DEVELOPMENT OF AN AGENT-BASED FACTORY SHOP FLOOR SIMULATION TOOL

Albert J. Whangbo

Advisors
Dr. Edward Lin
Dr. Jeffrey Herrmann

Computer Integrated Manufacturing Laboratory
Institute for Systems Research
University of Maryland at College Park

August 6, 1999

SUMMARY

Manufacturing systems of the future are expected to be agile and failure-tolerant. Current simulation tools are not well equipped to model these dynamically changing systems. Agent-based simulation represents an attractive alternative to traditional simulation techniques. This project aims to develop software for agent-based factory shop floor simulation. The current version of the factory simulation software is implemented in Java. Although the program lacks important features of a decision support tool, it provides a flexible agent-based framework for modeling and testing shop floor configurations.

TABLE OF CONTENTS **PAGE**

SUMMARY	ii
INTRODUCTION	1
BACKGROUND	2
Factory Simulation	2
Software Agents	2
The Cybele Infrastructure for Autonomous Agents	3
PROGRAM STRUCTURE	4
Overview	4
Shop Floor Agents	4
The Display Agent	6
Agent Interaction Functions	6
Flow Control Rules and Methods	6
The Graphical User Interface	7
Program Outputs	8
EXAMPLE SIMULATION RUN	8
SHORTCOMINGS AND FUTURE ENHANCEMENTS	12
General Simulation Problems	12
Agent Function Problems	12
User Interface Problems	13
ACKNOWLEDGEMENTS	13
BIBLIOGRAPHY	14
APPENDIX: PROGRAM DOCUMENTATION	15

LIST OF FIGURES **PAGE**

Figure 1	Organization of the program components	4
Figure 2	Interaction of shop floor agents	5
Figure 3	A "push" interaction implemented with Cybele	7
Figure 4	A "pull" interaction implemented with Cybele	7
Figure 5	The initial dialog box	7
Figure 6	Representation of the factory shop floor	8
Figure 7	Initial dialog box	9
Figure 8	Shop floor diagram	9
Figure 9	Parameter entry dialog for part generator agents	9
Figure 10	Arrival pattern dialog	10
Figure 11	Process plan dialog	10
Figure 12	Queue agent parameter dialog	10
Figure 13	The shop floor diagram during a simulation run	11
Figure 14	Statistics dialog	11
Figure 15	Log file output	11

INTRODUCTION

Manufacturing systems of the future are expected to be agile and failure-tolerant. Furthermore, these systems should be rapidly configurable to meet demands for custom products produced efficiently in small lots (Nagel and Dove, 1991).

Current simulation tools are not well equipped to model these dynamically changing systems. Presently, simulations must be modified to reflect each change in a simulated environment. These software changes are cost- and labor-intensive. Agent-based simulation represents an attractive alternative to traditional simulation techniques. Agent-based systems are potentially more robust, adaptable, and flexible than conventional models because tasks are carried out locally and in a modular fashion (Agre, et al., 1995).

The goal of this project is to develop software for agent-based factory shop floor simulation. The shop floor will be modeled as a community of autonomous, interacting agents. The agent controls will be separate from the user interface. This structure will ensure that the agent framework remains modular and transparent to the user. Finally, the program will lend itself to operation over a network of computers.

The simulation software will be a versatile tool for factory shop floor design and analysis. The modularity of the agent-based system will allow users to build and test shop floor models of nearly any size or complexity. The software will help users gauge shop floor performance by collecting a variety of simulation statistics. In addition, users will be able to alter the shop floor environment during a simulation run. This function will help users to forecast how events such as changes in equipment or capacity affect the factory performance.

The current version of the factory simulation software is implemented in Java and runs on a single processor. Although the program lacks important features of a decision support tool, it provides an agent-based framework for modeling and testing shop floor configurations. The program employs Cybele, a software package developed by Intelligent Automation, Incorporated, to support essential agent functions such agent creation and messaging. A simple graphical user interface (GUI) accepts shop floor parameters from the user and updates the status of the factory components during the simulation. The GUI also reports some basic simulation statistics.

This report provides some background information on factory simulation and software agents. The report also details the structure of the current factory simulation program. Finally, a sample simulation run and program deficiencies are discussed.

BACKGROUND

Factory Simulation

Computer simulation is a popular technique for modeling and analyzing manufacturing systems. While mathematical programming models of manufacturing systems are often unfeasible, simulations can provide insights into the complex and often unpredictable behaviors of these systems (Kassicieh et al., 1997). Simulation is also a powerful tool for forecasting the effects of changing system characteristics. The results of factory simulations can provide decision-makers with valuable cost and production information (Krishnamurthi et al., 1997).

Many approaches to simulation exist. In discrete event simulation (DES), the state of the modeled system is assumed to change only at discrete points in simulated time. Changes in the system's state are dependent on the current state of the system and on events such as messages or component failures. Events in a discrete event simulator are processed and managed by a global event queue, which can activate and deactivate system components as needed during the simulation (Craig, 1996). Each event in the queue bears a time stamp indicating when the event is to be processed. The discrete event simulator operates by continuously removing from the queue and processing the event with the smallest time stamp. In this sense, discrete event simulations are sequential.

In a distributed simulation, the system is divided into parts that are then modeled in parallel. Each component of a system is assigned a process that runs on a local clock. Managing all the processes on a global scale requires careful synchronization. Perfect synchronization is very difficult to achieve, since it may be difficult or impossible to determine the precedence of two local events on a global scale. Despite these challenges, distributed simulation offers a means of modeling large manufacturing systems in which equipment, information, and expertise are distributed due to physical separation.

Software Agents

Software "agents" are software entities that operate continuously and autonomously, often within a community of other agents (Bradshaw, 1997). Individual agents often possess specialized knowledge or expertise that can be enlisted by other agents in the community. Ideally, agents require minimal human intervention or guidance; instead, agents should be responsive to changes in their environments. Furthermore, agents within a community should be able to interact with each other and with humans to complete their individual problem solving tasks or to collaborate on larger problems.

