MARS: AN OBJECT-ORIENTED DISCRETE-EVENT SIMULATOR TO ANALYZE SEMICONDUCTOR FABRICATION MANUFACTURING STRATEGIES

Stephen P. Smith
Intel, CH2-68
5000 W Chandler, Chandler AZ, 85226
SSmith@FA.Intel.COM

ABSTRACT

This paper describes the software analysis, design, and implementation of an object-oriented simulation system. This system allows the synthesis and analysis of manufacturing strategy rules which control semiconductor fabrication factories (Fabs). These rules capture resource tradeoff decisions such as when to prefer to run one step over another on a machine that can run both steps.

Because of the complexity of the manufacturing process, simulation is the only effective technique to answer many key questions about strategy rules. However, because of the need to experiment with alternative strategy rules, we felt it was worthwhile to develop our own low-level simulation model using the CLOS Object-Oriented programming language.

This paper discusses the MARS project and the object-oriented approach used. We focus on some interesting design and implementation tradeoffs needed to make MARS both computationally efficient and extensible to new rules and behaviors. Our conclusion is that the application of a few OO-derived techniques produced a simulator with advanced modeling power and remarkable extension capability.

BACKGROUND

Semiconductor fabrication is the complicated process by which raw silicon wafers are converted into microelectronic devices by deposited layers and patterns of metal and semiconductor materials. The process is characterized by complex re-entrant product flow, random yields, random machine availability, diverse equipment characteristics, etc. [Uzoy 1992]

An important problem is the choice of management strategies in wafer fabrication facilities (Fabs) to maximize output while holding down inventory and keeping quality high. These strategies are the rules that make resource tradeoff decisions (when to run Step A as opposed to Step B on a machine that can run both), resource setup decisions, and other control decisions (such as how to assign repair technicians to broken machines, when to introduce starting wafers into the line, when to bring a production machine down for preventative maintenance or to run a line-yield experiment, etc.).

The MARS project began after extensive talks with manufacturing managers and our own work on real-time scheduling systems for Intel's Fabs [Kempf 1994]. Key questions of strategies kept emerging. What is the long term effect of short term decisions made every shift in the Fab? How do I manage a toolset that is used over multiple steps in the line? Should I manage steps by their output rates? Should I manage a constraint toolset with an inventory buffer? If I have a number of near constraints, how do I make sure they feed each other correctly?

We decided to focus on strategies directly related to the movement of WIP (WIP-management strategies, as opposed to, say, equipment maintenance strategies) as we could delineate reasonable WIP-management rules, some of them supported by simple analytic models. However, for our complex Fabs, we still could not prove any one was better than another without fall back to religious convictions; Thus to answer the above questions, we resorted to simulation models. Unfortunately, while some previous simulation work studied the semiconductor fabrication area [Glassey 1990, Miller 1990, Wein 1988] and some excellent commercial simulators exist [AutoSimulations 1993], no commercial simulation package included the ability to quickly define new WIP-management rules. Therefore, we developed ***MARS,*** *The Manufacturing Rules Simulator*.

BASIC DEVELOPMENT TECHNIQUES

We used the following software development techniques. The system was

(1) analyzed and designed using an Object-Oriented methodology, and
(2) designed around a simulation "micro-kernel", and
(3) implemented using the CLOS OO rapid prototyping language [Steele 1990, Keene 1989, Paepcke 1993].

The first technique ensured those objects with stable domain identities (machines, steps, rules, etc.) were encapsulated into the software via well-designed simulation interfaces. We felt that the meaning of machines/steps etc. would be less likely to change (and more likely to be merely extended) than other possible primitives. For instance, when new types of strategy rules are added, their previous supporting components (for example parsers and printers for high-level rule input/output) are immediately available for incremental redefinition.

Technique (2) ensured that supporting code could be developed incrementally around a stable simulation core. This core consists of all of the main code for maintaining a simulation event queue and all basic events sent to objects. For instance, to include random machine downtime models, a supporting program downloads machine "availability calendars" into the simulation kernel. By default, the standard class of machines has an availability calendar whose shift-long buckets are each filled with a default taken from model input.

