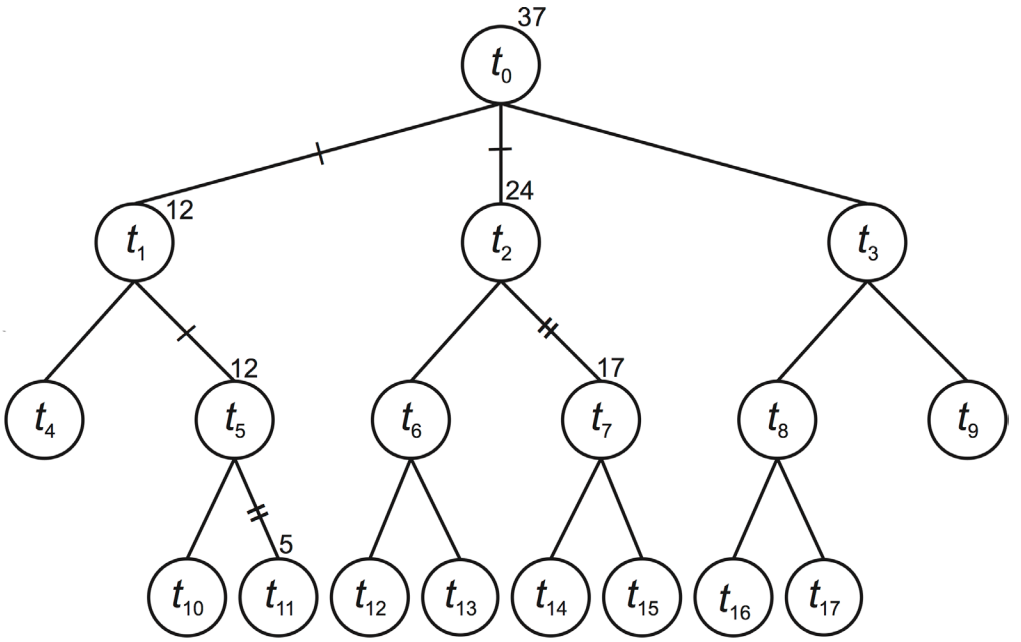
## Failure

Of the three steps of transactions, we assume that *manufacturing* is the only step prone to failure; i.e., *order* never fails, and (if manufacturing is successful) *fulfillment* never fails. Failure could be due to a variety of different reasons: a communication failure, sudden withdrawal from the enterprise, refusal to honor prior commitments, and so on. Failure can be modeled as a disconnection of an edge in the transaction tree.

To analyze the *cost* of transaction failure, we make several assumptions. First, payment is part of fulfillment. Second, all the subtransactions of a given transaction are placed simultaneously, and it is not possible to *cancel* orders that have already been placed. Finally, transactions cannot be *reversed*; i.e., it is not possible to return the goods and obtain a refund.

Under these assumptions, the client of a failed transaction does not pay for the cost of the product (as fulfillment never took place),

*Figure 4. The distribution of failure costs*




and it bears only the cost of the transaction overhead (as this cost was already invested). The client, however still pays for all the other component products that have been delivered, and for the cost of manufacturing (although manufacturing was not completed). Assuming the supplier of the failed transaction completed all its subtransactions promptly (if not, there had been earlier failure), it bears the complete cost of the product it was committed to produce.

Consider the transaction tree in Figure 4. For simplicity, assume that all manufacturing costs are 5, and all transaction overhead costs are 1. The overall price is then 107. Assume now two failures: in $t_7$ and in $t_{11}$. The former failure propagates to $t_2$, and the latter propagates to $t_5$ and $t_1$, and consequently $t_0$ fails. The external client does not pay the virtual enterprise the price of 107, and this loss is distributed as follows: $t_{11}$: 5, $t_5$: 12, $t_1$: 12, $t_7$: 17, $t_2$: 24, and $t_0$: 37. Thus, only members on a failure path bear the price of failure ($t_0$, $t_1$, $t_5$, and $t_{11}$ are on the path of the first failure, and $t_0$, $t_2$, and $t_7$ are on the path of the second failure). Essentially, the price they pay is their overhead for the subtransactions they launched, and the price of products that were delivered but proved unnecessary because the product for which they were intended was not manufactured.

**Proposition:** The losses borne by the participating members add up to the price of the ultimate product.

The proof is straightforward. The price of the ultimate product is the sum of all node costs and all edge costs, and our distribution of the cost assigned each of these costs to exactly one member. One obvious conclusion is that because the price of products increases along the supply chain, "high level" suppliers tend to pay a higher share of the loss. Although this higher risk would normally be mitigated by higher profit margins, it is important to minimize the risk of failure.

## COORDINATED AND AUTONOMOUS ENTERPRISES

So far, we outlined the basic concepts that are shared by all virtual enterprises. We now turn our attention to the specific variants discussed earlier: coordinated and autonomous virtual enterprises. The distinction between coordinated and autonomous enterprises stems from their approach to information sharing and strategic decisions (i.e., optimization).

