Proceedings of the 2008 Winter Simulation Conference
S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, J. W. Fowler eds.

UNIFYING SIMULATION AND OPTIMIZATION OF STRATEGIC SOURCING AND TRANSPORTATION

Malak Al-Nory
Alexander Brodsky

Dept. of Computer Science
George Mason University
4400 University Drive, USA

ABSTRACT

Proposed and developed is a framework and an extensible library of simulation modeling components for strategic sourcing and transportation. The components include items, suppliers, volume-discount schedules, aggregators, procurement rules, and less-than-truck-load delivery. Service models are classes in the Java programming language extended with decision variables, assertions, and business objective constructs. The optimization semantics of the framework is based on finding an instantiation of real values into the decision variables in the service object constructor, that satisfies all the assertions and leads to the optimal business objective. The optimization is not done by repeated simulation runs, but rather by automatic compilation of the simulation model in Java into a mathematical programming model in AMPL and solving it using an external solver.

1 INTRODUCTION

To ensure the procurement function is aligned with the organization's long-term objectives such as savings and profitability, many organizations have transitioned to strategic sourcing to acquire their raw materials and service requirements (Carr and Smeltzer 1999). Strategic sourcing should be a systematic and comprehensive process to determine the procurement plan that minimizes linked costs in the supply chain, and maximizes the value of purchased goods and services. This is done by determining the best suppliers for needed goods or services and the conditions under which to use their services in a way that achieves the best value and contributes to the organization's long-term objectives (Anderson and Katz 1998). To fully achieve the alignment of the procurement process with the organizational objectives, transportation should be an essential element in the strategic sourcing plan. The reason is that strategic sourcing and transportation are interrelated processes and one of them can not be optimized in isolation of the other. For example, if the delivery cost of the purchased items was not incorporated in the model, these costs can tremendously increase the overall costs. However, the full integration of strategic sourcing and transportation is very complex and challenging in practice, and therefore organizations have increasingly used different methodologies to solve this problem.

The most widely used tools apply simulation and/or optimization techniques to support strategic sourcing and transportation decisions (Padmos et al. 1999; Phelps, Parsons, and Siprelle 2000; Terzi and Cavalieri 2004). Analytical optimization models are challenging and require high level of expertise in Operations Research (OR) techniques and mathematical modeling languages such as AMPL (Fourer, Gay, and Kernighan 2003) and GAMS (Boisvert, Howe, and Kahaner 1985). The elements of an OR model are abstract constraints, which have only an indirect connection to elements of a real-world process. Also, the notions of order and timing of events are usually not explicit in OR models, which puts additional burden on the modeler. Even when the OR expertise is available, classical Linear Programming (LP) and Mixed Integer Linear Programming (MILP) do not provide means for modeling complex relations of highly complex problems such as strategic sourcing and transportation. The problems are usually simplified to a great extent to be modeled mathematically and solved by a mathematical programming solver.

Simulations such as discrete-event simulation and Monte Carlo simulation are more realistic and provide means for incorporating complex model behavior using easier model development methodologies and tools (Law 2007). The elements of the simulation model are state-variables and state-transitions, which have clear one-to-one correspondence with elements of a real-world process. Also, real-world time and sequence of events corresponds to time and sequence in the running simulation in an obvious way.

Combinations of simulation methods have been also widely used to develop simulation environments that can get the most out of simulation. For example, combining object-oriented simulation with discrete-event simulation as in Jwrap (Bizaro and Silva 1998) or with processoriented simulation as in Silk (Healy and Kilgore 1998). Simulation modeler can practice modern object-oriented software engineering to build arbitrarily complex modular models from simple building blocks.

While simulation offers numerous advantages in ease of modeling, testing and extensibility, it is optimized by choosing parameters manually. An optimization layer can be added by running a simulation multiple times with possible heuristics. The improving strategies for the simulation objectives are mainly a trial-and-error procedure (Kelton, Sadowski, and Sturrock 2004). Thus, simulations lack systematic optimization.

In the last decade, there has been work on combining simulation and optimization by adding an optimization model on top of a simulation model. See (Swisher and Hyden 2000; Fu 2001) for excellent simulation optimization surveys. The optimization model is used to optimize a set of user-selected system parameters with respect to some performance measures of the simulation model. The user does not know that an optimum has been reached (Fu 2002). Thus, the optimization uses the simulation as a black-box and the parameters of the actual problem are not used directly in the optimization strategy. Furthermore, computational cost for the simulation-based approaches is still very expensive, in spite of the advancement in the computing power, which makes classical search procedures inefficient. Advanced techniques such as OCBA by (Chen et al. 2000) used for allocating simulation budget are required to enhance the efficiency of the simulation for optimization and to determine the number of runs required.

When reviewing the literature we found limited number of papers that deal with optimization by simulation in the context of strategic sourcing and/or transportation in supply chains. The paper of (Lee and H.Kim 2002) combined simulation and optimization for a production-distribution system. The iterative hybrid analytic-simulation procedure uses simulation result to adjust the parameters of the optimization model. The goal is to produce a solution representing a more realistic optimization model which is also within the constraints of the stochastic simulation model. The work of (Truong and Azadivar 2003) combines simulation, Mixed Integer Programming (MIP) and Genetic Algorithm (GA) to solve supply chain configuration problem such as facility location and partner selection. MIP and GA are used to optimize qualitative and quantitative variables respectively. Simulation is used to evaluate performance of each configuration design. (Almeder and Preusser 2007) developed a framework for network flow in the supply chains that embeds abstract deterministic LP or MILP model within a complex discrete-event simulation model to improve the overall performance. Their hybrid methodology uses loops between the simulation and the optimization until the model reaches convergence to a table solution based on adapting decision rules. The approach requires modeling the supply chain as a discrete-event simulation as well as an optimization model. In addition, the decision rules of the simulation model might change and might necessitates a recalculation of the parameters for the optimization model. This might result in a situation where convergence is not possible. In general, most of the work in the literature require developing multiple models, each of which optimizes some of the problem parameters. There has been work on algorithms for clearance in combinatorial reverse auctions, e.g., (Sandholm and Suri 2001; Sandholm et al. 2001; Sandholm and Suri 2003) and their extensions. However, the optimization problem is formulated as extended reverse auction and do not provide the level of flexibility of simulation-based models.

