*Proceedings of the 1998 Winter Simulation Conference*
*D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.*

# IMPLEMENTATIONS OF DISPATCH RULES IN PARALLEL MANUFACTURING SIMULATION

Chu-Cheow Lim
Yoke-Hean Low
Boon-Ping Gan
Sanjay Jain

Gintic Institute of Manufacturing Technology
71 Nanyang Drive, Singapore 638075, SINGAPORE

## ABSTRACT

Most features in commercial simulation packages are often omitted in parallel simulation benchmarks, because they neither affect the overall correctness of the simulation protocol nor the benchmark's performance. In our work on parallel simulation of a wafer fabrication plant, we however find several features which complicate the implementation of the simulation protocol and affect the program performance. One such feature is the dispatch rules which a machine set uses to decide the priority of the waiting wafer lots. In a sequential simulation, the dispatch rule can be implemented in a straight-forward fashion because the whole system state is at the same simulation time, and the rule simply reads the state variables (of any machine, resources, etc.) In a parallel simulation, the dispatch rule computation may be complicated by the fact that different portions of the simulated system can be at different simulation times. This paper describes our study of the implementation of dispatch rules in parallel simulation. We note that this is actually an instance of the little-studied problem of providing shared-state information in parallel simulation. We briefly survey previous related work. We then outline two different approaches for a dispatch rule to access the shared-state information and compare them in terms of their ease of implementation.

## 1 INTRODUCTION

The background of this work is from an ongoing collaborative project between the Gintic Institute of Manufacturing Technology and the School of Applied Science in Nanyang Technological University, Singapore. The objective of our project is to study how parallel and distributed simulation (PADS) techniques can be applied in a virtual factory simulation (Jain 1995). The simulations will be plant-wide, and include the modeling of manufacturing and business processes, and communications network. Such a simulation environment will allow one to model and analyze the effects of different system configurations and control policies on actual system performance. The initial focus is the electronics industry, because it is a major contributor to the manufacturing sector in Singapore. We therefore begin our study with a parallel discrete event simulation of a wafer fabrication plant (without considering the business process and communications aspect).

In order to have a convincing demonstration of applying PADS for virtual factory, one of the main project objectives is that the parallel simulation tool be able to support features which are commonly found and used in commercial (sequential) packages. On the other hand, because this is R&D work, we cannot afford to build a prototype as elaborate as commercial tools e.g. ManSim (TYECIN 1996). We therefore include only features which are likely to have an impact on the implementation and performance of the parallel simulation protocol.

While studying a wafer fabrication simulation using the Sematech datasets (Sematech 1997), one feature which complicates the implementation of parallel simulation, is the *dispatch rules*. A dispatch rule is a rule which a machine set uses to decide the priority of the waiting wafer lots. For example, if a dispatching rule is first-come-first-served, then an earlier wafer lot has a higher priority. Another example is to consider the amount of backlog in each lot's next destination machine set. For each lot, we find the queue length in the lot's next destination machine set. (Each machine set has a common queue.) The shorter the queue length, the higher is the priority of the corresponding lot. (For brevity, we will refer to this dispatch rule as the QLNM rule - *q*ueue-*l*ength in *n*ext *m*achine set.)

In a sequential simulation, either of the two dispatch rules described above can be implemented in a straight-forward fashion because the whole system state is at the same simulation time. A rule simply reads the state variables needed (e.g. queue length at different machine sets.) In a parallel simulation, the dispatch rule computation may be complicated by the fact that different portions of the simulated system can be at different simulation times. For example, while computing the QLNM dispatch rule at machine set $M_i$, we need to access information from other machine sets which may be at different simulation times from $M_i$.

This paper describes our study of the implementation of dispatch rules in parallel simulation. The basic issue is to find a way to handle the shared-state information in parallel simulation. In Section 2, we give the motivation of supporting shared-state in parallel simulation and survey several general implementation approaches. Section 3 describes two approaches we studied, for a dispatch rule (e.g. QLNM rule) to access the shared-state information. Both use message-passing to transmit events to read or write shared-state information. We also briefly compare the two approaches, in terms of the ease of implementation. We conclude this paper and outline the future directions of our project in Section 4.

