Proceedings of the 1992 Winter Simulation Conference
ed. J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson

SIMULATION THROUGH EXPLICIT STATE DESCRIPTION AND ITS APPLICATION TO SEMICONDUCTOR FAB OPERATION

Mutsumi Fujihara
Kiyoshi Yoneda

Toshiba Corporation Systems and Software Eng Lab
70 Yanagi, Saiwai, Kawasaki, 210 JAPAN

ABSTRACT

A nonstandard methodology in simulator construction is introduced along with its application to online simulation of semiconductor fabs. The simulation proceeds by sequentially rewriting state description, written in a language. A text in the state description language consists of a collection of titled two-column tables. The description is stored in an in-core database equipped with a splay tree algorithm. The initial state representing the fab's present state is downloaded from the fab's database. The simulation is for lotwise trace prediction needed for day-to-day fab operation rather than to estimate steady state performance indices. Its use at production sites involves frequent model modification. A one-month simulation of a clean room operation involving 1,300 lots and 200 machines is processed in five to ten minutes on a workstation, using 30 megabytes of internal memory. The execution time is $O(logN)$, where N is the number of lots and machines; memory usage is proportional to N. The simulator enjoys an extremely small source code: 1,200 lines in C.

1 INTRODUCTION

This paper proposes to look at discrete event simulation as a form of *language processing*. The word "language" here means a system of symbols used to describe the *state* of the object to simulate rather than a simulation language such as SIMULA or GPSS.

Consider the case of simulating a semiconductor fab. The contents of a *state description* would be like this: "Lot L consists of W wafers of product type P. Lot L is presently being processed by machine M, scheduled to finish at time t. Machine M is processing lot L and will go through maintenance after time s. Etc..." The system of symbols used to describe this is called a *state description language*. The language may resemble a natural language as above, or may be more succinctly coded. Simulation proceeds by rewriting such a description step by step according to a set of rules. This process parallels interpretive execution of a computer language.

The language is to be used by a team involved in building a model for negotiating an agreement among themselves and to communicate with computers for discovering mistakes, delegating execution, and receiving the result reports. We design a common language for people and machines, taking them as equals with different characteristics and abilities. Consequently we start with what computers are rather than what they should be.

This is in contrast with modern computer languages which try to hit a balance between human and machine processing efficiencies by defining separate languages and bridging them with translators: a graphic model in a visual language is translated into a compiler language which is in turn compiled to a machine language to be finally executed; its output in table form is translated into animation. The language for human use is based on some powerful concept, computational model, or metaphor contending what computers ought to be like. This approach is unaffordable for us because it requires building a large amount of software.

Section 2 explains the background motivation for considering the state description language approach in semiconductor fab simulation. In Section 3 we give a rationale for limiting the scope of language design so it depends on efficient handling of ordered multisets. Section 4 summarizes the splay tree algorithm for ordered multisets. The design and implementation of the language, described in Sections 5 and 6, is based on the nature of this algorithm. The problem of balance between description flexibility and execution performance is addressed in Section 7. Section 8 relates experience in using the language for semiconductor fab online simulation. Finally in Section 9 we summarize our approach.

2 SIMULATION AS LANGUAGE PROCESSING

A qualitative change is taking place in demand for simulation that arises in semiconductor factories. Until several years ago simulation was treated as a phase in research type projects, as described by Burman *et al.* (1986), Dayhoff and Atherton (1987), and Miller (1990). Typically, a client would come up with two or more alternatives in production method or system configuration, looking for data with which to decide on the best. Simulation would be run offline; the decision is made by observing steady state performance indices. Numerical solution of queueing networks as in Chen *et al.* (1988) or Inoue and Yoneda (1989) often work better in such cases, especially when optimization is involved as in Yoneda *et al.* (1992). Today, a typical demand would be as follows.

Factory managers wish to run simulation routinely to use the results in instructing the operators on the work to be done during a time span. They prefer that the simulation be initialized automatically by downloading the *status quo* from production and quality control computers. The result they need is individual behavior of products and resources in the form of predicted *traces* rather than statistical performance measures. Since factories constantly change systems, easy model modification is essential. Numerical solution of queueing networks does not help not only because we need lot traces but also because we are not interested in the steady state.

