Service Composition Language to Unify Simulation and Optimization of Supply Chains

Alexander Brodsky [1,2], Malak Al-Nory [1], and Hadon Nash [3]
[1] George Mason University, Virginia, USA
[2] Adaptive Decisions, Inc., Maryland, USA
[3] Google, California, USA
brodsky@gmu.edu, malnory@gmu.edu, hadonn@gmail.com

Abstract

Proposed and developed is the language Service Composition (SC) CoJava, which extends the programming language Java with (1) a modular service composition framework; (2) an extensible library of supply-chain modeling components such as items, services and business metrics; and (3) decision choice constructs for program variables, assertions of constraints and a designation of a program variable to serve as the objective to be minimized or maximized. The SC-CoJava provides not only the procedural "simulation-like" semantics of Java, but also an optimization semantics. The optimization semantics of SC-CoJava amounts to (1) finding an optimal instantiation of values into the choice-variables, based on automatic construction of a standard optimization model and solving it using a mathematical programming solver, and then (2) executing the Java program procedurally, where all the decision choice values are taken from the optimization result.

1. Introduction

Motivation

Simulation and optimization of supply chains have been widely used to make better decisions, e.g., to minimize costs or maximize profitability in diverse enterprises. A supply chain can be viewed as a complex service, which is composed of interrelated sub-services, such as distribution, transportation, manufacturing, and sourcing. In turn, these sub-services may be composed of more basic services, and so on. In this paper we propose the language Service Composition (SC-) CoJava for fast simulation-like modeling of service compositions in a supply chain. SC-CoJava allows the use of a unified supply chain model for both simulation and decision optimization based on Mathematical Programming.

Consider an example of Emergency Response (ER) Service, depicted in Figure 1. This service is responsible for delivering ER packages (with water, food, medicine, blankets etc) to a number of designated ER locations. The service consists of a supply chain that involves (1) transportation services to deliver the packages to ER locations, (2) packaging services, to make the packages from individual products; and (3) supply services, to purchase and deliver individual products to packaging facilities. In turn, transportation services may involve multiple carriers of different kinds (e.g., full-truck-load or less-than-truckload). Similarly, there may be multiple packaging and supply sub-services (see sub-services in Figure 1 expansion). A typical decision required may be on how to deliver the packages to ER locations in the shortest amount of time subject to available resources, or for the minimal total cost within a fixed (short) amount of time. The outcome of such a decision is an actionable recommendation on precisely what packages, in what quantities should be delivered from which packaging facilities and by which transportation carriers, and which products should be purchased, and in what quantities from which suppliers.

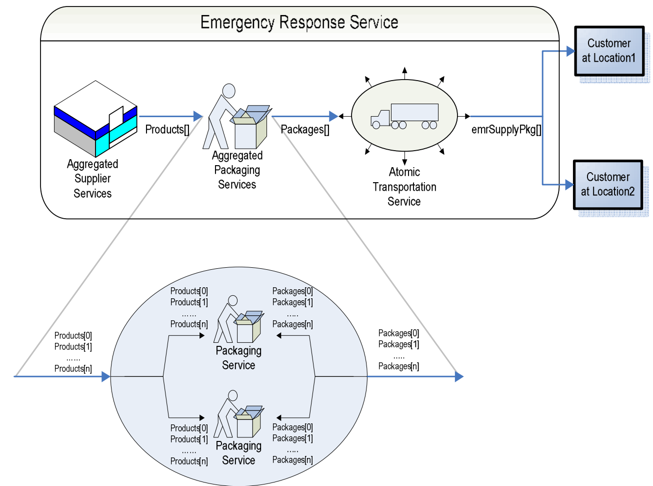

Figure 1: Emergency Response Service supply chain

The Challenge: Simulation vs. Optimization

To implement such an optimization problem, a typical Operations Research (OR) approach would be to construct a mathematical model, with decision variables, constraints, and an objective function, possibly using a modeling language such as AMPL[4] or GAMS[1], and then solve it using an existing mathematical programming solver, e.g., for Mixed Integer Linear Programming (MILP). However, to build such models is quite a challenging task, the more so for non-OR professionals, even for those with general computer science and software engineering skills.

As indicated in [2], the reason for that challenge is that the elements of an OR model are abstract constraints, which have only an indirect connection to elements of a real-world process. For example, one equation may combine elements from several real-world services or devices. In the example of ER service, one equation might involve quantities of each package/product produced by each service (greater than or) equal to the quantities of the same package/product produced by all the other services.

Also, the notions of order and timing of events are usually not explicit in OR models, which puts additional burden on the modeler. Furthermore, the execution of the optimization is typically a black box for the modeler, with no clear connection to the flow of the real world process. This makes debugging of an optimization model a challenging task. If the optimization fails there is no clear explanation for the failure. Finally, OR models typically lack the modularity of modern object-oriented (OO) programming languages, so they tend to become difficult to maintain over time (like "spaghetti code").