## 2 SHARED STATE IN PARALLEL SIMULATION

The problem of implementing a QLNM dispatch rule is the sharing of state information among different machine sets. A common solution is to group into a single logical process (LP), a machine set $M_i$ together with other machine sets whose state $M_i$ accesses. This ensures that all machine sets in the same LP are progressing at the same simulation time. If a machine set $M_i$, while handling an event, needs to read or write the state of another machine set $M_j$ in the same LP, the read/write action can proceed correctly, because the machine sets share the same LP simulation time.

Such an approach is useful if, after grouping resources or machine sets which access shared state, the model can still yield enough parallelism (in terms of the number of LP's). But the grouping of resources/machine sets is a transitive operation and we may end up with a small number of LP's. For example, while we group machine sets $M_i$ and $M_j$ into the same LP because of one shared variable $x$, $M_i$ and $M_k$ may also have to be grouped because of a different shared variable $y$. We end up with $M_k$ and $M_j$ in the same LP even though they do not share any state.

We illustrate this problem using the Sematech dataset (Sematech 1997) for wafer fabrication. The datasets specify the machine set and (human) operator set required to perform a processing step. Each machine set and operator set can be used in more than one step. There are cases when an operator set works with different machine sets, and a machine set needs different operator sets when performing different processing step. In the simulation model, a machine (from a machine set) has to acquire an operator (from an operator set) to process a wafer lot. The straight-forward way to implement this is to treat the operator set as a shared counter (variable). We check whether an operator is available by reading the counter, and decrement the counter to signify the acquisition of an operator. Each shared counter is read and updated by different machine sets, and a machine set may read and update more than one counter (depending on which processing step is to be performed by the machine.)

Table 1 shows the number of LP's (conflict sets) obtained from the Sematech datasets when we group into the same LP, machine sets which read/write a shared counter, and compute this in a transitive fashion (Turner et al. 1998). In particular, sets 5 and 6 result in relatively few LP's. Furthermore, we note that these numbers of LP's are only when we consider the use of operator sets in the simulation model. To build a realistic model, there will be other kinds of shared information, which will further reduce the number of LP's (and hence level of parallelism.) One such example is the QLNM dispatching rule.

Table 1: Statistics from Sematech Datasets

| Data set | No. of mach. sets | No. of operator sets | No. of conflict sets |
|---|---|---|---|
| 1 | 68 | 32 | 23 |
| 2 | 87 | 97 | 87 |
| 3 | 73 | 0 | 73 |
| 4 | 31 | 0 | 31 |
| 5 | 83 | 4 | 4 |
| 6 | 93 | 7 | 7 |

It is therefore useful to study alternative ways of sharing state among different LP's. Another approach is to provide a space-time abstraction for the simulation application (Ghosh and Fujimoto 1991, Mehl and Hammes 1993). Shared variables are read and written at virtual time instants. A read or write operation is considered as an event. The sequence of actions to handle each type of event depends on whether the shared variables are used together with a conservative or an optimistic parallel simulation protocol (Fujimoto 1990). For example, Mehl and Hammes (1993) extends the ideas in distributed shared memory (DSM) in different ways, to provide shared memory that is kept consistent with virtual time.

In an optimistic protocol, an LP will save its state at different virtual times, and execute events assuming they will be in time-order. If events arrive out of time-order, an LP will *rollback* back to an earlier state, and undo all its actions, including 'un-sending' event-messages it sent to other LP's. The central idea to implement a shared variable within an optimistic protocol, is to have a *multi-version* list for each shared variable. The list contains a list of triples *(sender id, new value, timestamp)* which indicate when the variable has been updated. A reader LP sends a read event to the owner of a variable and suspends its event handling. Assuming that the read event is at time $t'$, the owner finds the value v from the multi-version list, such that $(id, v, t)$ is a triple and $t$ is the largest value in the list, which is smaller than $t'$. The owner keeps track of the LP's which have read each variable. In case a write event arrives $(id'', v'', t'')$ such that $t < t'' < t'$, then the reader has received an incorrect value and has to be notified of the new value $v''$, so that it can perform any necessary rollback.