More specifically, we assume that in coordinated enterprises

1. **Information:** Each member shares its production capabilities (products it can manufacture), the components it needs for each product, and its operational costs and risks.
2. **Optimization:** The member who receives the external order chooses the optimization target, and constructs a complete transaction tree that achieves the optimum; each participating member is thus instructed as to which production plan to use for the component that it is required to supply, and whom to order from.

whereas in autonomous enterprises

1. **Information:** The only information shared by members is the products they manufacture, and, for each product, the associated price and risk.
2. **Optimization:** Each member conducts its operation according to its own interests, choosing its individual optimization targets.

## OPTIMIZING TRANSACTIONS WITH DGQL

In both the coordinated and autonomous cases, we choose to optimize the *expected profit* from offering a product, which we define. We note that other optimization targets are possible.

Let *m* be a member who considers offering a product with the following parameters: The manufacturing cost at *m* is *p* and the manufacturing risk is *r*, and the product requires procuring *n* component products, each involving price $p_i$, risk $r_i$ and transaction cost $t_i$.

The transaction costs add up to $\sum_{i=1}^{n} t_i$ and the component costs add up to $\sum_{i=1}^{n} p_i$. After adding the local manufacturing cost (recall that it incorporates the profit), the overall price of this product is

$$price = p + \sum_{i=1}^{n} (p_i + t_i) \tag{4}$$

The procurement risk of this product is $1 - \prod_{i=1}^{n} (1 - r_i)$. After factoring-in the manufacturing risk, the overall risk of this product is

$$risk = 1 - (1 - r) \times \prod_{i=1}^{n} (1 - r_i) \tag{5}$$

Altogether, the expected revenue to *m* from this transaction is

$$expected\ \ revenue = (1 - risk) \times price \tag{6}$$

While *m* bears the entire transaction costs, it only bears the cost of components that were delivered (and the manufacturing cost). Altogether, the expected cost to *m* is

$$expected\ \ cost = p + \sum_{i=1}^{n} (t_i + (1 - r_i) \times p_i) \tag{7}$$

Finally, the expected profit to *m* is defines as

$$\textit{expected profit} = \textit{expected revenue} - \textit{expected cost} \tag{8}$$

## COORDINATED ENTERPRISES

We now describe the optimization of expected profit in coordinated enterprises. Our description refers to the SQL/DGQL program in Appendix A. In this type of enterprise, the enterprise member receiving the external order is optimizing the overall execution of the order using data provided by every enterprise member. In relational database terms, all enterprise information is stored in three tables (the corresponding program statements are numbered 1-3):

*Manufacturing*
*=(Plan, Manufacturer, Product, Mcost, Mrisk)*

*Transaction*
*=(Client, Product, Supplier, Tcost)*

*Components=(Plan, Component_Product)*

*Manufacturing* describes production plans. For each product plan, it stores the manufacturing member, the product it manufactures and the associated (add-on) manufacturing cost and risk. *Transaction* specifies the overhead of a transaction in which a client member purchases a product from a supplier member. Finally, *Component* describes production plans (bills-of-materials): The set of component parts that comprise a production plan is described in a corresponding set of rows, each associating the production plan with a product. Note that *Plan*, *Manufacturer* (or *Client* or *Supplier*), and *Product* (or *Component_Product*) are identifiers unique to the entire enterprise. The output of the process is two tables defined by augmentation. These tables would contain eventually the optimal solution (statements 4-5):

*Procurement*
*=(Plan, Component_Product, Component_Plan)*

*Catalog*
*=(Plan, Manufacturer, Product, Mcost, Mrisk, Price, Risk)*

*Procurement* is an augmentation of *Components* with the field *Component_Plan* which indicates which production plan to order when procuring a product for use in a production plan. *Catalog* is an augmentation of *Manufacturing* with two fields: *Price*, which is the overall price of the product manufactured by this production plan, and *Risk*, which is the overall risk associated with ordering the product manufactured by this production plan. These two values are outcome of the procurement decisions. A sequence of three SQL views is now created to define the objective of the optimization (statements 6-8). First,

*Procurement_Metrics*
*=(Client, Possible_Plan, Price, Risk, Tcost)*

describes the product price, product risk, and transaction overhead incurred when a client member initiates a transaction to procure a particular plan (product version) from its manufacturer. Next,

*Catalog_Metrics*
*=(Plan, Price, Computed_Price, Risk, Computed_Risk)*

describes, for each production plan (product version), its overall price and risk (corresponding to Equations 4 and 5). This view also extracts the assigned price and risk from *Catalog*, and is used to *express constraints that the assigned values must be equal to the computed values. The final view,*

*Plan_Expected_Profit*
*=(Plan, Expected_Profit)*

specifies, for a given production plan, the expected profit it generates (corresponding to Equations 6, 7 and 8). Finally, a **maximize**

statement is used to specify *Expected_Profit* as the objective of the optimization (statement 9).

## AUTONOMOUS ENTERPRISES

We now turn to the optimization of *expected profit* in autonomous enterprises. Our description refers to the SQL/DGQL Appendix B. In this type of enterprise, each enterprise member assembles its catalog of offerings as follows. For each of its product versions (production plans) it chooses the suppliers to optimize its expect profit from that production plan. It then calculates the overall price and risk and publishes them.