In our previous work (Brodsky, Al-Nory and Nash 2008) we introduced the Service Composition (SC) CoJava language to unify simulation and optimization of supply chains, i.e., SC-CoJava is used to specify a simulation model, yet its complier automatically constructs a mathematical programming model and solves it using a mathematical programming solver. SC-CoJava also provides a Service Composition framework that allows to specify both atomic and composite services, which form a simulation model. However, SC-CoJava only provided a general framework, but did not focus on modeling specific components in a supply chain.

This paper focuses on modeling Strategic Sourcing and Transportation services in the Service Composition framework, and on simplification of SC-CoJava syntax and its optimization semantics. More specifically, the contributions of this paper are as follows.

First, we developed an extensible modular library of strategic sourcing and transportation modeling components, including items, services, business metrics, and procurement rules. The service models include (a) different types of suppliers and volume-discount schedules; (b) supplier aggregators with procurement rules such as the amount/percentage of awarded business; and (c) less-than-truck-load transportation. This library within the Service Composition framework allows quick construction of simulation models for composite services involving sourcing and transportation.

Second, we simplified the syntax and suggested a conceptually easier, yet equivalent, optimization semantics of SC-CoJava, which is being used with the sourcing and transportation library. A service is modeled as a Java class, in which the data part captures all the relevant service information, and constructors compute business metrics associated with a service instance (business transaction). However, a service class constructor may involve (a) *decision-choice* constructs for one or more program variables, (b) *assert* statements with Boolean conditions, and (c) a single variable *objective* in the service class, as well as the *min/maxFlag,* to indicate whether the objective is to be minimized or maximized. In addition to the procedural "simulation" semantics of Java, we also provide an optimization semantics, which is based on the notion of an *optimal* service object. Namely, it is an object of the service class, constructed so that (a) *decision-choice* variables are instantiated with real numbers, (b) all *assert* conditions are satisfied, and (c) the value of the variable *objective* in the service class is minimal/maximal among all service objects that satisfy (a) and (b). The optimization semantics of a service constructor is the one of the *optimal* service object; the rest is executed as a regular Java program. The *optimal* service object computation is done by automatic construction of a mathematical programming model (in AMPL) and solving it using a mathematical programming solver. Note that SC-CoJava does not optimize using multiple simulation runs, but rather using a mathematical programming solver. Also, important to note that while this framework constructs a mathematical programming model, very similar techniques can be used to construct reverse auctions and thus would utilize specialized optimization algorithms such as in (Sandholm and Suri 2001; Sandholm et al. 2001; Sandholm and Suri 2003).

Third, we developed a case study of the use of the strategic sourcing and transportation services library and SC-CoJava.

This paper is organized as follows. Section 2 explains the strategic sourcing and transportation problem. Section 3 describes the simulation semantics of the proposed framework and its individual components. Section 4 describes the optimization semantics. Section 5 exemplifies the use of the framework through a case study. Section 6 describes the implementation. Section 7 concludes the paper and briefly outlines directions for future work.

2 STRATEGIC SOURCING AND TRANSPORTATION PROBLEM

To understand the problem, consider an example of a Strategic Sourcing and Transportation (SST) service. This service is responsible of purchasing three different types of products (i.e., food, water, and medicine) and deliver these products to customers in two different locations. The SST service consists of two sub-services; a Supply service to purchase the products and deliver them to a number of stocking locations, and a Transportation service to deliver these products from the stocking locations to the customers. See Figure 1.

In turn, the supply service may involve multiple suppliers of different types and cost functions (e.g., fixed price supplier and volume-discount supplier) Similarly, there might be multiple carriers (e.g., full-truck-load or less-than-truck-load). In this specific case, we consider a less-than-truck-load (LTL) transportation. In LTL mode, only a fraction of the truck capacity is hired and the cost is proportional to the transported amount with specific fees depending on weight ranges and the destination zone (Kuo and Soflarsky 2003; Caputo, Fratocchi, and Pelagagge 2006).

