Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

OBJECT ORIENTED SIMULATION WITH SMALLTALK-80: A CASE STUDY

Jocelyn R. Drolet

Groupe de Recherche en Gestion
de la Logistique
Laboratoire de Productique
Université du Québec à Trois-Rivières

Colin L. Moodie

Engineering Research Center
Purdue University
West Lafayette, Indiana 47907

Benoit Montreuil

Groupe de Recherche en Gestion
de la Logistique
Université Laval

ABSTRACT

Object Oriented Simulation (OOS) provides a more natural way of modeling complex phenomenon and a higher productivity during the incremental phases of development but it is not a panacea. In this paper, the authors relate their experience with OOS. More specifically, the article begins with a few historical developments that lead to object oriented programming. Then, we present the objectives of the object oriented simulation model, which we developed as a case study. The third section introduces the virtual cellular manufacturing system which has served as the referent system for the experimentation. In the fourth section and the following, we present the object oriented modeling methodology which has been followed during the incremental phases of development and the strategic and tactical plan for sustaining the experimental investigation. Finally, the results of the experimental investigation that focus uniquely on the object oriented experience are presented. Results that are specific to the virtual cellular manufacturing system are not discussed in this paper.

KEY WORDS: object, simulation, SmallTalk, modeling, virtual, cell

1 INTRODUCTION

Computer simulation is one of many ways available for modeling certain phenomenon. Problems that are well understood can sometime be approximated by analytical models. Other problems involve collections of interacting entities whose low-level pair-wise interactions with each other are known but whose high level group interactions cannot be grasped. Simulation enables one to encode these low-level interactions and simulate them, dynamically approximating the overall behavior of the system.

2 THE EVOLUTION OF SIMULATION LANGUAGES

In this section, we present a few historical developments that have led to object oriented simulation. The first related milestone in the history of programming language is the development of FORTRAN in the 1950's. Although most simulation model being developed today are written in languages such as SLAM, GPSS and SIMULA, observers agree that most simulation models in use are still written in FORTRAN. With Fortran, programmers had for the first time the ability to directly write complex mathematical expressions in a familiar form. Also, Fortran's primary contribution was the introduction of subroutines which added modularity to large applications.

Modularity was further enhanced with the appearance of Simula [2] in the 1960's. Simula, as the name suggests, was designed as a general purpose simulation language. Its contribution was to introduce the concepts of classes and class objects that supports multiple concurrently operating processes.

Results obtained from computer language theory developed during the 1960's led to many new programming languages in the 1970's. One milestone was the development of Pascal which further emphasized program modularity. Among the most widespread simulation languages developed in that period are GPSS, GASP, and SLAM. In the late 1970's, many other languages used some form of class construct borrowed from Simula within a Pascal-like syntax. Another major achievement was undoubtedly Ada. Ada supports

structured generic data types. With Ada, the potential was almost unlimited but the price to pay was high; the goal with Ada was to satisfy the needs of so many application areas that it resulted in a very complex language.

In the mean time, Lisp and Prolog were proposed as the solution for artificial intelligence programming. Object-oriented simulation environments have tended to be implemented in Lisp but suffered from Lisp's scaling and integration problems. Efficiency and execution speed of LISP becomes problematic when scaling from a handful of objects to thousands.

Then, object oriented programming languages appeared. Object-oriented simulation traces its roots to SIMULA. Its modern form is represented by such languages as C++, SMALLTALK-80, LOOPS and EIFFEL.

Object oriented programming is a philosophy of design It has three characteristics; data abstraction, program abstraction and protection domains which define access rights and operations available to the user. It provides a new dimension in which to organize the elements of an application software. The object oriented concept allows the developer to create highly reusable, generic code, in an environment that favors prototype style of application development.

The object oriented paradigm provides several features not included in procedural languages. The variety of data structures; indexed collection, fixed size collection, bitmap, byte array, symbol, ordered collection, sorted collection, dictionary etc., are not found in any procedural language. Flexibility, modularity, inheritance, encapsulation and sophisticated programming environment that usually permit pseudo parallel processing are other advantages of this paradigm over procedural language. Objects are the basic entities within Object Oriented Programming (OOP). Booch [1] defines an object as follow:

An object is an entity that

1. Has a state.
2. Is characterized by the action that it suffers and that it requires of other objects.
3. Is an instance of some class.
4. Is denoted by name.
5. Has restricted visibility of and by other objects.
6. May be viewed by its specification or its implementation.

Objects should also have the following three characteristics:

1. Inherent processing ability
2. Message communication
3. Uniformity of appearance, status, and reference.

Generally speaking, objects are intended to represent entities capable of exhibiting different behavior. All objects in an application are organized in a class hierarchy that lets them inherit properties from higher level objects. The class structure definition depends on what OOP language is supported.