The information available to each member is stored in three tables as well:

*Catalog*
*=Plan, Manufacturer, Product, Price, Risk)*

*Transaction*
*=(Client, Product, Supplier, Tcost)*

*Components=(Plan, Component_Product)*

*Catalog* describes available production plans. For each production plan it stores the manufacturing member, the product it manufactures and the associated (total) price and risk. This is the only "outside" information available to each member (note the difference in the last two fields from the earlier *Manufacturing* table). *Transaction* specifies the overhead of a transaction in which a client member purchases a product from a supplier member. However, the field *Client* is always the local member. Finally, *Component* describes production plans (bills-of materials): The set of component parts that comprise a production plan is described in a corresponding set of rows, each associating the production plan with a product. Again, the plans described are only those offered by the local member. The output of the process is one table defined by *augmentation*. This table would contain eventually the optimal solution.

*Procurement*
*=(Plan, Component_Product, Component_Plan)*

*Procurement* is an augmentation of *Components* with the field *Component_Plan* which indicates which production plan to order when procuring a product for use in a production plan. A sequence of three SQL views is now created to define the objective of the optimization (statements 5-7). First,

*Procurement_Metrics*
*=(Client, Possible_Plan, Price, Risk, Tcost)*

describes the product price, product risk, and transaction overhead incurred when the local member initiates a transaction to procure a particular plan (product version) from its manufacturer. Next,

*Catalog_Metrics*
*=(Plan, Manufacturer, Price, Risk)*

describes, for each production plan (product version), its overall price and risk (corresponding to Equations 4 and 5). The final view,

*Plan_Expected_Profit*
*=(Plan, Expected_Profit)*

specifies, for a given production plan, the expected profit it generates (corresponding to Equations 6, 7 and 8). Finally, a **maximize** statement is used to specify *Expected_Profit* as the object of the optimization (statement 8).

## EXPERIMENT EVALUATION

In previous sections we described a model for transactions in virtual enterprises and its associated optimization problems, and we showed how these problems can be concisely modeled and solved using DGQL in a production database setting. We now describe a system that implements (compiles and executes) DGQL programs of the type given earlier, and provide details

on various experiments with specific virtual organization examples.

Our experimental study addresses four basic questions: (1) Does DGQL correctly solve the virtual enterprise problem as modeled, (2) how to classify the virtual enterprise optimization problem both in size and complexity, (3) is the implementation efficient with respect to the size of the input problem, and (4) are solvable virtual enterprises of sufficient size to be interesting in real world applications.

In presenting our results, we consider the problems of autonomous and coordinated enterprises separately. While any autonomous enterprise can be correctly modeled and solved as a coordinated enterprise, where each component product has only atomic plans for that product, our implementation showed that the simpler autonomous model had a significantly faster running time for the same virtual enterprise problems. This is discussed further in the upcoming sections

In both experiments the test platform used was a quad-core Xeon X3430 with 4GB DDR3 RAM running 64-bit Linux. The prototype DGQL implementation uses PostgreSQL 8.4 as the database backend and is written in Java. This setup is typical of how we expect DGQL will be used in a production environment. For these experiments, we have disabled multi-threading in our DGQL implementation to present an accurate and easily comparable performance profile of each problem. The algorithms presented use a form of stochastic search and are easily parallelized across multiple cores in a cooperative fashion.

## DGQL ALGORITHM

As described earlier, at optimization time DGQL translates a database program into an algebraic model and invokes an external solver. Matching the appropriate solver to a particular model greatly impacts the performance of DGQL. Based on problem features, DGQL can automatically detect and translate an algebraic model into the appropriate solver API.

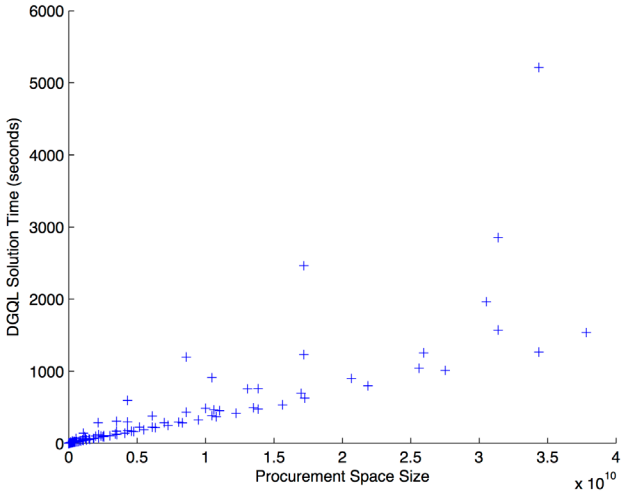
Both the autonomous and coordinated virtual enterprises are concerned with maximizing expected profit. The computations for expected profit involve computing the joint probability of an event (plan failure) and are inherently nonlinear. Additionally, the contribution of each component choice to both *expected revenue* and *expected cost* means that optimal choices for individual components cannot be made independently and later combined to give an optimal overall result. Hence these optimization problems are not decomposable. Finally, our decision variables indicate which production plan to select for each component product and are over a finite or enumerated domain.

