DEVELOPMENT OF MODEL COMPONENT TEMPLATES FOR SEMICODUCTOR MANUFACTURING PROCESS SIMULATION IN DEVSJAVA

by

Bin Xie

An Applied Project Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Computer Science

ARIZONA STATE UNIVERSITY

December 2004

DEVELOPMENT OF MODEL COMPONENT TEMPLATES FOR SEMICODUCTOR MANUFACTURING PROCESS SIMULATION IN DEVSJAVA

by

Bin Xie

has been approved
December 2004

APPROVED:

Hessam S. Sarjoughian, _____, Chair

Stephen Yau_____, Supervisory Committee

ACCEPTED:

_____
Head of Academic Unit

ABSTRACT

Modeling manufacturing processes is important to the operation of semiconductor supply-chain networks. As the scale and complexity of manufacturing processes increase, appropriate model abstractions of such network become important for carrying out simulation studies. Given the scale of models for semiconductor supply-chain systems, it is useful to simplify simulation model development. In this project, representative manufacturing process configurations have been examined and analyzed. Pipeline model templates with join and fork constructs are developed. Automatic couplings among the parts of a pipeline template are supported. Simulation of model templates for common process models is presented. These template models are implemented in the DEVSJAVA simulation environment and their use is demonstrated using an illustrative example.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

1. INTRODUCTION

Operation of a semiconductor supply-chain network poses a variety of challenges such as how best to maximize return on investment. A semiconductor supply-chain network has a manufacturing process which must be operated and managed to deliver products to users under some decision policies. The operation of a manufacturing process as shown in Figure 1 is complex in nature due to long production lead times, short product lifecycles, and large demand forecast variations. For example, the Fabrication plant can take ninety days to process raw silicon wafers and produce wafers with circuitry. Similarly, the Assembly Test plant can take twenty days to cut wafers and package them for delivery to customers.

Since it is impractical to directly experiment with a supply-chain network which costs in billions of dollars, we must use some other approach to manage from day-to-day to multi-month life-cycle operation of a *Semiconductor Manufacturing Process* (SMP) as shown in Figure 1.

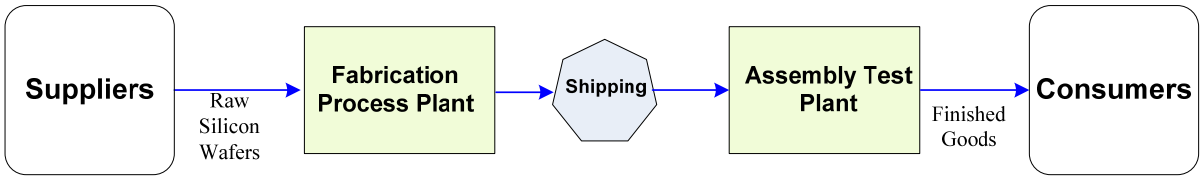

Figure 1: Semiconductor Manufacturing Process

Modeling and simulation presents itself as an attractive approach to understand the complexity of semiconductor supply-chain networks and therefore be able to operate its parts individually and collectively. Many simulation modeling tools and approaches have been developed for studying manufacturing process [1]. These tools provide a library of model components that can be used to synthesize different manufacturing plant layouts – e.g., an SMP model can have a Fabrication Plant, Assembly Test plant, and Shipping components (see Figure 1).

1.1. Background

There are a number of existing modeling methodologies and simulation frameworks which use discrete-event, discrete-time, or continuous worldviews to characterize manufacturing processes. Among these, discrete-event modeling is most suitable since dynamics of manufacturing processes are discrete event that occurs at arbitrary time instances – i.e., events occur asynchronously. The Discrete Event System Specification (DEVS) is a modeling formalism which supports describing dynamic systems as discrete event models [2]. In this framework, two types of models are supported – atomic and coupled. The former captures non-decomposable dynamics and the latter captures hierarchical composition of atomic and coupled models. The coupled models are composed by coupling (atomic and coupled) models via a coupling scheme that links the output ports of the models to the input ports of other models. The couplings of unidirectional – output ports are coupled to input ports. If this coupling is done correctly, the resulting coupled model is regarded as closed under coupling, which means that it can be treated as if it were an atomic model. This property allows development of complex systems in a hierarchical and modular manner with well defined input and output interfaces as well as time-based causal behavior.

DEVSJAVA is a Java-based environment for developing and simulating DEVS models [3][4]. User-defined atomic and coupled models are inherited from the classes in the DEVSJAVA packages. It can model complex model behavior in a systematic fashion. During simulation, models are presented in graphical manner for easy control. The behavior of these models in terms of their states and input/output trajectories can be automatically traced which facilitates both runtime observation as well as post-simulation analysis.