Agent-based systems are software systems that are designed and implemented in terms of agents. Because of their abilities to perform specialized functions and to share information, agents represent a powerful tool for making systems modular and scalable (Jennings & Wooldridge, 1998). Agents can be added to, removed from, or relocated

within a system as needed. Agent-based systems are therefore appropriate for modeling complex, changing, or unpredictable situations.

Agent-based systems are naturally suited for simulating environments in which information, expertise, and resources must interact but are physically distributed. In such cases, real-world entities and their interactions can be mapped directly into autonomous agents with their own resources and expertise, and which are able to interact with other agents to complete tasks.

Because of their ability to effectively simulate systems with distributed resources and expertise, agents are finding many applications to manufacturing. For example, a company may have many factories with different capabilities, and each physically separated from the others. Furthermore, each individual factory may contain a variety of different components, or operate under different resource or time constraints. A multi-agent approach, in which each factory and factory component is assigned an agent, can be used to model this system. Each agent can be given a set of plans representing its individual capabilities. The agents can interact with each other to either simulate or coordinate the entire factory network.

The Cybele Infrastructure for Autonomous Agents

Cybele, a software package developed by Intelligent Automation, Incorporated, provides an infrastructure for developing and running agent-based applications. Cybele is comprised of methods for agent creation, deployment, communication, and other functions. While Cybele supports essential agent roles and interactions, it remains separate from the application domain. Cybele is therefore well suited for use in a wide range of agent-based applications.

An important feature of Cybele is its subject-based messaging scheme. Rather than addressing a message to specific recipients, an agent posts its message to a subject string. All agents subscribing to this subject receive the message. This method forestalls dilemmas such as misdirected messages or irrelevant communications. More importantly, the subject-based addressing mechanism is independent of the location of the sender or recipient agents within the network. This feature allows agents to be added, removed, or relocated without disrupting the flow of messages.

Messages contain any number of "key-value" string pairs. "Keys" are analogous to labels, while "values" contain actual data. Key-value pairs are bundled with a new message before it is sent to a subject string. Recipient agents may extract the values by referring to the corresponding keys.

Messages and other events in Cybele are handled by an “activity.” Activities are analogous to discrete event simulation controllers in that they maintain event queues and process events. However, activities dispatch messages and invoke callback functions without regard to discrete state changes. Furthermore, the activity does not process

events according to a global schedule; it processes events as they arrive in the event queue.

PROGRAM STRUCTURE

The current factory simulation software is implemented in Java. The program is a collection of classes that define the shop floor agents, a display agent, agent interactions, and the graphical user interface (GUI). Interactions between the simulation components are achieved over the Cybele agent framework.

Overview

The relationships between the program components are illustrated in Figure 1. The user interacts with the GUI to set the agent characteristics and part process plans. The GUI passes this information to the display agent, which in turn creates the shop floor agents. The shop floor agents execute the bulk of the simulation. During a simulation run, the shop floor agents access the interaction functions to perform frequent tasks such as querying the status of other agents, passing parts, and collecting statistics. The shop floor agents send production information back to the GUI via the display agent. The user is then able to monitor the state of the shop floor model during the simulation.

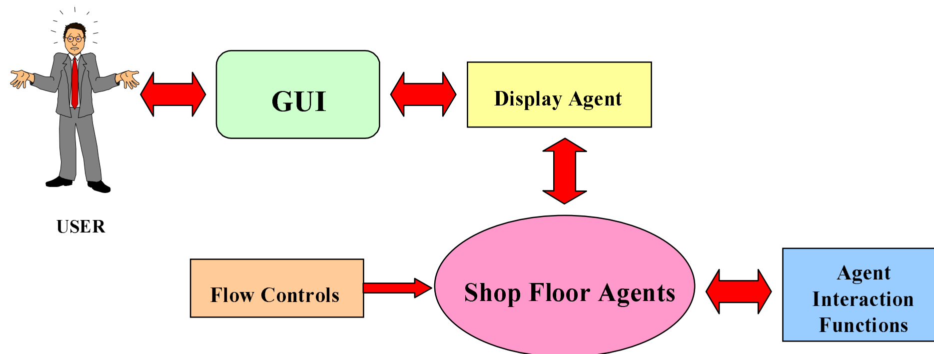


Figure 1. Organization of the program components

In addition to the information displayed in the GUI, the program outputs a log file of parts passed during the simulation. This file can be used to track individual parts through their process plans.

Shop Floor Agents

The program models the factory shop floor as a collection of autonomous agents. There are three types of shop floor agents: part generator agents, queue agents, and machine agents.

Part generator agents are responsible for introducing new parts onto the shop floor. Each part generator agent is assigned a unique part type. The agent also stores the

processing sequence, or process plan, that is shared by all parts of that type. The machine agents refer to this process plan to maintain the proper routing of parts through the shop floor. Part generator agents use a repeating timer to create new parts. For every part creation cycle, the timer interval is set to a uniformly distributed random value between two user-defined bounds. This method simulates random variations in part arrival times.

Each queue agent is paired with a machine agent. A queue-machine pair represents a single node in the shop floor network. Queue agents provide buffers for their corresponding machine agents. Queues are implemented in Java as vector data structures. When a queue agent receives a part, it checks the status of its companion machine agent. If the machine agent is busy, the queue agent will hold the part until the machine becomes available. On the other hand, if the machine is idle, then the queue passes the part to the machine for processing. Prior to the simulation, the user must define the size of each queue.

Machine agents simulate the processing of the parts. Machine agents have two states: idle and busy. A machine agent is busy if it currently processing a part. If a machine agent is idle, then it is eligible to receive a part from its queue agent for processing. Part processing is simulated by a time delay. The delay interval, which is similar to the part arrival interval, is set to a uniformly distributed random value between two user-defined bounds. A machine agent is also responsible for sending a processed part to the next node in its process plan. Because a single machine agent may process many different kinds of parts, the agent must recognize the type of part currently being processed, as well as the progress of the part relative to its unique process plan. Once this information is collected, the machine agent refers to the process plan in the appropriate part generator agent to determine the identity of the next node in the sequence. Finally, the machine agent passes the part to the node.

