A modular, scalable approach to modeling and analysis of semiconductor manufacturing supply chains

Gary Godding and Karl Kempf

Intel Corporation, Chandler, Arizona, USA

Introduction

A number of powerful influences drive the evolution of manufacturing supply chains. One is the demand generated by the end customer who increasingly desires a product customized for a specific need, priced as though it was produced in mass, and delivered overnight to a residential address. This is compounded by the globalization of business that means the potential customer base spans multiple cultures, geographies, and economic systems.

Another influence is the double effect of the technology treadmill. On the one hand, expanding technology supports the introduction of new products through innovative materials and packaging as well as production and distribution methods. On the other, advancing technology provides faster, better support for management through all facets of electronic commerce from data availability to integrated decision-making.

A third influence is the ever-increasing level and pace of competition. A broader more demanding customer base stimulates the profit motive to satisfy the need. A stronger, more versatile technology base raises the confidence that the need can be satisfied. Companies and supply chains compete for the perceived profit using every opportunity to gain an advantage.

In this evolutionary cycle it is difficult to separate cause and effect. Does demand drive technology or technology drive demand? Does competition foster demand, or vice-versa? Are technology and competition related? The answers are, of course, "Yes," and it is clear that, to be competitive, the managers of any supply chain must deal with all of these factors on a daily basis. It is equally clear that a key constraint on the evolution of supply chains is our collective lack of a scientific basis upon which to successfully design and operate them.

Establishment of such a scientific basis depends on overcoming a number of inherent difficulties. Supply chains are dynamic systems with many sources of non-linearity and stochasticity. At every point along a multi-echelon system there is variability in both supply and demand with demand volatility usually outpacing the responsiveness of the supply. Supply chains involve a wide variety of tradeoffs, and balances must be achieved to remain profitable including (at least) tactical versus strategic and local versus global. In making supply chain management decisions, the appropriateness of the metrics chosen as well as the availability, freshness, and accuracy of the data required are critically important. Since supply chains change size and markets over time, the scientific basis must easily scale and adapt to these changes. Ultimately, the basis for the design and operation of supply chains must be reduced to practice and then implemented.

Supply chain modeling and simulation

Given the expense involved and the profit at risk, experimenting on actual supply chains is not practical. The development of the science requires practical enterprise level modeling, simulation, and analysis techniques and tools. This in turn requires the representation of a number of quite different but interrelated flows including physical, financial, decision, and data. The physical flow represents the goods being produced, stored, and shipped while the financial flow concerns the payments for the materials and services required and the products supplied. The data flow represents data about the past, present, and the forecast future state of the physical and financial flows. The decision flow uses the data available about the physical flow, advice from the financial system, and its decision policies to provide direction for the physical system. The overall goal is to maximize the four customer service factors - the right product, in the right quantity, in the right place, at the right time - while minimizing the four major costs - materials, production, storage, transport.

An obvious difficulty is that a single tool to efficiently model and simulate physical, decision, financial, and data flows is hard to imagine. We have developed a software architecture to interconnect a variety of tools to address this problem as shown in Figure 1. To model the physical flow, the obvious choice is a discrete event simulator (DES) as developed in the Industrial Engineering community with which factories, warehouses, and transportation links along with their capacities and throughput times can be modeled. Decision policies, both heuristic and mathematical, are conveniently modeled in the current generation of expert systems (ES) as developed in the Artificial Intelligence community. The overwhelming majority of financial analysis is done with spreadsheets (SS) and so we include one of them. Finally, data flow is modeled in the interconnect mechanism between the DES, ES, and SS. A database is included to complete the data flow modeling. Although a variety of software modules can be connected through the method we describe in this paper, with multiple modules of like or similar functionality, our current implementation target includes just one of each type and utilizes commercial products.

See Figure 1

Using this architecture a wide variety of experiments can be imagined for exploring the behavior of demand networks. From the physical perspective, the number and connectivity of entities can be varied as well as the capacity and throughput time of each entity. With the financial module, calculations can be done at the end of a simulation for evaluating the overall run and during a simulation to support decision-making. Using the decision module, various mathematical and heuristic control policies can be investigated to quantify their impact. The advantage of one central controller versus many local controllers can be quantified by using one or multiple decision and financial modules. The utility of performing financial calculations and decision-making more or less frequently can be explored. The data that can flow between modules is the basis for their interaction, as is the associated database. Experiments can be imagined that test the advantage of having precise data immediately versus inaccurate stale data.