Existing commercial discrete event simulation packages such as ARENA [5] offer general process modeling capabilities such as delays, queues, branching, splitting, merging and resources. ARENA is flow-oriented simulation language based on SIMAN [5]. Models can be easily created by drag and drop. However, modeling a complex semiconductor supply-chain network with low-level modeling constructs (e.g., branching component) can be extremely time-consuming and error prone. ARENA similar to other contemporary tools

allows creating customized building blocks (components), it would not be practical for end-users to extend its modeling engine.

1.2. Related Work

A simulation model was developed to study the Intel's supply network from Assembly Test to the end customer using the DEVSJAVA environment [6]. An important aspect of this model is the separation of material processes from decision-making (refer to Figure 2). This approach supports the use of different modeling techniques for the Decision and Processing layers.

However, this model did not support model scalability in terms of modeling data and control interaction between the Processing and Decision layers. This lack of scalability led to development of an approach where the data and control connectivity follows a well-defined scheme [7].

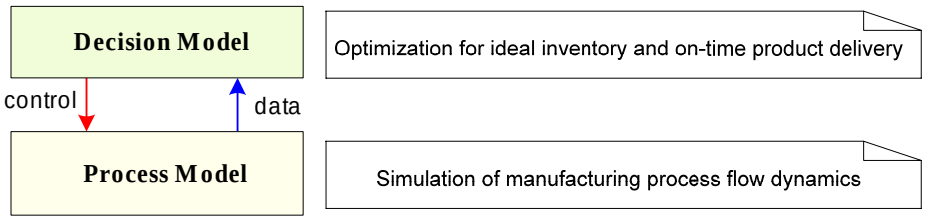

Figure 2: Two layer Abstract Model of Semiconductor Supply Network

As suggested in Figure 3. This approach introduces single control input/output into and out of the simulation model. It enables the messages from the Decision layer to be broadcasted to all components in the Processing layer. The simulation model components are designed in such a way that they accept messages that are directed to them. For a large model the ability to automatically route messages in a network of nodes is very important and valuable. As part of this approach, the model of the Processing layer (process model) is divided into two parts: Manufacturing Network and Logistic Network. This is a natural partition that separates components of the manufacturing process from the distribution process. An advantage of this approach is the ability to reuse logistic topologies with different manufacturing configurations or conversely with different logistics configurations. The manufacturing

process can be viewed as a pipeline network consisting of four components as shown in Figure 3 – Assembly Test plant, Semi-Finished Goods Inventory (SFGI), Finish Line and End Of Factory (EOF).

The components in Figure 3 can be modeled as atomic or coupled DEVS models. The *coupled models* are instances of a set of model templates called SNNode (Supply Network Node), FabNode (Fabrication Node), ATNode (Assembly Test Node), ManufacturingNode (Manufacturing Node) and LogisticsNode (Logistics Node). The ATNode and FabNode are coupled model components of the ManufacturingNode and the DistNode and TermMode are atomic component of the LogisticsNode model [7]. The SNNode represents a coupled model which characterizes the basic structure and behavior inherited by all other model components. This model component can dynamically interact with one or more model components via data input and output ports (dataIn and dataOut) and control input and output ports (cntrlIn and cntrlOut). With these template model components, the DEVSJAVA modeling engine can support automatic configuration of coupled models using cntrlIn and cntrlOut ports with the Decision model. As shown in Figure 3, the Manufacturing Network (MN) component is composed from the Assembly Test Factory and EOF components. This SMP process is divided to have the Manufacturing Network and the Logistic Network to allow independent reconfiguration of models.