A natural direction to take in order to fulfill the above requirements is the separation of data and program. That is, a general purpose database stores the downloaded data; all programs are designed to access a common database. Such architecture permits most work of model modification to be confined to data modification, having little to do with programs. For simulation purposes it is necessary that the database be totally incore since external memories are too slow. The data downloaded into the database represent the initial state for simulation.

In order to point out a resemblance between simulation and language processing, let us take a look at execution of a computer language. It proceeds by sequentially rewriting the contents of the memory according to the instructions given in the form of a program. Discrete event simulation parallels this in the sense that it is a process of sequentially rewriting state description according to the instructions given in the form of an event list. Figure 1 depicts this view taking a fab simulation as an example. The similarity is no coincidence, by the definition of "computer simulation." The object we wish to simulate is described in a language, which is in turn executed by a computer. Thus a simulator allows to be seen as a language processor.

The similarity is clearer in the simulation of a computer's instruction set. The instruction set of a newly designed computer is implemented in terms of that of an old computer, which amounts to writing an interpreter such that executes programs written in the new computer's machine language. The program in the new language represents the initial state of the finite state machine to simulate. The execution of the program by interpretation entails rewriting the states sequentially, starting from that initial state. Here the incore database or data structure to store the internal state of the machine plays an essential role in both ease of implementation and performance.

The largest difference between simulation of a computer and discrete event simulation regarding implementation is that while in the former the chronological order of execution is controlled by an instruction counter, in the latter the same is accomplished by an event list. Consider a computer address space or an event list as an *ordered multiset*, a set allowing multiple appearance of elements with a linear order defined among the elements. From an abstract point of view, both instruction counter and event list mechanisms are implementations of an ordered multiset of rules to *transform* states, such that the ordered multiset itself forms a part of the state: the rules contained in the ordered multiset transforms the ordered multiset itself, like *self-referencing* in languages.

Tables 1 and 2 summarize the similarities between simulation or database operation and language processing.

3 A MULTISORTER FOR LANGUAGE PROCESSING

A problem which arises immediately is how to hit a good balance between the flexibility in describing system states and the efficiency in analyzing its *representation.* A simple machine language is easy and fast to process but limited in convenience, while a language with large specification would be powerful but slow to process. With this problem in mind, in this section we propose to limit the scope of the state description language design. The class of languages we choose is such that their structures fit into ordered multisets: in short, we call multisorted table structure a language.

We pointed out in the previous section that event list and instruction counter are both ordered multisets. There is no doubt that representation of ordered multisets is fundamental in a language for describing

