Proceedings of the 2007 Winter Simulation Conference
S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, eds.

AUTOMATIC GENERATION OF SIMULATION MODELS FOR SEMICONDUCTOR MANUFACTURING

Ralph Mueller
Christos Alexopoulos
Leon F. McGinnis

School of Industrial and Systems Engineering
765 Ferst Drive, N.W.
Georgia Institute of Technology
Atlanta, GA 30332, U.S.A.

ABSTRACT

This article gives an overview of a framework for automatically generating large-scale simulation models from a domain specific problem definition data schema, here semiconductor manufacturing. This simulation model uses an object-oriented Petri net data structure. The Petri net based simulation uses the same enabling rules as classical Petri nets, but has extensions of time and priorities. This approach minimizes the effort of model verification. Each object identified in the problem data specification is mapped to corresponding Petri net fragments. The Petri net simulation model is synthesized from verifiable subnets. This allows ensuring the liveness of the final Petri net simulation model. The applicability of this approach is demonstrated by generating a simulation model based on the Sematech data set.

1 INTRODUCTION

The steady decline in computing cost makes the use of simulation very cost efficient in terms of hardware requirements. However, commercial simulation software has not kept up with the hardware improvements. It can take very long to build and verify large models with standard commercial-of-the-shelf (COTS) software.

Efficient simulation model generation will allow the user to simplify and accelerate the process of producing correct and credible simulation models. Model credibility is attained when the end user accepts the simulation model as a correct model. Improving the verification process can improve the model credibility, especially when the end user can retrace the verification process.

The proposed framework promotes a "bottom up" approach to simulation modeling. Small verifiable subsets are synthesized in a particular way to create a large model, which will maintain certain properties, in particular, absence of deadlock, in the resulting simulation model. The existing literature on the use of Petri nets for semiconductor manufacturing focuses almost exclusively on the modeling of sub systems of the manufacturing system under study (e.g., Zhou 1998). One exception is Becker (2003), whose approach combines queuing and Petri nets to simulate an entire semiconductor manufacturing system (only the first Sematech data set).

The suitability of the simulation framework is demonstrated for semiconductor manufacturing simulation. Semiconductor wafer fabrication is considered one of the most complex and capital-intensive manufacturing processes (cf. Ramirez-Hernandez et al. 2005), and typically involves several hundred processing steps. This makes it an ideal candidate for the automatic generation of simulation models.

The remainder of this paper consists of three parts. Section 2 introduces an object-oriented Petri net simulation framework. Section 3 describes a problem data specification schema for a particular application domain. Section 3 presents mappings from this data structure to a Petri net. These mappings are used to create a simulation model that is based on the framework.

2 OBJECT-ORIENTED PETRI NET SIMULATION FRAMEWORK

The proposed framework is based on a Petri net (PN) data structure. PNs incorporate the concept of distributed system state with rules that define how state changes occur. PNs are successfully employed to support many stages of the development of complex systems: rapid prototyping, formal specification, verification of correctness, performance evaluation, and documentation (Zurawski and Zhou 1994).

A Petri net graph is a weighted bipartite graph *(P, T, A, w)*, where *P* is a finite set of places and *T* is a finite set of transitions. Weights are associated with each arc in the graph. If an arc is not labeled with a weight, its weight is assumed to be one. The set of input places to a transition *t*

is ${}^{\bullet}t = \{p \mid (p,t) \in A\}$, the set of output places of is $t^{\bullet} = \{p \mid (t,p) \in A\}$, similar the set of input transitions to a place p is ${}^{\bullet}p = \{t \mid (t,p) \in A\}$, and the set of output transition is $p^{\bullet} = \{t \mid (p,t) \in A\}$. A comprehensive introduction to Petri nets is given by Murata (1989).

A unique feature of the simulation framework is that all relevant state information is contained within the model. The future event list (FEL) can be reconstructed from the current state of the model. This means that structural changes to the model can be made during a simulation run and the FEL can be updated accordingly.

2.1 Representation of Time

Classic Petri nets do not have any notion of time. A great number of extensions of Petri nets to capture time have evolved (see Wang 1998). A problem with theses extensions is that the interpretation of firing delays is different for each version. The firing of a transition often represents time durations, but in classical Petri nets it represents the instantaneous transition of a discrete-event system to a new state.

Van der Aalst (1993) introduced a different approach with interval-timed colored Petri nets, which are based on colored Petri nets but use timestamps on tokens to capture time. This simulation framework uses a similar but simplified approach.

2.2 Relation to Colored Petri Nets

Our framework also uses tokens that have additional data assigned. Primarily these are times and priorities. The enabling rule for a transition, however, is exactly the same as for classical Petri nets. Tokens can have different attributes, but these do not influence the enabling condition for transitions. No colored tokens (in a traditional sense) are used, as the attributes are not involved in enabling rules of transitions.