For these reasons, the virtual enterprise problem is best suited to a constraint programming (CP) solver. In our experimentation we use ILOG CP Optimizer 2.3 as the underlying solver technology. This solver uses automatically computed heuristics to steer its search path through the solution space. This increases the likelihood that an optimal solution will be found early on. However, ultimately the entire solution space must be considered to prove that a solution is, in fact, optimal. Although this optimal solution may be found early on, our experiments will measure total solution time. Future work on DGQL will look at estimating a solver cutoff time to capture this solution without exploring the entire space.

## AUTONOMOUS ENTERPRISES

In an autonomous virtual enterprise, the component products for a particular plan are selected based only on their published price and risk. In a plan that requires n components where each component has pi alternative plans, the total number of alternative production plans is $T = \prod_{i=1}^{n} p_i$ . When each component has the same number of alternative plans $p$, this equation becomes $T = p^n$. Hence, $T$ is the number of procurement "vectors" that a member of the enterprise must consider when attempting to satisfy a production plan, and is a good measure of the complexity of the problem. To show how

*Figure 5. Solution time vs. procurement space size for autonomous enterprises*




the values of $n$ and $p$ impact the performance of DGQL, our experimentation makes use of a simplifying assumption that all component products have the same number of alternative plans.

For this experiment, we generated random problems based on two parameters, the number of component products n and the number of alternative plans for each component product p. Each of these random problems was instantiated in the database and DGQL's maximize *Expected_Profit* command invoked. The values of $n$ and $p$ were chosen with the following algorithm that gives a good sampling of reasonably sized problems:

Input for the experiment of autonomous enterprises was generated using Algorithm 1 with values $M = 50$ and $K = 500$. These experiments took about 12 hours to run on the test platform. The results can be found in Figure 5. As expected, solution time is clearly linear in $p^n$. This is because we intentionally let the solver run to optimality and this requires considering all $p^n$ production alternatives. An attractive result is that our implementation appears to scale well. Even when the number of production alternatives is in the tens of billions, it still takes only minutes to solve.

Another interesting result is that the data all fall between two linear bounds. This is due to the fact that for the same overall procurement space size $p^n$, the computation expected profit takes fewer multiplications when the number of component products $n$ is smaller than when it is larger.

Consider the sample output from the autonomous experiment in Table 1. When $n = 24$ it took nearly 4× the time to walk the solution space as when $n = 6$. We can thus conclude that solution time is linear in $p^n$ with a coefficient linear in $n$.

## COORDINATED ENTERPRISES

In a coordinated virtual enterprise, the contracting member can choose not only which production plans to use for immediate component products, but also which production plans to use to construct those component products, and so on. The idea is that by controlling the entire transaction tree that goes into a particular product plan, the member can maximize its expected profit even if it means sacrificing the expected profit of other members. Here we must

*Algorithm 1. Selecting input parameters for the "autonomous" experiment*

```
Input: Max components and plan alternatives M, desired experimental runs K
Output: List of experiment parameters (n_1, p_1), (n_2, p_2), …, (n_K, p_K)
Method:
    1:  Let L= ∅
    2:  for i  = 1 to M,  j  = 1 to M do
    3:      Add (i, j) to L
    4:  end for
    5:  Sort L = {(n_1, p_1), …} based on p_i^n_i ascending
    6:  Return first K elements of L
```

consider the depth of the production tree when computing the solution space.

Similar to the autonomous case, we make a few simplifying assumptions for the purpose of experimentation. In addition to assuming that all component products $n$ have the same number of production plans $p$, we also assume that every production plan has the same depth $d$. The solution space thus forms a balanced and complete tree and we can compute the number of alternative production plans $T$ as:

$$T = f_T(n,p,d) = \begin{cases} p^d & when\ n = 1 \\ p^{n\frac{n^d-1}{n-1}} & when\ n > 1 \end{cases} \quad (9)$$

Clearly $f_T$ grows fast with $d$. Despite this fact, using these simplifying assumptions is convenient for our experimentation as it allows us to use a similar problem generation technique as the autonomous case. In the coordinated experiment, we generated problems based on three parameters, the number of component products $n$, the number of alternative plans $p$ and the production plan depth $d$. These random problems were inserted into the database and the solution time from invoking **maximize** *Expected_Profit* was measured. The values of $n$, $p$ and d were chosen with Algorithm 2 to give a reasonable sampling of coordinated virtual enterprise problems.

Input for experiment of coordinated enterprises where generated using Algorithm 2 with values $M$ = 15, $D$ = 5 and $K$ = 350. These experiments took about 16 hours to run on the test platform. The results can be found in Figure 6. Similar to our experience with the autonomous problem, we expected solution time to be linear in size of the procurement space. Again, this is due to the fact that we let the underlying CP solver run to optimality. Unlike the results from the autonomous experiment, the coordinated data suggests that there is a much tighter linear range that the data fall into. While not demonstrated by this data, we would expect the variation in linear bounds on solution time is due to the coefficient being a function of $n$ and $d$.