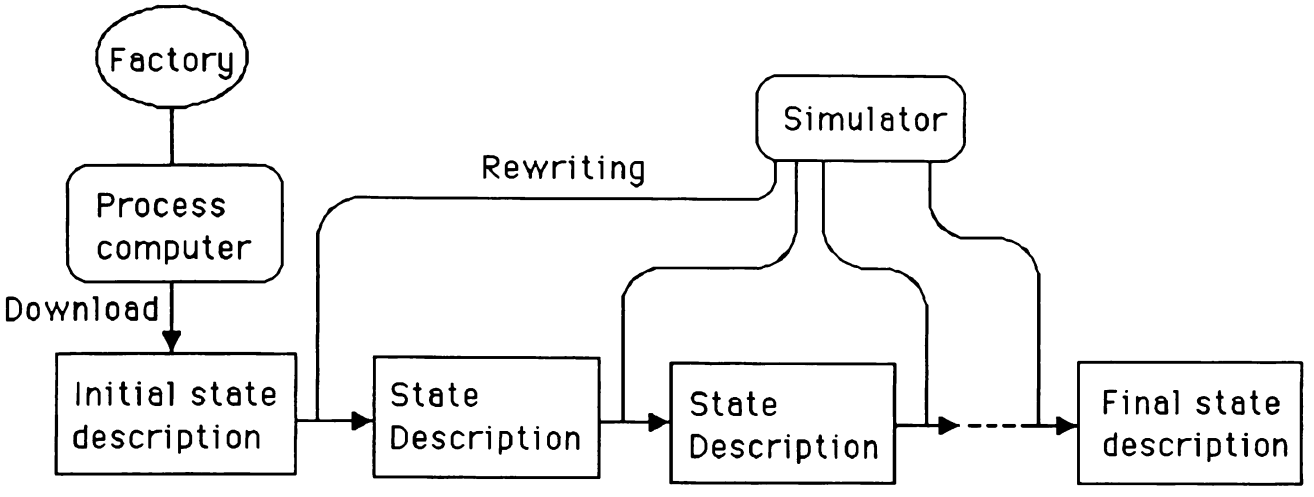

Figure 1: Simulation as a Sequence of State Description Rewriting

Table 1: Correspondence between Simulation and Language

Simulation	Language
State description represents a system state according to a format.	**Sentence** describes an idea according to a grammar.
Simulation sequentially transforms a state description according to rules in the event list.	**Interpretation** sequentially rewrites a string of symbols according to rules in the program.
Event list determines the sequence in which events occur.	**Execution control** determines the sequence in which instructions are executed.
Self-transforming instruction to change state is a part of the state itself.	**Self-referencing** is possible in many languages.

Table 2: Correspondence between Database and Language

Database	Language
Data access is performed according to relationship among data.	**Context analysis** is performed according to relationship among words.
View represents the same relationship seen from different angles.	**Voice** represents the same meaning under different forms.

states, even though there may be other useful structures. Therefore efficient handling of ordered multisets is essential for a state description language. The same can be said for an incore database. That is, a large portion of data access operations seem to require some kind of ordering: picking the n-th of an order, inserting new data according to the ordering, or checking an entry in an ordering to see the corresponding data in another ordering.

Looking at computation this way reveals that well over a half of the operations performed in nonnumerical computation, such as business type information processing or systems programming, seem to reduce to ordered multiset handling. Perhaps this is not surprising since in a Von Neumann type computer, data are represented as bit patterns sequenced according to the order defined by address. From this observation we conclude that the determination of data address from an arbitrarily defined data ordering is a fundamental operation in computing. The idea of associative memory is based on the same premise.

With these considerations we restrict our attention to the class of languages which allow efficient analyses provided there is some efficient means of handling ordered multisets. This amounts to limiting the language we consider to multisorted tables. We adopt the *splay tree* algorithm as the basis for this. The task of language design and its solution now reduces to the items shown in Table 3. The following three sections are arranged according to the Table: we briefly touch the splay tree, then describe the state description language, and finally explain the implementation.

4 SPLAY TREE

A splay tree is a binary search tree which, each time a node is accessed the tree structure is reorganized adaptively. The reorganization is performed in such a way that the most recently accessed node will be the root, which is the quickest position to access. At the same time, the path length from the old root to the new is reduced to about a half the previous length. The data structure and the algorithm were introduced by Sleator and Tarjan (1983, 1985).

The principle of splay tree is like cache or virtual memories in the sense that they all make use of the locality in data access. A single access to a node in a splay tree will in many cases be slower than in a fixed structure binary tree, since an extra task of reorganizing is involved. However, the average time measured over a sequence of such operations as retrieval, insertion, and deletion will usually turn out considerably shorter with a splay tree. Such phenomena have been studied using the concept of *amortized time complexity* introduced by Tarjan (1985). The *amortized time bound* for a splay tree is $O(logN)$, where N is the number of nodes, meaning that the access will not slow down drastically as the tree grows.

Experimental results concerning the adequacy of splay tree for event list implementation are shown in Jones (1986). There are even faster algorithms if we limit our attention to event lists. *Calendar queue* by Brown (1988) is, for instance, intuitively and empirically with $O(1)$ average performance. While event list operation constitutes of deleting the topmost element and inserting elements more or less randomly, an algorithm for generic ordered multiset manipulation requires a wider variety of operations, e.g. finding the position of a specific element in the multiset. This is the reason why splay tree is preferable to calendar queue in our case.

There are two major variations in splay trees: *top down* and *bottom up*. A top down splay tree is faster than the bottom up (with the same order time complexity), but can be used only when the data access starts from the root. So the top down does not substitute the bottom up. By experimenting with and without the top down code we decided that the increase in program complexity caused by the inclusion does not compensate the gain in speed.

The splay tree may seem memory expensive as an incore database algorithm, requiring three extra pointers besides a key. However, the rate of increase in RAM capacity over the years is well known to have been much faster than the decrease in RAM access time or the increase in CPU speed. Since there is no sign of change in this trend, the general strategy of spending more memory in order to gain in time would seem to remain valid through at least several years.

5 A MINIMAL LANGUAGE

This section describes DEUS90 (short for *deus ex machina*, or Discrete Event Universal Simulator), the present version of the language used for state description. For the sake of explanation we treat the system as an incore database.