2.3 Core Elements of the Framework

As with the classical definition of a Petri net, the core elements of the simulation framework are transitions, places, and tokens. All of these elements are implemented as objects. Arcs between places and transitions are not modeled explicitly, as places have references to transitions and vice versa, (each of these references represents an arc). With these simple elements, it is possible to model complex systems. Transitions represent events, whereas places and their marking represent states. The class diagram in Figure 1 shows how these elements relate to each other. We proceed with a short description of these elements.

A `Place` is an object that represents a place in the Petri net. Each `Place` has an `id`, which uniquely identifies it in the simulation model. The field `inTransitions` is a list of `Transition` objects that represent the input transitions to the place. The `Place` object can receive tokens from these transitions. This is equivalent to the set $\bullet p$ of a place p in a Petri net. The field `outTransitions` is a list of `Transition` objects, which represent the output transitions of the place. This is equivalent to the set $p\bullet$ of a place p in a Petri net. There are subclasses of `Place`, which can represent resources or operators.

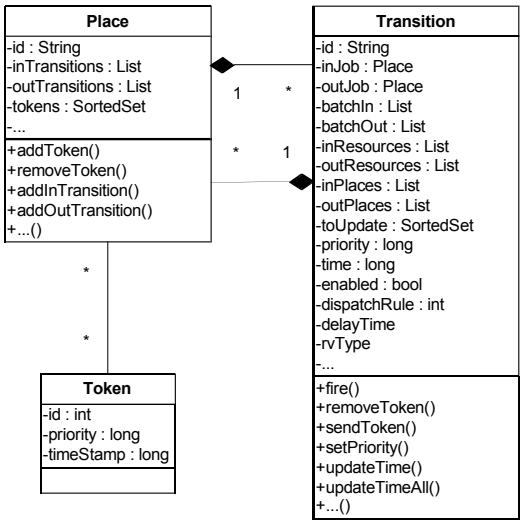

Figure 1: Core elements.

A `Token` is an object that represents a token in a classical Petri net. However, it has extensions for time and priority, which are represented by the fields `timeStamp` and `priority`. There is a subclass `JobToken`, which is used to represent lots that have to be processed.

The method `addToken()` is used during a simulation to add a `Token` object to the `tokens` set of the place. The method `removeToken()` is used to remove a token from the place. There are more fields and methods in the `Place` class, which are of auxiliary nature and are not shown in Figure 1.

A `Transition` is an object that corresponds to a transition in a classical PN. The fields `inJob`, `batchIn`, `inResources`, and `inPlaces` are used to hold places that represent the set of input places of the `Transition` object, similar to the set $\bullet t$ of a transition t in a Petri net. Different names are `used` to organize the simulation model better. The field `inJob` represent a place that holds `Token` objects representing jobs in the manufacturing system. The field `batchIn` represents a set of `Place` objects representing places with an arc weight w

greater than 1. The fields `outJob`, `batchOut`, `outResources`, `outPlaces` are used to hold places that represent the set of output places of the `Transition` object, as the set $t^\bullet$ of transition t. Their usage is analogous to the input places. The field `time` is used to indicate the firing time of the transition. The field `priority` is needed to resolve conflicts of transitions with the same firing time. The field `enabled` indicates if the transition is enabled, i.e., it is eligible to fire. The field `delayTime` stores the time an activity will take. The field `rvType` represent the name of the distribution that is used to model the delay time. Any distribution can be implemented, as long as the appropriate generator is available to the framework.

The method `fire()` will remove the appropriate `Token` objects from the input places and will place the appropriate tokens in the output places by invoking the methods `removeToken() and sendToken()`. This method can be called only when the transition is enabled. The method `setPriority()` is used to calculate the priority value of the Transition object based on the tokens in the input places. The method `updateTime()` will calculate the firing time of the transition. The method `updateTimeAll()` will calculate the firing times and check enabling conditions of all affected transitions after a transition has fired. The core elements above have a direct mapping to a classical Petri net *(P, T, A ,w)*. For each place $p \in P$ there is a corresponding `Place` object. Also for each transition $t \in T$, there is a `Transition` object. A simulation model is created by instantiating `Place` and `Transition` objects and adding appropriate references to each another. There are a number of sub classes, which are omitted here for conciseness.

2.4 Execution Mechanism

After the Petri net is generated, the simulation model can be executed. Simulated events correspond to firings of transitions. To simulate the Petri net, the simulator has to scan transitions and their input places to determine if they can fire. One can scan all transitions in the net after each transition firing, but this would be wasteful, since only some transitions are affected by the firing of a transition. For example, if transition t fires, only the transitions that have input places whose markings were changed by t are affected. It is not sufficient to check if a transition is enabled; its firing time also needs to be determined. Tokens carry timestamps, which determine when they will be eligible to be used by a transition. The firing time of a transition is determined by the maximum timestamp of the enabling token. If there is more than one token in a place and the respective arc weight is one, the token with the smallest timestamp is the enabling token for that place. The example in Figure 2 illustrates this.