By contrast, simulations are generally well understood by software developers. The elements of a simulation are state variables and state-transitions, which have a clear one-to-one correspondence with elements of a real-world process. Every quantity from the real-world process is represented by a single state variable, so there is little room for confusion. Real-world time and sequence of events correspond to time and sequence in the running simulation in an obvious way. In the ER Service example, the order in which the services are instantiated is indicated clearly in the simulation process. Also, the "cause and effect" progression of the simulation is easy to follow. If the simulation fails, the exact time and place of the failure is reported. Finally, simulation modelers can practice modern OO software engineering. Complex building blocks can be modeled using simpler building blocks, and so on. In fact, modern OO languages have been derived from early simulation systems. We will show in the next section how complex services in the supply chain, such as the one in the ER example, can be built easily from simple OO components.

While simulation offers numerous advantages in ease of modeling, testing and extensibility, OR modeling has one major advantage. If modeled correctly using a manageable constraint domain such as LP or MILP, an optimization problem can be solved efficiently using existing solvers with sophisticated optimization algorithms. By contrast, no such solvers exist for simulation models. Typically, simulations are optimized by choosing parameters manually. An optimization layer can be added by running a simulation multiple times, with possible heuristics. However, such a search cannot compete with performance of solvers on manageable constraint domains.

The CoJava language[2] proposed a unified model that offers both the advantages of simulation-like process modeling, and the capabilities of true decision optimization. However, in terms of modeling, CoJava only provides a basic environment of Java, whereas a higher level of abstraction is desirable for fast modeling of supply chains like the one in the Emergency Response example. This is exactly the focus of this paper.

Contributions

In this paper we propose the language Service Composition (SC) CoJava, describe its syntax and semantics, report on its implementation, and exemplify its use on an emergency response supply chain scenario.

Syntactically, SC-CoJava extends the programming language Java with (1) a modular service composition (SC) framework; (2) an extensible library of supply-chain modeling components such as items, services and business metrics; and (3) decision choice constructs for program variables, assert statements and a designation of a program variable to serve as the objective to be minimized or maximized.

A modular SC framework is based on the concepts of Items, Business Metrics, Service Information and (atomic or composed) Services, such as Distribution, Manufacturing, Transportation and Sourcing. Conceptually, a Service represents a transformation of incoming Items to outgoing Items and the associated Business Metrics, such as cost, profit or reliability. These SC framework concepts are modeled as Java abstract classes. A hierarchy of supply-chain modeling components that adhere to the SC framework are represented as subclasses of the abstract classes in a Java library. New service classes can be added, whether they are atomic or composed of previously defined service classes (e.g., from the SC library). A simulation semantics of an SC-CoJava program amounts to its execution as a regular Java program, where the variable choice statements are interpreted as a random selection of a value from within a given range, while assert statements and a designation of a program variable to be minimized or maximized are ignored.

The SC-CoJava provides not only the procedural "simulation-like" semantics of Java, but also an optimization semantics, which amounts to a two-step process. First, SC-CoJava finds an optimal instantiation of values into the choice-variables, i.e., one that would satisfy all the following assert statements, and lead to the minimal or maximal value of a designated program variable. This is done in SC-CoJava by automatic construction of a standard optimization model

and solving it using a mathematical programming solver. Second, SC-CoJava executes the program procedurally, as a Java program, where all the decision choice values are taken from the optimization result. Note that SC-CoJava does not optimize using multiple simulation runs, but rather using a mathematical programming solver.

To exemplify the SC-CoJava framework, we developed an application to make decisions in emergency response situations. In this application we model the emergency response as a Service that is composed of a number of sub-services, which involve transportation, packaging and sourcing.

This paper is organized as follows. Section 2 discusses the SC framework and SC-CoJava simulation semantics. Section 3 focuses on the optimization semantics of SC-CoJava. Section 4 describes related work and Section 5 concludes.

2. Service Composition Framework and Its Simulation Semantics

Conceptual SC Framework

Figure 2 shows a partially expanded library of supply chain components that adhere to the Service Composition (SC) framework of SC-CoJava. The most important concept is that of a Service, to represent services such as Distribution, Manufacturing, Packaging, Supply, and Transportation. Conceptually, a service represents a transformation of incoming Items to outgoing Items. For example, a Manufacturing service transforms Items of type Materials to Items of type Products. A Packaging service transforms items of type Products to items of type Packages. Or a Transportation service transforms Items of type Transportation Package 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.