Technique (3) ensured that the system is geared to the rapid introduction of fixes, changes, and enhancements. We were aware that new rule types and the need to store new supporting data in domain objects would mean that, even with the stability ingrained via technique (1), we would have to make code-level changes. We believed that most of the changes would come via specializing classes using CLOS forms, rather than via modifications to the underlying structure of the simulation micro-kernel. For instance, during initial month-long prototyping, we implemented three radically different forms of resource tradeoff rules by extending a basic parent rule class.

THE SOFTWARE DEVELOPMENT PROCESS

We now give an overview of the development process used for MARS and some of the key design decisions. Though our software development group had the formal requirement of using the Shlaer-Mellor methodology [Shlaer 1992] as the methodology of choice, we were less tied to one formal methodology, and borrowed from other methods [Booch 1994, Martin 1992] where appropriate. We took the MARS project through the formal sign-off process steps in our organization and produced some of the work products described below to meet the requirements of this process.

Requirements Phase

We began with some basic requirements. While we wanted to build a general and extensible simulation system, we were willing to narrow its scope to our style of manufacturing when we deemed it appropriate. We knew the basic workings of our Fabs and their current style of WIP-management rules. We had a few new types of rules we wanted to study, including those using constraint-management techniques [Goldratt 1986], but we knew that these rules, and the required data needed to support them, would be one of the most changeable parts of the system. We wanted the simplest of these changes to be achievable by non-programming manufacturing personnel. We wanted less than one hour of run time to get answers for full Fab-level simulations involving more than 300 process steps, and on the order of the same number of machines, for months worth of simulated time, while starting more than 4K wafers per week into the line.

All accounting rollups would be on a shift-by-shift basis. MARS would store all history data for later on-line analysis. MARS should collect enough information to plot:

- Lot "Outs" per shift per step,
- WIP per shift per step,
- Machine setups per shift,
- Machine utilization per shift, etc.

These requirements lead to some interesting compromises. We decided to model the Fab resources at the level of a number of single-usage, non-interruptible machines per process step. Additional delays would be used to represent pipelined machines. Resource availability would be modeled as “buckets of availability” per shift. There would be one “route” for product modeled. For runtime speed, true “lot” objects would not exist and we would merely count WIP queue lengths.

Some of these initial requirements changed, and we will discuss this effect in a later section.

Analysis and Design Phase

At the top-level of analysis, MARS was designed to operate by simulating the beginning and ending of *steps* (i.e., recipes) running in *batches* (some number of lots) on *machines.* Steps may run on a number of alternative machines, each with different parameters, such as cycle-time, the amount of operator time required, maximum batch size, etc. Each step has a WIP buffer of waiting lots in front of it.

The top-level resource assignment algorithm of MARS proceeds by looking for empty machines. For any empty machine, MARS determines from the model what steps may run on it. If none of these steps have WIP, then the machine remains idle until WIP arrives. If only one step has WIP, then MARS (if the strategy rules allow it) starts the maximum batch that it can on the machine. If more than one step has available WIP, then MARS uses the strategy rules given by the user to resolve which step should use the machine.

Figures 1 through 3 show some of the main work products of the OO analysis phase. Figure 1 shows a portion of the basic object information model, with classes and relationships. This diagram is simplified by omitting inverse relations and cardinality notations (one-to-one, one-to-many, etc.).

This is an early analysis phase diagram. An important point is that such diagrams evolve as analysis, design, and implementation proceed. We will discuss major changes to this diagram below. However, take the case of the activity class, which represents the physical act of running a batch of lots on a machine. In the implementation of MARS, activity objects have two attributes (besides their start and stop times and the lots undergoing the activity) one of which is a pointer to the step object they are performing and the other a pointer to the machine object on which they run. Likewise, a machine has an attribute which is a pointer to its current activity. However, step objects only have pointers to the current set of lots processing at the step, so finding the activity objects associated with a step involves searching all the machines that can run the step. This was deemed appropriate given the algorithms used. Thus, because of the computational needs of the algorithm, the actual class relations implemented has changed from the overview given in Figure 1.