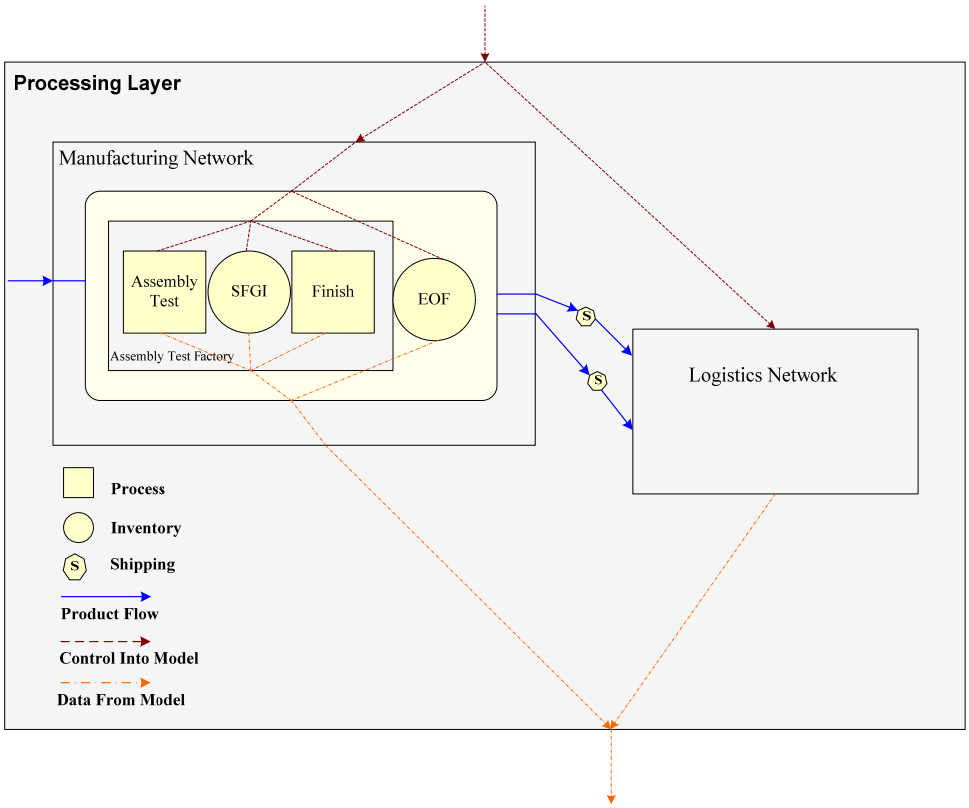

Figure 3: Simulation Model for Processing Layer of a Semiconductor Supply Network

In addition to the above models, there are three types of *atomic models* called inventory, process and shipping (see Figure 4). These are derived from the Supply Network Entity (SNEntity) model which provides a set of functions to track the simulation time and to report inventory levels.

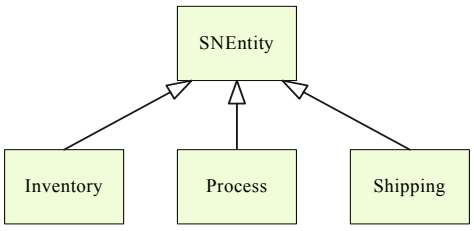

Figure 4: Semiconductor Manufacturing Process Atomic Models

1.3. Project Scope

To support rapid model development of a *Semiconductor Manufacturing Process*, it is important to develop a set of reusable coupled model templates that can be readily customized independent of their scale. Our objective is to extend the *simple pipeline model* templates described in [7] to a set of configurable templates for *complex pipeline models*. Any number finite number of model components can be coupled automatically using "fork" and "join" modeling constructs. These complex pipeline model templates are important for modeling realistic semiconductor manufacturing process configurations (topologies). For example, a fork element can be readily configured to couple one "sender" model component to two or more "receiver" model components. Since the SMP models need to be composed with other kinds of models, they must adhere to the data and control interaction schemes developed in previous work [7].

With the new template models, arbitrary structures such as one having multiple Finish components can be configured (see Figure 5). These provide higher level of model abstraction which extends model reuse and enable scalable model development. The model used for this project is different from the previous one. These model templates are used in the DEVSJAVA simulation environment and validated with a semiconductor manufacturing process example.

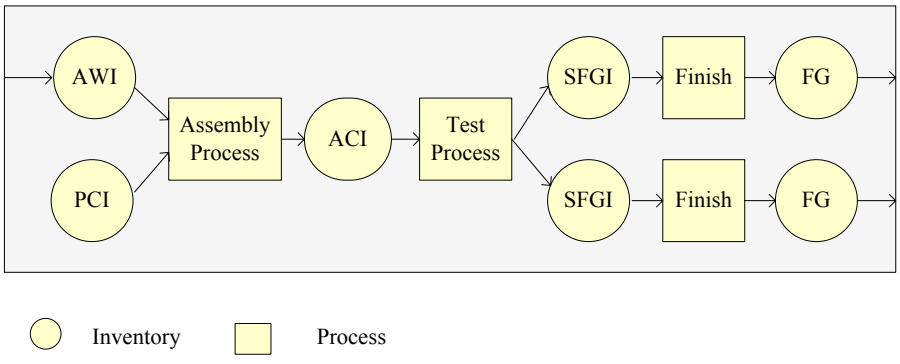

Figure 5: An Example of Assembly Test Process Configuration

CHAPTER 2

2. SEMICONDUCTOR MANUFACTURING PROCESS TEMPLATE MODELS

A semiconductor manufacturing process can have many hundreds of parts, each of which can be modeled and simulated individually or collectively. In this chapter, we describe different types of model configurations that may be found in a typical semiconductor manufacturing processes. The mapping of these configurations into the complex template models are described next followed by their implementation in the DEVSJAVA modeling and simulation environment.

2.1. Hierarchical Model Abstractions

Modeling depends on abstractions that simplify detailed behavior. The simplifications can focus on hiding details from the end user, removing details that are not necessary for the modeling task, or some combination. Appropriately developed abstractions offer important advantages such as shielding domain experts from specifying how control and data to be enforced between model components. Equally important is for modelers to benefit from pre-built model templates and thus depend on model reuse. Furthermore with composite model templates in comparison to atomic model templates, we can tackle model scalability.

Increasingly larger and more complex model can be developed from smaller, simpler model components. Therefore, how to identify the building blocks plays a key role in model abstractions. The manufacturing process configurations we are interested to model are shown in Figure 6. Based on the domain specific roles and responsibilities of components shown in Figure 6, it is not difficult to find that each manufacturing network can be partitioned into several fabrication (FAB) processes, assembly test (AT) processes and shipping. The corresponding model abstractions are shown in Figure 7.