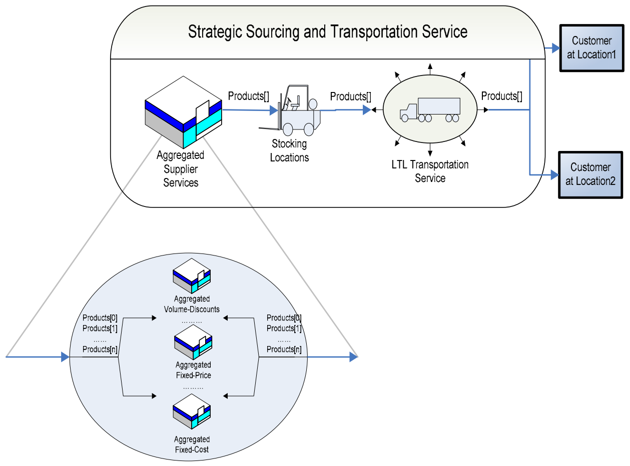


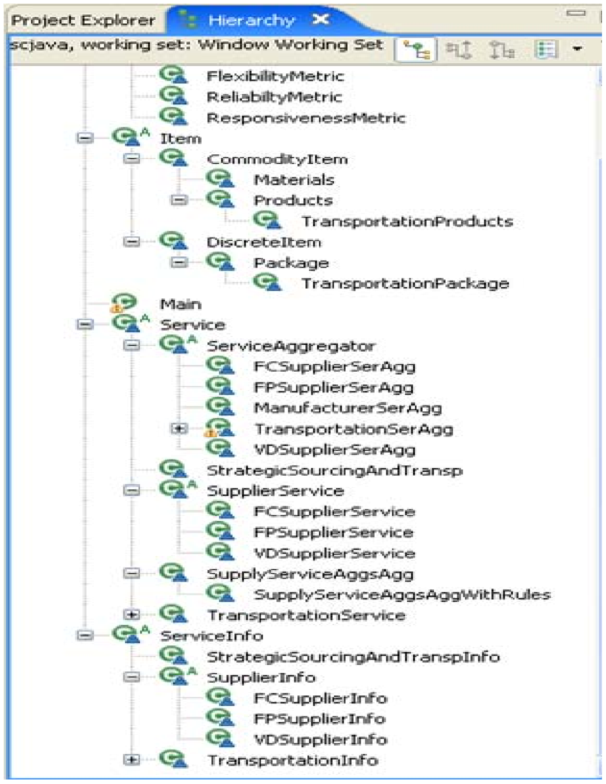
Figure 1: Strategic Sourcing and Transportation

A typical decision required may be how to purchase and deliver the products to the customers' locations in the shortest amount of time subject to available resources, or for the minimal total cost within a fixed amount of time. The outcome of such strategic decision is an actionable recommendation involving quantitative and qualitative variables. Specifically, what products, in what quantities should be delivered by which transportation carriers, to which locations, and which products should be purchased, in what quantities from which suppliers, and which stocking locations should be used, for which products, in what quantities.

3 SIMULATION SEMANTICS

3.1 Conceptual Service Composition Framework

Figure 2 shows a partially expanded library of strategic sourcing and transportation components that adhere to the Service Composition (SC) CoJava framework proposed in (Brodsky, Al-Nory, and Nash 2008). All strategic sourcing and transportation components are represented in the framework as subclasses of the corresponding abstract classes: `Item`, `Service`, `ServiceInfo`, and `BusMetric`. The most important concept is that of a Service, to represent services such as Supply and Transportation. Conceptually, a service represents a transformation of incoming Items to outgoing Items. For example, a Transportation service transforms Items of type Transportation Package or Transportation Product to Items of the same type (with an instance indicating a different location). Some services have only incoming, but no outgoing Items, or the other way around. For example, a Supplier service, has only outgoing Items of type Products, whereas a Demand service has only incoming Items.



```
Project Explorer   Hierarchy
scjava, working set: Window Working Set
        FlexibilityMetric
        ReliabiltyMetric
        ResponsivenessMetric
  Item
        CommodityItem
              Materials
              Products
                    TransportationProducts
        DiscreteItem
              Package
                    TransportationPackage
  Main
  Service
        ServiceAggregator
              FCSupplierSerAgg
              FPSupplierSerAgg
              ManufacturerSerAgg
              TransportationSerAgg
              VDSupplierSerAgg
        StrategicSourcingAndTransp
        SupplierService
              FCSupplierService
              FPSupplierService
              VDSupplierService
        SupplyServiceAggsAgg
              SupplyServiceAggsAggWithRules
        TransportationService
  ServiceInfo
        StrategicSourcingAndTranspInfo
        SupplierInfo
              FCSupplierInfo
              FPSupplierInfo
              VDSupplierInfo
        TransportationInfo
```

Figure 2: Strategic Sourcing and Transportation Modeling Components Library

Incoming and outgoing Items used in Services are characterized by multiple attributes such as quantity and location, which differ in Items of different types. Services are also associated with one or more Business Metrics, such as Reliability, Responsiveness, Flexibility, Assets, and Cost. Also, each Service has an associated Service Information. While Service instance represents a specific dynamic transaction (transformation), its corresponding Service Info instance represents more static parameters. For example, a Supplier Service Info may hold a price list of Items supplied by the Supplier Service, as well as volume-discount and their steps. This data is being used, for example, to compute the Business Metric Cost for a specific set of Items (and their quantities) supplied by a Supplier Service.

Services may be composed of other, more basic services. A composite service transforms its incoming items (if applicable) to outgoing items (if applicable) as if it was an atomic service. All of the operations of determining the items' quantities and the partners selection are encapsulated in the composite service. For example, our SST service is a composite service that is composed of two other sub-services: Aggregated Supplier service, and an atomic Transportation service. In turn, the Aggregated Supplier service is composed of multiple levels of aggregated Supplier services of different types of suppliers.