In a conservative protocol, an LP will only execute events if it is safe to do so (i.e. if it is certain that there will not be any violation time-order causality constraint.) Mehl and Hammes (1993) suggested two general approaches to implement shared variables for a conservative protocol, without any rollback. In the first approach, an LP maintains a multi-version list for each variable that it owns. When an LP receives an write event $(id, v, t)$, it simply inserts the triple into the multi-version list. The write event $(id, v, t)$ is ordered with other simulation events, and is therefore correctly handled by the LP when the LP reaches the simulation time $t$. For a read event at time $t'$, the owner LP waits until it receives guarantees (e.g. via additional null messages) that there will not be any write events with timestamp smaller than $t'$. It then retrieves the correct value for the variable from its multi-version list. Mehl and Hammes (1993) however did not note the following: that if the read event is handled according to time-order with other events, using a conservative protocol, then when the LP handles the read event at timestamp $t'$, the protocol will guarantee that there cannot be any write event with timestamp smaller than $t'$. It is therefore unnecessary to maintain a multi-version list if all read and write events are handled with respect to their timestamp order.

One disadvantage with the first approach is that the reader LP has to suspend its event handling until a result is received from the LP which owns the variable. A second approach suggested by Mehl and Hammes (1993) is to cache a copy of the variable at each reader. The cached copy has a time-guarantee associated with it, which ensures that the value is valid up to the guaranteed time. The reader LP sends a read event, if the copy is unavailable or invalid.

We have earlier described a way of providing shared variables (operator set counters) by coalescing multiple LP's (machine sets) into a single LP. We can envision using the algorithms in Mehl and Hammes (1993) as an alternative approach to implement counters for operator sets. Similarly, we may use these algorithms for the QLNM dispatching rule. The queue length of a machine set (i.e. number of wafer lots waiting to be scheduled) is represented by a shared variable, and the reader LP's are the upstream machine sets which use each wafer lot's next destination machine set's queue length, to determine the lot's priority.

It is however possible to implement the QLNM dispatching rule without using multi-version lists because the accesses to the queue length shared variable, have a different characteristic from that of the operator set counter. While the operator set counter is read and written by multiple LP's, the queue length variable is updated by the owner LP and read by other LP's. We describe our implementations for the QLNM rule in the next section, which can be generalized to any write-local, read-by-remote shared variable.

## 3 WRITE-LOCAL, READ-BY-REMOTE SHARED VARIABLES

The next two subsections describe approaches to implement a shared variable, used in a conservative protocol, which is written only by its owner LP, and read by other LP's. Since the variable is written only by its owner LP, there are no write events, and all writes occur at the correct simulation time. We therefore only need to handle read events.

### 3.1 Request-Reply

One way to read a remote variable is to use a request-reply protocol; this is similar to the request-reply in typical message-passing, except that the request (i.e. read event) is handled at a specific virtual time instant. The requester (reader) LP sends out a read event at timestamp $t'$ and its simulation time is frozen at $t'' = t'$ while waiting for the reply (Figure 1). In general, there is no need for the owner LP to implement a multi-version list for the shared variable if read events are ordered with other simulation events, and handled in time-order. When the read event is executed, the shared variable will contain the correct value at that virtual time instant because it is not updated by other remote LP.

We note that while the reader LP is frozen in simulation time, it must not be suspended in terms of real execution. In particular, it must continue to receive read events (timestamped at $t'$) from other LP's. If not, deadlock may

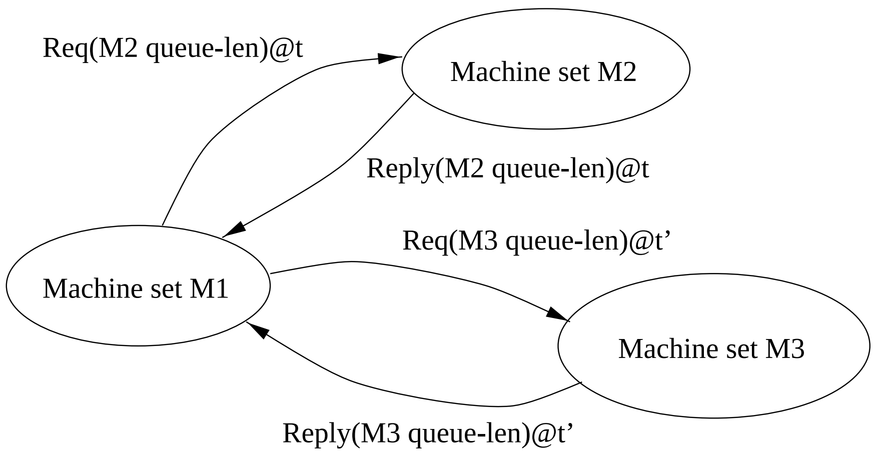

Figure 1: Reading a Remote Variable by Request-Reply Protocol

occur. For example, if $LP_i$ sends a read event to $LP_j$ at time $t'$ and vice-versa, both may be suspended waiting for the other to reply. In order to prevent suspension in real execution, the LP has to break out of the event-handling code just after the read event is sent out, and resume at the same location when the result is received. This complicates coding and we shall describe our implementation later.