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. We designed these metric classes in accordance to the performance measures of the de facto reference in supply chain services, i.e., Supply Chain Operations Reference Model (SCOR) [1]. 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 discounts 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. Figure 1 shows an example of Emergency Response (ER) Service, described in the Introduction. ER Service has only outgoing Items of type Transportation Package, each characterized by quantity, delivery location, and the description of the more basic Items that compose the package. The ER Service is composed of three sub-services: Aggregated Supplier Services, which supply products to multiple packaging locations; Aggregated Packaging Services, which package products into packages to be shipped to designated ER locations; and a Transportation Service, that delivers incoming Items of type Package from packaging facilities to designated ER locations. In turn, the Aggregated Packaging and Supplier Services are composed of multiple atomic Services.

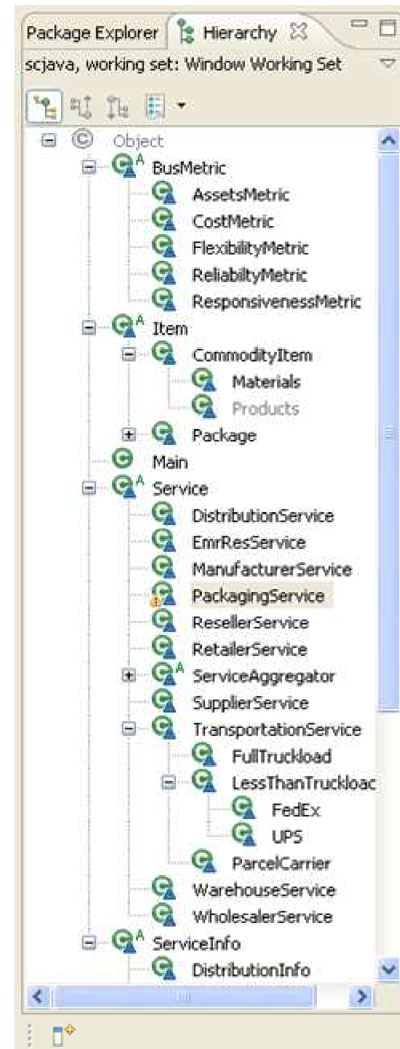

Figure 2: SC-Co Java library partially expanded

Java Representation and Simulation-like Semantics

For the purpose of this section, we assume the simulation-like semantics of SC-CoJava, which is the regular procedural semantics of Java. We explain the optimization semantics of SC-CoJava in the next section.

All SC framework components are represented in SC-CoJava as subclasses of the corresponding abstract classes: Item, Service, ServiceInfo, and BusMetric. The class Item has a number of concrete subclasses. For example, as shown in Figure 3, the class Products is a subclass of CommodityItem, which is in turn a subclass of Item.

```
class Products extends CommodityItem
{
        int locZone;
        double weight;

        Products(int iid, String des, double qty, int loc, double wght)
        {
           super (iid, des, qty);
           this.locZone = loc;
           this.weight = wght;
        }
}
```

Figure 3: Products class

The abstract class ServiceInfo has also a number of concrete subclasses. These classes provide specific data relevant to every service. Therefore, for every service type in SC-CoJava library (e.g., ManufacturerService, WarehouseService, RetailerService, etc.) we find a subclass of ServiceInfo to instantiate this service (e.g., ManufacturerInfo, WarehouseInfo, RetailerInfo, etc.) For example, TransportationInfo class is shown below in Figure 4.

```
class TransportationInfo extends ServiceInfo
{
        double[][] fees;
        double fxdCost;
        TransportationInfo(String tsId, double fxCost, double[][] fee)
        {
           super(tsId);
           this.fxdCost = fxCost;
           this.fees = fee;
        }
}
```

Figure 4: TransportationInfo class

To exemplify how concrete subclasses of type Service are represented consider the SupplierService class shown in Figure 5.

```
class SupplierService extends Service
{
   SupplierInfo supplierInfo;
   Products[] outProducts;
   CostMetric costMetric;

   SupplierService(SupplierInfo supplierInfo, Products[]outProd)
   {
         this.outProducts = outProd;
         this.supplierInfo = supplierInfo;
         this.costMetric = new CostMetric(supplyingCost());
         this.defaultOptObjective = optMetric() ;
   }

   private 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);
         }
         if ( cost  >  supplierInfo.thresh ){
            cost = supplierInfo.thresh +
            ((1 - supplierInfo.discount)*(cost - supplierInfo.thresh));
         }
      cost = cost + supplierInfo.fxdCost;
     }
   return cost;
   }

   CostMetric optMetric()
   {
   return this.costMetric;
   }
}
```

Figure 5: SupplierService class

Note that SupplierService computes its cost by iterating over outProducts and looking-up the price of each product at the supplierInfo's price table. The service provides a discount if the cost reaches the specified threshold by the supplierInfo. The service then sums the cost (after discounts) and adds its fixed cost to instantiate its costMetric.