Figure 2 shows a simple object communication diagram for the End_of_Shift event which diagrams the main messages used in responding to that event. For simplicity, other messages are not shown. The simulation control object triggers messages to all machines and steps telling them to collect their shift-by-shift history data. The control object decides if it should place another shift end on the simulation event queue or whether the simulation has completed its required number of shifts.

Figure 1: Portion of Basic Object Information Model

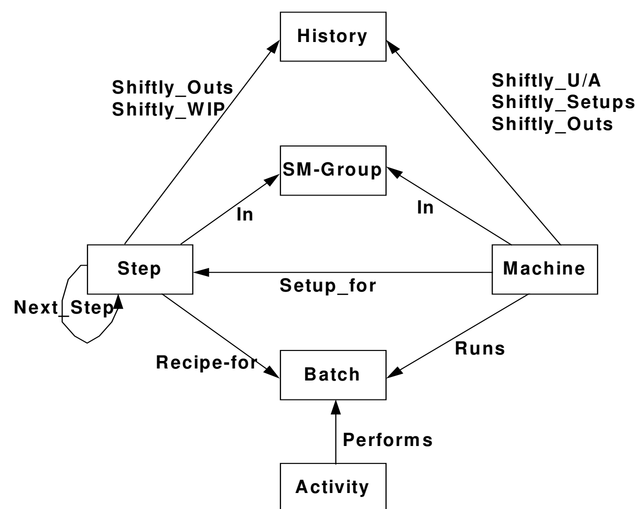
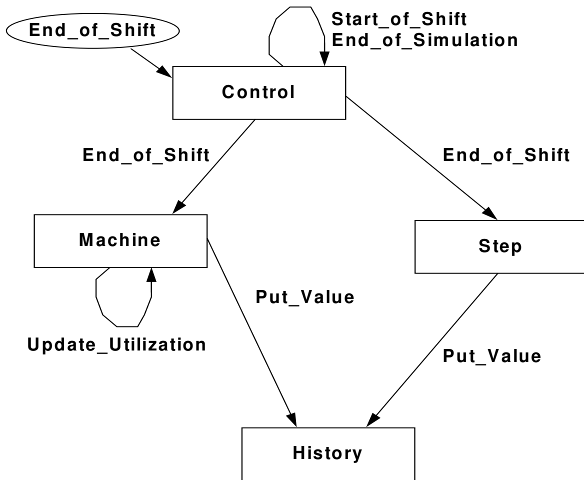


Figure 2:Object Communication Diagram for End_of_Shift Event



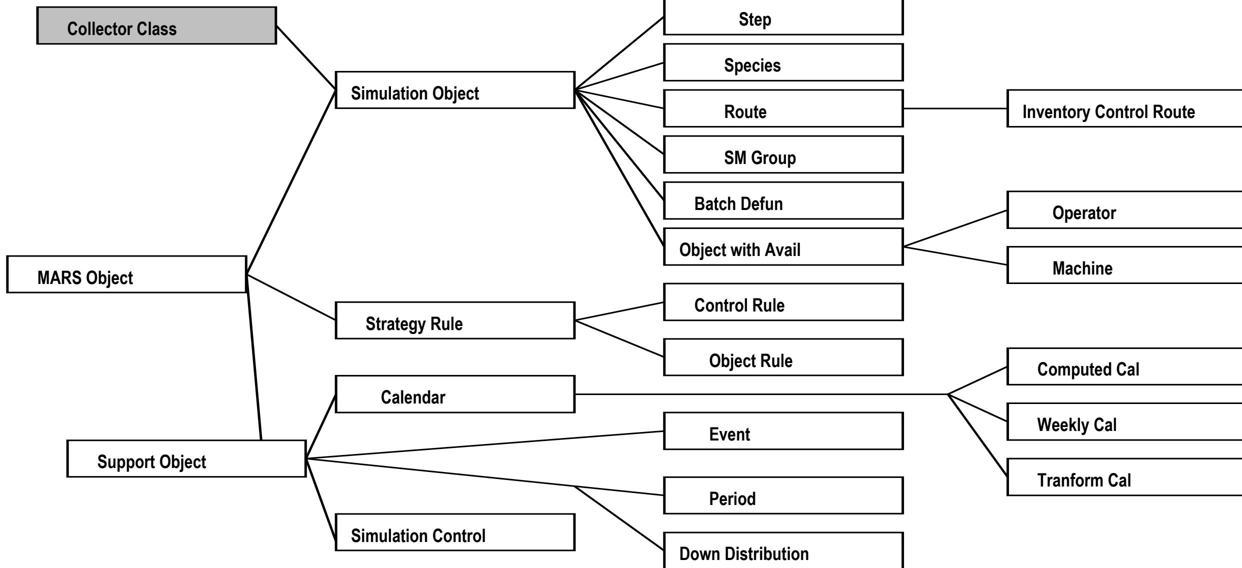
We produced communication diagrams for all events. We were careful to differentiate those communications that happened with delay (thus going through the discrete event simulation queue) and those that happened without delay.