Another interesting result is that the DGQL formulation of coordinated enterprises is noticeably more expensive than the formulation of autonomous enterprises of $d$ = 1. Figure 7 shows both sets of data on the same scale. Initial investigation suggests this is due to the extra decision variables and constraints introduced

*Table 1. Comparison of solution time for experimental runs with the same procurement space size*

| Components $n$ | Plans $p$ | Space size $p^n$ | Solution time (seconds) |
|---|---|---|---|
| 6 | 16 | 16777216 | 0.567 |
| 8 | 8 | 16777216 | 0.701 |
| 12 | 4 | 16777216 | 0.941 |
| 24 | 2 | 16777216 | 1.858 |

*Algorithm 2. Selecting input parameters for the "coordinated" experiment*

```
Input: Max components and plan alternatives M, max depth D, experimental runs
K
Output: List of experiment parameters (n_1, p_1,d_1), …, (n_K, p_K,d_K)
Method:
    1:  Let L= ∅
    2:  for i  = 1 to M,  j  = 1 to M, k = 1 to D do
    3:      Add (i, j,k) to L
    4:  end for
    5:  Sort L = {(n_1, p_1,d_1), …} based on f_T(n_i,p_i,d_i) ascending
    6:  Return first K elements of L
```

by the coordinated formulation to restrict the values of *Price* and *Risk*. Future work with DGQL will consider query optimization and rewriting using an optimization algebra similar to the relational algebra used in SQL query analysis. This will allow redundant algebraic information to be removed before translating the problem into a solver specific model.

## CONCLUSION

We described a simple yet powerful cost model for transactions in virtual enterprises. Simplicity is maintained with several assumptions, including: (1) Transactions involve only three phases: ordering, manufacturing and fulfillment; (2) all subtransactions are placed simultaneously (e.g., it is not possible to choose the second subtransaction after the first has been completed); (3) orders may not be cancelled or reversed (no refunds); (4) transaction cost is the sum of external procurement, transaction overhead, and internal manufacturing; (5) transaction risk is the combination of manufacturer risk and product risk, where the latter is the risk in procuring the product components; and (6) the failure of a node (a manufacturer) and any of its incoming edges (the subtransactions it issued) are independent events, allowing for simple calculation of risk based on probabilities of individual events only. With the exception of (6), the other assumptions can be relaxed at the cost of a more complex model, but DGQL should be able to manage this additional complexity. While the assumption on the independence of failures may sometimes be invalid, we maintain that the calculated risks and the outcome of the optimization would still be highly beneficial. Yet, the power of this simple model is in its ability to represent appropriately many real-world situations and perform several optimizations, including: (1) Virtual enterprises of different types of autonomy; (2) the freedom to order parts from different sources, and to assemble parts according to different schemes; (3) the just distribution of the cost of a failed transaction among the suppliers on the failure path, with higher costs being borne by suppliers higher on the supply chain; (4) the positive correlation between product complexity and product risk; (5) the propagation of product risks upward the supply chain; and (6) the optimization of individual member operations to reduce costs, risks, or expected losses.

We then focused on this final feature of optimizability and we presented DGQL, a language for expressing and executing optimization problems in a database setting. For enterprises already invested in database technology (software and programmers), DGQL offers a convenient and efficient method to extend that technology to solve optimization problems. While the DGQL code we presented may appear to be non-trivial, it must be stressed that to experienced SQL programmers, acquiring proficiency in DGQL requires only moderate effort (considerably less than mastering the techniques of mathematical programming and the requisite new software tools).

Our experiment has demonstrated the viability of DGQL as an optimization tool. It is numerically stable and correctly solves problems of the autonomous and coordinated

*Figure 6. Solution time vs. procurement space size for coordinated enterprises*

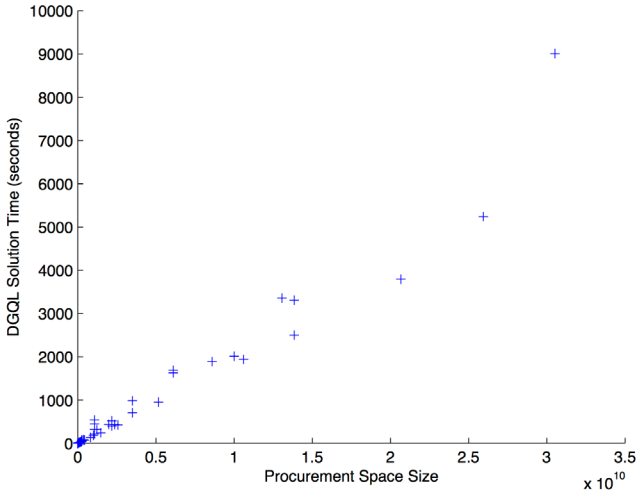
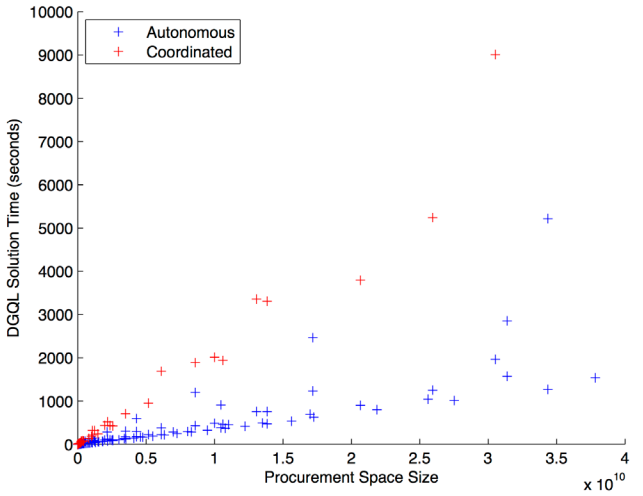