Similarly, PackagingService class constructor computes packaging costs by iterating over outPackages volumes, looking-up the corresponding unit fees in the packagingInfo fee table, and multiplying by the packages quantity. PackagingService also constructs its inProducts by extracting the products objects from the outPackages.

TransportationService is an atomic service represented as a class that extends Service. A constructor of the TransportationService class requires to provide a Transportation Info, the outgoing demanded packages, and the locations from which the packages are going to be transported. A private method constructs the consumed items (inPackages) by assigning a non-deterministic value to each package quantity to be transported from each location within the range 0 to the demanded quantity of each package,

```
pkgQty = Nd.choice ( 0, outPackages[p].iQty ) ;
```

In the simulation semantics of SC-CoJava, 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 example above, from the interval [0,outPackages[p].iQty]. (We will explain the optimization semantics of this statement in the next section). The private method also uses an assert statement

```
assert ( totPkgQty == outPackages[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 packages (inPackages) and looking-up the location and the destination pair in the fee table of the service, and then multiplying the fee by the package weight and quantity. The service then sums up the costs of all packages, adds the fixed cost of this service, and uses this cost to instantiate the service's CostMetric.

To exemplify the Service Aggregator abstract class, consider its concrete subclass Supplier Service Aggregator (denoted as SupplierSerAgg), shown in Figure 6. The constructor of the Aggregated Supplier service takes as arguments SuppliersInfo[] and the items that should be produced by the service, i.e., Products[]. SupplierSerAgg does a similar job to the atomic Supplier service, but also constructs the atomic supplier services in the composition. The private method aggSuppliers() returns an array of SupplierService[]. For each possible supplier service provided by SuppliersInfo[], 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 supplier services are aligned with the service aggregator outProducts.

Note that the assert statement specifies that the total quantity of each product in outProducts from all supplier services must be equal to this product quantity in the outProducts of the aggregator, but this statement is ignored in the simulation semantics. A quantity of zero for a specific service would indicate that this service is being omitted from the composition. The aggregator computes suppliers' costs by iterating over each service object and summing up the costMetric objectives.

Other Aggregation Services, such as Transportation and Packaging 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.

```
class SupplierSerAgg extends ServiceAggregator
{
   SupplierInfo[] suppliersInfo;
   Products[] outProducts;
   SupplierService[] supplierServices;
   CostMetric costMetric;

   SupplierSerAgg(SupplierInfo[] si, Products[] oItems)
   {
        this.outProducts = oItems;
        this.suppliersInfo = si;
        this.supplierServices = aggSuppliers();
        this.costMetric = new CostMetric(supplyingCost());
        this.defaultOptObjective = optMetric();
   }

   private SupplierService[] aggSuppliers()
   {

   SupplierService[] ss = new SupplierService [ suppliersInfo.length ];
   Products[] totOutProducts= new Products[suppliersInfo.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<suppliersInfo.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<suppliersInfo.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 + suppliersInfo.length;
             }
        ss[s]= new SupplierService(suppliersInfo[s], newOutProducts);
        }
        return ss;
   }

   double supplyingCost(){
        double cost = 0;
        for (int i=0; i<supplierServices.length; i=i+1){
             cost = cost + supplierServices[i].costMetric.objective();
        }
        return cost;
   }

   CostMetric optMetric()
   {
        return this.costMetric;
   }
}
```

Figure 6: Supplier Service Aggregator class

Emergency Response Service Example

Once we have a library of atomic and aggregated services, building a new service such as ER service in Figure 1, can be easily done as follows:

```
class EmrResService extends Service{

   EmrResServiceInfo emrResServiceInfo;
   Package[] outItems;
   CostMetric costMetric;

   EmrResService(EmrResServiceInfo ersInfo,
                 Package[] emrSupplyPkg,int[] locations)
   {
         this.emrResServiceInfo = ersInfo;
         this.outItems = emrSupplyPkg;
         TransportationService TS = new TransportationService(
                 emrResServiceInfo.transportationInfo,
               outItems, locations);
         PackagingSerAgg PSA = new PackagingSerAgg(
               emrResServiceInfo.packagingInfo, TS.inPackages);
         SupplierSerAgg SSA = new SupplierSerAgg(
               emrResServiceInfo.suppliersInfo, PSA.inProducts);
         this.costMetric = new CostMetric(
               TS.costMetric.objective()+
               PSA.costMetric.objective()+
               SSA.costMetric.objective());
         this.defaultOptObjective = optMetric();
   }