The general type of supply chain that we are interested in modeling is shown in Figure 2.

See Figure 2

Our focus to this point has been the communication between the DES modeling the physical flow and the ES modeling the decision flow for supply chains like this. This is the minimal

communication required to make our system operational. The data that flows from the DES to the ES is basically inventory status. Many decisions are based on the current inventory holding at each position in the supply chain. Included in this data are details of the timing to be expected. For example, on a transport link that has an average transit time of three days, it is useful to know what inventory is expected to be delivered in one day, two days, and three days. The decision engine occasionally needs an update of its basic model of the supply chain. This includes the forecast capacity and throughput time for each entity in the supply chain based on the historical data of each entity.

The data that flows from the ES to the DES is basically control commands. For factories, the important control input is how many of which product to release into production over what time period as well as how to prioritize the work in progress already making its way through the factory. Transportation links need commands about how much of which product to move to which destination at what time by what transport method. The amount of each product to release for which customer is the basic commend to warehouses as well as what is on its way to them.

Clearly, in the long term, storing this data in a database will make both status data a control data available to any module that needs them. Not so clear to us at this point is the communication between the ES decision module and the SS finance module. One example of a message from the ES might be that the physical system will have extra capacity of some amount over some duration. The SS will respond, after performing the appropriate calculations, that the extra capacity should be left idle or should be put to work to pre-build some volume of a product(s) to be held at some inventory position. At this point we have not designed the communication vocabulary between these modules.

Implementation issues

Once we have defined the types of messages required, we need to think about how to synchronize the communication between the different applications. In our model we are simulating the physical aspects of a supply network [Knutson et. Al, 2001] and the decision policies to run the network [Armbruster et. al., 2001]. In the real world this would correspond to the coordination between the planning, manufacturing, and shipping organizations.

Typically, the planning organization creates manufacturing plans at some regular interval by looking at market conditions, current inventories, and available manufacturing capacity. Manufacturing and shipping organizations generally execute to the goals provided by planning, not changing their goals until getting the next sequential update from planning. However, there may be exception conditions that occur where planning would want to adjust manufacturing goals immediately. Examples of this could be unexpected market changes, or unexpected yield problems in manufacturing.

Another factor to consider is how often planning gets data updates and how often updates are sent to manufacturing. We could easily envision scenarios where planning is very reactive by looking at current system state very frequently and attempt to make corrections immediately. We also could create scenarios where planning looks at data very infrequently letting manufacturing run on its own for long periods of time.

In our simulation tool we want to be able to model all the scenarios discussed above. To do this, we need a flexible synchronization scheme that not only allows the decision policy to update the plan at a regular interval, but also enable normal operations to be interrupted to change direction immediately.

In our initial implementation, we are able to see the supply network data at a regular interval and make adjustments as needed. The time interval at which this coordination takes place is configurable. The ability to interrupt on exception is not supported directly, however, it could be by increasing the synchronization frequency above the update frequency and allowing changes between updates only on exception. For example, if you are updating on a weekly basis, you could synchronize daily, but only send updates weekly. If an exception occurred during the week, you could change course within a day.

The synchronization protocol works as follows: The simulation runs for some interval of time and at the end of the interval it sends all the expected data to the expert system. The expert system analyzes the data, decides if updates are needed, and sends back the new results (perhaps nothing more than "continue executing the previous plan"). After the simulation gets the results, it runs for another time period. This cycle repeats until the simulation is complete. After the simulation completes the financial analysis tool can look at the data and tell us how much profit was made.

Implementation of the interconnect facilities

To implement the architecture discussed in the previous sections, we must be able to connect the different types of applications and enable them to communicate to each other using the data protocols described above. The types of applications that need to communicate with each other are a discrete event simulator, a spreadsheet program, a database, and (multiple) rule based decision systems. After doing some analysis, we found that some of these applications may have to run on different computer platforms. To facilitate an interactive communication bus that allowed the applications to communicate across different computers and operating systems, we chose to build our communication architecture on top of TCP sockets. TCP sockets will easily enable a distributed computing environment and are supported by all the mainstream operating systems.

The next problem to address was how to connect the application to the communications bus. All of the applications we had looked at had facilities to call dynamically linked libraries (DLL's). So we were able to create a DLL to interface with the communications bus. In the next sections we will describe how this was implemented, and the message protocols developed to support the experiments.

Communications bus

Figure 3 identifies the major components of the communication bus. There is a standalone socket server module, and an application interface.