3.2 Strategic Sourcing Service

The abstract class `SupplierService` extends `Service` class with three fields and an abstract method `supplyingCost()` which must be overridden by the concrete subclasses that extend the `SupplierService` class.

```
abstract class SupplierService extends
Service{
  SupplierInfo supplierInfo;
  Products[] outProducts;
  CostMetric costMetric;
  abstract protected double supplyingCost();
}
```

In the Strategic Sourcing & Transportation framework, `SupplierService` is extended by three other types of suppliers differentiated from each other by their cost functions. `FCSupplierService` class represents suppliers with a fixed cost price structure. This model might be more suitable for trading services rather than goods. The `supplyingCost()` method of the `FCSupplierService` class computes the cost by iterating over `outProducts` and summing the fixed price of each product regardless of its quantity, then at the end, it adds the supplier's fixed cost. Similarly, `FPSupplierService` class represents suppliers with a fixed price, however, the product quantity is an important factor of determining the cost of a `FPSupplierService`. Its `supplyingCost()` method computes the cost by multiplying the price of each product by its quantity, at the end, it adds the supplier's fixed cost. `VDSupplierService` class is used for suppliers with a price structure that provides varying levels of discounts for certain threshold amounts that are reached by the buyer.

To exemplify the class `SupplierService`, consider one of its subclasses, `VDSupplierService`, below:

```
class VDSupplierService extends
SupplierService{
  VDSupplierInfo supplierInfo;
  Products[] outProducts;
  CostMetric costMetric;
  //class constructor
  VDSupplierService(VDSupplierInfo sInfo,
  Products[] outProd){
    this.supplierInfo = sInfo;
    this.outProducts = outProd;
    this.costMetric =
    new CostMeric(supplyingCost());
    this.defaultOptObjective = optMetric();
    this.minFlag = true;}
  //method to compute service cost
  protected double supplyingCost(){
    double cost=0;
    if(this.outProducts.length>0){
      for (int i=0; i<outProducts.length;
      i=i+1){
        double itemPrice =0;
        for(int p=0; p<supplierInfo.
        products.length; p=p+1){
            if(outProducts[i].iID==
            supplierInfo.products[p]){
                itemPrice =
                supplierInfo.unitCost[p];}}
        cost=cost+(itemPrice *
        outProducts[i].iQty);}
        for(int t=0;
        t<supplierInfo.thresh.length; t=t+1){
          if(cost>=supplierInfo.thresh[t]){
            cost = supplierInfo.thresh[t]+
            ((1 - supplierInfo.discount[t])*
            (cost-supplierInfo.thresh[t]));}}
          if (cost>0){
            cost=cost+supplierInfo.fxdCost;}}
    return cost;}
  //optMetric method is omitted here
}
```

Note that the Boolean variable `minFlag` is assigned a true value, which indicates that the `defaultOptObjective` of this service should be minimized. In this case the `Nd.checkMinObjective()` method is called for the `defaultOptObjective`.

Note also that this service provides level of discounts based on the total cost before discount (i.e., threshold specified by the corresponding `supplierInfo`). The discount is applied on the amount greater than (or equal) the threshold. The service then sums up the cost (after applying appropriate discounts) and adds its fixed cost to instantiate its `costMetric`.

3.3 Transportation Service

`TransportationService` is an atomic service represented as a subclass of the `Service` class. The constructor of the `TransportationService` class requires to provide a Transportation Info, the demand for outgoing products, and the locations from which the products are going to be transported. Similar to `SupplierService` class, the `TransportationService` constructor computes the transportation cost based on the carrier fee structure. In the LTL model, transportation cost is computed per origin-destination zone, and is proportional to the transported products weight and quantity. However, `TransportationService` has a special method to construct its incoming items' quantities and locations since we only provide the outgoing items in the list of this class constructor parameters.

```
private   TransportationProducts[]findProdsND
(int[] locs){
  TransportationProducts[] inProds =
  new TransportationProducts
  [locs.length*outProducts.length];
  int i=0;
  for (int p=0; p<outProducts.length;
  p=p+1){
  double prodQty = 0;
  double totProdQty = 0;
    for(int l=0; l<locs.length; l=l+1){
      prodQty=Nd.choice(0,outProducts[p].iQty);
      totProdQty = totProdQty + prodQty;
      inProds[i] = new TransportationProducts(
        outProducts[p].iID,
        outProducts[p].iDes,prodQty, locs[l],
        outProducts[p].locZone,
        outProducts[p].weight);
      i=i+1;}
    assert(totProdQty==outProducts[p].iQty);}
  return inProds;
  }
```

A private method constructs the consumed items `(inProducts)` by assigning a non-deterministic value to each package quantity to be transported from each stocking location within the range 0 to the demanded quantity of each product,

```
prodQty = Nd.choice(0, outProducts[p].iQty);
```

In the simulation semantics, the default interpretation of the `Nd.choice` methods is the random selection of a real value (of type double) from the interval with boundaries indicated by its actual parameters. In the line of code above, from the interval [0,outProducts[p].iQty]. (We will explain the optimization semantics of this statement in the next section). The private method also uses an *assert* statement

```
assert (totProdQty == outProducts[p].iQty);
```

which is ignored in the simulation semantics (except for an error message if the assertion is violated). The transportation service computes its cost by iterating over the consumed products and looking-up the fee in the fee table of the service, and then multiplying it by the product weight and quantity. The service then sums up the costs of all products, adds the fixed cost of this service, and uses this cost to instantiate the service's `CostMetric`.

3.4 Composite Sourcing Service

To exemplify the composite sourcing services, consider the Service Aggregator abstract class and its concrete subclass Volume-discount Supplier Service Aggregator (denoted as `VDSupplierSerAgg`) The constructor takes as arguments an array of suppliers Info, i.e., `VDSuppliersInfo[]`, and the set of items that should be produced by the service, i.e., `Products[]`.