For example, early versions of the SMALLTALK language supported a class structure with a strict single inheritance rule. Although more efficient, multiple inheritance has some problems. In fact, when an object has more than one superclass, it might inherit conflicting variables or even conflicting methods, thus, multiple-inheritance hierarchy is really for experienced programmers.

3 SIMULATION OBJECTIVES

The main objective of the study is to demonstrate the viability of a virtual cellular manufacturing system [3],[4],[5],[6] and to comprehend the high level interactions of its many components. More specifically, the following questions shall be answered, although in some cases, partial results shall be sufficient given the infancy of the concept.

1. Will the virtual cellular manufacturing concept work?
2. Since these systems operate in Just-in-Time mode, how long will the average queue length be?
3. How long will the maximum queue length be in a typical run?
4. How long will the average time-in-system be?
5. How fast will the optimization based scheduling algorithm react upon the occurrence of an unexpected event such as a breakdown?
6. Typically, how many parts, jobs, or virtual cells will be active in the system simultaneously?
7. Typically, how many interactions will be present between virtual cells? In other words, how many workstations will be shared between two or more virtual cells?
8. What burden will be put on the material handling system?
9. How efficient in terms of machine utilization, time-in-system, and throughput is the virtual cellular system?

The variety of questions, the complexity of virtual cellular systems, and the importance of testing the overall behavior of the system, while being in transient

state, forces us to reject a queuing theory based methodology since these techniques are valuable only in systems of simple to moderate complexity that will reache a steady state. Simulation is the only methodology currently capable to answer the questions just listed.

The overall goal can thus be formulated as follows:

To develop an object oriented simulation model of a Virtual Cellular Manufacturing System (VCMS) which permits answering the questions just presented. In addition, the model should consider various entities and resources: machines, workstations, material handling, jobs, parts, tools, and virtual cell controller. The model should permit the generation of random breakdowns. Also, one should be able to interface the model with a variety of existing algorithms already written in the C programming language; among them, a linear programming algorithm, a generator of linear programs and a parsing code. Finally, it should be noted that it is not necessary to have a one-to-one correspondence between every aspect of the referent system and every aspect of the model.

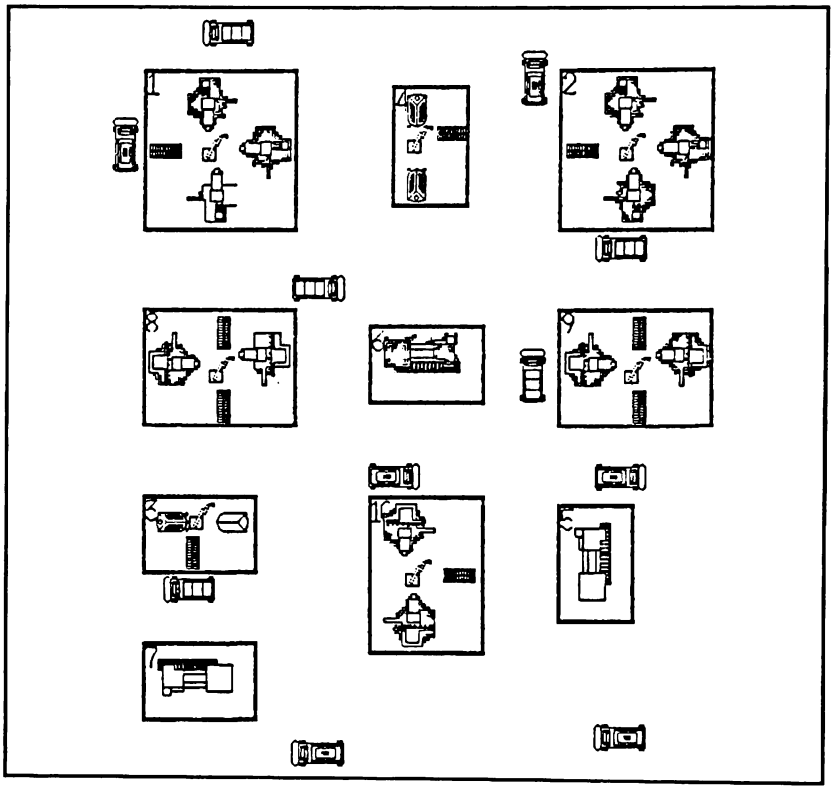

Figure 1: An experimental VCMS

4 DESIGN OF AN HYPOTHETICAL VCMS

We do not know of any existing virtual cellular based manufacturing system to date. Given this, it is necessary to conjecture an hypothetical VCMS which will be the referent system for our experimental investigation. The referent of a simulation model must be well defined, it must serve as reality from the perspective of the model. This section focuses essentially on the development of an hypothetical VCMS.