See Figure 3

The socket server is a central application that routes messages to named mailboxes and the application interface is a piece of software that enables the application to connect to the socket server. The application interface has been implemented as a dynamic link library (DLL)

that all applications can use on a particular operating system. Applications communicate to each other via mailbox names. When an application starts, it can create multiple connections to the socket server, but each connection must have a unique mailbox name associated with it. The socket server uses these names to keep track of the connections and to route messages. Every message has the sender and sending mailbox name in the header. The communications interface module automatically adds this header to a message when it is sent.

Application Interface

The application interface can be broken into three different modules; the communications interface, the message parser, and the custom application software as shown in Figure 4.

See Figure 4

The custom application software is written in the applications native language and it utilizes the services of the communication interface and message parser to send, receive, decode, and encode messages.

The message parser is used to decode the message into a format that the application can understand or to encode messages back to the communication message protocol. In our implementation, the message parser is written in the native application language. However, the application software could call a message parser DLL. This is important because it provides an easy way to implement a different message protocol language where third party off-the-shelf DLL's are available. XML is an example of possible message protocol we could switch to in the future.

The communication interface is used to send, receive, and wait for messages. The communications interface consists of 3 layers. The communications API, the message buffer, and the TCP interface.

The communications API is a simple protocol consisting of five different functions. The functions are:

1. Open connection – Opens a connection to the socket server. The mailbox name is set and a socket is opened.
2. SendMessage – Used to send message to any other mailbox
3. WaitForMessage – Blocking call that will return when a message is received
4. GetNextMessage – Gets the next message from the message queue assigned to the callers mailbox
5. CloseConnection – Closes the connection and frees up the socket.

These five functions enable the application to send, receive, and sync with the other applications. The sync can be accomplished by using WaitForMessage. This function blocks until a message is in the mailbox.

The message buffer holds incoming messages until the application is ready to receive them. Messages are queued up in FIFO order. The buffer is required because other applications may be sending messages while the receiving application is busy doing something else.

The TCP socket interface connects to the operating system TCP socket services to manage the creation, deletion, and data passing through the socket. For incoming date, the TCP interface reads the data stream, separates each message, and puts it in the message buffer. For outgoing messages, it converts the message to a data stream and then sends it out on the socket.

In our implementation, the communication interface has been implemented as a DLL. A COM wrapper has also been created. Applications can connect using the native DLL API, or they can use the registered COM interface.

Message protocol between Discrete Event Simulator and Expert System

In our current implementation, we have developed an interactive message protocol between the expert system and DES. The two systems can pass three types of messages; initialization message, update value messages, and synchronization messages.

Initialization messages – At system startup, the DES will send a series of initialization messages to the expert system. There will be one initialization message for each parameter that the expert system listens to or updates a value for. The expert system uses these initialization messages to create a local model of the network being simulated.

Update messages – These messages are sent by the DES or expert system. These messages are used to notify the other system when a value has been updated. The expert system would use this message to update a release value, and the simulation would use it to update an inventory level.

Synchronization messages – The DES will send a synchronization message to notify the expert system when a new day is about to start. The expert system will send a message to tell when the simulation can continue.

The DES and expert system use the synchronization and initialization messages to enable a synchronization scheme. At initialization, the expert system calculates how many update messages will be received from the simulation on each time interval by counting the number of initialization messages for simulation update parameters. It uses this value during the simulation to verify it has received all updates for a particular time period. When the simulation is about to start a new time interval, the simulation will send the expert system a new time interval message and stops to wait for a proceed message to be sent back from the ES. When the expert system receives a new time interval message, it will check to see if it got all the updates from the previous time period. If it has, it will make any decisions needed, send down updates for new values, and then send a proceed message. If it has not, the ES will wait for all updates before making decisions and sending the proceed message. When the simulation gets a proceed message, it will read all the new values from the ES and then start the simulation for the next interval.