   CostMetric optMetric()
   {
         return this.costMetric;
   }
}
```

Figure 7: Emergency Response Service constructor

As shown in Figure 7, the constructor of the class Emergency Response Service (denoted as EmrResService) encodes a simulation procedure which uses three of the services in the library; transportation, packaging, and supplier. We instantiate EmrResService using three parameters; the service info (i.e., emrResServiceInfo that has the info of all the services that are possibly going to be used), the demand (i.e., emrSupplyPkg), and the possible locations of the packaging facilities that packages are going to be transported from. The constructor of EmrResService does a number of instantiations that simulate the entire supply process. First it uses all its three parameter arguments to instantiate the transportation service object (TS). Then it instantiates the packaging service aggregator object (PSA) using the packages that were constructed by TS as inPackages. Then it instantiates the supplier service aggregator object (SSA) using the products that were constructed from PSA as inProducts. Every instantiated service takes as its own outItems, the inItems which was constructed by the previous service and in turn it constructs its own inItems. Finally, since EmrResService 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 services BusMetrics objectives.

The main program shown in Figure 8 exemplifies a typical application of an Emergency Response Service (we omit the instantiation of input data due to space limitations). It instantiates emrResServiceInfo using one transportation serviceInfo, three packaing serviceInfo, and two supplier serviceInfo. It also instantiates the demand emrSupplyPkg. Having all the input data instantiated, it invokes the constructor of the EmrResService class, which simulates the entire process described in the Emergency Response Service example. Lastly, we designate the objective to be the costMetric objective, which is the total cost of all the services. Note that in the simulation semantics, this statement is ignored, except for a message indicating the value of the objective in the result of the simulation.

```
public class TypicalUsage
{
   public static void main(String[] args)
   {
      /* instantiate demand */
      // instantiate emergency supply package

      /* instantiate some ServiceInfo objects */
      // instantiate some  TransportationInfo, insert into TransportationInfo[] ti
      // instantiate some PackagingInfo, insert into PackaingInfo[] pi
      // instantiate some SuppliersInfo, insert into SuppliersInfo[] si

      /* instantiate EmrResServiceInfo using instantiated serviceInfos */
      EmrResServiceInfo emrInfo = new EmrResServiceInfo ("ERS1",ti,pi,si);

      /* instantiate EmrResService using Info, demand, and packaging
        facilities locations */
      EmrResService ERS = new EmrResService(
        emrInfo, emrSupplyPkg, new int[]{1,2,3});

      /* define objective */
      Nd.checkMinObjective(ERS.costMetric.objective());
      }
}
```

Figure 8: TypicalUsage class illustrates main program

Every time this application is run as a regular Java program, it produces a simulation to the entire Emergency Response Service example. Because a random selection is used by the Nd.Choice method to select such values as quantities of each package to be shipped from each location to each ER 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 the Emergency Response service. For example, we might run the application three times, and see the following output (produced by the Nd.checkMinObjective method in the simulation semantics):

```
objective: 3240404.73
objective: 2812901.62
objective: 2483801.83
```

Note that these numbers do not give the minimum cost for ER service, but merely the costs corresponding to the random selections of values in the simulation runs. The optimization semantics, discussed in the next section allows

to automatically make a selection of values that would minimize the cost of ER service.

3. SC-CoJava Syntax and Optimization Semantics

SC-CoJava Syntax

By design, the syntax of SC-CoJava is identical to that of the Java programming language, extended with the SC framework and library. Also, it is extended with one special class Nd, and a few restrictions on how its methods can interact with the rest of the program. More specifically, SC-CoJava adds the following special class to Java:

```
public class Nd {
public double choice(double min, double max) {...}
public double checkMinObjective(double objective) {…}
public double checkMaxObjective(double objective) {…}
}
```

As mentioned earlier in the examples, the simulation semantics interprets the methods of the class Nd as follows. The method Nd.choice(min,max) returns a single specific value between min and max, inclusive. Note that the value for min may be negative infinity, and for max positive infinity. The user can use her own implementation of the choice method, or use the default SC-CoJava implementation (which is currently a random selection using uniform distribution).

The methods checkMinObjective and checkMaxObjective, in the simulation semantics, do nothing but output the value of the parameter objective. An SC-CoJava program can also use the Java command