We have assumed that read events are sent out at the simulation time of the reader LP. The reader LP may also try to prefetch a value by sending out a read event at timestamp $t'$ while its simulation time is at $t'' < t'$. (The situation when an LP sends out a read event at timestamp t' while its simulation time is at $t'' > t'$ is not useful because the event should have been sent out when the LP was at timestamp $t'$.) In this case, the reader LP can continue its event execution, because the reply will arrive at a later simulation time, and be handled then.

One differences between our approach and that in Mehl and Hammes (1993) is that we do not keep a multi-version list for each shared variable. This is because we process the read events in timestamp order together with other simulation events. By the guarantee of the conservative protocol, when an LP processes a read event at time $t'$, it will also have completed all local updates of the corresponding variable.

In order for a multi-version list to be useful when using request-reply to read shared variables, read events should be treated as special events, independent of other simulation events. So read events arriving at an LP constitute an independent stream from other time-ordered events. This may allow the owner LP to proceed ahead in simulation time, while the reader LPs lag behind, thus providing more parallelism. In the case for the QLNM dispatching rule, it depends on the characteristic of the conservative protocol, in particular whether the simulation time of a shared variable's owner LP can proceed ahead of the reader LPs' simulation time.

Consider a synchronous simulation protocol (e.g. Cai, Letertre and Turner (1997) and Lim, Low and Turner (1998)) which calculates a *safetime* for every LP at every super-step. The safetime for $LP_i$ $(ST_{LP_i})$ is a guarantee that events arriving at $LP_i$ in the next super-step will be at least at timestamp $\geq ST_{LP_i}$ . There is also no lookahead information. Suppose we have machine set $M'$ sending wafer lots to machine set $M''$. In such a protocol, the simulation time of $M''$ $(Time_{M''})$ cannot exceed the simulation time of $M'$ $(Time_{M'})$. When $M'$ handles a lot arrival event at $t'$ and sends a read event to $M''$ at $t'$, $M''$ may not be able to return a value to $M'$ immediately because $Time_{M''} \leq Time_{M'} = t'$. We note that even if the shared variable's value has a time-guarantee as lookahead, a multi-version list is still unnecessary, because only the latest value is read.

In an asynchronous protocol (e.g. Chandy and Misra (1979)) with lookahead and deadlock prevention, $M'$ may send a null message to $M''$ to advance $M''$'s simulation time ahead of $M'$'s. If $M'$ then sends a read event at $t'$, it is now useful for $M''$ to maintain a multi-version list because it may be reading an older value of the variable, now that $Time_{M''} > Time_{M'} = t'$. Note that the null message from $M'$ only ensures $M''$ that $M'$ will not send any non-read event before the guaranteed time. Read events can still arrive before the guaranteed time.

## 3.2 Cached-Copy

A second way is to cache the remote shared variable in the reader LP. While Mehl and Hammes (1993) examined a cache-on-demand approach, we tried an always-update-by-writer approach. There is now an additional link from the owner/writer LP to each reader LP (if none exists previously.) The cache-update events will travel along each link in timestamp order, with any other simulation events. At the beginning of the simulation, the writer sends an initial value to all the variable's readers. Because in the QLNM dispatching rule, the machine set queue length is updated only by the owner LP, there cannot be any simultaneous write events from different LP's. During execution, every time the queue length changes, an updated value is sent to all readers. Neither the reader nor the writer maintains any multi-version list.

When the reader needs the value of a shared variable, it reads from its local cache without being suspended. In Mehl and Hammes (1993), a time-guarantee is associated with each cached copy to ensure that it correctly reflects the current value in the original variable location. This is because the reader only receives cache-update events when its cache becomes invalid, and it has to send out requests for updates.

On the other hand, in our approach, there is no need to associate a time-guarantee for every cached copy, because the reader LP is notified of every update. Since the cache-update events arrive in timestamp order, with other simulation events, we are guaranteed that an LP at time $t'$ reading cached variable $v'$ is reading its latest value because any other cache-update events for $v'$ have to arrive at or after $t'$. (If not, they would have arrived before $t'$ and used to update $v'$ before the current attempt to read $v'$ at $t'$.)

