Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

A DISTRIBUTED COMPUTING ARCHITECTURE FOR SIMULATION AND OPTIMIZATION

Yijia Xu
Suvrajeet Sen

Systems & Industrial Engineering Department
College of Engineering
University of Arizona
Tucson, AZ 85721, U.S.A.

ABSTRACT

We design a generic framework to integrate distributed simulation and optimization models. Many problems require the integration of these two types of models. For example, stochastic programming can use simulation models as a scenario generator for optimization models; in some other cases, simulation models need optimization models to help determine system parameters. The framework is shown to be able to provide various services to help the integration of simulation and optimization models. We illustrate our implementation with a product-mix example. The example integrates a discrete event simulation of a product-mix problem with a linear programming (optimization) model of such a system. The simulation updates the parameters in the optimization model, which as a result will generate a new production plan.

1 INTRODUCTION

Simulation has become an important method of systems analysis, and is widely used in engineering and science. The application areas of this method include manufacturing, financial engineering, military games, and many other fields. Meanwhile, practical optimization has seen sustained developments in all facets, including modeling, algorithms, and software. These developments, together with the need for advanced modeling and simulation in industry, military, and government have provided the impetus for integrating simulation and optimization for complex engineered systems.

There are several reasons motivating interactions between simulation and optimization models. Fu (2002) cites the following interactions between optimization and simulation models:

- Simulation for Optimization. Simulations work as a scenario generator which provides the optimization with a sample space. One of the more common settings for this arises in stochastic programming, where the term "sample-path optimization" has been coined by Robinson and co-workers. Plambeck, Fu, Robinson, and Suri (1996) apply sample-path optimization method to optimize convex stochastic performance functions. Gürkan, Özge, and Robinson (1999) broaden the method's applicability to include the solution of stochastic variational inequalities. Such methods are also referred as "retrospective optimization".
- Optimization via Simulation. In this category the optimization component orchestrates the simulation of a sequence of system configurations, so that the system configuration obtained eventually is an optimal or near optimal solution. Suri and Leung (1989) used a stochastic approximation method to optimize a simulation model in a single simulation run of an M/M/1 queue problem. Other methods include the stochastic ruler algorithm (Yan and Mukai, 1992), variants of simulated annealing altered to accommodate randomness (Gelfand and Mitter, 1989; Gutjahr and Pflug, 1996; Alrefaei and Andradóttir, 1999), and Andradóttir's (1996) random search algorithms. Pichitlamken and Nelson (2003) report a combined procedure which consists of a global guidance system, a selection-of-the-best procedure, and local improvement for use when the performance measure is estimated via a stochastic, discrete-event simulation, and the decision variables may be subject to deterministic linear integer constraints. This particular arena of research has also attracted the attention of simulation software vendors (AutoStat, OptQuest, OPTIMIZ, SimRunner, and WITNESS Optimizer) who already provide some optimization capability, although optimality in these systems is difficult to verify.

Another setting in which it becomes important to provide services for data exchange between optimization and simulation is to enable models that accommodate multiple fidelities. For instance, tractability of an optimization model might require a coarse-grain stochastic model, whereas, a detailed (fine-grain) simulation model may be used to examine the consequences of the solution provided by the optimization model. Such an approach was reported in Sen, Doverspike and Cosares (1994), and, with increasing popularity of "fluid approximations" of queueing systems, this approach is becoming increasingly popular. The fluid approximations relax discrete nature of the objects in a flow control system and may lead to efficient approximation or heuristics procedures and sometimes even to efficient optimization algorithms. Applications of "fluid approximations" can be found in papers reported by Chen and Mandelbaum (1991), Bertsimas and Gammarnik (1999), Boudoukh, Penn, and Weiss (2001), and others. Finer gain simulation models that capture detailed discrete object flow features are often used to evaluate the efficiency of the method.