Figure 3 shows a portion of the current implementation inheritance hierarchy. We talk about some of the design tradeoffs present on this diagram in the next section.

DESIGN TRADEOFFS

In this section, we give details of some of the more interesting design tradeoffs in MARS.

Figure 3: Portion of Inheritance Hierarchy



Local versus Global Decision Making

A natural tendency when using an OO approach is to focus on algorithms composed of local decisions made by objects based mainly on their own local state; It is well known such greedy heuristic methods can be sub-optimal [French 1982].

To account for more global algorithms, our solution is to include the decision making object as another object class. Thus in Figure 1, you see an SM-Group class, which actually embodies the rules needed to make all resource tradeoff decisions for all steps which potentially interact by running on an overlapping set of machines.

Another interesting approach which solves a piece of this problem uses CLOS multi-methods. These encapsulate the protocol among multiple class into one method. Rather than dispatching on a single class, as in most common OO language such as Smalltalk, multi-methods can dispatch based on multiple classes at once. For instance, the method definition:

```
(defmethod compute-cycle-time ((step basic-step)
                               (machine pipeline-machine))
  ...)
```

could potentially behave differently for the (basic-step, pipeline-machine) (step, machine) class pair than any other pair of step and machine classes. However, we have used multi-methods sparingly in MARS.

Collecting Groups of Objects

We have used various forms of object collectors in MARS.

Note the meta-class CollectorClass in Figure 3. Using the CLOS Meta-Object Protocol (MOP) [Kiczales 1991], it allows classes built from it to collect their instance, a behavior not supplied by default by CLOS. The simulation uses this behavior to iterate over all objects of a certain type. This has one drawback, namely that an object can only be in one such collector (since in CLOS an instance is a member of only one class at a time). To remedy this, we occasionally use a generic collector class, such as a queue, which can hold any type of object.

In our implementation, an instance of one of the rule classes shown in Figure 3 is actually the definition of one generic rule type. As such, it is the instances themselves which function as collectors of the rule forms input from the user.

Finally, the single instance of the Simulation_Control object provides a place for more global collectors such as the cached set of ready-to-run SM-Group objects.

Machine Types and Classes for Machine Types

If you have two types of machines, say a certain type of Lithography machine and a certain type of Etch machine, a natural tendency is to create a class for each and inherit from the generic machine class. If these "Litho" and "Etch" classes are to be part of the input of the user model, rather than hardcoded into the simulation, this can create a problem for some compile-time only class languages, such as C++.

In fact, though dynamic class creation is quite easy in CLOS, we have avoided it in this context in MARS for two reasons. First, our initial requirements did not determine a need to have method dispatches based on these semantic categories. Second, specialized classes based on the behavior modification required, not on one simple "type" of the machine, is a better model of the true semantics of the situation. This is because machines may be in a number of different such categorizations. In our implementation, each "machine" object has a pointer to one or more "equipment family" objects which control what summary data is collected and reported.

Multiple Inheritance

CLOS supports multiple inheritance, though some OOAD techniques suggest avoiding its use unless the class structure lattice is of a certain type [Booch 1994]. Though the inheritance diagram in Figure 3 does not show them, we have used multiple inheritance occasionally, specifically, to share the structure of objects which control WIP queues (steps and species) and those which are resources (species and machines).