The approach in Mehl and Hammes (1993) has the advantage that cache-update events are only sent when needed by the reader. But if the lookahead is small or even zero, and the shared variable is read frequently, then the overheads may be higher from more request messages. Our approach avoids the round-trip needed to get a cache update.

In the QLNM dispatching rule, the shared variable is written only by the owner LP. If a shared variable is written by multiple LP's, our scheme can be simply extended by having each writer update all readers whenever a writer updates the variable. This may potentially result in many cache-update messages, but since we are only interested in the single-writer shared variable, we are not affected by such a situation.

### 3.3 Comparison of Approaches

In this section, we briefly compare the request-reply and cached copy approaches in terms of ease of implementation. The main complication when coding the request-reply protocol is that the reader LP has to suspend its event execution after sending out a read event. But we cannot simply suspend the reader LP in real execution; otherwise a deadlock might result. Intuitively, the LP is frozen at simulation time $t'$ but continues to handle other events at $t'$ (if they do not interfere with the interrupted event.) The natural sequence of event-handling actions is now broken into two portions of code, reducing modularity (Figure 2). There are also additional execution overheads, because local variables used in the earlier statements may need to be saved, to be restored after receiving the reply.

One way to retain the continuity of the event-handling code is to make use of the language exception mechanism (Figure 3). After sending out a read event, an exception is *thrown*. The caller of the event-handling code must be ready to accept such exceptions. When the reply is received, the exception is *resumed*. The language must provide the mechanism to both throw and resume exceptions. Because C++ only has the *throw* mechanism, the programmer must still explicitly save and restore local variables for the LP to resume its event-handling. Ideally, the code using exception mechanism is as in Figure 3.

The cached-copy approach is simpler and does not require any code mangling. All we do is to introduce a new type of cache-update events, and implement the local caches for each LP. If no message link existed previously from the owner LP to a reader LP, a new link is added; otherwise, the cache-update events share the same link as other simulation messages. The only change to the sequential sequence of event-handling actions is to look up the local cache for the value of a shared variable if it is remote.

## 4 CONCLUSIONS

In this paper, we try to illustrate the relevance of supporting shared state in a parallel manufacturing simulation, especially if we want to implement in parallel, similar features found in commercial sequential simulation packages. Two examples of shared state in a parallel wafer fabrication model are the operator set counter and machine set queue length. The former is used when a processing step at a machine needs to acquire an operator before it can proceed. The latter is used to decide priority of wafer lots in the QLNM dispatching rule. While the operator set counter is read and written by multiple LP's, the machine set queue length is written only locally by the owner LP.

We find that in the implementation of shared variables, if the read and write events are treated just like any other simulation events and handled in timestamp order, there is no need to implement mechanisms such as multi-version list (as described in Mehl and Hammes, 1993). The latter is useful only if read/write events are to be treated in a separate time stream from other events.

Our simulation model is still relatively simple because it only looks at wafer manufacturing. We are currently integrating the various aspects of a virtual factory model,

```
$statement_0$, ... $statement_i$          ⟹     $statement_0$, ... $statement_i$
read variable $v$                               send read-event for $v$
$statement_{i+1}$ ...
                                                [On receiving reply with $v$'s value]
                                                $statement_{i+1}$ ...
```

Figure 2: Request-reply approach to read shared variable using manual code partition

```
$statement_0$, ... $statement_i$                [On receiving reply with $v$'s value]
read variable $v$                               resume exception $E$
throw exception $E$
$statement_{i+1}$ ...
```

Figure 3: Request-reply approach to read shared variable, using exception mechanism

including business processes, manufacturing and communications network support, and evaluating the different protocols to find out which will give better performance for our application. We plan to consider the use of lookahead information in our parallel simulation. This provides time guarantees to advance each LP's simulation time. We can also look into using this lookahead information to provide time guarantees in the shared variable implementation. This may help to reduce the number of read and cache-update events, and/or increasing the amount of parallelism.

## ACKNOWLEDGMENTS

This research is supported by National Science and Technology Board, Singapore, under project: *Parallel And Distributed Simulation of Virtual Factory Implementation*. The project is currently located at the Center for Advanced Information Systems at the Nanyang Technological University, Singapore. The authors would also like to acknowledge the other members of the project: Wentong Cai, Wen Jing Hsu and Shell Ying Huang.