In addition to the integration of simulation and optimization models, there are also needs to integrate simulation models or optimization models themselves respectively. In large-scale system simulations, models can be partitioned and developed by different groups. Simulations developed by each group are required to be able to integrate with those developed by other groups so that the entire system can be studied as a whole. Similarly in large-scale optimization, problems are often decomposed so that smaller (and perhaps easier sub-problems) can be solved. Interesting research results include Lee's (2004) implementation of a disjunctive cutting-plane algorithm in a distributed memory environment, Blomvall's (2003) parallel algorithm for multistage stochastic programming, and so on. While there are many examples of efforts to solve optimization problems with distributed computing techniques, the software developed for these examples are rarely designed for re-use by different classes of models. Such research often focuses on implementing a certain algorithm, rather than constructing a generic architecture to enable a variety of algorithms. While it is clear that distributed computing is playing an increasingly important role for simulation and optimization, software engineering practices remain limited to customized implementation for specific applications. The efforts to develop a distributed computing system for a specific problem are usually difficult and time consuming; therefore it is necessary to develop a systematic architecture for this purpose. The architecture is required to be composable such that different models, including decision models and simulation models can be integrated. It should be interoperable to support communication and synchronization at runtime. In other words, the architecture should be able to provide such services as to dispatch tasks, control runs, and exchange outputs of each type of model.

In this paper we discuss the design of a distributed computing system, which is part of the Simulation Platform for Experimentation and Evaluation of Distributed Computing Systems (SPEED-CS) project. The system sets up a flexible framework that provides services to integrate simulation and optimization models. We illustrate its uses with a product-mix problem as an example. The organization of the rest of this paper is as follows. In the next section, we address the requirements and design of the generic architecture. In section 3 we present the implementation of an experimental system with the SPEED-CS architecture. We present an example of interoperability between product-mix optimization and simulation in section 4. Some concluding remarks and future directions are presented in section 5.

2 SPEED-CS SYSTEM DESIGN

In order to achieve composability and interoperability between model components, there are three major aspects of design that bear reflection. These are the design of system architecture, system modeling, and data models. System architecture designs the coordination of the components. System modeling specifies the mechanism of modeling objects and coupling model components. Data models are necessary to provide a common data format for components to interact with each other. Although the actual implementation varies, any distributed system with the scope we have identified requires a resolution of these issues.

2.1 SPEED-CS Architecture

The SPEED-CS system is constructed with multiple layers as illustrated in Figure 1. The layers include an Object Request Broker (ORB) layer, a SPEED-CS layer, a Components layer, and a User Interface/Modeling layer. The coupled model is defined in the Modeling layer, while the model components are implemented in the Components layer. The dashed line indicates that the model components are linked remotely. With the services in the SPEED-CS layer and ORB layer, a remote object can be referenced. Each layer has functionality as follows.

- Object Request Broker (ORB) layer. The ORB layer provides the distributed communication services. It is important in that it hides network programming from the developers so as to save time of designing, implementing and debugging the software components involved in networking.
- SPEED-CS layer. The main service provided by this layer is to manage the model components. It maintains a registry of all the model components by names, server machine URLs and ports. When there is a request for a component, the SPEED-CS layer locates the component by facilitating the ORB layer Naming Service with the information in the registry.
- Components layer. It maintains a collection of simulation model components and decision model components. By specifying the names and coupling relations of the components, users can construct a composite model.

- User Interface/Modeling layer. This layer provides a modeling interface for user to construct the models. High level modeling languages and graphical user interfaces (GUI) can be developed for a more user-friendly system.