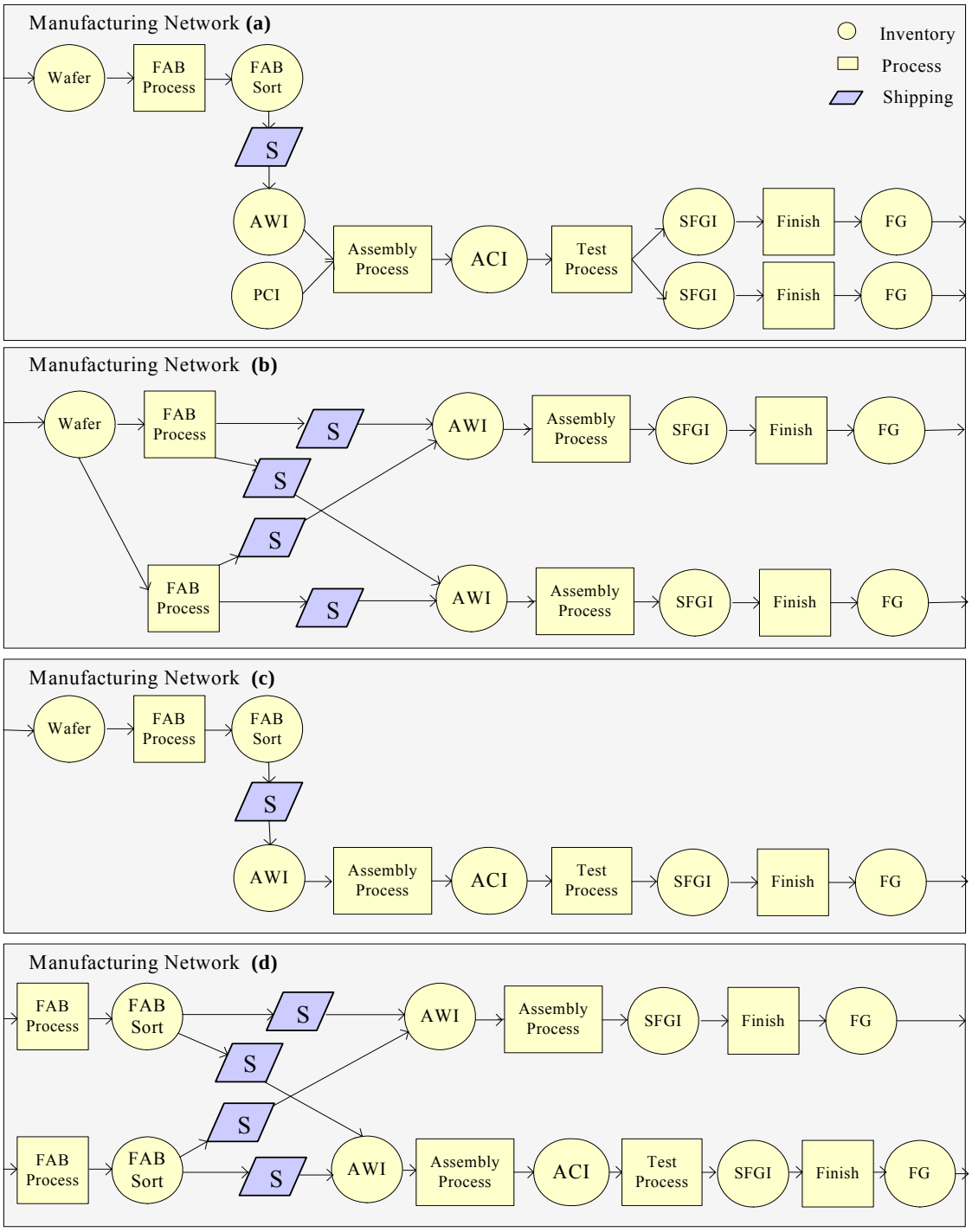

Figure 6: Manufacturing configurations with different number of FAB and AT Processes