T_1 and T_2 are the transitions that represent the beginning of processing involving resource R_1. The input places for T_1 and T_2 are $\{R_1, P_1\}$ and $\{R_1, P_2\}$, respectively. R_1 has one token with timestamp <5>, P_1 has two tokens with timestamps <11> and <25>, and P_2 has three tokens with timestamps <15>, <35> and <40>. Both transitions T_1 and T_2 are currently enabled. The enabling time for T_1 is 11 because this is the value of the smallest timestamp in P_1. The enabling time for T_2 is 15 since that is the value of the smallest timestamp in P_2. Note that P_1 and P_2 together represent the queue waiting for resource R_1.

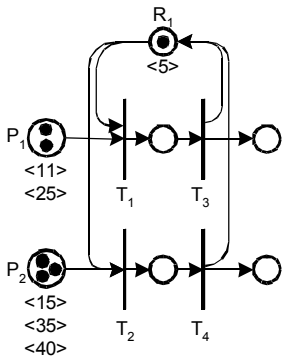

Figure 2: Timing example.

Job tokens also have priorities that are assigned to the enabled transitions and are needed to establish a firing order for transitions that are enabled at the same time. They also provide a mechanism to implement different dispatch rules (see also Section 4). The enabled transitions are added to the FEL, where they are ordered according to time and priority. This allows ordering of transitions that are enabled simultaneously with identical firing times. This is not directly possible with classical and colored Petri nets, as the firing of transitions is undetermined. The detailed description of the mechanisms can be found in Mueller (2007).

2.5 Transition Firing

When a transition t actually fires, the following steps are undertaken:

- Remove tokens form all input places $p \in {}^\bullet t$.
- Add tokens to all output places $p \in t^\bullet$.
- Update time of all affected transitions.

When a transition t fires, it removes the enabling tokens from each input place p. The removal of a token of type `JobToken` is different. As this type of token is representing jobs or lots that move through the system, they should not be discarded as they can carry attributes with

additional information. Instead, they are removed from the `inJob` place and are added to the place `outJob`.

The last phase of the transition firing is to add the appropriate number of tokens to the output places. First, the timestamp of the `JobToken` object is set to *t.time* + *t.delayTime*, which is the firing time of the transition plus the delay time for the transition. The delay time can be either deterministic or it can be a random variate generated from the distribution specified in the field `rvType` of transition *t*. The delay time can also be zero, depending on what the transition represents. The `JobToken` object is then added to the place that is specified in the field *t.outJob*. Furthermore, new `Token` objects are created with a timestamp of value *t.time* + *t.delayTime* for each output place of transition *t*. If the arc weight to the output place is greater than one, the appropriate number of tokens will be generated and added to the place.

After a transition fires, all affected transitions must be updated. As the state of the system is represented by the marking of all places, the change of state is represented by the change of tokens in these places. This also means that only transitions that have at least one of these places as an input place are affected. Note that transitions that only have one of these places as an output place cannot be affected. Places that received tokens can enable their output transitions, and places that have tokens removed might disable their output transitions.

3 SIMULATION DATA SPECIFICATION

Here the term Simulation Data Specification refers to the precise description of the data that will be used to generate the simulation model. There are very few data specifications for discrete-event simulation in the literature. An instance can be found in Lee and Yan (2005) and Lu, Qiao, and McLean (2003), where the NIST XML simulation interface specification is used, introduced first by McLean et al. (2002) and still under development.

The Sematech data set, available at `<http://www.eas.asu.edu/~masmlab>` represents a limited form of specification; however, it is in table format and cannot express explicitly the relationship between all the entities in the simulation model. Therefore an object-oriented model was developed.

This simulation data specification describes how the modeled elements of semiconductor manufacturing systems are represented. From this specification, the simulation model is generated automatically. This approach has several advantages because the data is represented in the application domain. Any changes in the simulation model are made in this domain. This avoids programming errors, as there is no simulation code to change. The following classes or object types are the main elements used to represent the fabmodel:

- Fabmodel
- Process Route
- Process Step
- Tool Set
- Operator Set

The `Fabmodel` class represents the root for all the other elements, i.e., it contains all the other objects for the simulation model.

The `ProcessRoute` class holds all the information of one specific process route. It contains all the constituent process steps of the route.