A *word*, which represents an *item* in the database, is one of the following types:

String-string pair:	`TIME EVENT`
String-integer pair:	`SEC 1198`
Integer-integer pair:	`-61232 24`
Real:	`29.89`

The words are totally ordered by rules such as: paired words are ordered first by the left element and

Table 3: Language Design Tasks and Solutions

Task	Solution
Devise an efficient algorithm for handling ordered multisets.	Employ the splay tree algorithm.
Give a minimal specification of a state description language.	Describe states in the form of a table whose entries are sorted in various ways.
Implement the language using the algorithm.	Organize the table as interlaced splay trees.

then by the right; strings are ordered by appearance; strings precede integers; paired words precede reals; and numbers are ordered naturally.

A *line*, corresponding to a database record, comprises three words as in

$$w_L \; w_C \; w_R$$

e.g.

```
SEC 1198        TIME EVENT        lot_arr 78354
```

Here w_j are individual words, with $j = L, C, R$ standing for left, center, and right, respectively. The design rationale is that we would need to describe at least an entity, an attribute, and a value. Thus in many cases in practice $\{L, C, R\}$ is used as if it meant {entity, attribute, value}, even though there is no such rule. With less than three words per line practical state description would be impaired.

A line

$$w \; x \; y$$

appears in the following four *forms*, or views in database terminology, simultaneously:

(w, l) Left w center-sorted:
the lines whose left (L) word is w, sorted with respect to the center (C) words.

(x, L) Center x left-sorted:
the lines whose center (C) word is x, sorted with respect to the left (L) words.

(x, R) Center x right-sorted:
the lines whose center (C) word is x, sorted with respect to the right (R) words.

(y, r) Right y center-sorted:
the lines whose right (R) word is y, sorted with respect to the center (C) words.

Note that this is not symmetric with respect to $\{L, C, R\}$: there is an emphasis on the center C word. A consequence is that accessing from C to L or R and vice versa will be easier than from L to R and back, both notationally and operationally and hence in terms of performance.

A *text* is a denotational representation of a form, i.e.:

$$\begin{matrix} w_{1L} & w_{1C} & w_{1R} \\ w_{2L} & w_{2C} & w_{2R} \\ w_{3L} & w_{3C} & w_{3R} \\ & \cdots & \end{matrix}$$

where w_{ij}, are words in lines $i = 1, 2, \ldots$ If the form is, say, center-x right-sorted (x, R), then all words w_{iC} in the center are x; if $h \leq k$ then w_{hR} weakly precedes w_{kR}.

Since there are four forms for each line, a line gives rise to four different texts. Upon registration of a line, the line is conceptually stored into four different but mutually related texts. When a line is modified, many texts may go through restructuring. For instance, suppose line $w \; x \; y$ is modified to $z \; x \; y$. Then:

(w, l) text gets line $w \; x \; y$ deleted.

(z, l) text gets line $z \; x \; y$ sorted in.

(x, L) text gets resorted.

(x, R) text gets line $w \; x \; y$ changed to $z \; x \; y$.

(y, r) text gets line $w \; x \; y$ changed to $z \; x \; y$.

A change in the center word causes similar but more extensive changes.

A line does not stand alone: it is retained only in the form of its representation which consists of the four texts. A line should be considered a unit for registration and deletion in texts rather than an entity whose projections are the four texts.

In order to be able to identify the context in which a line appears, a line is dealt with as assuming one of the forms at a given time, even though the four texts always exist. That is, given a line, one can ask

the form in which the line is currently in. If the user wishes to look at the same line in another form, the line goes through a transformation that changes its form.

Now we introduce a shorthand text notation. Suppose the text in the above example is in the center-$w_{.C}$ right-sorted $(w_{.C}, R)$ form. Then instead of writing extensively, we write the common center word $w_{.C}$ once at the top, followed by two-column lines of L and R words:

$$
\begin{array}{cc}
w_{.C} & \\
w_{1L} & w_{1R} \\
w_{2L} & w_{2R} \\
w_{3L} & w_{3R} \\
\multicolumn{2}{c}{\cdots}
\end{array}
$$

Similar notations apply to the other forms. With this notation, the state description language shown as a text in a specific form, is a collection of titled two-column tables, with each entry a word consisting of a pair or a real:

```
TIME EVENT
SEC 1198    lot_arr 78354
SEC 1354    stat    82016
...

STEP RECIPE
MPU_EQA3   723240    Furnace141 1440144
DRAM_3EQA4 721800    Etching993 8640144
...
```

The availability of four forms with equal handling efficiency turns the design of state description considerably more flexible than doing the same with a mere table such as the spreadsheet. This may easily be appreciated by designing a state description format using the language.