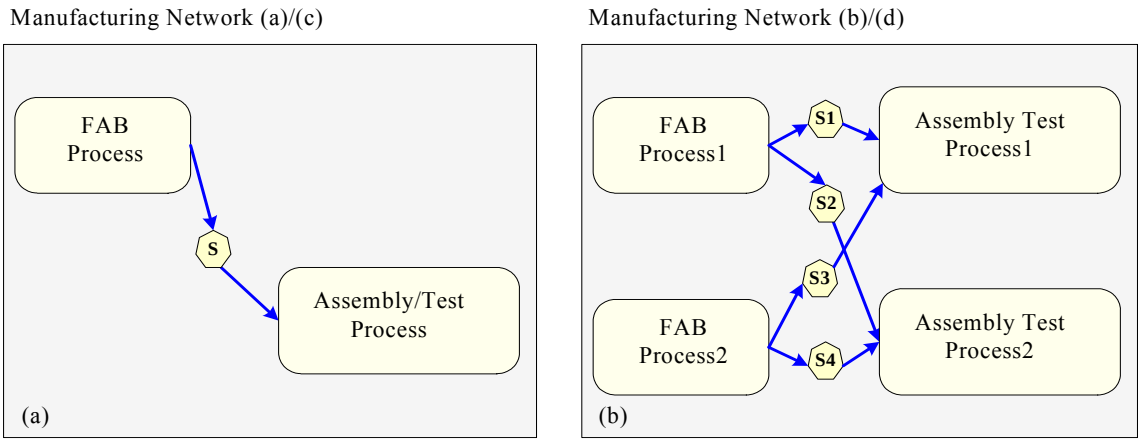

Figure 7: Abstract Manufacturing Network Models

Given different parts of the models in Figures 6 (a) and (c), we can derive an abstract model shown in Figure 7 (a). Similarly, the models shown in Figures 6 (b) and (d) can be abstracted to the model shown in Figure 7 (b). Therefore, in general, the manufacturing process can contain any number of FAB processes, AT processes and shipping. FAB process, if there is any, is connected to AT process via shipping.

The number of the coupling between FAB process and the other components ranges from one to many. More generally, any two types of model components can have multiple couplings since these models can have fork and join structures. For example, in Figure 7 (b), the connection between FAB Processes (FabProcess1 and FabProcess2) and the Assemble Test Processes (ATProcess1 and ATProcess2) is listed in Table 1 as {“S1”,S1} , {“S2”,S2}, {“S3”,S3} and {“S4”,S4} that denote the shipping link model components. The Shipping Link is an atomic model denoting the time required for outputs of a fabrication plant to arrive at the input port of an assembly test plant.

FAB Processes \ Assemble Test Processes	ATProcess1	ATProcess2
FabProcess1	{“S1”,S1}	{“S2”,S2}
FabProcess2	{“S3”,S3}	{“S4”,S4}

Table 1: Adjacency Matrix for Manufacturing Network

2.2 Hierarchical SMP Components

Three different types of components – ATProcess, FabProcess and Shipping – have been identified for configuring SMP models. The FabProcess and ATProcess are aggregated into logical units since they provide abstractions that can be used more than once in a semiconductor manufacturing process model. The structures of these two model configurations are described below. The detail of the Shipping model can be found in [6].

2.2.1 Assembly Test Process

Given the conceptual partitioning in Figures 6 and 7, the ATProcess can belong to one of three types (or configurations) as shown in Figure 8. Each of these configurations can have one to many external input and output ports. However, these models are distinguishable from one another since their internal couplings among the components differ. The configurations fall into two categories: *simple pipeline* and *complex pipeline*. With the simple pipeline configuration, materials (outputs) flow through the model components in feed-forward assembly fashion (see Figure 8 (b) and (c)). A simple pipeline configuration may have multiple input ports or output ports. With complex pipeline configuration materials can flow through multiple model components inside via fork structure or materials are received from multiple model components via join structure (see Figure 8 (a)). Next we will identify the common configurations that constitute the ATProcess shown in Figure 8.

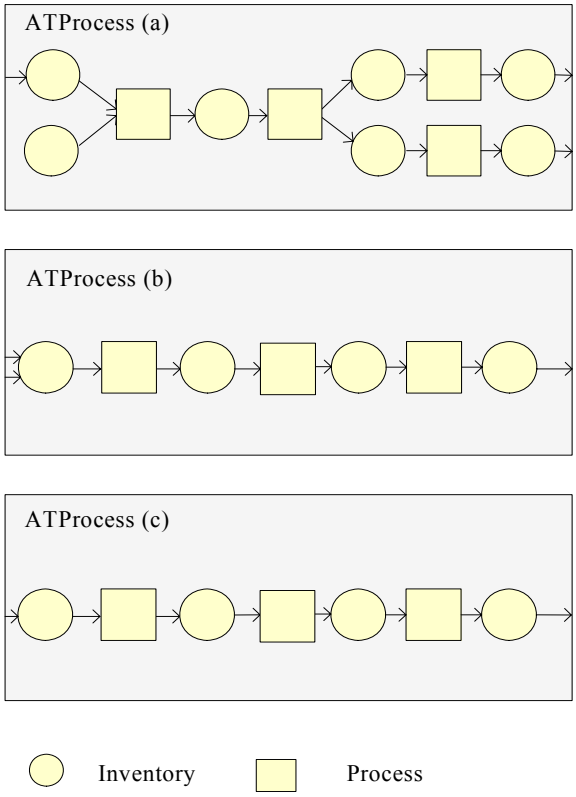

Figure 8: Assembly Test Process Model Templates

2.2.1.1 PLNode

For the rest of this report, we denote the constituents inside any coupled model as "nodes". By observing the specific simple pipeline ATProcess model configurations and considering more general cases, the following configurations are identified: (i) nodes in the ATProcess can be either a Process immediately followed by an Inventory, or an Inventory immediately followed by a Process, so the two atomic models are aggregated into logical units to form the new nodes called PINode and IPNode (see Figure 9); (ii) the number of the models in the ATProcess can be either odd or even resulting in PLNode as shown in Figure 10. The PLNode can have four configurations. In Figure 10 (a) and (b), the number of model components is even, and the number of model components is odd in Figure 10 (c) and (d). The model components in the PLNode can be IPNode and PINode.