Figure 2 illustrates the properties of the shop floor agents, as well as the general flow of parts through the shop floor.

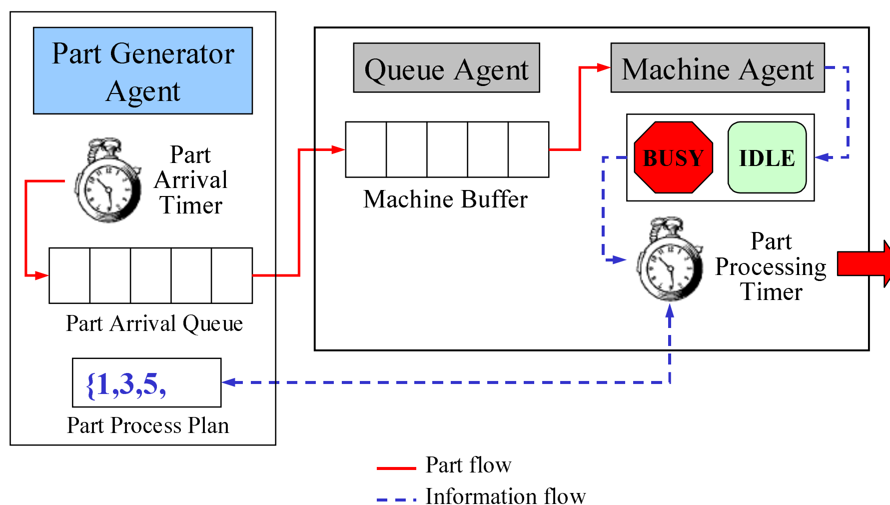


Figure 2. Interaction of shop floor agents

The Display Agent

The display agent is responsible for starting and stopping the simulation. When the user clicks the "Run Simulation" button in the GUI, the display agent creates the shop floor agents based on the user-defined constraints. The display agent tallies the agents as they are created. Once the display agent detects that all the shop floor agents are present, it commands the part generator agents to begin producing parts, thereby starting the simulation. When the user clicks the "Exit" button, the display agent collects several production statistics and terminates the simulation.

Agent Interaction Functions

The factory simulation program supports a number of functions that are useful for performing shop floor interactions. These functions, which are accessible to all of the shop floor agents, encapsulate behaviors that are performed repeatedly during a simulation run. Examples include status inquiries, passing parts between agents, collecting statistics, writing to an output file, and sending information to GUI. By expressing these often-used utilities as Java classes, instances of redundant code are avoided.

Flow Control Rules and Methods

Several rules govern the flow of parts through the simulated factory. Parts are stored in and dispatched from agent queues using "first-in, first-out" (FIFO) logic. Under FIFO rules, parts are processed and passed in the order that they are received. Parts received by an agent are placed at the end of the agent's vector. The first part in the vector ("first in") is always processed and passed forward first ("first-out").

The passing of parts between agents follows two sets of rules. The first set of rules, called "push rules," governs the movement of parts forward through the simulated factory. The second set of rules, called "pull rules," moderates the retrieval of parts from preceding agents in the process plan. Implementing push and pull rules simultaneously minimizes the idle times of the agents.

At every step in the process plan, the current agent checks the status of the next agent in the sequence. If the next agent is available, the current agent sends or "pushes" a part to that agent. If the next agent is unavailable, the current agent holds the part in its vector. The current shop-floor simulation program employs Cybele's messaging framework to check agent statuses and to send parts. Figure 3 illustrates a "push" interaction between two agents.

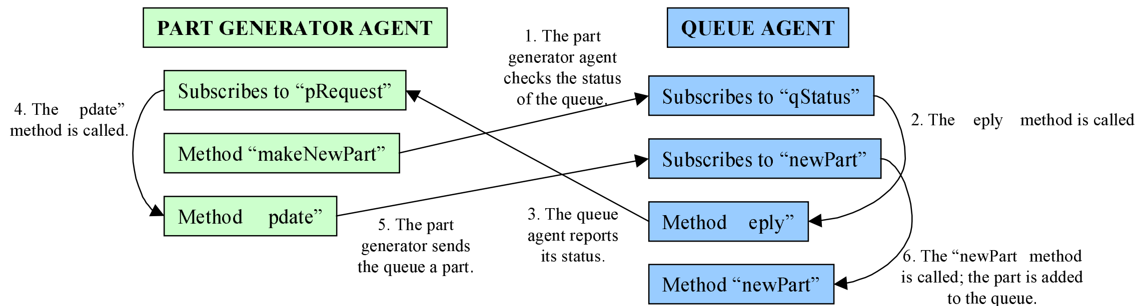


Figure 3. A “push” interaction implemented with Cybele
NOTE: The Cybele activity structure is omitted for clarity.

After each agent processes and sends a part, it checks its own status. If it is eligible to receive a new part, the agent will request and “pull” a part from the previous agent in the process plan. Pull rules can become complicated if an agent receives parts from several different sources; the agent must keep an inventory of all its sources, and broadcast part requests to only those agents. Figure 4 illustrates a pull interaction between two agents as implemented using the Cybele messaging framework.

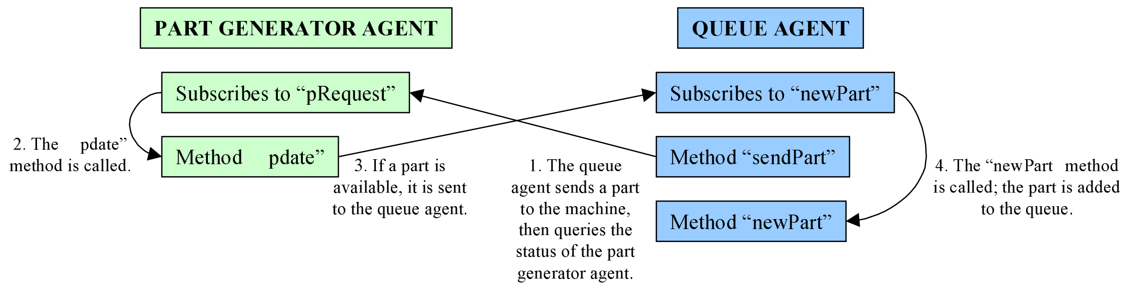


Figure 4. A “pull” interaction implemented with Cybele
NOTE: The Cybele activity structure is omitted for clarity.