At one point in the design, step inherited multiply from classes queue, wipped-object, and linked-list.

Class Generality versus Specific Encoding of Behavior

There is a certain dynamic tension in most OO development. For us, steps represent (at least!) two main things: a logical place where WIP queues and a specification for a recipe type runnable on some set of machines. As such, steps need both queuing behavior and behavior to retrieve relevant activity parameters, such as cycle times.

The reasoning in [Glassey 1990] would suggest steps represent a placeholder for both of these, more fundamental, behaviors. The "step" class should add them, either through aggregation or through multiple inheritance. We agree and have used both techniques where appropriate.

However, this sometimes becomes a question of implementation convenience. Very often we create the desired behavior "on the fly" in the object to avoid the syntactic pain of mapping domain specific method names into the generic method names (of course, this assumes the behavior is easy to replicate). In fact, the queuing behavior for steps became so specialized and we used it so often that it became semantically untenable, syntactically clumsy, and computationally troublesome not to bury very specialized queuing code inside the step class.

As a specific example, for a FIFO lot selection rule, it became computationally expensive to insist that the Add_to_WIP method associated with a step call the Add_to_Queue method with the sort field set to a lot's step entry time. Therefore we created a specialized subclass of queue which did only FIFO entry but did it very efficiently. But, since the FIFO lot selection rule is used by default by MARS, all baseline step objects actually call internal FIFO entry functions directly, without going through an intermediate queue object. A simple specialization of the baseline step class allows more complex behavior to come by using an external queue object.

High-level User Rules

One of the most troublesome of the initial requirements was that rules be modifiable by non-programming manufacturing personnel. In some sense we were schizophrenic on this requirement. We knew that the support of most experimental rules would come from programming-level class specialization. However, we worked quite a bit on "high-level" user input macros for both the rules and the domain objects in the user's models. To do so required us to focus on one highly adaptable form of constraint-based rule.

Constraint-based strategy rules are much like standard dispatching rules except they have a more global character. Rather than decide what to run next based on the arrival of lots in the WIP queue for a resource class (for instance, a FIFO rule), one should decide what to run based on the rate and buffer goals at the constraint.

Because of the re-entrant flow of lots in Fab manufacturing, in which the constraining resources are typically used in multiple steps in the process flow, we distribute the notion of the constraint resource to the *constraint steps* that utilize the constraining resource. For these steps, the user sets the desired production rate by giving each a *drumbeat* which specifies the output desired for the step per shift. Likewise, the user may give a *buffersize* to each step, which specifies the amount of WIP it is useful to maintain ahead of the step for safety reasons. It is of primary importance to meet the drumbeat of constraint steps every shift. Of secondary importance, the buffers at these steps should be full.

MARS uses these priorities to determine the strategy for allocating WIP at steps to idle machines for both constraint machines and non-constraint machines. At the beginning of every shift, MARS calculates the number of lots that must move into a constraint step to meet both its buffer and drumbeat goals. The calculation needed is:

$$Lots_needed = drumbeat + (buffer_size - WIP)$$

which is the number of lots which need to flow into the constraint step and, hence, an *implied* or *calculated* drumbeat for the step feeding the constraint step. This calculation iterates backwards up the line assigning drumbeats to each non-constraint step until the necessary WIP is found.

Note that there is a subtle object interaction with this design. We designed the class SM_Group to handle all these resource tradeoff decisions. However, step objects belonging to an SM_Group which use a constraint rule have user-set buffer and drumbeat information. We decided, by default, that all steps would store this information in internal slots.

Figure 4 shows a portion of the user-level input of a constraint rule. Certain model specific items are abbreviated in italics to save space.

Figure 4: Example User-Level Constraint Rule

```
(def-strategy-rules
  (base-route-order list-of-routes)
  (step-ordering default
    (under-drumbeat :wip) (overfull-buffer :wip) (underfull-buffer
:wip))
  (step-ordering Litho-Steps
    (under-drumbeat :step-filter ((list-of-critical-steps :pull) (t :pull)))
    (overfull-buffer :pull)
    (underfull-buffer :pull))
   (drumbeat litho-steps num-of-lots-per-shift)
   (buffersize litho-steps num-of-lots))
```

This rule states that certain routes are more important than others (the base-route-order form). It says that all step/machine groups should run the step with the largest current buffer, except for steps performing lithography. For these steps, any step under its shift drumbeat goal which is in the *list-of-critical-steps* runs before all other steps. In all categories, the step closest to the end of its process (the :pull rule) is run before another step.