Figure 1 illustrates the manufacturing system which plays the role of referent. The manufacturing system consists of 19 CNC machines of 4 types grouped in 10 workstations.

Each workstation contains from 1 to 3 identical machines, a material handling device for local handling, and at least one input/output station which can hold a few pallets. Input/output stations are the interfaces permitting the linkage between different material handling systems.

For the purpose of this study, it was decided to use robots and automated guided vehicles (agv) as principal medium for material handling. Robots would handle the parts within workstations having multiple machines. Generally, they would carry the parts from an input

station, which in most occurrence is one side of a bi-directional conveyor, to the destination machine and then, once the machining is done, from the machine back to the output station i.e., the other side of the bi-directional conveyor.

The system also includes a fleet of free ranging automated guided vehicles responsible for handling the flow of parts between workstations. The agvs can circulate freely in the manufacturing system. These intelligent AGV's are assumed to be of the latest generation, either controlled by infra red beam or radio frequency. The number of AGV's which will be required depends of many factors; AGV's characteristics (speed, acceleration, loading and unloading time, frequency of breakdown and duration, loading capacity, communication delay and throughput, quality of the control algorithm (i.e. myopic or optimized based)), production's characteristics (processing time, number of stations to visit, number of products per fixture-mounted pallet), and system's characteristics (distance between I/O stations, density of each virtual cell, variety of workstation, layout, replication of workstations, processing speed). Given the intent of the article, the specific parameters are not specified but will be included in a forth coming paper which will focus on the virtual cellular concept.

5 OBJECT ORIENTED MODELING

Industrial people think of manufacturing systems in terms of machines, parts, jobs or, more generally, "objects"; classical programmers think in terms of programs and data files, more specifically they apply functional decomposition techniques to encode the problem specific information. There are major differences between object oriented decomposition and functional decomposition. Booch [1] studied these differences and showed that in most cases object oriented decomposition allows a more natural representation.

OOP is a philosophy of design, thus the approach to model building turns out to be quite different from that associated with procedural programming. Booch [1] recommends the 6 following steps when OOP philosophy is used as a tool for model building.

1. Create a data flow diagram.
2. Identify the objects and their attributes.
3. Identify operations sustained by and required of each object.
4. Establish the visibility of each object.
5. Establish the interface of each object.
6. Implement each object.

The first step is carried out irrespective of the approach in model building. This step specifies the inputs and outputs of the system to model. In the second step, one tries to decompose the system into concrete entities or objects that act in the problem space. The objects that the system is decomposed into should be real actors in the system. In the third step, specification and behavior of each object are defined. In step four, "visibility of each object" really means to establish what objects need to communicate with which other objects. Then, one needs to establish how each object is going to communicate with the others. Finally, object implementation is the code that must be written for each object. In other words, for each object, one has to define their methods, instance variables, access, and behavior.

The 6-step methodology proposed by Booch is used as a guide. OOP is an environment that favors incremental prototyping style of application development, thus, these steps should not be executed sequentially. We would rather be thinking of these steps as an integrated approach to object oriented design philosophy.

The nature of the simulation tools forces the analyst to adopt a certain style, a certain structure during the course of model development. The object oriented simulation environment of Sim (Sim is a discrete event, object oriented framework written in SmallTalk. Sim contains a set of tools that may help the designer during the modeling phase.), for instance, involves the sub-classing of class Sim to define the simulation world in which the simulated entities will interact. Also, each object or entity which the analyst wishes to simulate in that world must be a subclass of SimObject. As such, these objects inherit some variables and the ability to interact with other objects. While sub-classing reduces code size and enhance its readability, the analyst must structure the model in a way that not only fits this framework but also takes advantage of it. During the model building phase, it was found that although inheritance by sub-classing reduces coding, it requires a much higher level of planning. Normally, it is unnecessary to subclass at more than two or three level down the hierarchy. Our model uses only one level of sub-classing, hence, every object is a child of class SimObject.

Data flow diagram

The creation of a data flow diagram is suggested as a design aid to help with conceptual design of complex systems. The data flow diagram should indicates data files and processes involved in the scheduling process. In this case, the scheduling process is initiated in four pre-determined situations; at the beginning of every simulation run, upon the occurrence of a breakdown, when the processing advance deviates more than a certain

threshold from what was planned, and when the detailed planning horizon has elapsed. When any of these situations occur, the classScheduler initiates the scheduling process. The simulation continues with the new schedule until the scheduling process is initiated again.