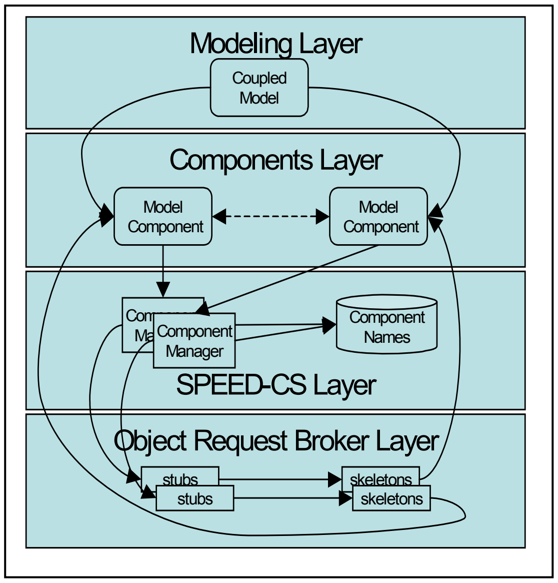

Figure 1: SPEED-CS architecture

2.2 Modeling and Simulation Formalism

In order to facilitate interoperability, we will choose a common formalism for both simulation and decision models. In the SPEED-CS system, we adopt the Discrete Event System Specification (DEVS) formalism which provides a means of specifying a mathematical object representing a discrete-event dynamic system. Such a system has a time base, inputs, states, and outputs, and functions for determining next states and outputs given current states and inputs (Zeigler and Sarjoughian, 2003). While decision models, especially static decision models, may not seem like discrete-event dynamic systems, it is important to view these models as components within a simulation in which they provide outputs when required. Thus, decision models will also be treated as DEVS models.

At the lowest level, an atomic DEVS model describes the autonomous behavior of a discrete-event system as a sequence of transitions responding to external input (event) and internal input (event). At the higher level, a coupled DEVS describes a system as a network of atomic or coupled models. The connections in the network denote how models influence each other (Zeigler and Sarjoughian, 2003).

In our design, both simulation model components and decision model components are developed as atomic or coupled DEVS models. Their common interfaces allow the couplings among the model components in a distributed environment. When the coupled system runs, the components communicate with one another following the parallel DEVS simulation protocol (Zeigler et al., 1999).

2.3 A Common Data Model

In the process of integrating heterogeneous applications, it is important to have a common Data Model as the understandable data format among the applications. A widely used cross-platform, extensible standard for common Data Model is the XML (eXtensible Markup Language).

In the system, we use XML as the standard format to encode the data transferred between different models. The information can be outputs generated by the simulation, or it can be user input data to formulate the models. In order to obtain the data expressed with XML format, a specific parser is designed for this task. The parser seeks keywords for extracting data. As an example, a linear programming model encoded as a XML data stream should express information such as whether it is a maximizing or minimizing problem, how many variables the model has, and others. Therefore the keywords of “*maxOrMin*”, “*numberOfVars*”, and others are introduced to the parser. With these vocabularies, the parser is able to extract information and thus to construct the LP model for the solver.

3 IMPLEMENTATION

The implementation of the architecture involves in choosing the middleware, developing the component management layer, component layer, and the modeling layer. We inherit CORBA’s middleware services to manage remote components. This feature allows the integration of components developed with different programming languages (such as JAVA and C++). In the following we focus our presentation on the implementation of decision model components and SPEED-CS layer, and we briefly introduce the construction of simulation model components.

3.1 Decision Model Components

Decision Model Components are objects that provide decisions, usually solutions of an optimization model, to the system. A decision model component contains APIs to input model data, run the algorithm, and output solutions. In order to deploy over the network, a decision model component needs to export a few functions for remote invocations. We present the design and implementation of using linear programming as a decision-making paradigm. Other decision models can be developed in a similar manner. An LP model contains a set of decision variables, an objective function and a set of constraints. The following formulation represents a general LP.

$$\begin{aligned} &\text{objective function: } \min \mathbf{c}^T\mathbf{x} \\ &\text{subject to: } \mathbf{A}_1\mathbf{x} = \mathbf{b}_1 \\ &\qquad\qquad \mathbf{A}_2\mathbf{x} \geq \mathbf{b}_2 \\ &\qquad\qquad \mathbf{A}_3\mathbf{x} \leq \mathbf{b}_3 \\ &\qquad\qquad \mathbf{lb} \leq \mathbf{x} \leq \mathbf{ub} \end{aligned}$$

where **x** is the decision variables, and vectors **c**, **lb**, **ub**, and matrices $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$ are coefficients.

Equations and inequalities form the set of constraints. Instead of writing a model file in certain common format (such as MPS), the LP model component provides a generic skeleton of a linear programming for callable functions of solvers. Modelers formulate the linear programming and prepare the input dataset. The process of running a linear programming model can be abstracted as the following sequential process.