6 THE LANGUAGE PROCESSOR

There are forty or so language processing functions provided by DEUS90, which may be classified as follows:

Initialization. Clear the incore database.

Word composition/decomposition. Given strings and numbers, compose a word and backwards.

Line registration/deletion. Create a new line and position it. Given a line, delete it.

Word comparison. Given two words, compare their precedence.

Form identification/switching. Given a line, find its form. Switch the form to another.

Line extraction. Given a word and a form, find a line in the corresponding text.

Word extraction/modification. Given a line, pick its words. Given a line, change its word to another.

Line position finding. Given a line, find its position in the text.

Current line pointing. Change the line in focus.

Input/output. Read in or write out lines or texts.

Subroutine linkage. Link subroutines which, given an event name, tell DEUS90 how to rewrite the state description. These subroutines represent the lowest layer executor in interpreting the state description.

The data structure used for implementation of the above functions is illustrated in Figure 2. There are six lines registered in this example, listed in the middle part of the Figure. Six words, u to z, are involved. Any word, say x, defines four distinct forms, (x, l), (x, L), (x, R), and (x, r). To each form a text is associated. Each text is represented by a splay tree. The tree's node is a pointer. The pointer points to a word in a line. The line pointed to is such that appears in the text represented by the tree. The word pointed to appears in the line, in the position specified by the form as the sorting position: left if $(., L)$, right if $(., R)$, and center if $(., l)$ or $(., r)$. Each line appears in four distinct texts. Therefore, the three words in a line are pointed to by four nodes in four distinct splay trees: the left word is pointed to by a node in a tree representing $(., L)$, the right word by $(., R)$, and the center word is pointed to twice, by $(., l)$ and $(., r)$.

7 FREEDOM VS. PERFORMANCE

The design and implementation explained above may seem redundant having to maintain the four texts concurrently. In this section we discuss a way to reduce the redundancy and claim that the present design hits a good balance in convenience for processing either by human or by computer. Jakobson's remark (1959) "Languages differ essentially in what they *must* convey and not in what they *may* convey." applies here, in that language design concerns how