The `ProcessStep` class holds all data that refer to a single process step. These data are processing times, required resources, operation description, loading and unloading times, scrap and rework probabilities, and travel times. This basic class is used to model process steps that process wafer lots one at a time. It is the most common type of process step. It serves as a basis for two subclasses:

- Batch Process Step
- Process Step with Setup

The batch process step represents process sub-steps that can batch lots together and process them at the same time. Lots from different process routes can be batched together when the `batchID` fields are identical. After processing is finished, each lot will continue its own process route. Each batch process step has a minimum and a maximum capacity for the number of wafers that can be processed at the same time. The process step with setup is used to model steps that require a setup of the tool. There is a specific setup time and a group setup time. The specific setup time is needed for every lot that has to be processed, whereas the group setup time is only needed when the previous processed lot belonged to a different setup group.

The `ToolSet` class describes the machines that are used to process the wafers. It has fields for `id`, `description`, and `quantity`. It there is more than one tool, all tools are treated as equivalent. Further, there are fields that indicate if the tool has to be loaded and/or unloaded by an operator. If the tool is used in a setup process step, the setup states are listed in the field `setupStates`. Downtimes are also listed in the field `downtimes` with description, duration, and time between downtimes.

The `OperatorSet` class describes the operators that are needs to operate the tool sets in the waferfab. It is very similar to the `ToolSet` class.

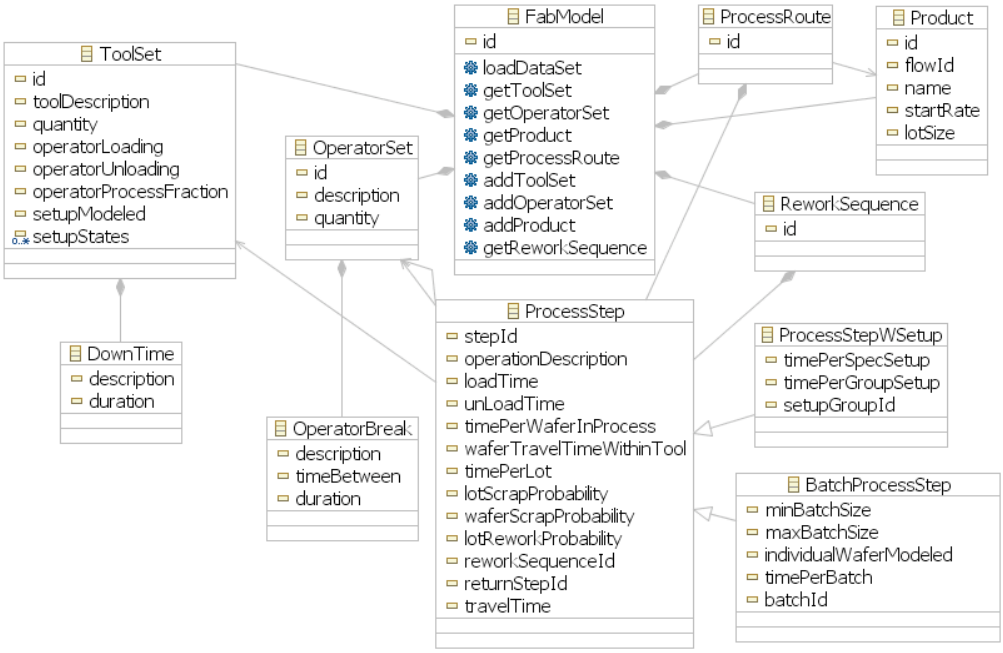

Figure 3: Class diagram of data specification.

Rework sequences have the same data format as process routes. A rework sequence also consists of process steps. If the rework probability is greater than zero at a particular process step, there is a chance that a wafer lot will have to follow the specified rework sequence after that step is completed. After the lot goes through the rework sequence, it returns to the original process route.

4 SIMULATION MODEL GENERATION

The typical way to create a simulation model is to use a COTS simulation package. These packages usually have graphical user interfaces and reduce the programming effort to dragging and dropping of modules into a screen and connecting these modules. A big advantage for the user of these packages is that it is not necessary to be familiar with the underlying simulation language. However, this process can become cumbersome for large models. Debugging can be difficult, as data is entered in many different dialogs. The graphical representation is also limited since not all details of the model can be presented. Another problem involves proprietary issues, as software vendors are reluctant to make source code available. Therefore, it is often not clear how specific modules behave on a detailed level.

A very different approach to building simulation models is automatic generation. According to Mathewson (1975), a simulation generator is a software tool that translates the logic of a model into the code of a simulation language, enabling a computer to mimic a modeler's behavior. One of the earliest examples for simulation model generator is presented in Mathewson (1975), which is based on entity cycle diagrams and in Mathewson (1985), an early PC implementation of the former. In Gong and McGinnis (1990), a simulation code generator for an automated guided vehicle system is presented. In Lee, Cho, and Jung (2000), a simulation model for shop floor control systems is generated automatically from graph-based process plans.

In this paper, a different approach is introduced: the simulation model is generated from an input file that adheres to the simulation data specification. The simulation model is represented as a Petri net data structure, which is built with the object-oriented framework that was introduced in the previous section. An instance of a simulation model is created by populating this data structure, but no compilation takes place.