- Initialization. This procedure sets the solver environment and gives necessary information to the solver such as the number of decision variables, the numbers of equalities and inequalities, and the sense of the objective function (i.e., maximize or minimize) the problem. Solvers may need these data to estimate the size of the problem. They reserve memories based on the problem size and the particular algorithm of the solver.
- Data Input. This procedure populates the component with data to construct decision model, including the objective function, constraints and bounds. For integer or mixed integer programming, the input data needs to indicate to the solver which decision variables should be treated as integer.
- Solve. This procedure solves the linear programming problem after the model is set up. User may be able to choose different algorithms to use. For example, many linear programming solvers allow the user to choose either Simplex method or Interior Point method. It is the user's decision to apply which algorithm to use in solving the problem.
- Output and Sensitivity Analysis. The component outputs the activities and duals when the solve process is accomplished. It can also output the status of the solver and model.

A linear programming model component is defined with the standard interface definition language (idl) to provide remote access from other models. The procedures are specified with several exporting methods in the idl. When compiling the interface file to a programming language, a server program and a client program will be generated. The server program implements the methods and the client program packages the component to a DEVS atomic model with communication information. The atomic model can be coupled with other DEVS models to construct more complicated models. When the atomic model (client) receives an event from another DEVS models, it will respond with the state transitions. The change of the model state may result the running of one of the functions defined in the interface file. For example, the client can be in one of the following states: "passive", "initializing", "waiting for data", "loading data", "optimizing", and "done". Initially the state of the component is "passive". Driven by the event from another model to initialize the algorithm, the component transitions to "initializing" and thus invokes the initialization function. After the initialization is done, the internal state transition updates the state to "waiting for data", which means the client is ready to accept input model data. The transitions go on until the model is solved and the client remains "passive" state unless another event activates the component. When the solving process is done, the solution, including activities and dual values are obtained. The solutions are converted to the XML format and output to designated components.

3.2 SPEED-CS Layer

The SPEED-CS layer is the software to manage the components. The management includes maintaining a list of components, adding or deleting components, and initiating access of remote objects. There are two equally important services supporting the management: the naming service and event service. The naming service provides users with a way to develop applications within a distributed computing environment, without sacrificing the advantages of a local environment. The event service provides controls over the execution of the whole system.

1. Naming Service

CORBA offers the basic naming services and the SPEED-CS layer extends the CORBA naming services to manage the components by maintaining a list of the properties of the components that users may want to reference and invoke. The property list includes the name of the component, the name or IP address of the server computer on which the component is implemented, and the port of the server.

The SPEED-CS layer checks the user-input name with those in the property list. If the user-input name does not match the names in the property list, it means that the requested component is not available. The request of the component is then rejected and an exception is thrown to inform the user that there is a mismatch of the name and component. If there is a match, it means the component is available. The SPEED-CS layer will use the name as a key to query other information of the component such as the computer and the port on which the component server is located. With all the information, the component can be referenced.

It is a straightforward process to maintain the property list. If new components are added to the server, the administrator just needs to add to the property list the name, server machine on which the object is located, and the ports to connect. In this way the SPEED-CS layer is open to variant CORBA objects of different decision models and simulation models.

2. Event Service

The event service in the SPEED-CS platform is realized by the DEVS formalism. DEVS provides an event - based state system, in which the states of an object transition according to the events scheduled as responses to received messages.

Time-management is an important issue to implement with event services. One of the ways is to maintain the SPEED-CS layer clock as a universal clock and thus it can monitor the time advance and event scheduling. The model components utilize DEVS functionalities, such as state transitions, receiving and sending messages, and reacting to received messages, etc. If a received message is to call a method of the remote component, the component first references the remote model by utilizing the naming service and then invokes the method. Modeling with the DEVS formalism also facilitates the hierarchical modeling to couple the components to larger coupled models.

3.3 Simulation Components

The design of simulation components is very similar to that of the decision model components. A simulation component also includes a server program and a client program. The server is modeled with DEVSJAVA and implements the discrete-event simulation application, and the client provides the naming service for users to locate the component. The idl interface declares the DEVS simulator methods, such as executing transition functions, getting output from the models and so on. The SPEED-CS coordinator will call these methods to provide event services when running the integrated system.