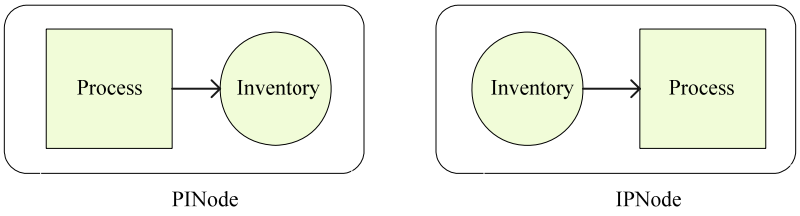

Figure 9: Base Nodes PINode and IPNode

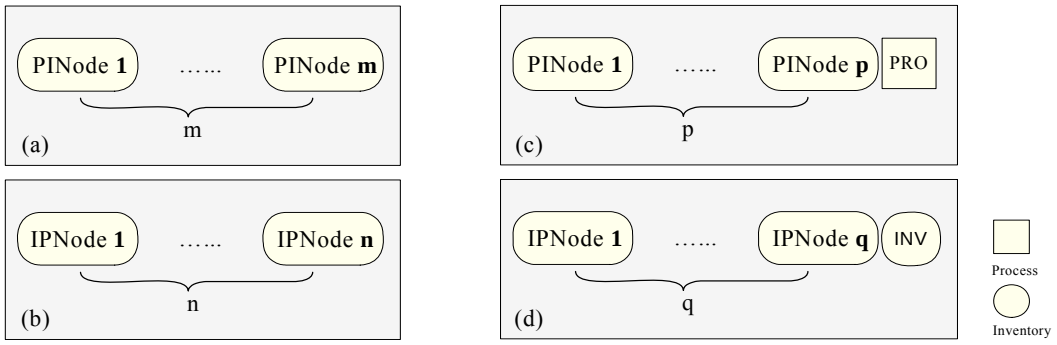

Figure 10: PLNode Model Templates

2.2.1.2 JoinNode and ForkNode Configurations

In the complex pipeline configuration, three base nodes can be identified: JoinNode, ForkNode (see Figure 11) and PLNode. The JoinNode represents the logical unit where several SNEntity model components are connected to one SNEntity model component, while in ForkNode, one SNEntity model component connects to several SNEntity model components. Based on this result, the ATProcess can be abstracted as shown in Figure 12. An example of how this design can be applied to real Assembly Test process configuration is also illustrated in Figure 12. The cardinality of couplings for data input and output ports ranges from one to many.