Use of Dynamic Classification

We have not made extensive use of object state diagrams [Shlaer 1992], so we haven't made extensive use of the *dynamic classification* method of [Martin 1992], though the CLOS change-class form [Steele 1990] could support it. We do use it in two places:

- as inputs of model objects are being read they change into instances of more refined object classes as more of their details become known. These details include both the form of the external syntax read and the detailed semantics contained in the form.

- after a model is read, non-baseline extensions are made by placing an object in some extended class which has new behaviors.

As an example of the later, consider this simple case of a route which, rather than releasing work onto the floor in a fixed amount per shift, as does the baseline route class, would maintain the inventory on the floor at some fixed amount. That is done with the definition show in Figure 5. It works by overridding the default behavior of the num-of-starting-lots-for-shift method.

Figure 5: Example Code to Allow A Route to Control Inventory

```
(def-class inventory-controlling-route (route)
   (desired-inventory 0))

(defmethod num-of-starting-lots-for-shift
                        ((route inventory-controlling-route)
                         (shiftnum integer))
   ;;;code to compute and return number of lots to maintain
inventory
   )

(defmacro declare-route-to-be-inventory-controlling
                                                        (route desired-inventory)
  `(progn (change-class ,route 'inventory-controlling-route)
           (setf (desired-inventory ,route) ,desired-inventory)
           ,route))
```

REQUIREMENT CHANGES AND THEIR DESIGN TRADEOFFS

After initial requirements, analysis and design, we began an iterative process of prototype development followed by re-analysis, design, and implementation. However, for the most part, new formal analysis workproducts were not produced. We believe this is unfortunately typical in most OO projects, and it was encouraged by our advanced CLOS development environment [Symbolics 1990] which focuses on supporting rapid changes at the CLOS code level rather than at the analysis level.

We will now discuss some highlights of the requirement changes and design tradeoffs produced during the iterative prototyping.

Laundry List of Requirements Changes

The main requirement changes added more detail about the complex behavior of objects:

- Operators as an additional resource in operations,
- Multiple routes, including rework routes,
- Lots as full fledged objects,
- Collection of lot throughput times,
- Transport times between operations,
- Complex rules for setup control,
- Non-constraint style rules based on lot priorities, etc.

In addition, we made a number of changes to maintain fast simulation run times.

In general, most changes consisted of simple specialization to the current set of base objects and the addition of more message coordination between objects to ensure locations to hook in new behavior. Early on, these additions effected the "simulation core" as we explained it previously, but these type of additions tended to decrease.

Obviously, this later stability came via design. For instance, to include transport times, we merely modeled step-to-step times and placed lots in a new IN-TRANSPORT state while holding them in the WIP buffer of the step to which they were moving. A new event type releasing lots from the transport state was also needed. Then, the method that gave a step's runnable WIP was modified to filter out lots which were IN-TRANSPORT. However, a much bigger change to the simulation core would have had to have been made if we had chosen to model a full transport system, with resource contention among delivery vehicles.

Some requirements changes greatly effected our implementation class lattice. When there was only one route, steps inherited from a link-list *mixin* [Paepcke 1993], so that they directly stored their previous and next steps. However, when multiple routes were introduced, steps could no

longer take this liberty and the route object itself had to take over the next/previous step storage.

Our early focus on implementation efficiency allowed us to design in only those abstractions that we knew were required. On the other side of the argument, as the previous linked-list example shows, we did occasionally have to tear down much of the structure of MARS to incorporate a radical requirements change.

Changes Related to Run Time Reduction

Many of the most effective changes to hold run time in check while increasing behavioral complexity were from caching information. Our original design followed the principle that a piece of information should only be held in one place. For instance, in the initial design only steps contained pointers to batch information. Asking what steps ran on a machine involved looping through all steps in the SM-Group of the machine and collecting a list of steps. However, this was asked often enough that it was effective to pre-compile this list into a slot on each machine object.