The Graphical User Interface

The current graphical user interface (GUI) serves two functions: to gather simulation parameters from the user, and to return production information. When the program is run, the initial dialog box (Figure 5) prompts the user to enter the number of part types and the number of nodes in the shop floor model.

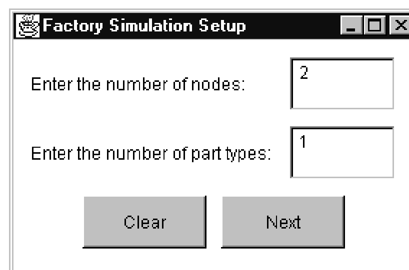


Figure 5. The initial dialog box

The GUI then generates and draws a representation of the shop floor layout. The layout is displayed as a window containing icon "buttons" to represent each component on the shop floor (Figure 6). Different icons indicate whether a component is a part generator, queue, or machine agent. The user clicks on an icon to define the corresponding agent's characteristics. For part generator agents, the user provides part arrival characteristics, a process plan, and processing time distributions for each step in the process plan. The user must also specify the capacity of each queue agent. At present, machine agents do not require any user inputs. However, as the program is enhanced to include more functions, some user-defined machine characteristics may be required.

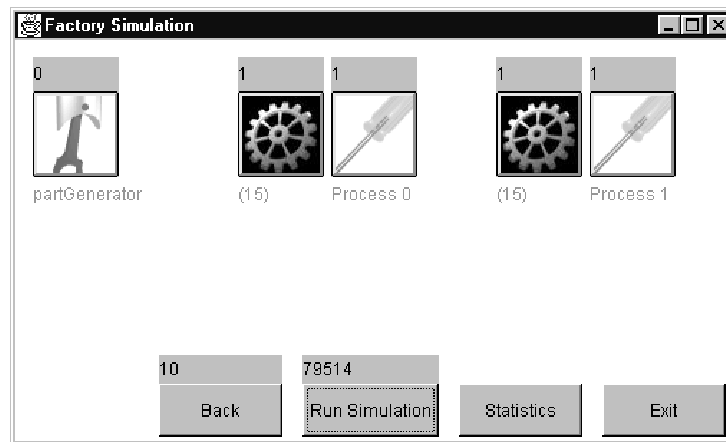


Figure 6. Representation of the factory shop floor

During a simulation run, the GUI displays the number of parts currently held by each agent. This makes it possible for the user to observe the accumulation of parts in queues, as well as the statuses of the machine agents. The GUI also displays the total number of outputs and the simulation duration.

Program Outputs

In addition to the information displayed by the GUI, the program generates a text file containing a log of parts passed during the simulation. Each line in the output file represents the passing of a single part from one agent to another. The line of text includes details such as the sender agent, the recipient agent, the part ID number, the time the part arrived at the sender, and the time the part was passed. The output file allows the user to trace the paths taken by the parts through the shop floor.

EXAMPLE SIMULATION RUN

This section demonstrates the current capabilities of the simulation software. Suppose the user wishes to simulate a simple shop floor containing two nodes that process a single part type in sequence. In the initial dialog box (Figure 7), the user must

specify the number of nodes and the number of part types. Once this information is entered and the user clicks "Next," the program generates and displays a diagram of the shop floor in a second dialog box (Figure 8).

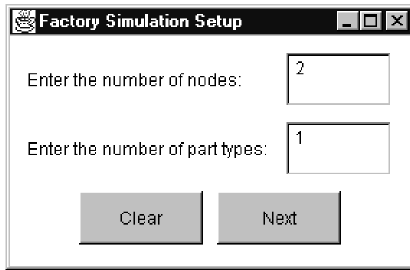


Figure 7. Initial dialog box

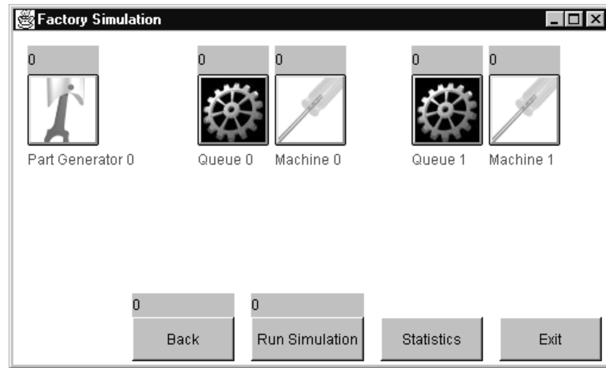


Figure 8. Shop floor diagram

Red button labels serve as visual cues for the user to enter simulation parameters. To edit an agent's characteristics, the user clicks on its icon button. When the user clicks on the "Part Generator 0" button, the following dialog box appears:

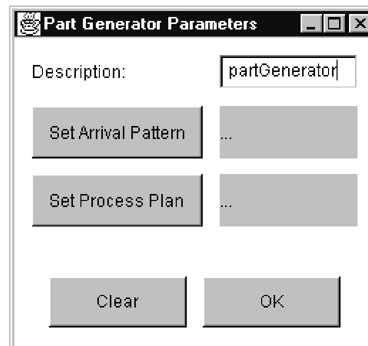
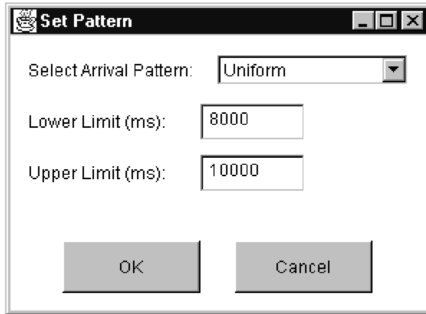


Figure 9. Parameter entry dialog for part generator agents

The "Part Generator Parameters" dialog allows the user to enter both the part arrival characteristics and the process plan associated with the part generator. To define the part arrival constraints, the user clicks the "Set Arrival Pattern" button. A new window opens (Figure 9). The user selects a distribution type from the "Select Arrival Pattern" list, and then enters the upper and lower bounds of the arrival interval. The arrival characteristics are stored when the user clicks the "OK" button.