4 AN EXAMPLE

In this section, we illustrate the system by presenting an example of product-mix optimization and simulation as an interacting system. A product-mix problem (Schrage, 1997) has a collection of products that compete for a finite set of resources. Associated with each product is a profit contribution per unit, and associated with each resource is availability. The objective is to find how much to produce of each product so as to maximize profits subject to resource constraints. A discrete event simulation will be created for the product-mix problem, where processing time of each product on each machine is a random variable whose distribution is known. However, because of queuing delays, the distribution of the total time at a machine is not known a priori. Because of this, the product mix model uses an estimated average processing time for each machine, and this estimate is updated through interactions between the LP formulation and the discrete-event simulation.

4.1 The LP formulation

In this particular example there are 3 machines and each machine can produce 7 types of products. The machines are not identical and they process each type of product with different processing times. We assume that the jobs are processed with the sequence of operations on machines 1, 2, and 3. Initially the decision model has an estimation of the expected processing time as shown in Table 1. The simulation model has another table of expected processing time as shown in Table 2. The net profit of each product is shown in Table 3. Both the decision model and the simulation model share the same data for profit.

Note that there are several data mismatches, and such discrepancies arise in instances where the optimization model and the simulation model are "owned" by different groups within the organization. Alternatively, one may view the simulation as representing the real world, and the LP model simply an outdated approximation. This exercise then illustrates how the SPEED-CS architecture allows the decision model to match up with more realistic data.

Table 1: Initial Expected Processing Time in Minutes for LP

	Machines		
Products	1	2	3
A	5.0	5.0	5.0
B	7.0	7.0	7.0
C	8.0	7.0	7.0
D	7.0	5.0	3.0
E	6.0	6.0	3.0
F	7.0	5.0	2.0
G	7.0	10.0	2.0

Table 2: Expected Processing Time in Minutes for Simulation

	Machines		
Products	1	2	3
A	12.0	8.0	5.0
B	7.0	9.0	10.0
C	8.0	4.0	7.0
D	10.0	5.0	3.0
E	10.0	6.0	3.0
F	7.0	11.0	2.0
G	7.0	11.0	2.0

Table 3: Net Profit of each Product

Decision Variables	Definition (per week)	Profit (per Unit)
A	Number of units of A	\$3.0
B	Number of units of B	\$3.0
C	Number of units of C	\$3.0
D	Number of units of D	\$3.0
E	Number of units of E	\$2.0
F	Number of units of F	\$3.0
G	Number of units of G	\$2.0
M_1	Hours of machine 1 used	-\$4.0
M_2	Hours of machine 2 used	-\$4.0
M_3	Hours of machine 3 used	-\$3.0

Some other constraints include: at most 20 units each can be produced of products D and E, and each machine can be run 128 hours (a week). There are no lower bounds for the products or the use of machine time. Let x_1 through x_7 be the decision variables of the number of product (A to G) to produce, and let x_8 through x_{10} be the time used on machine 1 to 3. To simplify the problem we model the problem with linear programming and round x_i to integers. The LP model can be represented with equation constraints as follows:

$$\text{Maximize} \sum_{i=1}^{10} p_i x_i$$

subject to

$$A_{eq} x = r_{eq}$$
$$lb \le x \le ub$$

where $x = [x_1 \quad x_2 \quad \cdots \quad x_{10}]^T$ and p_i is the profit of each product and the cost of running each machine. The initial equation coefficient matrix is:

$$A_{eq} = \begin{bmatrix} 5.0 & 7.0 & 8.0 & 7.0 & 6.0 & 7.0 & 7.0 & -60.0 & 0 & 0 \\ 5.0 & 7.0 & 7.0 & 5.0 & 6.0 & 5.0 & 10.0 & 0 & -60.0 & 0 \\ 5.0 & 7.0 & 7.0 & 3.0 & 3.0 & 2.0 & 2.0 & 0 & 0 & -60.0 \end{bmatrix}$$

, and the equation's right hand side is $r_{eq} = [0 \quad 0 \quad 0]^T$. The lower bound of the decision variables is specified as vector lb, which is 0 for both the product variables and machine time usage variables in this example. The upper bound is specified as vector ub: for product D and E (x_4 and x_5) the upper bound is 20.0 and for machine time usages variables the upper bound is 128.0. All variables are non-negative.