## REFERENCES

Cai, W. T., E. Letertre and S. J. Turner. 1997. Dag consistent parallel simulation: a predictable and robust conservative algorithm. In *Proceedings of 11th Workshop on Parallel and Distributed Simulation (PADS'97)*, 178-181. Lockenhaus Austria, June 1997.

Chandy, K. M., and J. Misra. 1979. Distributed simulation: a case study in design and verification of distributed programs. *IEEE Trans. On Software Engineering* SE-5 (5): 440 – 452, Sept. 1979.

Fujimoto, R. M. 1990. Parallel discrete event simulation. *Communications of the ACM* 33 (10): 31 – 53, Oct. 1990.

Ghosh, K. and R. M. Fujimoto. 1991. Parallel discrete-event simulation using space-time memory. Technical report GIT-ICS-94/27. January 10, 1991. College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA.

Jain, S. 1995. Virtual factory framework: a key enabler for agile manufacturing. In *Proceedings of 1995 INRIA/ IEEE Symposium on Emerging Technologies and Factory Automation*. Paris, Oct. 1995, Vol. 1, 247–258. IEEE Computer Society Press, Los Alamitos, CA.

Lim, C. C., Y. H. Low and S. J. Turner. 1998. Relaxing safetime computation of a conservative simulation algorithm. In *Proceedings of 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. Las Vegas, Nevada, USA. July 13 – 16, 1998.

Mehl, H. and S. Hammes. 1993. Shared variables in distributed simulation. In *Proceedings of 7th Workshop on Parallel and Distributed Simulation (PADS'93)*, 68 – 75.

Sematech. 1997. Modeling data standards, version 1.0. Technical report. Sematech, Inc., Austin, TX78741, 1997.

Turner, S. J., C. C. Lim, Y. H. Low, W. T. Cai, W. J. Hsu and S. Y. Huang. 1998. A methodology for automating the parallelization of manufacturing simulations. In *Proceedings of 12th Workshop on Parallel and Distributed Simulation (PADS'98)*. Banff, Alberta, Canada, May 26 – 29, 1998.

TYECIN Systems Inc. 1996. ManSim/X User's Manual Release 3.8. May 13, 1996. TYECIN Systems, Inc., Four Main Street, Los Altos, CA 94022, USA.

## AUTHOR BIOGRAPHIES

**CHU-CHEOW LIM** is a Research Fellow with the Manufacturing Systems Modeling & Simulation group in the Gintic Institute of Manufacturing Technology, Singapore. From 1994 to 1997, he was with the Defence Science Organization, Singapore, as a Project Engineer. He received a B.S. in Mathematical and Computational Sciences and M.S. in Computer Science from Stanford University, California (1987 and 1988 respectively), and Ph.D. in Computer Science from University of California at Berkeley (1993). His research interests include parallel computing, simulations, compilers and e-commerce.

**YOKE-HEAN LOW** received the B.A.Sc in Computer Engineering from Nanyang Technological University, Singapore in 1997. He is currently a Research Associate with the Gintic Institute of Manufacturing Technology, Singapore. His research interests include parallel discrete event simulation and parallel processing. He is a member of the IEEE.

**BOON-PING GAN** received his Bachelor of Applied Science degree (major in Computer Engineering), and Master of Applied Science degree from Nanyang Technological University of Singapore in 1995 and 1998 respectively. Currently, he is working on the application of parallel discrete event simulation strategy on virtual factory implementation. His research interests are parallel programs scheduling, parallel discrete event simulation, and genetic algorithms.

**SANJAY JAIN** is a Senior Research Fellow with the Manufacturing Systems Modeling & Simulation group at Gintic Institute of Manufacturing Technology, Singapore. His research interests are in the area of using modeling and analysis techniques in development and operation of manufacturing systems, and in improving performance of simulation systems through parallel and distributed execution. Prior to joining Gintic, he worked for several years as a Senior Project Engineer with General Motors North American Operations Technical Center in Warren, MI, USA. He has a Bachelors of Engineering from University of Roorkee, India, a Post Graduate Diploma from National Institute for Training in Industrial Engineering, Bombay, India, and a PhD in Engineering Science from Rensselaer Polytechnic Institute, Troy, New York. He is a member of Institute of Industrial Engineers and the editorial board of International Journal of Industrial Engineering.