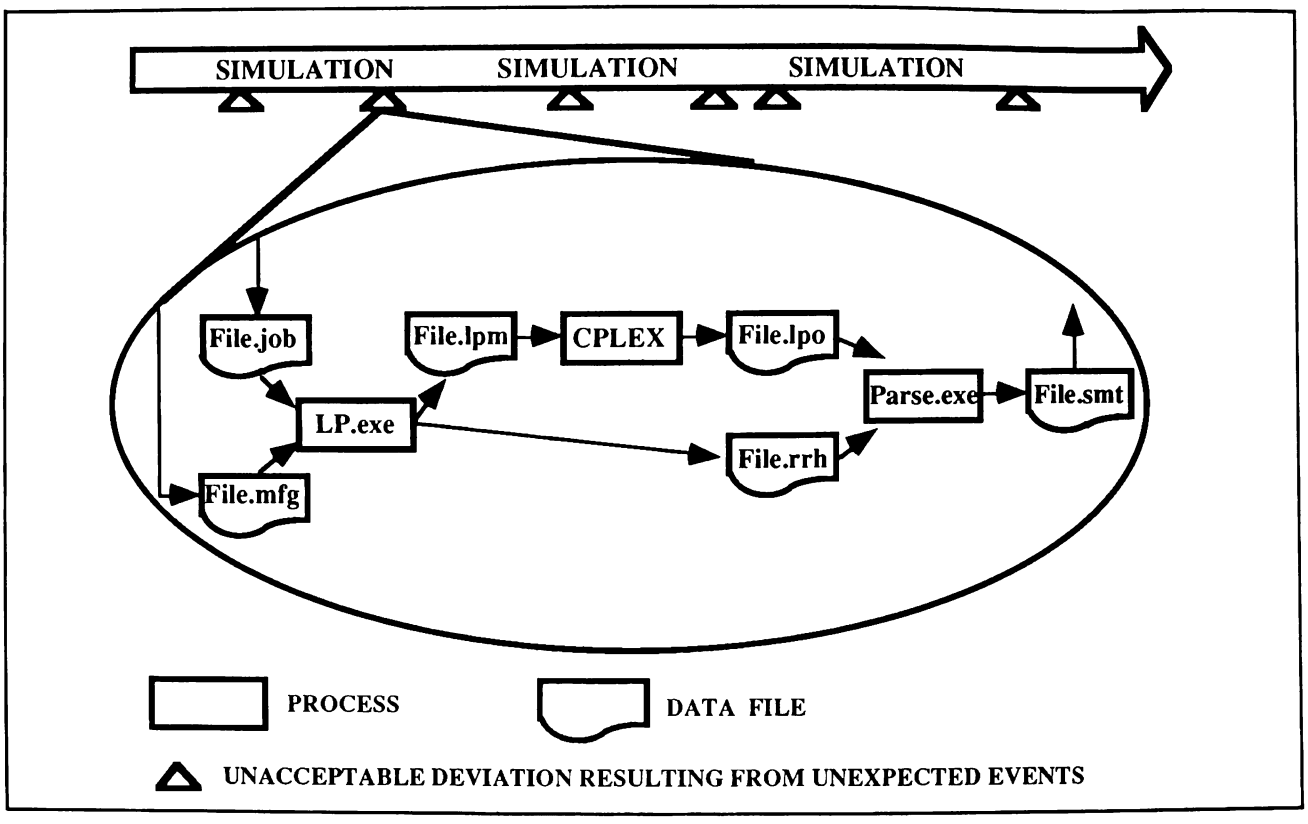

Figure 2: Data flow diagram

Figure 2 illustrates the data flow diagram that is specific to this. This data flow diagram is only one of many but it was chosen to show that the scheduling process comprises several steps, one of which is the resolution of an optimization based scheduling model. The object oriented simulation model can be interfaced with any code written in the C language. Also, if the model requires one or many number crunching functions, it is possible to write "object code" in the C language.

Object identification and attributes

The second step consists of decomposing the system into concrete entities or objects that act in the system space. Objects can be any entity that plays a specific role, they should be real actors in the referent system. Objects that shall be modeled are those that can perform certain tasks or methods when requested by a message. In general, it is possible to identify objects by their properties and attributes. The likely candidates in a manufacturing environment are: machines, parts, employees, tools, etc. A specific object named drillingMachine will be an instance of class DrillingMachine. The object will have a set of private variables or attributes named instance variables. The object will also have a set of procedures or methods which can access and modify those instance variables. Messages are a form of procedure calls; and methods correspond to procedures.