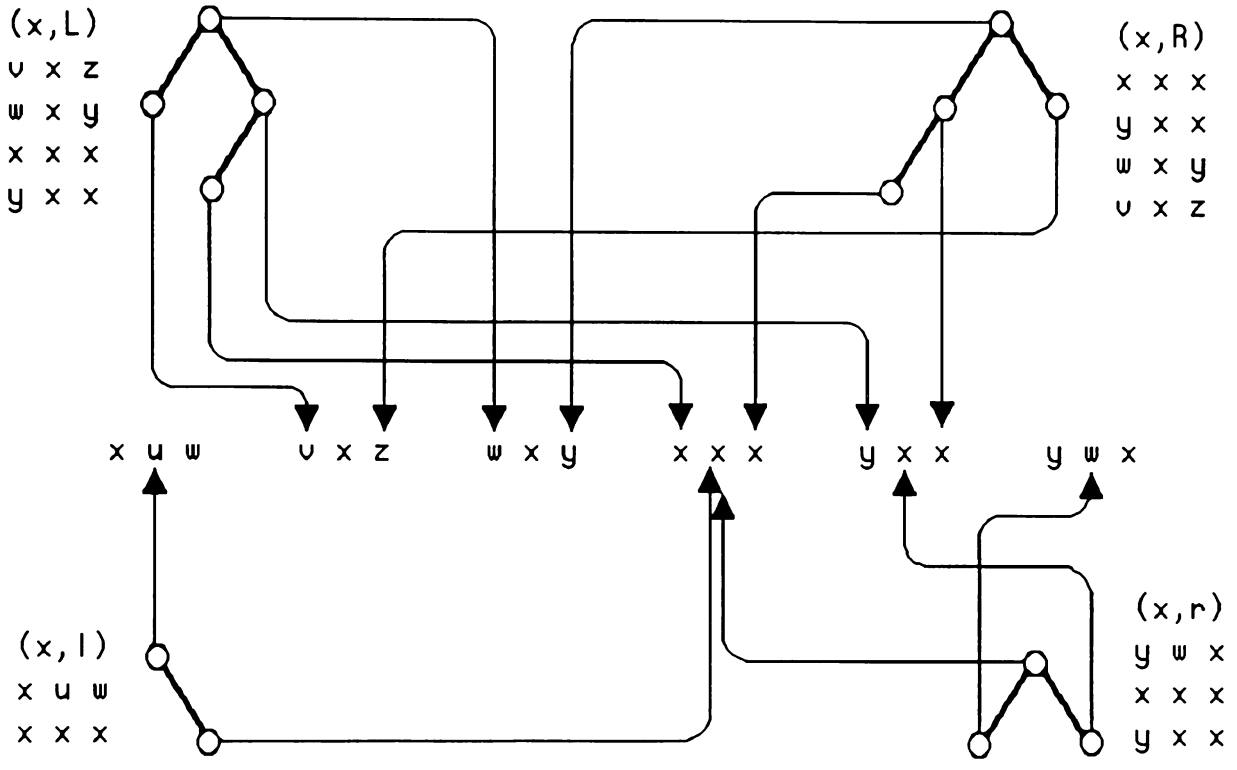

Figure 2: The Implementation

Table 4: Simulator Code Size

Language	Module	Size in lines	G/F	Comment
C	Splay tree	300	G	Total 1,200 lines
	Incore database	500	G	
	Event list processing	100	G	
	Event execution	300	F	
DEUS90	Downloaded data on lots and machines	3K	F	Total 4.5K lines
	Downloaded data on routing	1.5K	F	
	Initial state description	30K	F	After expansion
	Month's final state description	300K	F	Trace records

G: for general simulator; F: fab simulator specific.

to describe or ease of linguistic analyses rather than what is described or phenomena to simulate.

The state description format for the semiconductor fab simulation can be designed so as to use only $(., L)$ and $(., R)$ forms, without $(., l)$ and $(., r)$. For instance, consider this (`ProductA Lot1D1`, l) text:

```
ProductA Lot1D1
LOT STEP         Diffusion    1200
LOT SIZE         Wafers         20
LOT MACHINE      DifMCA         60
LOT DELIVERY     Before     921224
...
```

All this information on `ProductA Lot1D1` can be picked up using only $(., L)$ form but various texts with different C words:

```
LOT STEP
ProductA Lot1D1     Diffusion 1200
ProductB Lot1D2     Test      3600
...

LOT SIZE
ProductA Lot1D1     Wafers 20
ProductB Lot1D2     Wafers 10
...

LOT MACHINE
ProductA Lot1D1     DifMCA  60
ProductB Lot1D2     TesterB 40
...

LOT DELIVERY
ProductA Lot1D1     Before 921224
ProductB Lot1D2     Before 921226
...
```

This eliminates the necessity of two among the four texts, hence reducing the maintenance at the cost of losing options among the forms, meaning less freedom in state description.

An experiment in doing this *without actually reducing the number of forms supported by the language processor* resulted in a 30% increase in execution time. This may be explained as follows: less options in texts implies less freedom in choosing the shortest among the texts; during the simulation the program ended up searching longer texts.

The execution time of an update operation will be halved by limiting the number of forms from four to two. However, in other kinds of operations the execution time depends on text length and access pattern rather than the number of forms. Therefore the execution time for the limited version would be over a half the original: the 30% increase mentioned above may eat up the gain.

From this observation we conclude that the human and the machine information handling efficiencies in DEUS90 are well balanced with respect to our present concern, which is semiconductor fab simulation.

8 THE FAB SIMULATOR

With the state description language we built the semiconductor fab simulator. The simulator is being used in various DRAM and ASICs fabs, including both clean room wafer processing and chip packaging fabs. Presently the simulation is deterministic, involving no random numbers. The simulator's major advantages have been found to be quick model specification and extremely small size of the simulator's source code. Kamimura *et al.* (1992) describes the fab model in some detail, which explains why the program can be made small.