The problem with pre-compiled information is that it makes on-the-fly changes more troublesome and error prone. Our solution is to expose cached information using three techniques: (1) Every cache has a dirty flag to say when it needs to be recomputed, (2) Every method that changes a value cached elsewhere is given an *:after* method that sets this dirty flag, and (3) The method that reads the cache value has a *:before* method which checks the dirty flag to see if the cache needs to be recomputed. Of course, the latter two technques depend on CLOS's support for before/after daemons.

EXAMPLE OF BEHAVIOR SPECIALIZATION

As mentioned before, using an Object Oriented approach allows us to focus on two key items: (1) the domain objects, so the design is stable, and (2) the object methods, so that the interacting objects are de-coupled and encapsulated while places to "hook" new behaviors are exposed. While MARS doesn't have as many hook points as Autosched [AutoSimulations 1993] or the class generality of BLOCS [Glassey 1990], it does contain enough of both to allow some interesting modification to be achieved very simply. In addition, the choice of CLOS as the object-oriented language of MARS allows some interesting behavioral specializations to occur that would be difficult in other languages.

Example: Adding Calendars to Arbitrary Slots

One of the most interesting uses for a simulator is to look for good strategy rules when a Fab is *ramping*, that is increasing its wafer starts per week, its number of resources, its run rate of resources, and its process yield.

Initially, MARS was designed with the ability to represent ramps of machines, yields, and wafer starts. Occasionally, we wanted to represent and use cycle times which changed during the simulation period. This was accomplished through use of a calendar object, which stores a value indexed by the current shift number. Using CLOS, it was simple to replace the integer minutes of cycle time with a calendar object. The CLOS code segment shown in Figure 6 uses this calendar to return the desired, time-dependent, cycle time.

Figure 6: CLOS :around Method to Add Ramping Cycle Times

```
(defmethod batch-cycle-time :around ((bd batch-def))
  (let ((res (call-next-method)))
    (if (typep res 'calendar)
        (calculate-value res (current-shift-number))
        res)))
```

Interestingly, with only a minor performance penalty, this technique can be made general for *all* slots using the MOP without the need for the slot-by-slot :around method used in Figure 6.

Example: Counting the Number of Concurrently Used Machines

An interesting user request was for the average number of machines concurrently used on some step. The code in Figure 7 shows a simple specialization which gathers that information.

Figure 7: Code for Counting Concurrent Machines

```
(def-class step-concur (step)
   (concur-count 0)
   (concur-history nil))

(defmethod start-of-simulation :after ((step step-concur))
   (setf (concur-count step) 0)
   (setf (concur-history step) nil))

(defmethod start-running :after ((step step-concur) event)
    (incf (concur-count step))
    (push (cons (current-time) (concur-count step))
          (concur-history step)))

(defmethod end-running :after ((step step-concur) event)
    (decf (concur-count step))
    (push (cons (current-time) (concur-count step))
          (concur-history step)))
```

From this gathered information, we could then compute the expected number of machines running a step, using the integration technique outlined in [Glassey 90]. Since MARS also has a start-running-event message dispatching on *both* a machine and step, one could also easily count concurrent usage for only a certain classes of machines.

INFORMATION ABOUT THE CURRENT SYSTEM

MARS currently contains 38 base classes and 22 rule classes (WIP management, setup, and wafer start rules). In a complex Fab environment, it simulates one week of simulated time in about 5 minutes of real time. It was built in less than a year by two software engineers, one full time and one part time. The first prototype was up and running in less than one month and all changes there after were intermingled with modeling and simulation experiments.

MARS has been used at Intel to help delineate and test manufacturing strategies for most of our high-volume advanced microprocessor Fabs. We have used it to study new manufacturing strategies which would be too costly to try on the floor. It is currently being used to help design the strategy and machine set of our newest billion dollar Fab.

CONCLUSIONS

Were we successful? We are quite happy with our management of the competing system requirements of (1) rapid change, (2) modeling generality and (3) swift run times. For instance, the turn around time on most strategy experiments suggested by manufacturing personnel has been less than one week.