Objects may correspond to conceptual or physical entities of interest in the referent system. A reduced list of those class objects with their respective instance variables is presented hereafter.

```
SimObject subclass: #DrillingMachine
    instanceVariableNames: 'myWorkstationId'

SimObject subclass: #Agv
    instanceVariableNames: 'travelingTime currentTime
    timeToGo willBeThereAt myDirection myStatus'

SimObject subclass: #Part
    instanceVariableNames: 'jobId quantity processTime
    tooling partRouting indexRouting
```

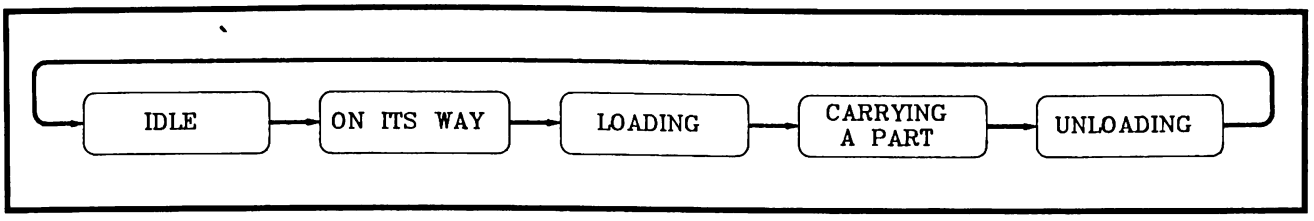

Figure 3: Operating cycle of an AGV

Object behavior

Since objects are supposed to be the real actors in the system, they must have a certain behavior. This section focuses on defining the behavior which characterizes the most interesting objects listed in the previous section. At this stage, we are not concerned with messages or procedures, we are only concerned with there behavior or operating cycle. According with Booch's methodology, an operating cycle shall be defined for each object. The operating cycle shall be precise enough to identify all operations sustained by and required of each object.

Because it is a representative example, we will now explore the operating cycles of an Automated Guided Vehicle (AGV). Agv's are responsible for the handling of parts between workstations. The operating cycle of an agv begins in the idle state. This is so simply because they are in this state at creation time. The agv is waiting until a customer requests transportation. When a call is received, the state of the agv passes from "idleAgv" to "agvOnItsWay" meaning that the agv is on its way to pickup a part. Traveling time depends on the distance to travel. Upon the arrival of the agv, the part is loaded, carried to its destination, and unloaded. The agv becomes available again and the cycle is repeated. Two reasons motivated our decision not to model the breakdown of agvs. First, agvs were assumed completely interchangeable. Second, one or more spare agvs could be kept nearby.Figure 3 illustrates the sequences involved in the operating cycle of an AGV.

Object visibility

Establishing the visibility of each object is to determine what objects need to communicate with which other objects. Communication between Sim objects takes the form of an object sending a message to another object, and receiving a reply. Two or more simulated objects may have to communicate directly with each other. In those dialogs, blocks of code and messages go back and forth between Sim objects causing the suspension and resumption of processes.

The dialog between an instance of class Part and an instance of class Agv, is a good example of that. Loosely speaking, the part tells the agv its initial location and its destination when requested by the AGV. The agv does not have any other way to obtain this information since these values are instance variables of the part and thus, private information. More specifically, upon the reception of a message, the part activates the corresponding method accessing its private date and then returning it to the agv that requested it. Figure 4 illustrates this point.

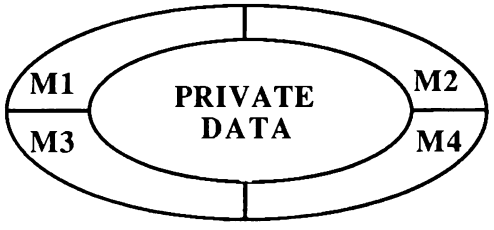

Figure 4: Encapsulation

Object interaction

Object interaction really means to establish how each object is going to communicate with each other and with the simulated world as well.

It may happen that many objects evolve in the same simulated world without directly interacting with each other. This is the case of more than half of the objects present in our simulation. In these cases, every object has a set of methods which permits it to read and/or modify some information held by the central control point which is the simulated world. It would be much too tedious to detail all these methods, moreover, those methods are very much language dependent.

Sometimes, objects need to communicate directly with each other. The dialog between an instance of class Part and an instance of class Agv is now the subject of a meticulous description.

Communication between Sim objects

Communication between Sim objects takes the form of an object sending a message to another object, and receiving a reply.