To set the process plan, the user clicks the "Set Process Plan" button in the "Part Generator Parameters" dialog. The "Set Process Plan" dialog opens (Figure 10). To add a machine to the process plan, the user highlights the desired machine from the "Machines Available" list and clicks the "Add" button. The selected machine now appears in the "Process Plan" list. This procedure is repeated until the user has added all the desired machines to the process plan. Machines can be removed from the process

plan in a similar manner: the user selects the unwanted machine from the "Process Plan" list, then clicks the "Remove" button.



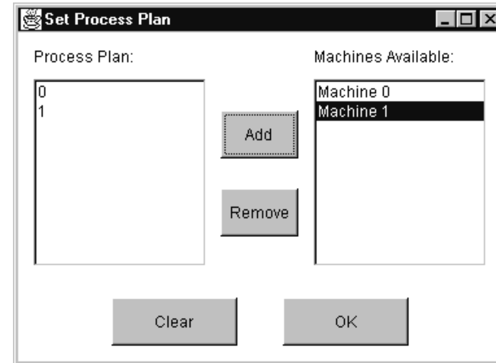


Figure 10. Arrival pattern dialog

Figure 11. Process plan dialog

In addition to specifying the process plan, the user must enter the time required for each machine in the plan to process the part. By clicking on the name of a machine in the "Process Plan" list, the user brings up a "Set Pattern" dialog box. The user enters processing time characteristics in the same manner as the part arrivals. This procedure is repeated until each machine in the process plan has an associated processing interval. The process plan is set once the user clicks the "OK" button in the "Set Process Plan" dialog. At this point, the part generator is fully defined. If there are multiple part generators, the entire procedure must be repeated until each part generator is defined.

The user must specify the capacity of each queue. When a queue icon button is clicked, the "Queue Parameters" dialog appears (Figure 12). This dialog box prompts the user to enter the capacity of the queue.

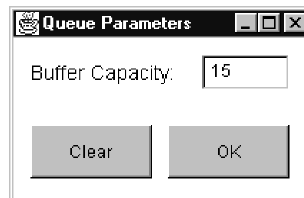


Figure 12. Queue agent parameter dialog

When the specifications for a shop floor agent are entered, the text label for that agent becomes green. Once all of the text labels are green, it is possible to run the simulation by clicking the "Run Simulation" button.

During a simulation run, the labels above the shop floor components reflect the number of parts held by each agent. The labels above the "Back" and "Run Simulation" buttons display the total output and total elapsed time, respectively. The screen below (Figure 13) shows that the part generator agent currently holds no parts, while each of the

other agents is holding a single part. Both machines are in the "busy" state. The total output after approximately 80,000 milliseconds (80 seconds) is 10 units.

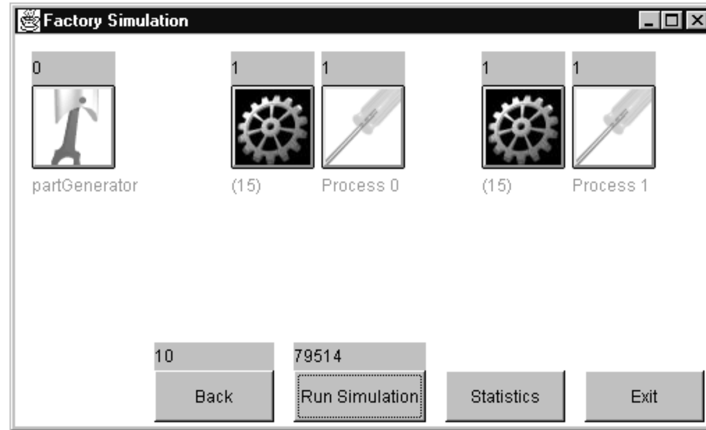


Figure 13. The shop floor diagram during a simulation run

Statistics such as throughput, average waiting time, and average processing time can be viewed by clicking the "Statistics" button during a simulation run. The statistics are not updated dynamically; each time the "Statistics" button is pressed, the "Simulation Statistics" box offers a "snapshot" of the shop floor performance (Figure 14).

The program also outputs a text file (Figure 15). This file can be accessed from the main program directory. Each line in the text file describes the passing of a part between two agents. The columns from left to right represent the sender, recipient, part ID number, time of arrival at the sender, and time sent.

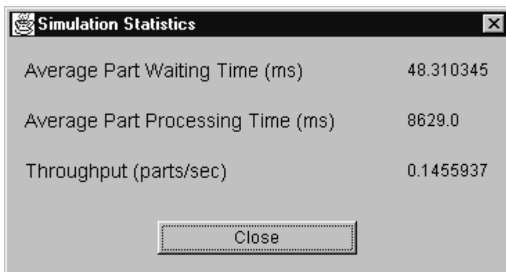
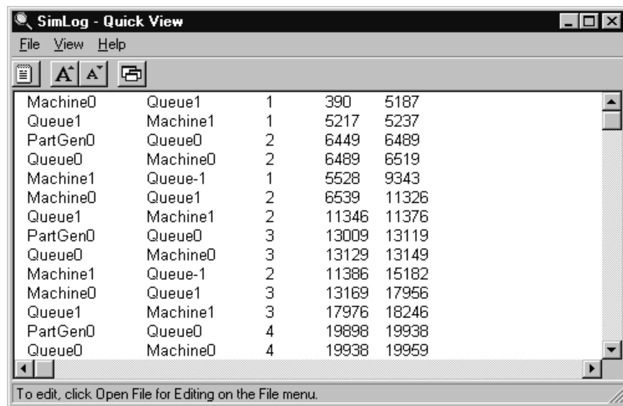