*Figure 7. Solution time of autonomous enterprises vs. coordinated enterprises of depth 1*




virtual enterprise. For both types of enterprise it parameterized the input size in terms of the number of component products $n$, plan alternatives $p$, and for coordinated enterprises, the depth of the production tree $d$, and these parameters were used to measure the problem complexity. Our implementation was shown to be efficient with respect to its complexity, with running time that is linear in the number of procurement alternatives.

Our work continues, with future work related to the transaction model includes: (1) Relax some the aforementioned model restrictions; (2) add delivery time as a third major

parameter (paralleling *risk* and *price*). (3) allow members to maintain stock and thus benefit from economics of scale (for example, transaction overhead would be paid less frequently); and (4) allow members to place redundant orders to further improve some variables (e.g., reduce risk). With respect to optimization we plan to focus on two issues. (1) How to use heuristics to discover good approximate solutions that do not require examining the entire set of procurement alternatives. One promising approach involves a combination of constraint programming techniques with an estimated solver cutoff time. (2) As our experiment showed, there is room for improvement in the preprocessing phase before DGQL is translated into the solver model. This will involve query optimization and rewriting and hopefully should close the performance gap between the autonomous case and the coordinated case with depth 1.

## REFERENCES


Barbini, F. M., & D'Atri, A. (2005). How innovative are virtual enterprises? In *Proceedings of the 13th European Conference on Information Systems* (pp. 1091-1102).

Boisvert, R. F., Howe, S. E., & Kahaner, D. K. (1985). GAMS: A framework for the management of scientific software. *ACM Transactions on Mathematical Software*, *11*(4), 313–355.

Bonferroni, C. E. (1936). Teoria statistica delle classi e calcolo delle probabilità. *Istituto Superiore di Scienze Economiche e Commerciali di Firenze*, *8*, 1–62.

Brodsky, A., Bhot, M. M., Chandrashekar, M., Egge, N. E., & Wang, X. S. (2009). A decisions query language (DQL): High-level abstraction for mathematical programming over databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 1059-1062).

Brodsky, A., & Nash, H. (2006). CoJava: Optimization modeling by nondeterministic simulation. In F. Benhamou (Eds.), *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming* (LNCS 4204, pp. 91-106).

Brusco, S. (1992). The idea of industrial districts: Its genesis. In *Industrial districts and inter-firm cooperation in Italy* (pp. 10–19). Geneva, Switzerland: International Institute of Labour Studies.

Camarinha-Matos, L. M. (2003). New collaborative organizations and their research needs. In *Proceedings of the Fourth Working Conference on Virtual Enterprises: Processes and Foundations for Virtual Organizations* (pp. 3-14).

Camarinha-Matos, L. M., & Afsarmanesh, H. (2004). Elements of base VE infrastructure. *Journal of Computers in Industry*, *51*(2), 139–163.

D'Atri, A., & Motro, A. (2008). VirtuE: A formal model of virtual enterprises for information markets. *Journal of Intelligent Information Systems*, *30*(1), 33–53.

D'Atri, A., & Motro, A. (2010). Virtual enterprise transactions: A cost model. In D'Atri, A., & Saccà, D. (Eds.), *Information systems: People, organizations, institutions, and technologies* (pp. 165–174). Berlin, Germany: Physica-Verlag.

Davidow, W. H., & Malone, M. S. (1992). *The virtual corporation: Structuring and revitalizing the corporation for the 21st century*. New York, NY: HarperCollins.

De Moivre, A. (1718). *Doctrine of chances - A method for calculating the probabilities of events in plays*. London, UK: Pearson.

Elmagramid, A. K. (1991). *Database transaction models for advanced applications*. San Francisco, CA: Morgan Kaufmann.

Fourer, R., Gay, D. M., & Kernighan, B. W. (2002). *AMPL: A modeling language for mathematical programming* (2nd ed.). Pacific Grove, CA: Brooks/Cole.

Fritzson, P., & Engelson, V. (1998). Modelica - A unified object-oriented language for system modelling and simulation. In *Proceedings of the 12th European Conference on Object-Oriented Programming* (pp. 67-90).