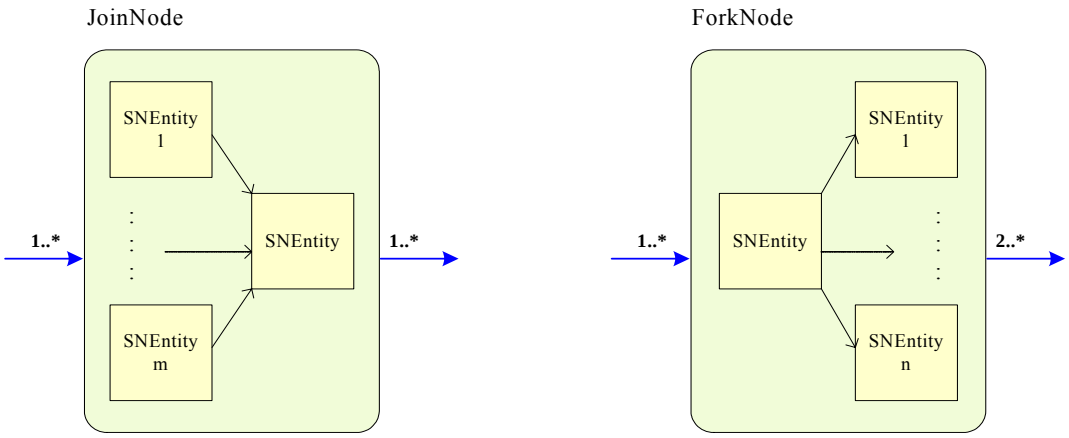

Figure 11: JoinNode and ForkNode Model Templates

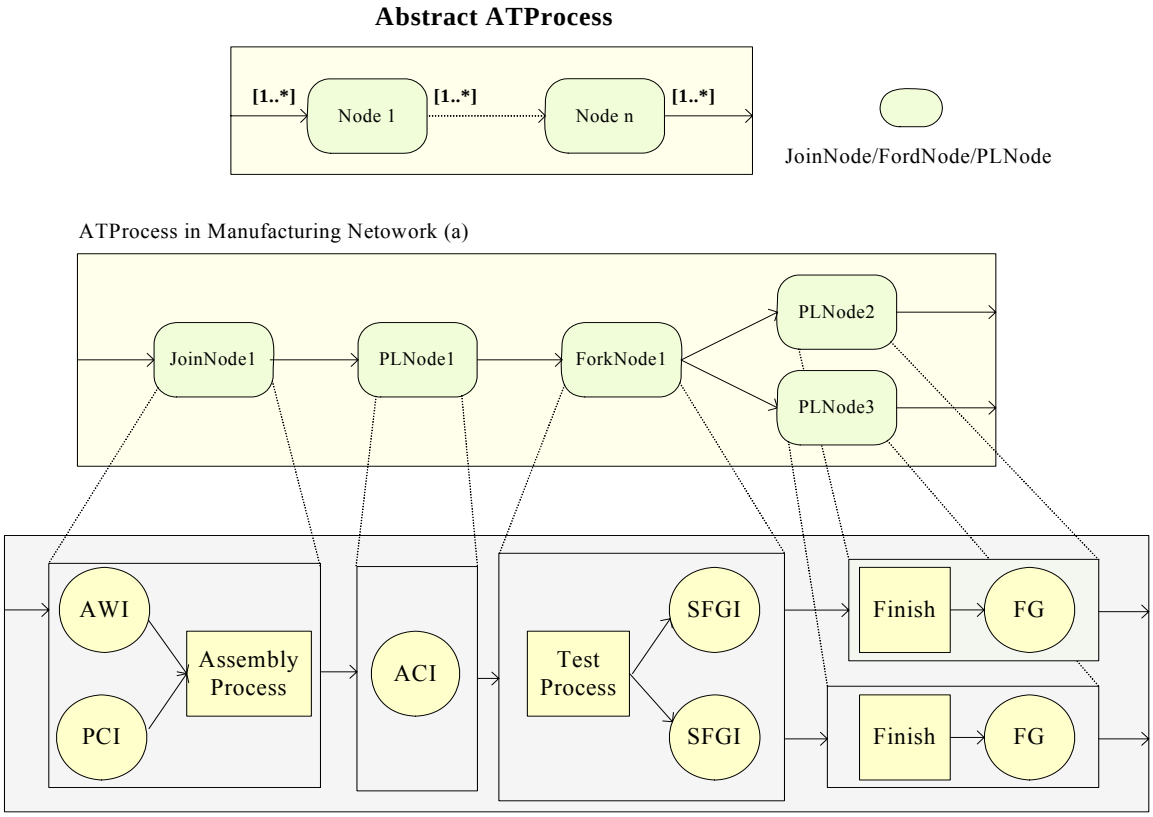

Figure 12: Abstract ATProcess Model Template and an ATProcess Example

2.2.2 FAB Process

Similar to the ATProces, three configurations of FabProcess are identified (see Figure 13). These falls into the category of the simple pipeline identified for the ATProcess – i.e., FabProcess configurations are not unique w.r.t. the ATProcess configurations.

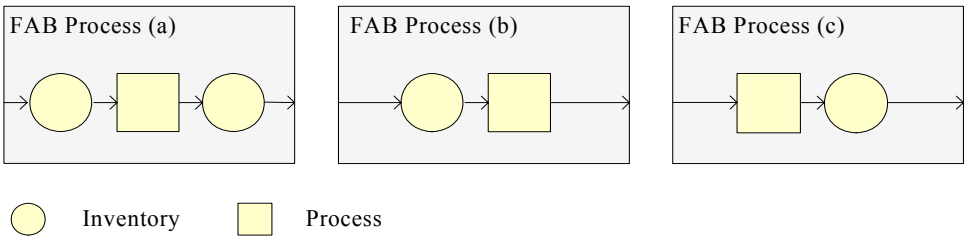

Figure 13: FabProcess Configurations

2.2.3 SMP Manufacturing Configurations

With the above model configurations, next we will illustrate how they can be used to model manufacturing configurations such as those shown in Figure 6.

2.2.3.1 Manufacturing Network (a)

For manufacturing network (a) shown in Figure 7, the inter-connection is listed by the adjacency matrix in Table 2 and the nodes inside it are shown in Fig 14. The FabProcess is a PLNode type (c) where there is one occurrence of PINode (see Figure 10 (c)). ATProcess is comprised of five nodes as shown in Figure 14. The connections among these nodes are shown in Table 3. Note there are no Shipping model components in the ATProcess model component, instead the connectivity among the components are ATProcess internal output to input port couplings.