Figure 5 shows the SmallTalk code that makes communication possible between two Sim objects, in the occurrence, an instance of class Agv and an instance of class Part. The code is executed sequentially from top to bottom, thus the control alternates from one object to the other. The example has been chosen for its simplicity and because every other object uses a similar code for communicating between them. One shall mention that the code is a slightly modified version of the original one, in fact, tracing and statistic gathering instruction have been removed for clarity. In Figure 5, instances of SimObject Agv are playing the role of servers while instances of SimObject Part are playing the role of customers. In the upcoming discussion instances of Agv and instances of Part will be respectively termed agv and part except when stated otherwise. The first line shows the name of both methods; 'actions' and 'transport'. The second line list all local variables. The scope of local variables is limited to the method in which they are declared. Both processes take turns executing and being suspended, while the customerServer protocol passes blocks of code between them.

```
Instance of SimObject: Agv                    Instance of SimObject: Part

     Server                                   Customer

1 actions                                     transport
2     | aPart travelTime from to |                 | agv1 |
3     aPart <-- (simControl getQ:'agv')          agv1 <-- (simControl getQ:'agv')
                findCustomers: 1                          findServers: 1
                for Server: self                          forCusomer: self.
4     aPart resumeProcessWhileIWait.
5                                                 agv1 reply: [self location].
6                                                 agv1 resumeAllProcessesWhileIWait.
7     from <-- aPart getBlock value.
8     aPart resumeProcessWhileIWait.
9                                                 agv1 reply: [self destination].
10                                                agv1 resumeAllProcessesWhileIWait.
11    to <-- aPart getBlock value.
12    travelTime <-- 3 * ((simControl workstationPosition:from)
                dist:(simControl workstationPosition:to)) + 1.
13    icon <-- simControl agvOnItsWay rotateBy:myRotation.
14    simControl terminateObject: self.
15    simControl animate: self.
16    simControl waitFor: travelingTime waitingObject: self.
17    icon <-- simControl agvBusy rotateBy:myRotation.
18    simControl TerminateObject: self.
19    simControl animate:self.
20    simControl waitFor: travelTime waitingObject: self.
21    icon <-- simControl agvIdle rotateBy:myRotation.
22    simControl terminateObject: self.
23    simControl animate: self.
24    simControl queueArrivalOf: self afterWait: 0.0
25    aPart resumeProcess.
26                                                indexRouting <-- indexRouting + 1.
27                                                transportDone <-- #true

instance variable:                            instance variable:
     travelingTime, myRotation                       indexRouting, transportDone
```

Figure 5: Two objects communication code

In the agv's method, line 3 shows:

```
3 aPart <-- (SimControl getQ: 'agv')
        findCustomers: 1 forServers: self.
```

The agv needs a customer. This line tells the agv to go to the customerServer queue named 'agv' and get a pointer to a control block that represents a part waiting to be carried. This control block is assigned the pointer aPart, which is how the agv knows who his customer is during the upcoming conversation. The method findCustomer:forServer: returns a control block. The control block is an instance of the class SimWaitForCustomerServer. Meanwhile, the part asks for an agv, representing its server.

```
3 agv1 <-- (SimControl getQ: 'agv')
    findServers: 1 forCustomer: self.
```

The part goes to the same customerServer queue 'agv' and gets a pointer to a control block representing the agv. This control block is assigned the pointer 'agv1', which is how the part knows its server while the conversation goes on. The method findServers:forCustomer: returns an instance of the class SimWaitForCustomerServer. Both instances of SimWaitForCustomerServer are complementary; each has a pointer to the other. The result is that both know their counterparts. The Q named 'agv' is a kind of rendez-vous point for the part and the agv. They can now pass blocks of code between them.

As soon as a customerServer has taken each of the two processes out of its queue and matched them in this way, the server process proceeds. In this case, this means that the agv executes the next piece of code in its method. Line 4 shows this code:

```
4 aPart resumeProcessWhileIWait.
```

This code has the effect of the agv telling the part to go ahead and resume its process. The agv process suspends itself. The next piece of code i.e. line 5 and 6 in the part's method are therefore executed.

```
5 agv1 reply: [self location].
6 agv1 resumeAllProcessWhileIWait.
```

Agv1 is the pointer by which the part knows the agv, i.e. its server at this Q. It replies with a block of code. This code asks the part to get the workstation number at its location. However, the code in the block is not yet evaluated. The block is merely passed as a whole to the agv. It represents the part's reply to its server agv. In effect, the part is telling the agv how to find out about its location.

Having passed its reply, the part then tells the agv to resume its process and suspend its own process. The agv's process resumes. The part's instance of SimWaitForCustomerServer passes the block to the agv's instance of SimWaitForCustomerServer. Now, the agv can access the part's reply. The next piece of code, line 7 and 8 will now be executed in the agv's method.

```
7 from <- aPart getBlock value.
8 aPart resumeProcessWhileIWait.
```

Sending the message 'getBlock' to the part to be carried returns the reply block sent by the part. The message 'value' causes the block to be evaluated. The result of evaluating line 7 assigns the workstation number of the part to the local variable 'from'. Now, the agv tells the part to resume its process and suspend its own (agv) process. The part's process resumes.

The next piece of code in the part's method is now executed.

```
9 agv1 reply: [self destination].
10 agv1 resumeAllProcessWhileIWait.
```

This code has the same effect as lines 5 and 6, except that the block of code which is passed to the agv is different. This time, the code asks the part to get the workstation number for its destination instead of its location. As previously explained, the code in the block is not yet evaluated. The block is passed as a whole to the agv. Having passed its reply, the part tells the agv to resume its process, then it suspend its own process. The agv process resumes.