The quick modeling is due to the human readable nature of the state description language. The existence of the state description, a document common to both factory engineers and model builders helps make clear who is responsible in case the model behaves in an unexpected way. Interpretation of event is discussed referring to two state descriptions, before and after the event. The document eliminated unreasonable requests from both sides, such as asking for unavailable input or impossible output data.

The code size is as in Table 4. That the program can be made small supports that the set of elementary operations has been chosen adequately. Smaller code means less bugs. The portion depending on the fab model is only the "event execution" portion; the rest is common to any discrete event simulation.

The model, which is the present state of the fab, is downloaded from the production control computer's master file. The model size listed in the Table is about three times the number of lots (1,300) and machines (200) involved. The downloaded data are first expanded into the initial state, by for instance generating all the lots that will eventually arrive during the simulation period. The state description grows considerably as simulation proceeds, since the event list and result records grow: both of them make part of the state description. Taking traces of each lot and machine increases the number of lines considerably. Adding one line to the state description uses roughly an additional 100 bytes. Thus a one month run requires about 30 megabytes main memory; execution time is 5 to 10 minutes on a SUN4 workstation.

Table 5: Directions in Simulation

Conventional approach	Simulation as language processing
Graphic input with icons and state transition diagrams such as Petri nets.	**Explicit state description** in a common language for people and machines.
Visual simulation such as animation.	**State transition** as a sequence of rewriting the state description.
Persuasion is the main purpose of simulation.	**Efficiency** in description and execution is of concern.
Small models are the norm.	**Large models** are of interest.
Large software is inevitable.	**Small software** suffices.

9 CONCLUSION

The time and space efficiency measures in the previous section suggest that the simulator is ready for larger fab models of the next generation which may require waferwise trace prediction. From the amortized time bound for a splay tree and the state description of the fab, the execution time is $O(logN)$ where N is the sum of the number of lots and the number of machines.

Table 5 summarizes the difference between the present paper's and the conventional approaches. The language provides a common platform for communication among people and computers involved.

REFERENCES

Brown, R. 1988. Calendar queues: a fast $O(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM* 31:1220–1227.

Burman, D. Y. *et al.* 1986. Performance analysis techniques for IC manufacturing lines. *AT&T Technical Journal* 6,4:46–57.

Chen, H. *et al.* 1988. Empirical evaluation of a queueing network model for semiconductor wafer fabrication. *Operations Research* 36:202–215.

Dayhoff, J. E., and R. W. Atherton. 1987. A model for wafer fabrication dynamics in integrated circuit manufacturing. *IEEE Transactions on Systems, Man, and Cybernetics* SMC-17:91–100.

Inoue, G., and K. Yoneda. 1989. VLSI production analysis using queueing network model. *Proceedings of the 176th Meeting, the Electrochemical Society.*

Jakobson, R. 1959. On linguistic aspects of translation. In R. A. Brower (ed.) *On Translation*, 233–239. Harvard University Press.

Jones, D. W. 1986. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM* 29:300–311.

Kamimura, S., *et al.* 1992. Realtime simulation for semiconductor fab operation. *Proceedings of the Pacific Conference on Manufacturing.* To appear.

Miller, D. J. 1990. Simulation of a semiconductor manufacturing line. *Communications of the ACM* 33:98–108.

Sleator, D. D., and R. E. Tarjan. 1983. Self-adjusting binary trees. *Proceedings of the 15th Annual ACM Symposium on Theory of Computing* 235–245.

Sleator, D. D., and R. E. Tarjan. 1985. Self-adjusting binary search trees. *Journal of the ACM* 32:652–686.

Tarjan, R. E. 1985. Amortized computational complexity. *SIAM Journal on Algorithms and Discrete Mathematics* 6:306–318.

Yoneda, K., *et al.* 1992. Job shop configuration with queueing networks and simulated annealing. *Proceedings of IEEE International Conference on Systems Engineering.* To appear.

AUTHOR BIOGRAPHIES

MUTSUMI FUJIHARA's research interests include natural and artificial language processing, design and analysis of algorithms, discrete event simulation, and computer utilization in humanity studies. Phone +81-44-548-5636, fax +81-44-533-3593, e-mail fujiha@ssel.toshiba.co.jp .

KIYOSHI YONEDA is Secretary of Production Planning and Scheduling SIG jointly sponsored by the Operations Research Society of Japan and Japan Industrial Management Association. Same phone and fax numbers; e-mail yoneda@ssel.toshiba.co.jp .