Goldman, S. L., Nagel, R. N., & Preiss, K. (1995). *Agile competitors and virtual organizations: Strategies for enriching the customer*. New York, NY: Van Nostrand.

Grefen, P. (2002). Transactional workflows or workflow transactions? In A. Hameurlain, R. Cicchetti, &_R. Traunmüller (Eds.), *Proceedings of the 13th International Conference on Database and Expert Systems Applications* (LNCS 2453, pp. 60-69).

Ilyas, I. F., Beskales, G., & Soliman, M. A. (2008). A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, *40*(4), 11.

Monge, P., & DeSanctis, G. (Eds.). (1999). Special issue on virtual organizations. *Organization Science*, *10*(6).

Mowshowitz, A. (Ed.). (1997). Special section on virtual organizations. *Communications of the ACM*, *40*(9), 30–64.

Smith, G. E., Venkatraman, M. P., & Dholakia, R. R. (1999). Diagnosing the search cost effect: Waiting time and the moderating impact of prior category knowledge. *Journal of Economic Psychology*, *20*, 285–314.

*Amihai Motro is a Professor of Computer Science at George Mason University. He holds a BSc degree in mathematics from Tel Aviv University, an MSc degree in computer science from the Hebrew University of Jerusalem, and a PhD degree in computer and information science from the University of Pennsylvania. He was previously on the faculty of the Computer Science Department at the University of Southern California. Dr. Motro's main areas of research are database management, information systems, information retrieval, and collaborative enterprises; specializing in subjects such as intelligent integration of information, cooperative user interfaces, information uncertainty, information quality and integrity, and service composition models. In these areas, Dr. Motro has designed and implemented several innovative systems, including Multiplex, Fusionplex, Autoplex and Retroplex, for integrating multiple, heterogeneous and inconsistent information sources; and Vague, Flex, Baroque and ViewFinder which implemented novel methods for accessing information in databases. Dr. Motro has published over 100 research papers in professional journals and international conferences, and has edited several books and journal issues. He organized several conferences and workshops, has been on the editorial board of the* Journal of Intelligent Information Systems *and the* Journal of Data Mining and Knowledge Discovery*, and was the principal investigator on research projects funded by agencies such as the National Science Foundation, DARPA, and AT&T.*

*Alexander Brodsky is Associate Professor of Computer Science at George Mason University. His current research interests include decision-guidance and support systems, decision optimization, and their applications to energy and smart power grids. Dr. Brodsky has over 20 years of experience in leading R&D projects in private industry, defense, and academia. For his research work on Constraint Databases and Programming, Dr. Brodsky received a National Science Foundation (NSF) CAREER Award, NSF Research Initiation Award, and grants from the Office of Naval Research and NASA. He has authored or co-authored over seventy scholarly peer-reviewed journal and conference papers and co-edited a LNCS volume on constraint databases and programming. Dr. Brodsky recently served as a program committee co-chair of the IEEE ICDE (International Conference on Data Engineering) workshop on Data-Driven Decision Guidance and Support Systems (DGSS 2012), and as a general vice co-chair of ICDE 2012. In the past, he served as conference chairman of the fifth International Conference on Principles and Practice of Constraint Programming, and PC co-chairman and/or organizer of a CP workshop on Constraints and Databases, of CDB98, and of an ILPS workshop on Constraints, Databases and Logic Programming. Prior to George Mason University, Dr. Brodsky worked at IBM's T.J. Watson Research Center, at Israel Aircraft Industries and was an R&D officer in the Computer Division of Communications, Electronics and Computer Corps, Israel Defense Forces. He earned his PhD and prior degrees in Computer Science and/or Mathematics from the Hebrew University of Jerusalem.*

*Nathan Egge is a PhD candidate at George Mason University working on a dissertation on the decision-guidance query language. He holds dual master's degrees in mathematics and computer science from Virginia Tech. His research interests include decision-guidance and support systems, database management systems, constraint and mathematical programming, distributed computing, and enterprise software architecture. He has over 10 years experience in the industry, designing and building enterprise applications for Fortune 500/Global 1000 companies.*

*Alessandro D'Atri was a Professor of Business Organization and the founder and director of the Center for Research on Information Systems (CeRSI) at LUISS "Guido Carli" University in Rome. His doctoral degree and his post-doctoral specialization were, respectively, in Electronic Engineering and in Computer and Control Engineering, both from "La Sapienza" University in Rome. His previous academic positions include an Associate Professor in the Faculty of Engineering at "La Sapienza" University, and a Professor in Computer Engineering and Dean of the School of Electronics at the Faculty of Engineering of L'Aquila University. Professor D'Atri was the founder and president of the Italian Association for Information Systems (the Italian chapter of the Association of Information Systems), and a scientific consultant to the European Commission and to the Italian Ministry of Research. His research and teaching career spanned a large number of different areas, including graph theory, computational complexity, database theory, cooperative user interfaces to databases, human-computer interaction, information systems, medical informatics, multi-media systems, telematics, e-commerce, e-learning, e-government, and virtual enterprises. Professor D'Atri published more than 150 scholarly articles in journals and conferences, he led and participated in over thirty sponsored research projects, and was involved in the organization of over fifty conferences and workshops. Professor D'Atri passed away in April 2011 at the age of 60.*