4.2 Transferring data from the LP to Simulation

The optimization results are returned with the notations of x_i, while the simulation requires some meaningful descriptions of the results. For this purpose a converter is designed to interpret the decisions to the variables which the simulations can understand. A map file helps the converter link decision variables and simulation variables. The decision variable names are defined as the attribute of the tag "LPVAR". As shown in Figure 2, the simulator uses variable "JobA" for decision variable x_1 to generate job type A. With the mapping the simulator is able to convert the decisions from the LP models to the simulation inputs.

```
<map>
   <from>LP</from>
   <to>Simulation</to>
   <numOfVars>7</numOfVars>
   <LPVAR name="x1"> JobA </LPVAR>
   <LPVAR name="x2"> JobB </LPVAR>
   <LPVAR name="x3"> JobC </LPVAR>
   <LPVAR name="x4"> JobD </LPVAR>
   <LPVAR name="x5"> JobE </LPVAR>
   <LPVAR name="x6"> JobF </LPVAR>
   <LPVAR name="x7"> JobG </LPVAR>
</map>
```

Figure 2: The Mapping for LP to Simulator

4.3 The Simulator

We design a discrete event simulation model to simulate the running of the production system. A job generator generates each kind of jobs with a uniformly distributed inter-arrival time within the range [15.0, 25.0]. The jobs must be processed by machine 1 first, then machine 2, and finally machine 3. Each machine maintains a queue for the jobs and the queue capacity is large enough. Figure 3 depicts the simulation setup in the product-mix example.

The machines are not identical and a machine can process all the types of jobs. A machine's processing time of a certain type of job is a random number generated from a uniform distribution whose expected processing time is shown in Table 2. The production plan comes from the LP decision and the jobs are generated according to the plan. In the simulation, processing time intervals (including delays) of each job on a machine and the counts of each type of jobs are observed. The time intervals are averaged by the number of jobs produced and the results will be sent back to the optimization component for revisions of the production plan. The revised production plan will then trigger a new simulation.

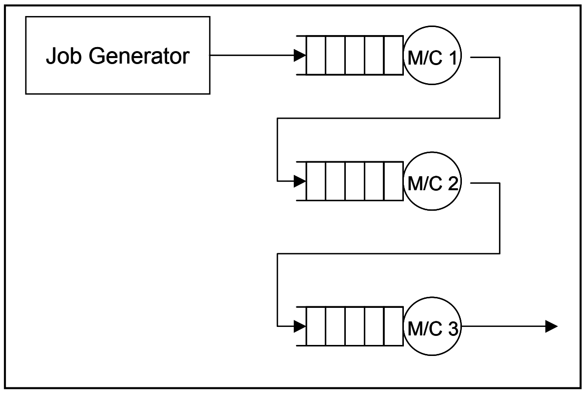

Figure 3: Product-mix Problem Simulation

4.4 Transferring data from the Simulation to LP

The LP model requires generic inputs such as equation matrix, bounds, right hand side vectors, etc. The simulation provides application-related output data, and in this example, the simulation updates the average processing time of each job on each machine.

A converter to translate simulation output to the LP model is therefore necessary. In the implementation, the construction of an LP is partitioned into tasks of loading objective functions, equation matrix, bounds, etc. Therefore we introduce a few keywords for data prepared for these operations. The keywords include "*EquationMatrix*", "*UpperBounds*", "*LowerBounds*", and so on, which are related to the partitioned LP tasks. Other key words are "*columnIndex*" and "*rowIndex*", which is used to map the simulation data to the position of the data in matrix (or vectors) for the LP model. When the converter receives the output data from the simulation, it extracts the data according to the relationship defined in the map file and modifies the coefficient matrices or vectors for the LP.

Figure 4 shows the map file for this example. The simulation outputs the average processing time as shown in Figure 5, which is prepared for the equation matrix A_{eq} in the LP model. Each data element is the average time for a certain job spent on a machine. The job type and machine id are specified in the data attributes. However the tag "avgTime" is not meaningful to the LP model, which is expecting "EquationMatrix" tag with "columnIndex" and "rowIndex". With the help of the map file, the converter can modify corresponding data elements in the equation matrix A_{eq}. The updated matrix will be input to the LP model to modify the constraints $A_{eq}x = r_{eq}$. The updates of bounds and other data entries for the LP model are similar. The map file provides modelers with flexibility to interpret application-related simulation data to generic LP inputs.

```
<map>
  <from>Simulation</from>
  <to>LP</to>
  <EquationMatrix incomingNname="avgTime"
            columnIndex="productid"
            rowIndex="machineid"
            outputName="Aeq">
  </EquationMatrix>
</map>
```