```
assert(booleanCondition)
```

with the standard procedural semantics, namely the program will report an error if the booleanCondition is not satisfied.

Restrictions on the Non-deterministic (Nd) Class Methods

Certain restrictions on how the methods choice(...), checkObjective, and the command assert interact with the rest of SC-CoJava program are imposed to make the optimization semantics well-defined and computable. To formulate the restrictions, we use the notion of nondeterministic values, or ND-values for short, which we define recursively as follows:

- The output of a choice method is a ND-value
- A variable is a ND-value, if it appears on the left-hand side of an assignment with a ND-value on the right-hand side.
- A variable is a ND-value, if it appears on the left-hand side of an assignment that appears in the THEN or ELSE part of a conditional statement, where the Boolean condition is a ND-value.
- The result of an arithmetic or Boolean operation on one or more ND-values is a ND-value.

We also say that a conditional statement is ND, if its Boolean condition is ND. A LOOP statement is ND, if its exit Boolean condition is ND. A method call is ND, if it is done from within a ND conditional statement. We also say that a variable, expression, conditional statement etc. are deterministic to mean the negation of being nondeterministic.

The following simple restrictions are imposed in order to make the number of values computed by the program independent of the nondeterministic choices, and associate at most one checkObjective method call with each choice call and assert statement.

- No ND loops
- No ND recursive method calls
- No ND calls for checkObjective.

The first restriction controls the number of iterations of each loop. As long as the loop's exit conditions are deterministic, the loop will continue for a deterministic number of iterations. If a loop executed a non-deterministic number of iterations, it would compute a nondeterministic number of values.

The second restriction controls the depth of recursive calls. By prohibiting recursive calls within non-deterministic conditionals, we prevent the depth of recursion from depending on nondeterministic choices.

We do allow arbitrary non-recursive method calls, whether or not they are deterministic, and also recursive method calls as long as they are deterministic. Note that the conditions above are sufficient, but not necessary, to control the number of states in the program. More flexible conditions are subject for further work.

The third restriction, namely that no checkObjective is called from a ND conditional statement, makes sure that per given input to the program, (1) all checkObjective method calls have a total ordering, which is deterministic, and (2) that every execution path that goes through a specific choice or assert statement will deterministically "continue" to a unique "nearest" checkObjective call (if there is any). In this case, we say that such a choice or assert statement is in the scope of that nearest checkObjective call.

Under the restrictions on the use of nondeterministic values, the reduction to the standard optimization formulation is guaranteed to always work and be correct. However, the resulting formulation may be beyond the constraint domain handled by the available external solver. Thus, similar to OR modeling, a SC-CoJava developer needs to be aware to only use arithmetic expressions that can be handled by the solver used. For example, if an MILP solver is used, a SC-CoJava program can only use arithmetic expressions that are linear in nondeterministic values (but arbitrarily complex in deterministic values). Or if a non-linear solver such as

MINUS or SNOPT is used, a SC-CoJava program can use non-linear arithmetic expressions, but no nondeterministic conditional statements.

SC-CoJava Optimization Semantics

A sequence of specific selections in nondeterministic choice statements corresponds to an execution path. We define a feasible execution path as one that satisfies (1) the range conditions in the choice statements, and (2) the assert-constraint statements. An optimal execution path is a feasible path that produces the optimal value in a designated program variable, among all feasible execution paths.

Case 1: Single checkObjective Call, as Last Program Statement

We assume here that all the restrictions outlined are satisfied. Given a SC-CoJava program **P**, input **I**, and the checkObjective(v) statement as the last program statement S, we denote by EP the set of all feasible execution paths e, i.e., execution paths that reach S. For a particular feasible execution path e in EP, we denote by v(e) the value of the program variable v at the statement S. An optimal execution path is a solution to the following optimization problem OP:

Optimize v(e) s.t. e in EP

where Optimize stands for Minimize in the case of checkMinObjective and for Maximize in the case of checkMaxObjective. Note that a solution to this problem may not be unique, as more than one feasible execution path e in EP may have the minimal/maximal v(e). An optimal execution path e defines the values for each execution of a choice method.

An execution of the program P according to the SC-CoJava optimization semantics is a regular procedural execution where the values returned by each choice statement are those corresponding to an optimal execution path e.

Case 2: No checkObjective in the program

In this case, consider a satisfaction problem SP: Find e in EP, where EP is the set of all feasible execution paths, i.e., those that reach a special NOOP method (which does nothing) at the end of the program.

An execution of the program P according to the SC-CoJava optimization semantics is a regular procedural execution where the values returned by each choice statement are those corresponding to a feasible execution path e.