4.1 Mapping of Fabmodel Elements to the Petri Net Simulation Model

This section discusses how each of the elements of the fabmodel is represented within the Petri net. This is the basis for the algorithms that generate the PN simulation model.

4.1.1 Tool Sets

Each tool set is represented by a place in the PN. The corresponding object type is the Resource type of the framework. Each place that represents a tool set will have tokens, representing the number of tools available. Some tool sets have different setup states. These tool sets are

represented by a set of places in the PN. For each setup state of the tool, there is a corresponding place. The number of tokens in one place will correspond to the number of tools available that are in the corresponding setup state.

4.1.2 Operator Sets

Operator sets are modeled in the same fashion as tool sets. Each operator set is represented by the Operator type of the framework. Each set represents a set of operators, who are considered to be identical, i.e., they are all able to perform the same tasks with identical distributions for task duration. The number of tokens in a place indicates the number of operators available.

4.1.3 Process Routes

A process route consists of a series of process steps. Each route describes how a wafer lot is routed through the wafer fab. The PN simulation model generation for the process routes works as follows: For each process route, each of its process steps is generated sequentially, beginning with the first.

4.1.4 Process Steps

There are three main types of process steps:

- Basic Process Step
- Batch Process Step
- Process Step with Setup

The basic process step corresponds to the object type `ProcessStep` of the data specification. Each of these three main types has a mapping to the PN.

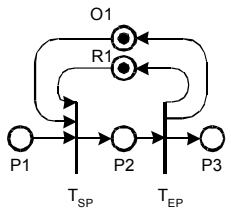

Figure 4: Basic process step.

Figure 4 shows the basic process step in its simplest form. P_1 is the input place. A `jobToken` will first arrive here. Place P_2 represents processing and place P_3 represents the completion of the process step. P_3 will also be the input place of the next process step. Transition T_{SP} represents the beginning of processing. It consumes a token from the tool set place R_1 and the operator set place O_1. T_{SP} will also add the processing time to the token. After processing is finished, T_{EP} will fire and add a token to each of the tool set and the operator set places, i.e., these resources are released. Note that the operator and tool set places can have arcs to other process steps, which are not shown here.

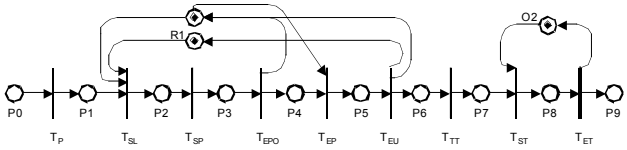

Figure 5: Complex process step.

Figure 5 shows a more complex process step. For each process sub-step there is a transition at the beginning, which will set the priority for the job token to the appropriate value (here T_P). Different dispatch rules can be implemented by assigning the appropriate value to the job token. This transition is present at the beginning of each process sub-step, but is omitted in most illustrations. The interpretation of the transitions in Figure 5 is as follows. T_{SL} marks the start of the loading process, T_{SP} represents the end of the loading and the beginning of the processing with operator, and T_{EPO} corresponds to the end of the processing with operator because it releases the operator seized by T_{SL}. T_{EP} is the end of processing and the beginning of the unloading process. T_{EU} represents the end of the unloading process and releases all resources. T_{TT} represents the travel time within the tool. T_{ST} seizes the operator for the transport to the next tool, and T_{ET} releases the operator again. Note that the tool is seized during the entire time, starting with the loading process at T_{SL} until the unloading has finished at T_{EU}. The transitions T_{SL}, T_{SP}, T_{EP}, T_{EPO}, T_{TT}, and T_{ST} all add the appropriate times to the token time stamp, to represent the respective time delays.

4.1.5 Batch Process Step

The batch process step has the ability to batch several lots together and process them at once, e.g., wafer lots that are processed together in an oven. The wafer lots that are batched together can also come from different process steps, as long as the `batchID` fields are identical. The `batchID` field is an identifier that marks lots that can be processed together. However, the data sets specify minimum and maximum batch sizes. This makes it very hard to model it in a straightforward manner. Since the capacity of the process step is greater than the minimum lot size, the number of lots that have to be processed can vary between the minimum and maximum batch size.

Figure 6 shows a simple batch process step that addresses this problem. Here the most basic version is introduced. It has a minimum batch size of two wafer lots and a maximum batch size of five lots. The process route that wafer lots will follow is represented by the path P_1, T_{S1}, P_2, T_{S2}, P_3, T_{EP}, and P_4, where P_1 represents the arrival place and P_4 the end of processing place. P_3 represents the

processing stage and P_2 is an intermediate place needed to model the batch mechanism. The liveness of this structure is ensured by assigning certain priorities to specific transitions. A detailed analysis can be found in Mueller (2007), who considers more complex operations (e.g., loading and unloading).