Some exceedingly complex rules can be encoded in our rule language. Most other changes can be made by specialization on the current set of base classes. Early on, however, we came to the conclusion that, due to other commitments, MARS would never be a simulator that was delivered to manufacturing personnel. Thus, all naive user ease-of-use considerations were dropped as requirements.

The simulation-building approach taken in MARS used a state-of-the-art software development environment based on OO principles and augmented with simulation class libraries. For the high-power user interested in implementing complex new behaviors, we believe that this approach produces better systems faster than the use of off-the-shelf commercial simulation tools. This is due primarily to the fact that there is much more industry effort and competition in developing general software development environments than in more specialized simulation environments.

Given the range of system changes we made and the additional functionality we provided in MARS, we believe that focusing on the key OO techniques outlined in this paper has proven worthwhile in keeping it an extensible and speedy simulation system.

REFERENCES

AutoSimulations, Inc., *AutoSched and the Simulator System User's Manual*, Bountiful, Utah, 1993.

G. Booch, *Object-Oriented Analysis and Design with Applications*, 2nd Edition, Benjamin/Cummings, 1994.

S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Ellis Horwood, 1982.

C.R. Glassey and S. Adiga, "Berkeley Library of Objects for Control and Simulation of Manufacturing (BLOCS/M)", in *Applications of Object-Oriented Programming*, ed. L.J. Pinson and R.S. Wiener, Addison-Wesley, 1990, pp. 1-27.

E.M. Goldratt, *The Goal: A Process of Ongoing Improvement*, North River Press, 1986.

S.E. Keene, *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1989.

K.G. Kempf, "Intelligently Scheduling Semiconductor Wafer Fabrication", in *Intelligent Scheduling*, M. Zweben and M. Fox, eds., Morgan Kaufman, 1994.

G. Kiczales, J. des Rivieres, D.G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

J. Martin and J.J. Odell, *Object-Oriented Analysis and Design*, Prentice Hall, 1992.

D.J. Miller, "Simulation of a Semiconductor Manufacturing Line", *Communications of the ACM*, Vol. 33, No. 10, October 1990, pp. 98-108.

A. Paepcke, ed, *Object-Oriented Programming: The CLOS Perspective*, MIT Press, 1993.

S. Shlaer and S.J. Mellor, *Object Lifecycles: Modeling the World in States*, Prentice-Hall, 1992.

G.L. Steele Jr., *Common Lisp: The Language*, 2nd Edition, Digital Press, 1990.

Symbolics, *Symbolics Common Lisp Programming Constructs*, Burlington MA, 1990.

R. Uzoy, C.Y. Lee, and L.A. Martin-Vega, "A Suvey of Production Planning and Scheduling Models in the Semiconductor Industry Part I: System Characteristics, Performance Evaluation and Production Planning", *IIE Trans. on Scheduling and Logistics*, Vol. 24, 1992, pp. 47-61.

L.W. Wein, "Scheduling Semiconductor Wafer Fabrication", *IEEE Trans. on Semiconductor Manufacturing*, Vol. 1, No. 3, August 1988, pp.115-130.

BIOGRAPHY

Dr. Smith recieved the B.S., M.S., and Ph.D. degress in Computer Science from Michigan State University in 1977, 1979, and 1982, respectively.

Since 1992, Dr. Smith has worked at Intel on various problems involved in the planning, scheduling and control of semiconductor manufacturing. His current interests are computer models which capture important manufacturing tradeoffs and the process by which such models become orgranizationally useful and supportable.

Prior to joining Intel, Dr. Smith was a research scientist at Northrop's Research and Technology Center. He worked on a wide variety of advanced AI-based software systems, including those for manufacturing scheduling and planning, image understanding, pattern recognition, and computer-supported collaboration.

Dr. Smith has more that dozen technical publications and has served as a technical reviewer for the National Science Foundation, IEEE Trans. PAMI, IEEE Trans. SMC, IEEE Expert, and ACM Computer Reviews. He is a member of IEEE, ACM, AAAI, and Phi Kappa Phi.