# APPENDIX A

## DGQL Optimization for Coordinated Enterprises

```
1.
create table Manufacturing (
    Plan integer,
    Manufacturer integer,
    Product integer,
    Mcost numeric,
    Mrisk numeric,
    primary key (Plan));
2.
create table Transaction (
    Client integer,
    Product integer,
    Supplier integer,
    Tcost numeric,
    primary key (Client, Product, Supplier));
3.
create table Components (
Plan integer,
Component_Product integer,
primary key (Plan, Component_Product));
4.
create view Procurement as
augment Components C with tuple in
    select M.Plan as Component_Plan
    from Manufacturing M
    where M.Product = C.Component Product;
5.
create view Catalog as
    augment Manufacturing with
    Price numeric,
    Cost numeric;
6.
create view Procurement_Metrics as
    select T.Client, C.Plan as Possible_Plan, C.Price, C.Risk, T.Tcost
    from Transaction T, Catalog C
    where C.Product = T.Product and C.Manufacturer = T.Supplier;
7.
create view Catalog_Metrics as
    select C.Plan,
        C.Price,
        C.Mcost + sum(M.Price + M.Tcost) as Computed_Price,
        C.Risk,
        1 - (1 - C.Mrisk) * prod(1 - M.Risk) as Computed_Risk
    from Catalog C
    left outer join (
        select *
        from Procurement P, Procurement_Metrics PM
        where PM.Possible_Plan = P.Component_Plan
   ) as T
    on T.Client = C.Manufacturer and T.Plan = C.Plan
    group by C.Plan, C.Price, C.Mcost, C.Risk, C.Mrisk
    check Price = Computed_Price
    check Risk = Computed_Risk;
```

```
8.
create view Plan_Expected_Profit as
    select C.Plan,
        (1 - C.Risk) * C.Price              /* expected revenue */
        - (C.Mcost + sum(PM.Tcost))         /* manufacturing and overhead
costs paid*/
        - sum((1 - PM.Risk) * PM.Price)      /* expected components cost
        */
    as Expected_Profit
    from Catalog C
        left outer join (
        select *
        from Procurement P, Procurement_Metrics PM
    where PM.Possible_Plan = P.Component_Plan
   ) as T
    on T.Client = C.Manufacturer and T.Plan = C.Plan
    where C.Plan=${Plan};
        /* substitute ${Plan} with the production plan specified in the exter-
nal transaction
         */
9.
maximize Plan_Expected_Profit.Expected_Profit;
```

# APPENDIX B

## DGQL Optimization for Autonomous Enterprises

```
1.
    create table Catalog (
    Plan integer,
    Manufacturer integer,
    Product integer,
    Price numeric,
    Risk numeric,
    primary key (Plan));
2.
create table Transaction (
    Client integer,             /* Client is always the local member */
    Product integer,
    Supplier integer,
    Tcost numeric,
    primary key (Client, Product, Supplier));
3.
create table Components (
Plan integer,                  /* Plan is a production plan of the local member
*/
    Component_Product integer,
    primary key (Plan, Component_Product));
4.
create view Procurement as
augment Components C with tuple in
    select Plan as Component_Plan
    from Catalog CT
    where CT.Product = C.Component_Product;
5.
create view Procurement_Metrics as
    select T.Client, M.Plan as Possible_Plan, M.Price, M.Risk, T.Tcost
    from Transaction T, Catalog C
```

```
    where C.Product = T.Product and C.Manufacturer = T.Supplier;
6.
create view Catalog_Metrics as
    select C.Plan, C.Manufacturer,
        ${Mcost}+ sum(PM.Price + PM.Tcost) as Price
            /* substitute ${Mcost} with the local manufacturing cost */
        1 - (1- ${Mrisk}) * prod(1 - PM.Risk) as Risk
            /* substitute ${Mrisk} with the local manufacturing risk */
    from Catalog C
    left outer join (
        select *
        from Procurement P, Procurement_Metrics PM
        where PM.Possible_Plan = P.Component_Plan
   ) as T
    on T.Client = C.Manufacturer and T.Plan = C.Plan
    group by C.Plan, C.Manufacturer;
7.
create view Plan_Expected_Profit as
    select CM.Plan,
        (1 - CM.Risk) * CM.Price              /* expected revenue */
        - (CM.Price + sum(PM.Tcost))         /* manufacturing and overhead
costs paid */
        - sum((1 - CM.Risk) * CM.Price)     /* expected components cost*/
    as Expected_Profit
    from Catalog_Metrics CM
    left outer join (
        select *
        from Procurement P, Procurement_Metrics PM
        where PM.Possible_Plan = P.Component_Plan
   ) as T
    on T.Client = CM.Manufacturer and T.Plan = CM.Plan
    where CM.Plan=${Plan};
            /* substitute ${Plan} with the production plan being optimized by
the local
             member */
8.
maximize Plan_Expected_Profit.Expected_Profit;
```