`VDSupplierSerAgg` constructs the atomic volume-discount supplier services in the composition. The private method `aggSuppliers()` returns an array of `VDSupplierService[]`. For each possible supplier service provided by `VDSuppliersInfo[]`, it constructs a new `outProducts` with non-deterministic order quantities and use these products to instantiate a new supplier service, such that all the `outProducts` from all atomic volume-discount supplier services are aligned with the service aggregator `outProducts`.

```
class VDSupplierSerAgg extends
ServiceAggregator{
  VDSupplierInfo[] vdSuppliersInfo;
  Products[] outProducts;
  VDSupplierService[] vdSupplierServices;
  CostMetric costMetric;
  //constructor
  VDSupplierSerAgg(VDSupplierInfo[] si,
  Products[] oItems){
    this.outProducts = oItems;
    this.vdSuppliersInfo = si;
    this.vdSupplierServices = aggSuppliers();
    this.costMetric =
      new CostMetric(supplyingCost());
    this.defaultOptObjective = optMetric();}
  //method to aggregate suppliers
  private VDSupplierService[]aggSuppliers(){
    VDSupplierService[] ss =
      new VDSupplierService
        [vdSuppliersInfo.length];
    Products[] totOutProducts=
      new Products[vdSuppliersInfo.length*
        outProducts.length];
    int i=0;
    for (int p=0;p<outProducts.length;p=p+1){
     double ordQty = 0;
     double totQty = 0;
     for(int s=0;s<vdSuppliersInfo.length;
     s=s+1){
        ordQty=
        Nd.choice(0,outProducts[p].iQty);
        totQty = totQty + ordQty;
        totOutProducts[i]= new Products(
         outProducts[p].iID,
         outProducts[p].iDes, ordQty,
         outProducts[p].locZone,
         outProducts[p].weight);
     i=i+1; }
    assert(totQty>=outProducts[p].iQty);}
    for(int s=0; s<vdSuppliersInfo.length;
    s=s+1){
     Products[] newOutProducts=
     new Products[outProducts.length];
     int count = s;
     for(int t=0; t<outProducts.length;
     t=t+1){
        newOutProducts[t]= new Products(
          totOutProducts[count].iID,
          totOutProducts[count].iDes,
          totOutProducts[count].iQty,
          totOutProducts[count].locZone,
          totOutProducts[count].weight);
        count=count+vdSuppliersInfo.length;}
     ss[s]= new VDSupplierService(
        vdSuppliersInfo[s], newOutProducts);}
    return ss;}
  //method to return supply cost
  private double supplyingCost(){
    double cost = 0;
    for (int i=0;i<vdSupplierServices.length;
    i=i+1){
        cost = cost + vdSupplierSevices[i].
          costMetric.objective();}
        return cost;}
  //optMetric method is omitted here
}
```

Note that the *assert* statement specifies that the total quantity of each product in `outProducts` from all supplier services must be greater than or equal to this product quantity in the `outProducts` of the aggregator, but this statement is ignored in the simulation semantics. The aggregator computes volume-discount suppliers' costs by iterating over each service object and summing up the `costMetric` objectives.

Other Supplier Service Aggregators perform similar tasks. They assign non-deterministic values to the quantities from each possible atomic service, while asserting that the items that are consumed and produced by the sub-services are aligned with the items that are consumed and produced by the aggregator.

3.5 Strategic Sourcing Service with Supplier Rules

In reality, the objective of the strategic sourcing and transportation (e.g., minimizing the cost of the total purchase) is usually constrained by some business rules which make the optimization problem a harder one. Our framework provides a `Service` class with supplier business rules which are formulated in a simple fashion. The rules are specified by a set of double numbers passed as parameters to the constructor of the `Service` class. Each of these numbers corresponds to a specific business rule to be added to the model.

The `SupplyServiceAggsAggWithRules` service class supports supplier business rules of the following types:

- The maximum quantity (or percentage of the total quantity) of each item procured from each supplier is bounded to limit exposure to few suppliers.
- The minimum quantity (or percentage of the total quantity) of each item procured from each selected supplier is controlled to reduce the overhead of managing large number of suppliers.
- The maximum amount (or percentage of the overall procurement cost) paid for each selected supplier is bounded to limit exposure to few suppliers.
- The minimum amount (or percentage of the overall procurement cost) paid for each selected supplier is bounded to eliminate suppliers with minor advantage to the procurement process.

4 SYNTAX AND OPTIMIZATION SEMANTICS

Assume that user defines a subclass `MyService` of the class `Service`. The method `Nd.choice(double min, double max)` is used to indicate unknown choice constant, i.e., a decision variable, and the *assert* construct is used to indicate Boolean conditions that must be satisfied.

The optimization semantics is based on the notion of optimal service objects as follows.

We say that an object *s* of the class `MyService` is *feasible*, if it is constructed by `MyService` constructor, where each invocation of `Nd.choice(a,b)` method returns a value in the interval [a, b], and all *assert* statements are satisfied. Thus, every feasible `MyService` object can be identified by choice constants *$(C_1,\ldots, C_n)$* where *Ci* is the value returned by the *i*-th invocation of the method `Nd.choice(ai,bi)`. Let *S* denote the set of all feasible objects *s*.

Given a specific input to `MyService` constructor, the constructor defines a function $g : R_n \to R$ as follows. Given choice constants *$c_1,\ldots,c_n$*, it returns a value of the variable objective in the instantiated `MyService` object. We define a vector *$(c_1,\ldots,c_n)$* of optimal choice values as *argmin* $g(x_1,\ldots,x_n) s.t. (x_1,\ldots,x_n) \in S$ if `minFlag`=true; otherwise *argmax* is used.