Figure 14. Statistics dialog



SimLog - Quick View

File View Help

Machine0	Queue1	1	390	5187
Queue1	Machine1	1	5217	5237
PartGen0	Queue0	2	6449	6489
Queue0	Machine0	2	6489	6519
Machine1	Queue-1	1	5528	9343
Machine0	Queue1	2	6539	11326
Queue1	Machine1	2	11346	11376
PartGen0	Queue0	3	13009	13119
Queue0	Machine0	3	13129	13149
Machine1	Queue-1	2	11386	15182
Machine0	Queue1	3	13169	17956
Queue1	Machine1	3	17976	18246
PartGen0	Queue0	4	19898	19938
Queue0	Machine0	4	19938	19959

To edit, click Open File for Editing on the File menu.

Figure 15. Log file output

SHORTCOMINGS AND FUTURE ENHANCEMENTS

The factory simulation program's shortcomings can be grouped into three categories: general simulation problems, agent function problems, and user interface problems.

General Simulation Problems

- *The simulation lacks proper exit conditions*. The program should allow the user to specify an exit condition, such as a specific simulation duration or output amount. This could be easily implemented by checking the elapsed simulation time or the total output of all the machines, respectively.
- *The simulation cannot run at faster than real time*. At present, simulation runs are carried out in real time; a 1:1 ratio exists between actual time and simulation time. To simulate very long processing runs in a practical fashion, the program should be able to run at faster than real time.
- *The simulation cannot save factory configurations*. The current program does not support the saving or loading of factory shop floor configurations. A new shop floor must be defined before every simulation run. In the future, the ability to save and load configurations will be essential, particularly for users who are modeling large and complex factory systems.

Agent Function Problems

- *The shop floor agents are not equipped to handle situations in which more than one part is required to begin a processing task*. This is not very realistic, since it is likely that some assembly operations require several different parts and/or more than one part of the same type. Because machine agents in the current program are very simply defined, steps should be taken to allow the user to define specific functions for each machine agent. Furthermore, queue agents must be given the ability to check the types of parts stored in their vectors, so that the proper combinations of parts can be bundled and passed to the machines for processing.
- *Machine agents lack the ability to check the status of forward queues prior to pushing parts*. Currently, machine agents automatically push parts to the next queue in the process plan. This can result in queues overflowing. In the future, machine agents should possess the ability to check the status of forward queues, and remain busy if the queues are ineligible to receive processed parts.
- *Some shop floor agents do not implement pull rules*. Pull rules are currently implemented between part generators and the first queue agents in each process plan. These queue agents pull parts from part generator agents. Machine agents are also capable of pulling parts from their queue agents. However, queue agents lying further along the process plan do not employ pull rules. This flaw arises from the fact that these queue agents may receive parts from multiple nodes. Locating and notifying these nodes, as well as handling the subsequent passing of parts, are difficult and inefficient tasks. Nevertheless, methods for fully implementing pull rules will be necessary to provide the user with a flexible decision support tool.

User Interface Problems

- *The GUI does not check dialog boxes for errors*. The GUI does not currently check the dialog boxes for valid parameters. As a result, it is possible for users to enter parameters under which the simulation program cannot run. Future versions of the GUI must perform error checking to ensure proper parameter entry.
- *Part passing rules are currently hard-coded into the program*. The factory simulation tool currently makes use of FIFO, push, and pull rules to store and pass parts. To make the program more flexible and useful, the user should be given the option of manually selecting or defining these rules.
- *Shop floor parameters cannot be changed during a simulation run*. Currently, the parameters entered prior to the beginning of the simulation remain fixed until the simulation terminates. This limitation prohibits the user from imposing changes on the environment such as machine failures or additions, changing queue capacities, or variations in part arrival or processing intervals. In the future, it will be important to allow the user to make such changes in the middle of a simulation run.
- *The GUI does not clearly illustrate the flow of parts through the shop floor*. The passing of parts between agents is very difficult for the user to detect, as the GUI does not provide any visual cues for the different routes through the shop floor. The GUI should also be capable of displaying other important shop floor properties, including the status of the machines and queues.

ACKNOWLEDGEMENTS

I would like to thank Dr. Edward Lin for his constant patience and guidance. In addition, I would like to thank Dr. Jeffrey Herrmann for his many insightful ideas and suggestions for the project. Dr. Kutluhan Erol of Intelligent Automation, Incorporated, deserves thanks for providing useful information about the nuances of the Cybele software.

BIBLIOGRAPHY

Agre, J., et al. “Autoconfigurable Distributed Control Systems.” ISADS 95, Second International Symposium on Autonomous Decentralized Systems, April 25-27, 1995, Phoenix, Arizona, USA : Proceedings. Los Alamitos, CA: IEEE Computer Society Press, 1995. 162-168.

Balasubramanian S. and Norrie D.H. “A Multi-Agent Intelligent Design System Integrating Manufacturing and Shop-Floor Control.” Proceedings of First International Conference on Multi-Agent Systems (ICMAS '95), San Francisco, California, June 12-14, 1995. 3-9.

Bradshaw, Jeffrey, ed. Software Agents. AAAI Press/MIT Press, 1997.

Cheng, Z., Capretz, M., and Minetada Osano. “A Model for Negotiation Among Agents Based on the Transaction Analysis Theory.” ISADS 95, Second International Symposium on Autonomous Decentralized Systems, April 25-27, 1995, Phoenix, Arizona, USA : Proceedings. Los Alamitos, CA: IEEE Computer Society Press, 1995. 427-433.

Craig, Donald. Discrete Event Simulation Page. Memorial University of Newfoundland. 8 July 1996. <http://www.cs.mun.ca/~donald/msc/node11.html>

Fulkerson, B., and Van Parunak. “The Living Factory: Applications of Artificial Life to Manufacturing.” ISADS 95, Second International Symposium on Autonomous Decentralized Systems, April 25-27, 1995, Phoenix, Arizona, USA : Proceedings. Los Alamitos, CA: IEEE Computer Society Press, 1995. 391-397.