Line 11-24 are now executed in the agv's method.

```
11 to <-- aPart getBlock value.
12 travelTime <-- 3 * ((simControl
workstationPosition:from)
    dist:(simControl workstationPosition:to)) + 1.
13 icon <-- simControl agvOnItsWay
rotateBy:myRotation.
14 simControl terminateObject: self.
15 simControl animate: self.
16 simControl waitFor: travelingTime waitingObject:
self.
17 icon <-- simControl agvBusy rotateBy:myRotation.
18 simControl TerminateObject: self.
19 simControl animate:self.
20 simControl waitFor: travelTime waitingObject: self.
21 icon <-- simControl agvIdle rotateBy:myRotation.
22 simControl terminateObject: self.
```

```
23 simControl animate: self.
24 simControl queueArrivalOf: self afterWait:0.0.
25 aPart resumeProcess.
```

The result of evaluating line 11 assigns the workstation number in which the part goes to the local variable 'to'. Line 12 obtains the position of both workstations 'from' and 'to' and heuristically computes the traveling time which is then assigned to the local variable 'travelTime'. The heuristic results in traveling times that are more or less equivalent to an agv operating at 15 feet/min with 20 seconds of loading time and 20 seconds of unloading time. Possibilities of congestion were not considered in this study.

Every instance of SimObject inherits certain variables. Examples of inherited instance variables are 'icon' and 'position'. In line 13, the message agvOnItsWay is sent to the simulation which returns a form. The unary message rotateBy:myRotation is then sent to the form which is further assigned to the instance variable 'icon'. Lines 14 tells the simulation to stop displaying the icon which was representing the agv, and then Line 15 tells the simulation to animate or display the newly defined icon. Then line 16 tells the simulation to place the agv in the time queue until the amount of time specified by its traveling time has passed. Thereafter, the instance variable 'icon' is once again assigned a new form, this time the form represents a busy agv which explains the message's name 'agvBusy'. Line 18 tells the simulation to stop displaying the previous icon and line 19 tells it to start displaying or animate the newly defined 'icon'. The travel time computed previously is then used for delaying the agv, this is achieved in line 20.

After carrying the part, the agv becomes idle for an undetermined period of time. Again, the instance variable 'icon' is assigned a new form which represents an agv idle. Line 22 tells the simulation to stop displaying the previous 'icon' and line 23 tells it to start displaying the newly defined 'icon'.

The role of the agv is now finished, all that remains is to send a message to the simulation in order to schedule the arrival of that agv without delay, in the time Queue, which is done at line 24. The agv then tells the part aPart to resume its process. The agv has nothing else to execute in the simulation, so it returns from its actions method. However, one must remember that the same agv has been scheduled for arrival without delay which means that it will enter into action instantly.

The next piece of code i.e., line 26 and 27 in the part's method are therefore executed.

```
26 indexRouting <-- indexRouting + 1.
27 transportDone <-- #true
```

Line 26 increments the instance variable indexRouting by one unit. The instance variable transportDone is assigned the value #true since the part just arrived at destination. The part has nothing else to execute in its method thus the simulation returns to the actions method in which the transport message was sent.

There are four other cases where a direct communication between objects is necessary. Every instance of any of the classes DrillingMachine, VerticalMillingMachine, HorizontalMillingMachine, Lathe, communicate directly with their customers parts to obtain information such as routings, processing time and so forth, i.e. the kind of information that is privately owned by the instance of class Part.

6 STRATEGIC AND TACTICAL PLAN FOR THE EXPERIMENTATION AND VALIDATION

Model coding should be carried out in an incremental fashion. Before anything else, the analyst should break the referent in modules or subsystems. Then, he/she should concentrate his/her effort on modeling the core of the system while abstracting most subsystems surrounding it. After validation of the basic model, the analyst should model other subsystems and add them incrementally until the model attains the level of detail needed by the experimentation. This is especially true when dealing with complex systems.

In the spirit of incremental development, we started with a few SimObjects. Instances of class Part were processed sequentially in two different machines, one instance of class DrillingMachine and one instance of class HorizontalMillingMachine. In this early plan of development, material handling components were not considered. Then, we added more drillingMachines and more HorizontalMillingMachines grouped in two workstations. Instances of class Part were queueing at one or both workstations depending on their load. The next incremental step permits the addition of class Agv which permits estimating the burden on material handling and at the same time considers the effect of material handling on the efficiency of the system. Then, we added the SimObject "Breakdown" which permit the generation of randomly distributed breakdowns. More classes were added permitting the interfacing of the simulation model with some programs written in the "C" language.