These examples give just a brief overview of the available types of process steps. For each element in the data, specification there is a mapping to an appropriate PN sub-net. The exact structure of the sub-net will depend on the given data. There is also a sub-net that can model setup times, which is not shown here.

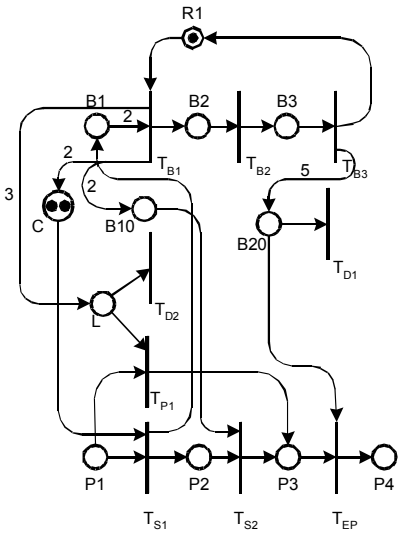

Figure 6: Batch process step.

4.1.6 Rework Sequence and Scrap Modeling

A rework sequence has the same basic structure as a process route. The difference is that it consists only of a few process steps. The Sematech dataset specifies for some process steps the probability that a lot has to go through a rework sequence. This is accomplished with a special type of a transition that can "switch" with a certain probability to a different place when firing. This place is the beginning of the rework sequence. At the end of the rework sequence, the lot is send back to the original sequence.

The modeling of scrap is similar: a special transition sends a token to a place with scrap probability, where all the scrapped lots are accumulated.

4.1.7 Dispatch Rules and Representation of Queues

Dispatch rules are used to establish an order for jobs that are waiting to be processed by a resource. Typical dispatch rules are FIFO or SPT (Shortest Processing Time), but the range of possible dispatch rules is very large. A dispatch rule in this framework assigns priorities to tokens that represent wafer lots. The assignment can be made with any function of the token attributes. The following attributes are available:

- Release Time
- Number of Operations
- Total Processing Time
- Number of Operations Remaining
- Processing Time Remaining
- Processing Time

The first three attribute are fixed values that are assigned to a job token when it is generated. The remaining attributes need to be updated at each process step. It is possible to extend this list to any conceivable attribute.

There is no explicit representation of queues in the framework. A queue is represented either by single place or by a set of places. If there is only one type of job waiting for a resource, the queue is simply the place that holds the job token that is waiting to be processed. If there is more than one job type, the queue is represented by the set of places that hold the job tokens that are waiting to be processed.

4.2 Generation of the PN Simulation Model

This section gives an overview of the simulation model generation procedure. The basic generation procedure consists of two steps. First, the specification is loaded from an XML file. This will instantiate a `FabModel` object (Section 3). This object is the root object for the wafer fab, i.e., it contains all the other objects that have the necessary information to generate the entire simulation model. The overall procedure is as follows:

- Create Tool Sets
- Create Operator Sets
- Create Process Routes
- Create Rework Routes
- Create Input Transition

First, all the places that represent tool sets and operator set are created. The number of tokens in each place represents the number of available resources. Then the process routes and the rework sequences are created by synthesizing the individual process steps together. Finally, the input transitions are created which are responsible for releasing lots into the system.

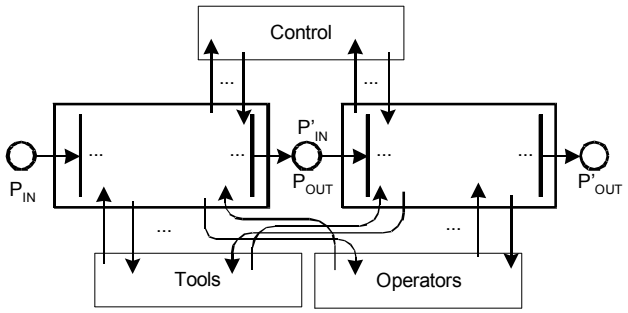

Figure 7: Synthesizing of process steps.

Figure 7 shows an example of two process steps joined at places P_{OUT} and P_{IN}. Each process step has one place for receiving and one place for sending job tokens to the next process step. These places serve as coupling points to generate a large simulation model. The boxes contain all transitions that model the different process stages of the process step. Arcs are connected to the tool and operator places in a specific way. For some process steps, such as batch process steps, there are also arcs to and from control places. In this fashion individual process steps are joined together to form process routes and rework sequences. The detailed algorithms can be found in Mueller (2007).

4.2.1 Example

A GUI was developed that allows loading and generating the model. The first step to generate the simulation model is to load the model data from an XML file. A small portion of the generated PN is shown in Figure 8 (data set 1). Also some arcs are not displayed for better readability (the arcs to the tool places and operator places have been omitted). The PN simulation model in this example has 6363 places and 3751 transitions. An overview of the size of the other simulation models generated from the Sematech data sets is given in Table 1.