Hamilton, John A., Jr., Nash, David and Udo W. Pooch. Distributed Simulation. Boca Raton, FL: CRC Press, 1997.

Jennings, N., and Michael J. Wooldridge, ed. Agent Technology Foundations, Applications, and Markets. Springer-Verlag, 1998.

Kassicieh, Suleiman K., Ravinder, H. V., and Steven A. Yourstone. “A Decision Support System for the Justification of Computer-Integrated Manufacturing.” Manufacturing Decision Support Systems. 1997.

Krishnamurthy, M., Jayashankar, R., and Don T. Phillips. “A Generalized Cost Analysis System for Manufacturing Simulation.” Manufacturing Decision Support Systems. 1997.

Nagel, R. and R. Dove. “21st Century Manufacturing Enterprise Strategy: An Industry-Led View.” Iacocca Institute, Lehigh University, 1991.

Appendix: Program Documentation

`public class` **`FactorySimulation2`**

Description Contains the `main` method for the simulation program. Starts up Cybele, creates `TheDisplayAgent`, and opens the first dialog box.

Constructors No constructors.

Methods `Public static void` **`main`**`(String args[])`

Starts up Cybele, creates the log file, and creates `TheDisplayAgent`. Creates and opens an instance of the `Setup1` dialog box.

`public class` **`TheDisplayAgent`** `implements IAIHandler`

Description `TheDisplayAgent` uses information from the graphical user interface to create the other factory agents. The `TheDisplayAgent` class also contains methods for starting and ending the simulation.

Constructors `Public` **`TheDisplayAgent`**`(String title)`

Subscribes `TheDisplayAgent` to "`setup`" and "`terminate`" messages.

Methods `Public void` **`drawSetup2`**`(IAIEvent ev)`

The method `drawSetup2` is the callback function for "`setup`" messages. Creates and opens an instance of the `Setup2` dialog box.

`Public static void` **`makeAgents`**`(Vector partVector, Vector queueVector)`

Creates the `PartGenerator`, `TheQueue`, and `TheMachine` agents. `PartGenerator` agents are created from the `PartGen` objects contained in `partVector`. `TheQueue` agents are created from `QueueObject` objects contained in `queueVector`. One `TheMachine` agent is created for every `TheQueue` agent.

`public static void` **`addToList`**`(int type)`

Counts the number of agents created. If all of the agents have been created, `addToList` broadcasts the "`begin`" message to the `PartGenerator` agents.

`public void` **`terminate`**`(IAIEvent ev)`

Terminates the simulation. Closes the log file, reports final production statistics, and terminates Cybele.

`public class` **`PartGenerator`** `implements IAIHandler`

Description

Instances of the `PartGenerator` class are agents that manage the creation and distribution of `Part` objects. Each `PartGenerator` is dedicated to a unique part type. It is assumed that every `Part` object of a particular type will follow the same process plan.

Constructors

`public` **`PartGenerator`**`(String l, PartGen partGen)`

Sets the parameters for a new `PartGenerator` using the information contained within a `PartGen` object. Establishes the process plan and the probability distribution of part arrival times. Includes a Vector of infinite capacity for holding new `Part` objects as they are created. Subscribes to "`checkPlan`," "`pRequest`," and "`begin`" messages specific to the `PartGenerator`'s ID number.

Methods

`public void` **`begin`**`(IAIEvent ev)`

Responds to the "`begin`" message sent by `TheDisplayAgent`. Sets the initial real timer to begin part creation.

`public void` **`MakeNewPart`**`(IAIEvent ev)`

This method creates a new `Part` object. The `Part` object is assigned an ID number and added to the `Part` Vector. The method then queries the first queue in the process plan for availability. Sets a real timer that allows the method to repeat itself recursively.

`public void` **`update`**`(IAIEvent ev) throws IOException`

Sends or holds a `Part` object based on the status of the first queue in the process plan. If the queue is available, update sends the first part in the `Part` Vector; if the queue is unavailable, update holds the `Part` object.

`public void` **`checkPlan`**`(IAIEvent ev)`

Accepts inquiries from `TheMachine` agents to identify the next node in a `Part` object's process plan. Using a process indicator contained within the `Part` object, `checkPlan` locates the `Part` object's current status relative to the process plan.

`public class` **`TheQueue`** `implements IAIHandler`

Description Receives and stores incoming parts for the associated machine agent.

Constructors `Public` **`TheQueue`**`(String name, QueueObject queueObject)`

Sets the parameters for a new queue agent, with `queueObject` as a template. Subscribes to `qStatus`, `newPart`, and `mRequest` messages specific to the queue's ID number.

Methods `public void` **`reply`**`(IAIEvent ev)`

Returns the current queue status (full or available) to the previous step in the process plan.

`public synchronized void` **`newPart`**`(IAIEvent ev)`

Receives a new part from a part generator agent and stores it in a queue vector. Queries the status of the associated machine agent.

`public void` **`sendPart`**`(IAIEvent ev) throws IOException`

Verifies the status of the machine, then passes a part to the machine for processing. Collects some production statistics.

`public class` **`TheMachine`** `implements IAIHander`

Description Simulates machine functions such as part retrieval from the queue and part processing.

Constructors `public` **`TheMachine`**`(String name)`

Sets the parameters for a new machine with `name` as an ID number. Subscribes to `mStatus`, `process`, and `sendNext` messages specific to the ID number.

Methods `public void` **`reply`**`(IAIEvent ev)`

Returns the current machine status to the queue.

`public void` **`getPart`**`(IAIEvent ev) throws IOException`

Receives a new part from the queue. Adds the part to a vector, and sets the current status of the machine to "busy." Queries the Part Generator agent for the next node in the process plan.

`public synchronized void` **`process`** `(IAIEvent ev)`

Gathers part processing time information. Creates and sets a real timer according to user-defined constraints stored in the Part Generator.