Future work

In the current implementation, interactive communication only happens between the DES and ES. The next step is to identify the types of messages to send between the finance model and decision system [Ellram et. al., 2001]. Once all of our modules are communicating efficiently

and reliably, we believe there are a large number of important experiments we will be able to execute. One extreme set of experiments will focus on an individual manufacturing company with one decision algorithm that integrates all of the data available and issues all of the decisions required for operation. At the other extreme there will be multiple decision algorithms. This could be separate decision makers for materials, fabrication, assembly, and logistics, or separate decision makers for all aspects for each product, or it could be a separate decision maker for each geography. We will also study the issue of data quality and its impact on timing and location of decisions. There are two extreme cases: data can be timely but wrong (or a large associated uncertainty) or they can be correct but stale. Typical cases of the former are demand forecasts, which are associated with a large uncertainty. The latter usually come from *a posteriori* analysis of events where, by the time the analysis is available, the overall conditions of the supply chain have changed in such a way as to make those data virtually worthless. We will use a forecast that will become more and more accurate as due dates approach. The forecast will be exact N days before delivery. We will study the impact of the length of the certainty window N, the influence of positioning upstream and downstream inventories and the impact of thrashing due to overreaction to forecast changes.

Acknowledgment

Partial support by NSF grants DMS 0075041 and DMI-0075655 is gratefully acknowledged.

References

Armbruster, D., R. Chidambaram, G. W. Godding, K. G. Kempf, and I. Katzorke, "Modeling and analysis of decision flows in complex supply networks", Proc. IV SIMPOI/POMS 2001.

Ellram, L., K. G. Kempf, T. Callarman, C. Arnold, and B. Liu, "Modeling and analysis of financial flows in complex supply networks", Proc. IV SIMPOI/POMS 2001.

Knutson, K., J. Fowler, K. G. Kempf, and B. Duarte, "Modeling and analysis of material flows in complex supply networks", Proc. IV SIMPOI/POMS 2001.