The semantics of the `MyService` constructor is as follows. It operates exactly as the user specified constructor where the *i*-th invocation of `Nd.choice(ai,bi)` method for *i= 1,…,n* is replaced with the choice construct *Ci* from the vector *$(C_1,\ldots,C_n)$* of optimal choice values as defined earlier. Finally, the semantics of the Service Composition framework is identical to that of the Java language, with the exception of the `MyService` constructor.

5 A CASE STUDY

In this section, we exemplify the use and the semantics of the proposed framework using the example depicted in Figure 1. Once we have a library of atomic and aggregated services, building a new service such as the SST service can be easily done as follows:

```
class StrategicSourcingAndTransp extends
Service{
    StrategicSourcingAndTranspInfo sstInfo;
    Products[] outItems;
    CostMetric costMetric;
    boolean minFlg;
    //constructor
    StrategicSourcingAnTransp
     (StrategicSourcingAndTranspInfo info,
      Products[] products, int[] locations,
      double[] suppliersRules){
        this.sstInfo = info;
        this.outItems = products;
        //instantiate transportation
        TransportationService TS =
         new TransportationService(
          sstInfo.transportationInfo[0],
          outItems, locations);
        //instantiate sourcing
        SupplyServiceAggsAggWithRules SSAA =
         new SupplyServiceAggsAggWithRules(
          sstInfo.sInfo, TS.inProducts,
          suppliersRules);
    //define costMetric computation
        this.costMetric = new CostMetric(
          TS.costMetric.objective()+
          SSAA.costMetric.objective());
        this.defaultOptObjective= optMetric()
        this.minFlag = true;}
  //optMetric method is omitted here
}
```

The constructor of the class `StrategicSourcingAndTransp` (SST) encodes a simulation procedure which uses two of the services in the library; Transportation, and Supplier Aggregators Aggregator. We instantiate SST using the following parameters;

1. the service info (i.e., `StrategicSourcingAndTranspInfo` which carries the info of all the suppliers that are possibly going to be selected),
2. the demand (i.e., the products including specific demand locations),
3. the possible locations of the stocking facilities that products are going to be transported from,
4. and the business rules that constrain the sourcing.

Table 1: Data for SST Study Case

Demand and Suppliers Info						
	Fixed cost	Prod 11 (Water)	Prod 12 (Food)	Prod 13 (Medicine)	Volume-Discounts	
					%	after
Demand	-	100 at Loc 1	200 at Loc 1	200 at Loc 2	-	-
FC1	5,000	10,000	20,000	30,000	-	-
FP1	1,000	16	20	30	-	-
FP2	2,000	11	20	22	-	-
VD1	1,000	11	22	31.5	.2 .4	1,000 5,000
VD2	2,000	10.5	21	30.5	.1 .3	2,500 4,000

Transportation Info				
Zone	0	1	2	3
0	10	20	30	40
1	20	10	50	60
2	30	50	10	70
3	40	60	70	10
Transportation Fixed cost = 30				

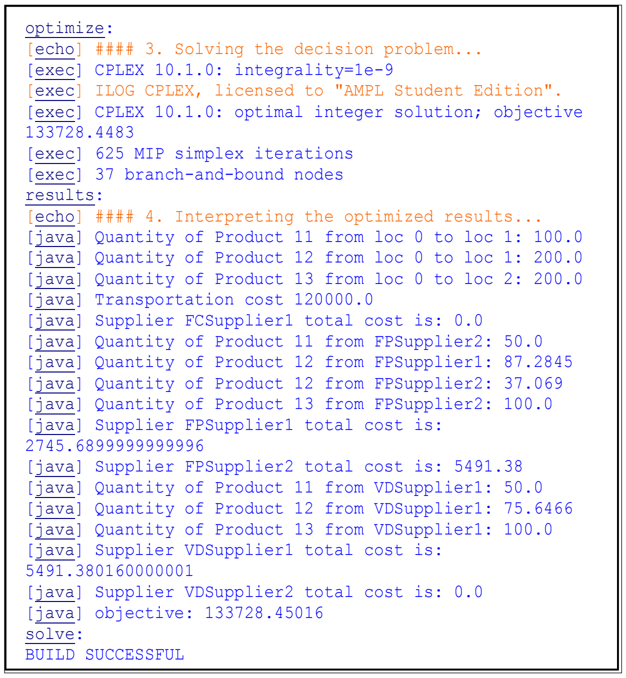
The constructor of SST does a number of instantiations that simulate the entire supply process. First it uses all its parameter arguments to instantiate the transportation service object (TS). Then it instantiates the supplier service aggregators aggregator object (SSAA) using the products that were constructed from TS as `inProducts`. SSAA in turn, instantiates a number of composite sourcing services (i.e., supplier service aggregators) each of which instantiates a number of atomic supplier services.

Since SST is a service itself, it instantiates a `BusMetric` that is defined here to be a cost metric; simply, it is the summation of all the sub-services `BusMetrics` objectives. Since the Boolean variable `minFlag` is true then the `defaultOptObjective of SST` will be minimized.

In the main program we instantiated `strategicSourcingAndTranspInfo` using one transportation `serviceInfo`, and five supplier `serviceInfo`; one fixed-cost supplier, two fixed-price suppliers, and two volume-discount suppliers. We also instantiated the demand products and the stocking locations (i.e., loc 0 and loc 3). We also defined two business rules to state that the procurement quantity from each supplier should not exceed 50%

of the total quantity of each product, and the amount paid for each supplier should not exceed 40% of the total procurement cost. See Table 1 for the case study data.