`public void` **`sendNext`**`(IAIEvent ev) throws IOException`

Sends the finished part to the next step in the process plan. Collects some production statistics and sets the machine status to "idle."

`public class` **`Part`**

Description `Part` is a class that defines the attributes of `Part` objects, including data that are required by every component in the simulation.

Constructors `public` **`Part`**`(int t)`

Creates an instance of the `Part` class with part type `t`.

`public` **`Part`**`(String p,String in,String out,int t,int i)`

Creates an instance of the `Part` class with part ID `p`, time-in stamp `in`, time-out stamp `out`, part type `t`, and index value `i`.

Methods `public void` **`assignPartId`**`()`

Assigns a part ID number to the `Part` object. ID numbers are incremented and assigned as `Part` objects are created.

`public class` **`PartGen`**

Description The `PartGen` class defines the attributes of `PartGen` objects. The characteristics of `PartGen` objects, in turn, are used to define `PartGenerator` agents.

Constructors `public` **`PartGen`**`(int i)`

Creates an instance of the `PartGen` class containing an ID number `i` and a Vector of probability distribution parameters.

Methods `public void` **`setPlan`**`(String[] plan)`

Establishes a process plan for the `PartGen` object. The process plan is passed to setPlan from the GUI via the plan array.

`public void` **`setTimes`**`(int index, int[] a)`

Adds elements to a Vector of probability distribution parameters. Each element contains an array of integers and represents the distribution of processing times at each step in the process plan.

`public class` **`QueueObject`**

Description The `QueueObject` class defines the attributes of `QueueObject` objects. `QueueObject` objects are used as templates for creating `TheQueue` agents.

Constructors `public` **`QueueObject`**`(int i)`

Creates an instance of the `QueueObject` class with ID number `i` and a queue capacity of 0.

Methods `public void` **`setCapacity`**`(int c)`

Sets the capacity of the queue to `c`.

public class **TimeKeeper**

Description `TimeKeeper` records the elapsed time of the simulation. This information can be imprinted as a time stamp on a `Part` object.

Constructors No constructors.

Methods `public static String` **`stamp`**`()`

Returns the elapsed time of the simulation (in milliseconds) as a String.

public class **Messenger**

Description The Messenger class contains methods that streamline the message-passing required for implementing push and pull rules between the various simulation components.

Constructors No constructors.

Methods `public static void` **`query`**`(String sender, String recipient)`

Sends a message from `sender` to `recipient` querying the current status of the `recipient` queue or machine.

`public static void` **`respond`**`(String sender, String recipient, String status)`

Sends a message from `sender` to `recipient` containing the current status of `sender`.

`public static void` **`checkPlan`**`(String sender, String recipient, string index)`

Sends a message from `sender` to `recipient` requesting information about a `Part` object at step `index` in the process plan.

public class **Dispatcher**

Description Contains a method for passing `Part` object data from one factory component to another.

Constructors No constructors.

Methods `public static void` **`sendToNext`**`(String dest, String ID, int t, int i)`

Passes the part ID (`ID`), the part type (`t`), and the current process index number (`i`) to agent `dest`.

`public class` **`Trace`**

Description The `Trace` class contains methods for creating, writing to, and closing a `FileWriter` object.

Constructors No constructors.

Methods

`public static void` **`create`**`(String filename) throws IOException`

Creates and attaches a `FileWriter` object to a new file called `filename`.

`public static void` **`put`**`(String sender, String recipient, String partId, String timeIn, String timeOut) throws IOException`

Adds a line of text to the output file every time a part is transferred from one factory component to another. The text includes the name of the sender (`sender`), the name of the recipient (`recipient`), the part ID number (`partId`), the time the part was received by the sender (`timeIn`), and the part the time was sent (`timeOut`).

`public static void` **`end`**`() throws IOException`

Closes the `FileWriter` object.

`public class` **`RandomNumbers`**

Description A uniformly distributed random number generator.

Constructors

`public` **`RandomNumbers`**`(int[] params)`

Creates an instance of the `Random` class. The seed value for the random number generator is derived from information contained within the `params` array. The `params` array takes the form {a,b,c}, where a is the type of distribution, b is the lower bound, and c is the upper bound.

Methods

`public double` **`sample`**`()`

Returns a random number. Random numbers returned by the `RandomNumbers` class are uniformly distributed between the lower and upper bounds specified in the `params` array.

`public class` **`DisplayCount`**

Description Contains a method for updating counter labels in the `Setup2` dialog window.

Constructors No constructors.

Methods `public static void` **`sendCountToDisplay`**`(String name, int id, int count)`

Passes `name`, `id`, and `count` to the `Setup2` dialog window by broadcasting a message to "`count`."

`public class` **`Statistics`**

Description The `Statistics` class contains methods for computing production statistics such as average part waiting time, average part processing time, and throughput.

Constructors No constructors.

Methods

`public static void` **`countTotalParts`**`()`

Increments the total number of parts produced by the PartGenerator agents.

`public static void` **`countQueueParts`**`()`

Increments the total number of parts received by TheQueue agents.

`public static void` **`countFinishedParts`**`()`

Increments the total number of outputs from the factory model.

`public static void` **`addQueueTime`**`(String timeIn, String timeOut)`

Calculates the interval in milliseconds between `timeOut` and `timeIn`. Adds this value to the total amount of time spent by parts in all queues.

`public static void` **`addMachineTime`**`(String timeIn, String timeOut)`

Calculates the interval in milliseconds between `timeOut` and `timeIn`. Adds this value to the total amount of time spent by parts in all machines.

`public static float` **`averageWait`**`()`

Returns the ratio of the total queue time (`timeInQueue`) to the total number of parts queued during the simulation (`queueParts`).

`public static float` **`averageProcessing`**`()`

Returns the ratio of the total machine time (`timeInMachine`) to the total number of finished parts (`finishedParts`).

`public static float` **`throughput`**`(String time)`

Returns the ratio of finished parts to simulation time.