Table 1: Size of PN simulation model.

Data Set	# of Places	# of Transitions
1	6,363	3,751
2	26,803	19,751
3	46,199	31,845
4	1,269	1,013
5	52,135	36,075
6	38,241	25,463

5 ANALYSIS OF GENERATED PETRI NET

It can be shown that some properties of PN simulation model are not impacted by the timing mechanism and the introduction of priorities (Mueller 2007). These properties are liveness and boundedness. The introduction of time and priorities will give an order to transitions that are enabled simultaneously. This means that the reachability graph of the PN simulating model will be a sub-graph of the underlying PN. This in turn makes it possible to analyze the PN simulation model in the same way as a classical PN. The generated PN simulation model can be shown to be live and bounded for most places. Furthermore, reachability of the end processing state for each process route can be established.

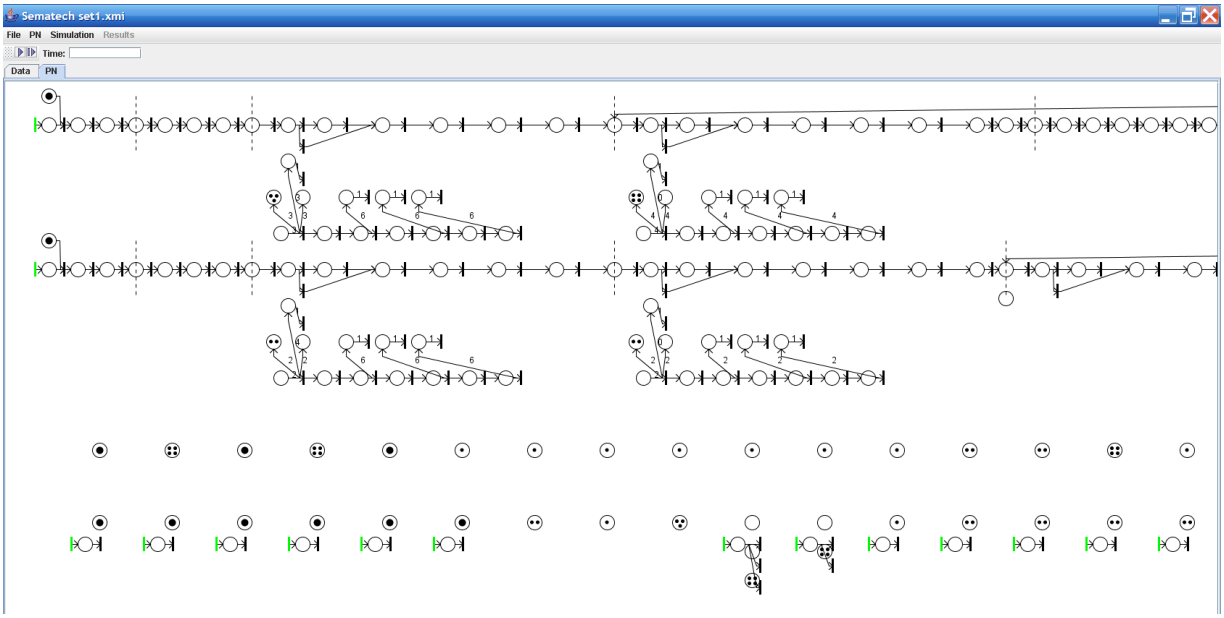

Figure 8: Example PN simulation model.

6 CONCLUSIONS AND FUTURE RESEARCH

This article introduced a novel approach to simulation modeling and simulation model generation by introducing a simulation framework that is based on an object-oriented PN data structure. This framework has several unique features because it uses as a basis the same execution rules as classical PNs, yet it has extensions for time as well as priorities for firing of transitions. This allows the representation of time as well as the implementation of dispatch rules. This is not possible with classical PNs.

For the simulation model generation an object model was developed, which servers as a basis for the simulation data specification. This object model contains all information needed for the generation of the simulation model. Instances of the simulation data specification can be stored in an XML file, which is based on this object model. Changes to the simulation model can be performed by editing this file and regenerating the simulation model.

The generation procedure that generates the PN simulation model is based on a mapping from the object model of the simulation data specification to the PN. Each of the main objects in the simulation data specification, such as resources and process steps, correspond to parts in the PN. The mapping serves also as an unambiguous description of the simulation model. The feasibility of this approach was demonstrated by means of the Sematech data set.