Finally, two more machine types and many more workstations were added to the model. An enhanced pop-up menu and several statistic gathering functions were also added.

Verification & Validation

Verification permits one to determine if the computer program is performing properly. If the input parameter translates in a correct output then this step has been completed. Validation enables the analyst to determine that the model is an accurate representation of the referent. Usually, the analyst makes sure that each component of the simulation model behaves properly. Since most models are developed in an incremental fashion, it is easy to systematically validate the components after every incremental step. Generally, the analyst isolates the component under study from every other except one or two with which it interacts, and he analyzes its behavior. Sometimes, it is best to incorporate these two steps i.e., verification and validation together. This was done in this study.

More specifically, let us explain how we started the validation process in our experiment. First, we ran the simulation model with one job, no breakdown, no critical tool, and only one agv. This permitted testing the process flows and processing times. In addition, it was possible to test the interface between the database and SmallTalk because every job's related information was loaded from the database into SmallTalk. The openness of SmallTalk permits tracking down every part in the system and following their behavior, making it very simple to debug any problem. Also, by running the simulation with one agv, one can easily trace it and record the time elapsed between two consecutive state changes. Thus, one can verify its idle time, busy time, and "onItsWay" time. We ran this configuration of the model repeatedly with different jobs.

Then we added the code which generates random breakdowns. Finally, we added the critical tools and processed many more jobs simultaneously. Several runs were executed with the unique objective of validating the model in its final version. When using a language which has the openness of SmallTalk, the debugging phase is usually not problematic.

7 CONCLUSION

The stages of simulation development outlined above have been presented in a sequential fashion; this does not mean however that one should perform them in a strict sequential manner. As far as we are concerned, the simulation development can be best described as an iterative process while progressing throughout the steps.

SmallTalk provides an excellent set of features, capable of supporting object oriented discrete event simulation. A simulator can model complex systems in an incremental fashion. The interactive user interface allows the designer to quickly zero in on the variables of interest while debugging capabilities undoubtedly lead to higher productivity during the incremental phases of development.

Speed of execution appears to be the only draw back. Efficiency is a major issue in the simulation world; advance in parallel processing appears to be the only way out for the development of large scale object-oriented simulation model.

ACKNOWLEDGEMENTS

This research is supported in part by: FCAR(91-NC-0637,91-ER-0685), CRSNG(OGP-0042280,OGPIN-020) and NSF(). Special thanks are addressed to Mrs. Verna Knapp from Tektronix Laboratories who gratefully accepted to send us a beta version of SimTalk, an object oriented discrete event simulation environment built on SmallTalk, which was used during the preliminary phases of development.

REFERENCES

[1] Booch, Grady, "Object-Oriented Development". IEEE Transactions on Software Engineering, Volume SE-12, Number 2, February 1986, p. 211.

[2] Dahl, O. J. and Nygaard, K., "Simula: A Language For Programming And Description Of Discrete Event Systems". Fifth Edition, Norwegian Computing Center, Oslo, 1967.

[3] Drolet, Jocelyn R., Moodie, Colin L., Montreuil, Benoit, "Decision Architecture For Scheduling Virtual Cellular Manufacturing Systems", International Federation of Automatic Control (IFAC), Proceedings of the international workshop on decisional structures in automated manufacturing, Genova, Italy September 18-21, 1989,pp.103-112.

[4] Drolet, J.R., "Scheduling Virtual Cellular Manufacturing Systems", Ph.D. Dissertation, Purdue University, June, 1989.

[5] Drolet, Jocelyn R., Montreuil, Benoit, and Moodie, Colin L.,Virtual Cellular Manufacturing Layout Planning, 1990 International Industrial Engineering Conference Proceedings, Institute of Industrial Engineers (IIE), San-Francisco, California, May 20-23,1990, pp.236-241.

[6] McLean, C. R., Bloom, H. M., and Hopp, T. H., "The Virtual Manufacturing Cell". Proceedings of Fourth IFAC/IFIP Conference on Information Control Problems in Manufacturing Technology, Gaithersburg, MD, October 1982.

AUTHOR BIOGRAPHIES

Jocelyn R. Drolet, Ph.D., Ing. is a Professor in Industrial Engineering at the University of Québec. He is member of the Groupe de Recherche en Gestion de la Logistique (GRGL) of Laval University and member of the Laboratoire Départemental de Recherche en Productique of the University of Québec.

Colin L. Moodie, Ph.D., Eng. is a professor in the School of Industrial Engineering at Purdue University. He is member of the Engineering Research Center at Purdue Univesity.

Benoit Montreuil, Ph.D., Ing. is a professor in Opérations et Systèmes de Décision (OSD) at Laval University. He is co-director of the Groupe de Recherche en Gestion de la Logistique at Laval University.