Figure 4: The Mapping from Simulation to LP

```
<avgTime>
  <data productid="1" machineid="1"> 11.97 </data>
  <data productid="1" machineid="2"> 23.07 </data>
  <data productid="1" machineid="3">  4.97 </data>
  <!-- other data points -->
  <data productid="4" machineid="1">  9.76 </data>
  <data productid="4" machineid="2">  4.82 </data>
  <data productid="4" machineid="3">  2.90 </data>
  <!-- other data points -->
</avgTime>
```

Figure 5: Average Time Data

4.5 Simulation Results

In the product-mix example, we design two cases in order to observe the closed-loop simulation-LP system. For both cases we keep job generation process the same and tune the variance of the machine processing time intervals. In one case, the machine processing time intervals are random numbers uniformly distributed with the expected processing time and ±10.0% deviation, whereas in another case the deviation is as much as ±20.0%. In the first case the system starts with data in Table 1 and generates the initial production plan as shown in the first row of Table 4. Then the simulation runs with the plan and the processing time intervals are observed. The average processing time intervals are very close to the data shown in Table 2 because there are no delays and the system converges after 6 iterations. The updates of the production plan are shown in Table 4.

In the second case the machines process jobs with more variance while the jobs are generated with same rate as in case 1. The simulation starts with the same initial plan as in the first case. The variance causes delays in the system, which as a result contributes to differences of the averaged processing time intervals from case 1. As shown in Table 5, the system takes more iterations to converge and it converges to a different plan from case 1 because delays affect the processing time intervals. The cases of the example have been run with different sets of seeds and the evolvements of the production plans are consistent for the runs.

Table 4: Production Plans with ±10.0% Deviation from the Average Processing Time

	Job Types						
Updates	A	B	C	D	E	F	G
1	1000	343	0	20	0	20	0
2	0	0	943	0	0	20	0
3	0	0	944	0	0	20	0
4	0	0	942	0	0	20	0
5	0	0	943	0	0	20	0
6	0	0	943	0	0	20	0

The example shows the ability of the architecture to achieve interoperability between LP model components and simulation components to form more complicated systems. If the LP model is complicated, the convergence of the system can become difficult, and in some situations the LP may not be able to find a feasible solution. The convergence is highly related to the applications.

Table 5: Production Plans with ±20.0% Deviation from the Average Processing Times

	Job Types						
Updates	A	B	C	D	E	F	G
1	1000	343	0	20	0	20	0
2	0	0	943	0	0	0	0
3	0	325	0	20	523	0	0
4	0	0	0	20	0	0	671
5	0	333	0	0	0	20	0
6	333	0	0	0	0	0	0
7	0	0	307	0	0	0	0
8	0	0	296	0	0	0	0
9	0	0	287	0	0	0	0
10	0	0	287	0	0	0	0

5 CONCLUSIONS

This paper proposes a distributed programming architecture for operations research studies that involve optimization models and discrete event simulations models. The architecture is a multi-layer system, which is composed of a middleware layer, the component management layer, component layer, and modeling layer from bottom up. We demonstrate the design of such a system by discussing a product-mix optimization and simulation interactive system. The example includes the implementations of each layer, the methods to manage each layer, and the converters between LP and simulations. The system is able to provide services to facilitate distributed computing, event services, naming services, and component management. We use XML as the common data format for the components. Another important feature is that the component sets can be updated and enlarged with different models adding in, as long as the models can be modeled as a discrete event model. This feature, however, raises the important issue of security of the system. There have been many research results on generic security issue on network systems, which can be implemented in the SPEED-CS architecture in the future work.

ACKNOWLEDGMENT

The research reported in this paper is supported under the Next Generation Software program of the National Science Foundation: DMI 9978780 and CISE 9975050. The authors also gratefully acknowledge financial support by a contract F49620-03-1-0477 from the AFOSR/MURI.