Having all the input data instantiated, we can now invoke the constructor of the `StrategicSourcingAndTransp` class, which simulates the entire process. Every time this application is run as a regular Java program, it produces a simulation to the `StategicSourcingAndTransp` service example. Because a random selection is used by every `Nd.Choice` method to select such values as quantities of each product to be shipped from each location to each demand location, and quantities of each product supplied by each supplier, each simulation run would result in a different outcome, including a different total cost of SST service. These outcomes do not give the minimum cost for SST service, but merely the costs corresponding to the random selections of values in the simulation runs. The optimization semantics, on the other hand, would minimize the cost of SST service, by automatically constructing a mathematical programming model and solving it on an external solver. The results of the solver are given in Figure 3.



```
optimize:
[echo] #### 3. Solving the decision problem...
[exec] CPLEX 10.1.0: integrality=1e-9
[exec] ILOG CPLEX, licensed to "AMPL Student Edition".
[exec] CPLEX 10.1.0: optimal integer solution; objective
133728.4483
[exec] 625 MIP simplex iterations
[exec] 37 branch-and-bound nodes
results:
[echo] #### 4. Interpreting the optimized results...
[java] Quantity of Product 11 from loc 0 to loc 1: 100.0
[java] Quantity of Product 12 from loc 0 to loc 1: 200.0
[java] Quantity of Product 13 from loc 0 to loc 2: 200.0
[java] Transportation cost 120000.0
[java] Supplier FCSupplier1 total cost is: 0.0
[java] Quantity of Product 11 from FPSupplier2: 50.0
[java] Quantity of Product 12 from FPSupplier1: 87.2845
[java] Quantity of Product 12 from FPSupplier2: 37.069
[java] Quantity of Product 13 from FPSupplier2: 100.0
[java] Supplier FPSupplier1 total cost is:
2745.6899999999996
[java] Supplier FPSupplier2 total cost is: 5491.38
[java] Quantity of Product 11 from VDSupplier1: 50.0
[java] Quantity of Product 12 from VDSupplier1: 75.6466
[java] Quantity of Product 13 from VDSupplier1: 100.0
[java] Supplier VDSupplier1 total cost is:
5491.380160000001
[java] Supplier VDSupplier2 total cost is: 0.0
[java] objective: 133728.45016
solve:
BUILD SUCCESSFUL
```

Figure 3: Optimization Results

6 IMPLEMENTATION NOTES

The Strategic Sourcing and Transportation framework was implemented by extending SC-CoJava (Brodsky, Al-Nory et al. 2008) with the sourcing and transportation modeling component library. The simulation procedure leaves the decision variables open which we call non-deterministic procedure. Then, the SC-CoJava (and CoJava) compiler translates a non-deterministic simulation procedure into an equivalent decision problem using a reduction algorithm. See (Brodsky and Nash 2005) for more details. The resulting decision problem consists of a set of constraints in the modeling language AMPL.

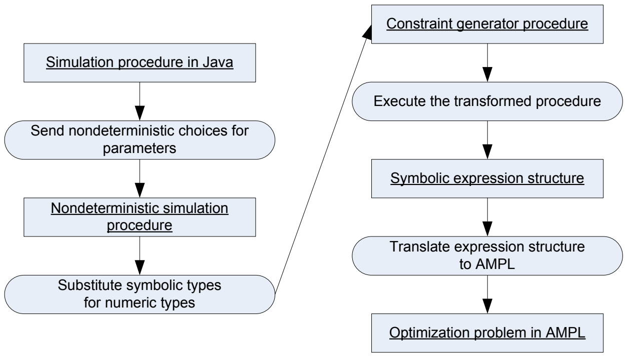

Simulation procedure in Java
Send nondeterministic choices for parameters
Nondeterministic simulation procedure
Substitute symbolic types for numeric types
Constraint generator procedure
Execute the transformed procedure
Symbolic expression structure
Translate expression structure to AMPL
Optimization problem in AMPL

Figure 4: Implementation Flow

The overall flow of the constraint compiler is shown in Figure 4. First, a simulation procedure is made nondeterministic by initializing it with values from the nondeterministic choice library, and designating its output as an objective value. This requires no change to the procedure itself, only to its parameters and return value. Next, the procedure is transformed to create a constraint generator procedure. This involves uniformly converting all of its numeric data types to symbolic expression data types. Next, the constraint generator is compiled and executed (using a standard java compiler). The result generated by this procedure is a set of symbolic expression data structures, represent the nondeterministic output of the simulation procedure. These symbolic expressions are translated into a mathematical programming language AMPL and are solved on a solver. Finally, the optimization results are used to run the simulation model deterministically. In the case of our case study example, it took 17 seconds to solve the problem using ILOG CPLEX (MILP) solver on a Dell OPTIPLEX GX260 machine with Intel® Pentium® 4 CPU 2.80GHz and 1 GB of RAM.

7 CONCLUSION AND FUTURE WORK

We proposed a new framework and a component library for making strategic sourcing and transportation decisions. Our approach allows quick construction of models for composite services with all the advantages of simulation model development, testing and extensibility, and yet allows optimization based on mathematical programming. Many questions remain for future research. They include extending our framework with stochastic and probabilistic models to allow for solving problems with uncertainty factors.