In summary, the simulation framework makes creating large-scale discrete-event (computer) simulation (DES) models for manufacturing systems more manageable. It has several advantages: there is an explicit description of the simulation model in form of a PN. The user can verify exactly how each component is working. The end user does not have to do any coding of the simulation model. The simulation model is described in a problem specific domain, in this case semiconductor manufacturing. The simulation model is specified as an instance of an object model that serves as a simulation data specification. The mapping from this object model to the PN is fixed; this means that the simulation model generation is a rigid process, which can avoid programming errors. Theoretically, there is no limitation for this framework to model any discrete-event system. In principle, all systems that can be modeled as finite-state ones can be modeled. However, there are some disadvantages. Due to the rigid control, it may not be possible to make changes in the behavior of the simulation model with ease. The behavior of the simulation model is determined by the mapping from the object model to the PN simulation model. Hence, this mapping has to be changed to implement different behaviors.

REFERENCES

Becker, M. 2003. Modeling and simulation of a complete semiconductor manufacturing facility using Petri nets. In *Proceedings of the 2003 IEEE Conference on Emerging Technologies and Factory Automation*, 153-155. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Gong, D.-C. and L. F. McGinnis. 1990. An AGVS simulation code generation for manufacturing applications In *Proceedings of the 1990 Winter Simulation Conference*, eds. O. Balci, R. P. Sadowski, and R. E. Nance, 676–682. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Lee, S., H. Cho, and M. Jung. 2000. A conceptual framework for the generation of simulation models from process plans and resource configuration. *International Journal of Production Research* 38:811–828.

Lee, Y. T. and L. Yan. 2005. Data exchange for machine shop simulation. In *Proceedings of the 2005 Winter Simulation Conference*, eds. M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 1446–1452. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Lu, R. F., G. Qiao, and C. McLean. 2003. NIST XML simulation interface specification at Boeing: A case study. In *Proceedings of the 2003 Winter Simulation Conference*, eds. S. Chick, P. J. Sanchez, D. Ferrin, and D. J. Morrice, 1230–1237. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Mathewson, S. 1975. Program generators. In *Interactive systems*, 423–39. Uxbridge: Online Conferences Ltd.

Mathewson, S. 1985. Simulation program generators: code and animation on a PC. *Journal of the Operational Research Society* 7:583–589.

McLean, C., A. Jones, T. Lee, and F. Riddick. 2002. An architecture for a generic data-driven machine shop simulator. In *Proceedings of the 2002 Winter Simulation Conference*, eds. E. Yüsescan, C.-H. Chen, J. L. Snowdown, and J. M. Charnes, 1108–1116. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Mueller, R. 2007. *Specification and automatic generation of simulation models with applications in semiconductor manufacturing*. Ph.D. thesis, H. Milton Stewart School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, Georgia.

Murata, T. 1989. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 7:541–580.

Ramirez-Hernandez, J. A., L. Heshan, E. Fernandez, C. McLean, and L. Swee. 2005. A framework for standard modular simulation in semiconductor wafer fabrication systems. In *Proceedings of the 2005 Winter Simulation Conference*, eds. M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 2162–2171.

Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Van der Aalst, W. 1993. Interval timed coloured petri nets and their analysis. In *Applications and Theory of Petri Nets 1993. 14th International Conference Proceedings*, eds. M. A. Marsan, 453–472. Berlin: Springer-Verlag.

Wang, J. 1998. *Timed Petri nets: Theory and application.* Boston: Kluwer Academic Publishers.

Zurawski, R. and M. Zhou. 1994. Petri nets and industrial applications: A tutorial. *IEEE Transactions on Industrial Electronics*. 6:567–583.

Zhou, M. and M. Jeng. 1998. Modeling, analysis, simulation, scheduling, and control of semiconductor manufacturing systems: A Petri net approach. *IEEE Transactions on Semiconductor Manufacturing,* 11:333-357.

AUTHOR BIOGRAPHIES

RALPH MUELLER is a Ph.D. candidate in the School of Industrial and Systems Engineering at the Georgia Institute of Technology. His research interests include simulation modeling and , as well as application of simulation in the semiconductor manufacturing. His email address is <ralphm@isyegatech.edu>.

CHRISTOS ALEXOPOULOS is an Associate Professor in the School of Industrial and Systems Engineering at the Georgia Institute of Technology. His research interests are in the areas of applied probability, simulation, and optimization of stochastic systems. He is a member of INFORMS and an active participant in the Winter Simulation Conference, having been Proceedings Co-Editor in 1995 and a member of the Board of Directors on behalf of the INFORMS Simulation Society. He is also the Simulation Department Editor of *IIE Transactions* and an Associate Editor of *Networks*. His e-mail address is <christos@isye.gatech.edu>.

LEON MCGINNIS is Gwaltney Professor of Manufacturing Systems at Georgia Tech, where he also serves as Director of the Product and Systems Lifecycle Management Center, Associate Director of the Manufacturing Research Center, and Director of the Keck Virtual Factory Lab. His research is focused on the representation of complex industrial systems, such as warehouses and factories, to enable analytic and simulation modeling to support performance assessment, behavioral prediction, and system design. His email address is <leon.mcginnis@gatech.edu>.