REFERENCE

Alrefaei, M.H. and S. Andradóttir. 1999. A simulated annealing algorithm with constant temperature for discrete stochastic optimization. *Management Science* 45 (5): 748-764.

Andradóttir, S. 1996. A global search method for discrete stochastic optimization. SIAM Journal on Optimization 6: 513-530.

Bertsimas, D. and D. Gammarnik. 1999. Asymptotically optimal algorithm for job shop scheduling and packet routing. *Journal of Algorithms* 33: 296-318.

Blomvall, J. 2003. A Multistage Stochastic Programming Algorithm Suitable for Parallel Computing. *Parallel Computing* 29 (4): 431-445.

Boudoukh, T., M. Penn, and G. Weiss. 2001. Scheduling job shop with some identical or similar jobs. *Journal of Scheduling* 4: 177-199.

Chen, H. and A. Mandelbaum. 1991. Discrete flow networks: Bottleneck analysis and fluid approximation. *Mathematics of Operations Research* 16 (2): 408-445.

Fu, M. C. 2002. Optimization for Simulation: Theory vs. Practice. *INFORMS Journal on Computing* 4 (3): 192-215.

Gelfand, S. and S. Mitter. 1989. Simulated annealing with noisy or imprecise energy measurements. *Journal of Optimization Theory and Applications* 62: 49-62.

Gürkan, G., A.Y. Özge, and S.M. Robinson. 1999. Sample-path solution of stochastic variational inequalities. *Mathematical Programming* 84: 313-333.

Gutjahr, W.J. and G.C. Pflug. 1996. Simulated annealing for noisy cost functions. *Journal of Global Optimization* 8: 1-13.

Lee, E.K. 2004. Generating Cutting Planes for Mixed Integer Programming Problems a Parallel Computing Environment. *INFORMS Journal on Computer* 16 (1): 3-26.

Pichitlamken, J. and B.L. Nelson. 2003. A Combined Procedure for Optimization via Simulation. *ACM Transactions on Modeling and Computer Simulation* 13 (2): 155-179.

Plambeck, E.L., B. -R. Fu, S.M. Robinson, and R. Suri. 1996. Sample-path optimization of convex stochastic performance functions. *Mathematical Programming* 75: 137-176.

Schrage, L., 1997. Optimization Modeling with LINDO. Duxbury Press.

Sen, S., R.D. Doverspike, and S. Cossares. 1994. Network planning with random demand. *Telecommunication Systems* 3: 11-30.

Suri, R. and Y.T. Leung. 1989. Single Run Optimization of Discrete Event Simulations — an Empirical Study Using the M/M/1 Queue. *IIE Transactions* 21: 35-49.

Yan, D. and H. MuKai. 1992. Stochastic Discrete Optimization. *SIAM Journal of Control and Optimization* 30(3): 594-612.

Zeigler, B.P., G. Ball, H. Cho, J.S. Lee, and H. Sarjoughian. 1999. Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions. In *Simulation Interoperation Workshop (SIW)*. Available online via http://acims.arizona.edu/PUBLICATIONS/SIW99Paper/SIWDEVSImplemHLARTI.pdf [accessed December 20, 2004].

Zeigler, B.P. and H.S. Sarjoughian. 2003. Introduction to DEVS Modeling & Simulation with JAVA: Developing Component-based Simulation Models. Available online via http://acims.arizona.edu/SOFTWARE/devsjava_licensed/CBMSManuscript.zip [accessed January 16, 2005].

AUTHOR BIOGRAPHIES

YIJIA XU is a Ph.D. student in Department of Systems and Industrial Engineering at the University of Arizona. His research interests include distributed simulation architecture and mobile agent system design and analysis. He is a member of INFORMS Simulations Society. His email address is `yxu@sie.arizona.edu`.

SUVRAJEET SEN is a professor of Systems and Industrial Engineering at the University of Arizona. His research is devoted to the theory and applications of large-scale optimization algorithms, especially those arising in stochastic programming. He has also applied these methods to practical problems arising in airlines, electric power, mining, telecommunications and transportation. He is a former Chair of the INFORMS Telecommunications Section and founded the INFORMS Optimization Section. Professor Sen's email address is `sen@sie.arizona.edu`.