Case 3: One or More checkObjective Calls According to Restrictions

We do not discuss this general case in detail in this paper, nor was it implemented. Here we only provide a general idea. Because of the restrictions, (1) all checkObjective method calls have a total ordering, which is deterministic, and (2) every execution path that goes through a specific choice or assert statement will deterministically “continue” to a unique “nearest” checkObjective call (if there is any). In this case, we say that such a choice or assert statement is in the scope of that nearest checkObjective call. The idea here is to consider an execution as split into sections, in the order of checkObjective calls, each with the choice and assert statements in its scope, and apply Case 1 on all but the last section, and Case 2 on the last.

Emergency Response Example as Optimization

The question is what should be an *optimal* selection of values in Nd.choice statements (instead of a random selection) in order to (1) satisfy all the requirements of the Emergency Response (i.e., all the assert statements) and (2) make the total cost, computed by the simulation, minimal. Answering such questions is exactly the purpose of the optimization semantics of SC-CoJava.

More specifically, the SC-CoJava compiler (under the optimization semantics) will first determine an optimal selection of values in all Nd.choice statements (instead of randomly selecting them), resulting in the minimum cost, according to Case 1 of Optimization Semantics. This will be done by an automatic reduction to a standard optimization formulation and solving it on a mathematical programming solver. Then, SC-CoJava compiler will run the user program as a regular Java program, where all Nd.choice statements select the values from the optimization result. In the example, emrSupplyPkg which corresponds to the demand, included two packages, each with different characteristics, such as quantity, delivery location, volume, weight, and the type of products included. For example, the first package has a quantity of 100, includes food and water, and must be delivered to location 2, while the second package has a quantity of 200, includes medicine, and must be delivered to location 1. (we omit the other details due to space limitations) Given the demand information, the locations of the facilities, and the serviceInfo of all the possible services in the composition, the optimum (i.e., with the minimum cost) ER service was instantiated. The SC-CoJava compiler first found the optimal selection of the values of the order quantities represented by the Nd.choice statements, satisfying all the assert statements, and consequently, producing the following minimum total cost (after 8 MIP simplex iterations): objective 1772493. This optimal objective corresponds to a specific composition of services; as a result, some of the services in the emrInfo were instantiated with an order quantity of 0.

According to the optimization results, first demanded package (Package 1111) must be delivered from location 1, while second demanded package (Package 1112) must be delivered from location 0. Both packages should be packaged at Facility 1. Product 13 (medicine) should be ordered from

Supplier 2, Product 12 (water) should be ordered from Supplier 1, while Product 11 (food) should be ordered from Supplier 2.

Implementation Notes

SC-CoJava was implemented by extending the language CoJava[2] and adding the SC framework and extensible library. The SC-CoJava (and CoJava) compiler translates a non-deterministic simulation procedure into an equivalent decision problem using a reduction algorithm (see[2]). The resulting decision problem consists of a set of constraints in the modeling language AMPL.

The overall flow of the constraint compiler is shown in Figure 9. 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. Finally, these symbolic expressions are translated into a mathematical programming language AMPL and are solved on a solver. In the case of our ER Service example, it took 14 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.

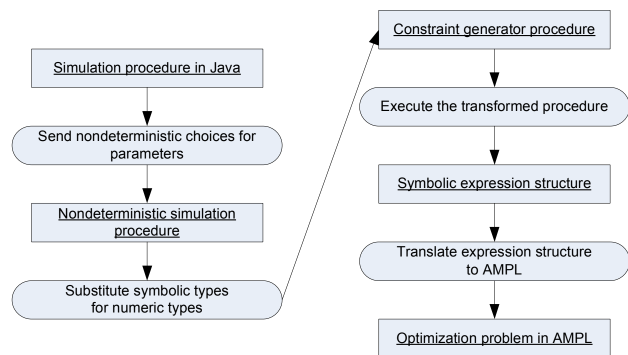

Figure 9: Implementation Flow

4. Related Work

SC-CoJava addresses the goals of constraint modeling and object oriented simulation for service compositions in supply chains. Object oriented simulation has traditionally been approached through procedural object oriented languages, such as Smalltalk[7] and Java. These languages start with a syntax for variable assignment and add support for modular organization of procedures. There are many specialized object oriented simulation languages such as Simula[3] and ModSim[18] and there are simulation environments layered on top of existing object oriented languages such as Silk[8] and Jwarp[15]. There are also productivity tools for supply chain simulation, such as Supply Chain Guru[13] and SDI Industry Pro[16]. These languages and tools allow complex models to be constructed and maintained effectively, but lack support for systematic optimization, based on mathematical programming. Typically, simulation approaches are used when the problem is too complex for existing optimization techniques (e.g., MILP). In contrast, SC-CoJava is designed for cases when existing optimization techniques (and solvers) can be used but allows to model the problem as a simulation-like process. This provides the benefits in ease of modeling, testing, and extensibility.

Constraint modeling has traditionally been approached through specialized constraint modeling languages, such as AMPL[5]. These languages start with a syntax for constraints and layer additional support for organizing them. They enable systematic optimization, but they require explicit definition and maintenance of constraints models.