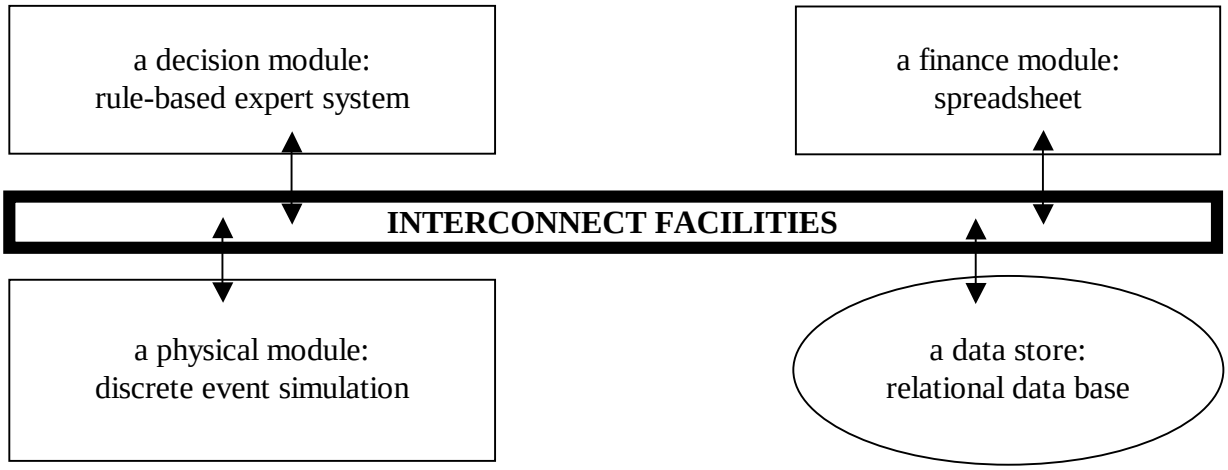

Figure 1: The basic components of the architecture

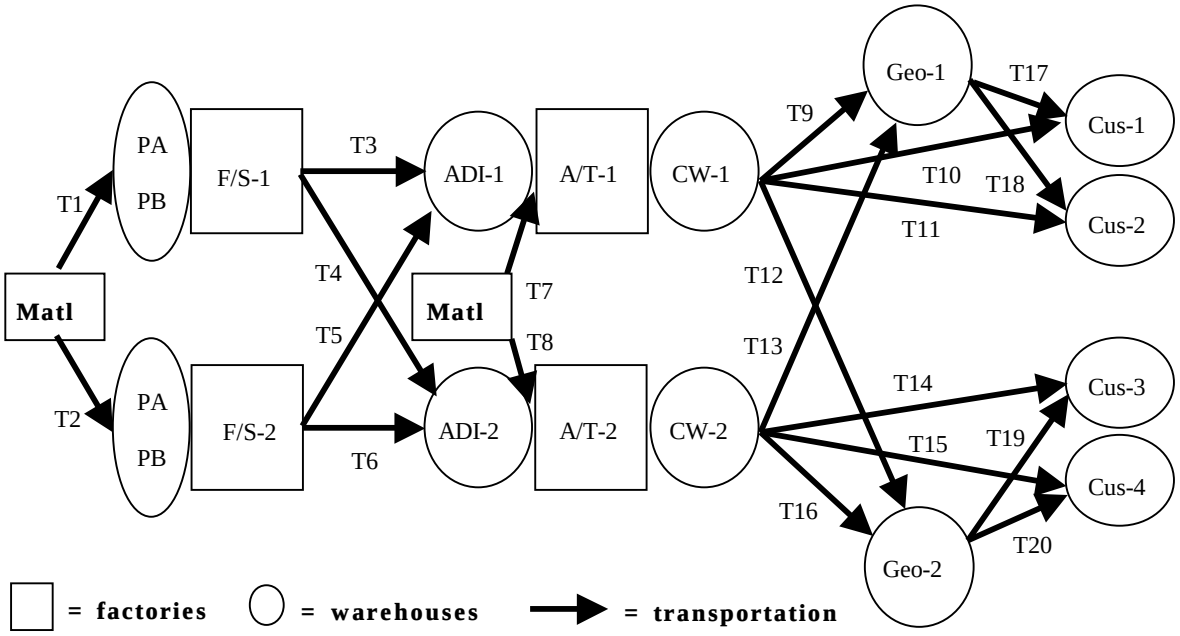

Figure 2: A very simple semiconductor manufacturing supply chain

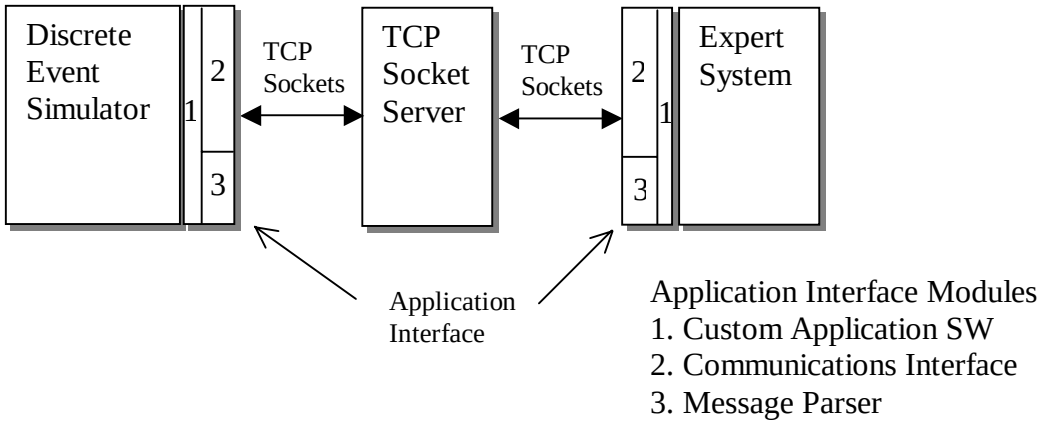

Figure 3: Communication Architecture

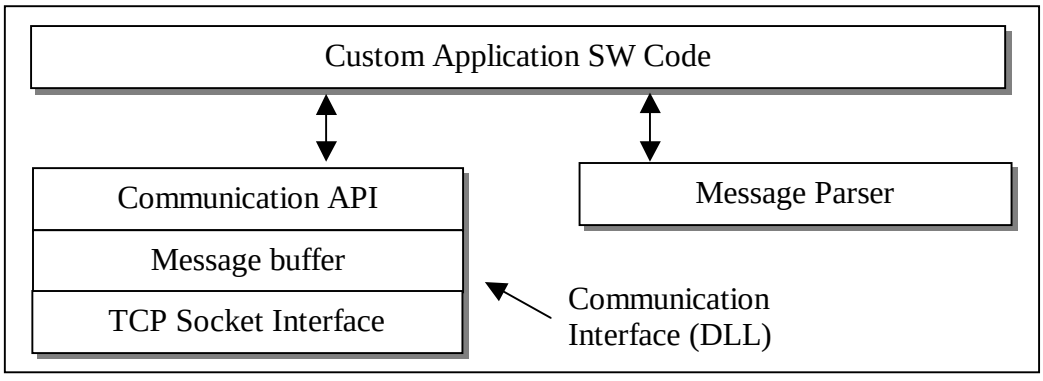

Figure 4: Application Interface Details