REFERENCES

Almeder, C., and M. Preusser. 2007. A Hybrid Simulation Optimization Approach for Supply Chains. *The 6th EUROSIM Congress on Modelling and Simulation*, Ljubljana, Slovenia.

Anderson, M. G., and P. B. Katz. 1998. Strategic Sourcing. *The International Journal of Logistics Management* 9(1): 1-13.

Bizaro, L. M. S. P., and J. G. Silva. 1998. Jwrap: A Java Library for Parallel Discrete-Event Simulation. *The ACM Workshop on Java for High-Performance Network Computing*.

Boisvert, R. F., S. E. Howe, and D. K. Kahaner. 1985. GAMS: A Framework for the Management of Scientific Software. *ACM Transactions on Mathematical Software (TOMS)* 11(4): 313-355.

Brodsky, A., M. Al-Nory, and H. Nash. 2008. Serivce Composition Language to Unify Simulation and Optimization of Supply Chains. *Proceedings of the 41st Hawaii International Conference on System Sciences*, Hawaii, USA, IEEE Computer Society Press.

Brodsky, A., and H. Nash. 2005. CoJava: Optimization Modeling by Nondeterministic Simulation. *Principles and Practice of Constraint Programming - CP2005*.

Caputo, A. C., L. Fratocchi, and P. M. Pelagagge. 2006. A Genetic Approach for Freight Transportation Planning. *Industrial Management & Data Systems* 106(5): 719-738.

Carr, A. S., and L. R. Smeltzer. 1999. The Relationship of Strategic Purchasing to Supply Chain Management. *European Journal of Purchasing & Supply Management* 5(1): 43-51.

Chen, C.-H., J. Lin, E. Yucesan, and S. E. Chick. 2000. Simulation Budget Allocation for Further Enhancing the Efficiency of Ordinal Optimization. *Dicrete Event Dynamic Systems: Theory and Applications* 10: 251-270.

Fourer, R., D. M. Gay, and B. W. Kernighan. 2003. *AMPL: A Modeling Language For Mathematical Programming*. Pacific Grove, VA, Brooks/Cole-Thomson Learning.

Fu, M. C. 2001. Simulation Optimization. *Proceedings of the 2001 Winter Simulation Conference*, Arlington, VA, USA, ACM.

Fu, M. C. 2002. Optimization for Simulation: Theory vs. Practice. *Informs Journal on Computing* 14(3): 192-215.

Healy, K. J., and R. A. Kilgore. 1998. Introduction to Silk and Java-Based Simulation. *Proceedings of the 1998 Winter Simulation Conference*, Los Alamitos, CA, USA.

Kelton, D. W., R. P. Sadowski and D. T. Sturrock. 2004. *Simulation with Arena*, McGraw-Hill Professional.

Kuo, C.-C., and F. Soflarsky. 2003. An Automated System for Motor Carrier Selection. *Industrial Management & Data Systems* 103(7): 533-539.

Law, A. M. 2007. *Simulation Modeling & Analysis*. New York, NY, Suzanne Jeans.

Lee, Y. H., and S. H.Kim. 2002. Production-Distribution in Supply Chain Considering Capacity Constraints. *Computers and Industrail Engineering* 43(1-2): 169-190.

Padmos, J., B. Hubbard, T. Duczmal, and S. Saidi. 1999. How i2 Integrates Simulation in Supply Chain Optimization. *Proceedings of the 1999 Winter Simulation Conference*.

Phelps, R. A., D. J. Parsons, and A. J. Siprelle. 2000. The SDI Industry Product Suite: Simulation from the Production Line to the Supply Chain. *Proceedings of the 2000 Winter Simulation Conference*.

Sandholm, T., and S. Suri. 2001. Market Clearability. *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)*, Seattle, WA.

Sandholm, T., and S. Suri. 2003. BOB: Improved Winner Determination in Combinatorial Auctions and Generalizations. *Artificial Intelligence* 145: 33-58.

Sandholm, T., S. Suri, A. Gilpin and D. Levine. 2001. Cabob: A Fast Optimal Algorithm for Combinatorial Auctions. *Proceedings of the 2001 International Joint Conferences on Artificial Intelligence*, Seatle, WA.

Swisher, J. R., and P. D. Hyden. 2000. A Survey of Simulation Optimization Techniques and Procedures. *Proceedings of the 2000 Winter Simulation Conference*, FL, USA, ACM.

Terzi, S., and S. Cavalieri. 2004. Simulation in the Supply Chain Context: A Survey. *Computers in Industry* 53: 3-16.

Truong, T. H., and F. Azadivar. 2003. Simulation Based Optimization for Supply Chain Configuration Design. *Proceedings of the 2003 Winter Simulation Conference*.

AUTHOR BIOGRPHIES

MALAK AL-NORY is a PhD candidate in the Information Technology Program at George Mason University. Her research interests include decision-guidance in supply chains, and unifying simulation and optimization in supply chains. Her email address is <malnory@gmu.edu>.

ALEXANDER BRODSKY is an Associate Professor of Computer Science at George Mason University. His research interests include decision-guidance systems and enterprise optimization. He also serves as Chief Technology Officer of Adaptive Decisions Inc., a start-up company that delivers Adaptive Enterprise Optimization solutions 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. Dr. Brodsky served as conference chairman of the fifth International Conference on Principles and Practice of Constraint Programming. He has authored a series of refereed journal and conference papers and co-edited a LNCS volume on constraint databases and programming. He earned his Ph.D. and prior degrees in Computer Science and/or Mathematics from the Hebrew University of Jerusalem. His email address is <brodsky@gmu.edu>.