Constraint programming (CP) languages, such as OPL[9], CLP[11], ECLiPSe[19], and ILOG SOLVER[17] allow developers to specify strategies for solving optimization problems, but their modeling part is based on declarative constraints, similar to AMPL. Languages such as Cob[12] and Siri[10] add object oriented modeling constructs, such as inheritance and encapsulation, to a constraint syntax. Similarly, COMET language [14] adds object-oriented modeling and control abstractions to constraint-based local graph search using randomization, check-pointing, and meta-heuristics. These languages provide a clean representation for steady-state optimization problems, but they do not model state changes in the direct way that procedural languages do.

The language Modelica[6] might be the most closely related to SC-CoJava in terms of the capability of translating procedural algorithms into declarative constraints. Modelica language supports unified models which can define both simulation and optimization problems. The models are translated to equations to be solved by an optimizer or compiled into a sequential procedure. Modelica can translate procedural algorithms into constraint equations within pure functions only (functions without side effects); therefore, it is efficient within a very limited context.

By contrast, SC-CoJava has a thoroughly procedural object oriented syntax and semantics inherited from Java. SC-CoJava eliminates the boundary between procedures and constraints. SC-CoJava gives developers the flexibility to move model components freely back and forth between procedural algorithms and declarative optimization models. The models developed using SC-CoJava can accommodate any supply chain design with any number of echelons. SC-CoJava approach provides robust true optimized evaluation of

alternative solutions and is more attractive than the complex mathematical programming models.

5. Conclusions and Future Work

To the best of our knowledge, ours is the first paper to propose a unified language for both simulation and optimization of service composition in supply chains. Our approach is unique as it offers the advantages of simulation-like model development, testing and extensibility, while providing true decision optimization based on mathematical programming.

Many questions remain for future research. They include extending SC-CoJava with stochastic and constraint programming capabilities, and utilizing the structure of high-level model to develop optimization heuristics that can work in conjunction with existing MP solvers.

6. References

1. Boisvert, R.F., Howe, S.E. and Kahaner, D.K. Gams: A Framework for the Management of Scientific Software. *ACM Transactions on Mathematical Software (TOMS)*, *11* (4). 313-355.
2. Brodsky, A. and Nash, H. CoJava: Optimization Modeling by Nondeterministic Simulation. in *Principles and Practice of Constraint Programming - CP2005*, 2005.
3. Dahl, O.J. and Nygaard, K. Simula: An Algol-Based Simulation Language. *Communication ACM* (9). 671-678.
4. Fourer, R., Gay, D.M. and Kernighan, B.W. *AMPL: A Modeling Language For Mathematical Programming*. Brooks/Cole-Thomson Learning, Pacific Grove, VA, 2003.
5. Fourer, R., Gay, D.M. and Kernighan, B.W. A Modeling Language for Mathematical Programming. *Management Science* (36). 519-554.
6. Fritzson, P. and Engelson, V., Modelica - a Unified Object-Oriented Language for System Modelling and Simulation. in *The 12th European Conference on Object-Oriented Programming*, (London, UK, 1998), Springer-Verlag, 67-90.
7. Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
8. Healy, K.J. and Kilgore, R.A., Introduction to Silk and Java-Based Simulation. in *WSC'98 in The 30th Conference on Winter Simulation*, (Los Alamitos, CA, USA, 1998), IEEE Computer Society Press, 327-334.
9. Hentenryck, P.V., Michel, L., L, P. and Regin, J.C., Constraint Programming in OPL. in *The Internation Conference on Principles and Practice of Declarative Programming (PPDP'99)*, (London, UK, 1999), Springer-Verlag, 98-116.
10. Horn, B.L. *Siri: A Constrained-Object Language for Reactive Program Implementation.* School of Computer Science, Carnegie Mellon University, Pitsburgh, PA, USA, 1991.
11. Jaffar, J. and Lassez, J.L., Constraint Logic Programming. in *The 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, (New York, NY, USA, 1987), ACM press, 111-119.
12. Jarayaman, B. and Tambay, P. Semantics and Applications of Constrained Objects. Technical Report 2001-15, 2001.
13. LLamasoft. Supply Chain Guru, www.llamasoft.com.
14. Michel, L. and Hentenryck, P.V., Comet in Context. in *ACM Internation Conference, Proceedings of Paris c.Kanellakis memorial workshop on Principles of computing & Knowledge:*, (San Diego, California, USA, 2003), ACM, 95-107.
15. P.Bizaro, L.M.S. and Silva, J.G., Jwrap: A Java Library for Parallel Discrete-Event Simulation. in *the ACM Workshop on Java for High-Performance Network Computing*, (1998).
16. Phelps, R.A., Parsons, D.J. and Siprelle, A.J., The SDI Industry Product Suite: Simulation from the Production Line to the Supply Chain. in *the 2000 Winter Simulation Conference* (2000), 208-214.
17. Puget, J.F. and Leconte, M. Beyond the Glass Box: Constraints as Objects. *International Logic Programming Symposium*. 513-527.
18. Thomasma, T. and Madsen, J., Object Oriented Programming Languages for Developing Simulation-Related Software. in *WSC'90 in The 22nd Conference in Winter Simulation* (Piscataway, NJ, USA, 1990), IEEE Press, 482-485.
19. Wallace, M., Novello, S. and Schimpf, J. ECLiPs: A Platform for Constraint Logic Programming. . *ICL Systems Journal* (12). 159-200.