FabPrcesses/ATProcesses	ATProcess
FabProcess	{“S”, S}

Table 2: Adjacency Matrix for Semiconductor Manufacturing Process (a)

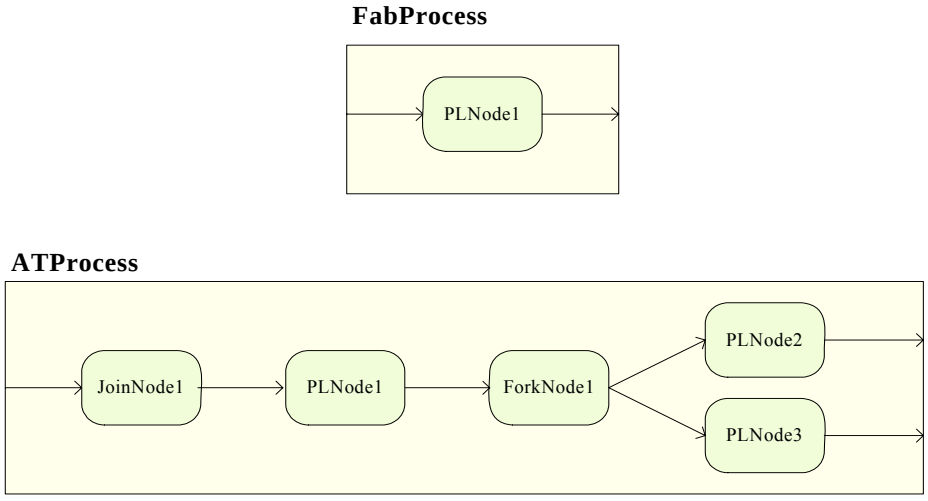

Figure 14: FabProcess and ATProcess in SMP (Figure 6 (a))

Couplings	PLNode1	ForkNode1	PLNode2	PLNode3
JoinNode1	√			
PLNode1		√		
ForkNode1			√	√

Table 3: Connections (internal couplings) for the ATProcess Coupled Model Component

2.2.3.2 Manufacturing Network (d)

Semiconductor manufacturing process model (d) depicted in Figure 7 (b) is also used to demonstrate the use of model templates. The corresponding connections and nodes are shown in Table 4 and Figure 15, respectively.

FabPrcesses/ATProcesses	ATProcess1	ATProcess2
FabProcess1	{“S1”,S1}	{“S2”,S2}
FabProcess1	{“S3”,S3}	{“S4”,S4}

Table 4: Adjacency Matrix for Semiconductor Manufacturing Network (d)

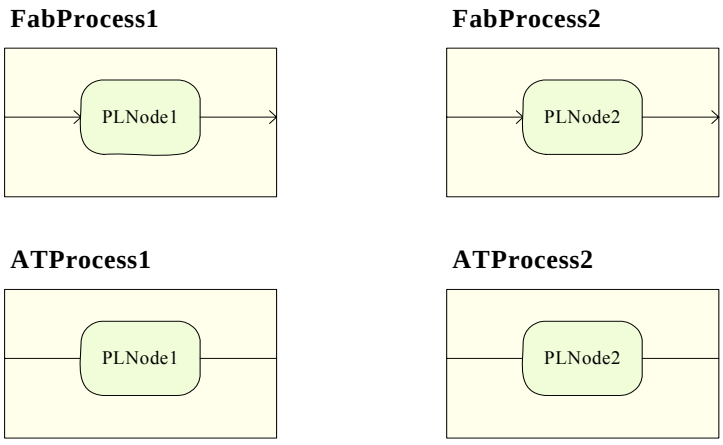

Figure 15: FabProcesses and ATProcesses in Manufacturing Network (Figure 6 (d))

2.3. Software Design and Implementation in DEVSJAVA

The template models described in Section 2.2 are developed in the DEVSJAVA simulation environment. The UML specification for these template models are derived from the ConfigurableCoupledModel class as shown in Figure 16. The classes IPPINode, PLNode, ForkNode, JoinNode, ManuProcess, ATProcess, FabProcess, and ManufacturingNetwork correspond to the model templates presented in Section 2.2. These classes and their relationships are depicted inside the dash-line box in Figure 16.

These models inherit input and output ports ctrlIn, dataIn, ctrlOut and dataOut. This allows automatic coupling for the dataIn and dataOut ports within complex pipeline models. This capability is supported with the introduction of two new methods – addAdditonalDataInportEdge and addAdditonalDataOutportEdge – to the SNNode. These methods are necessary to handle one or more couplings between any two components of a model such as FABProcess and ATProcess classes.

The ManufProcess class provides the functions to add PLNode, JoinNode and ForkNode into a Semiconductor Supply Network model. The ManufacturingNetwork class provides the functions to be able to add any number of FabProcess, ATProcess and Shipping. The data flow connections within each class inside the dash-line box are automatically handled and are constructed at runtime. Automatic configuration of complex models is key in

requiring manual configuration of models and thus can greatly reduce errors in model development. An important feature of automatic model configurations is to easily develop large-scale models consisting of many tens or hundreds of components.

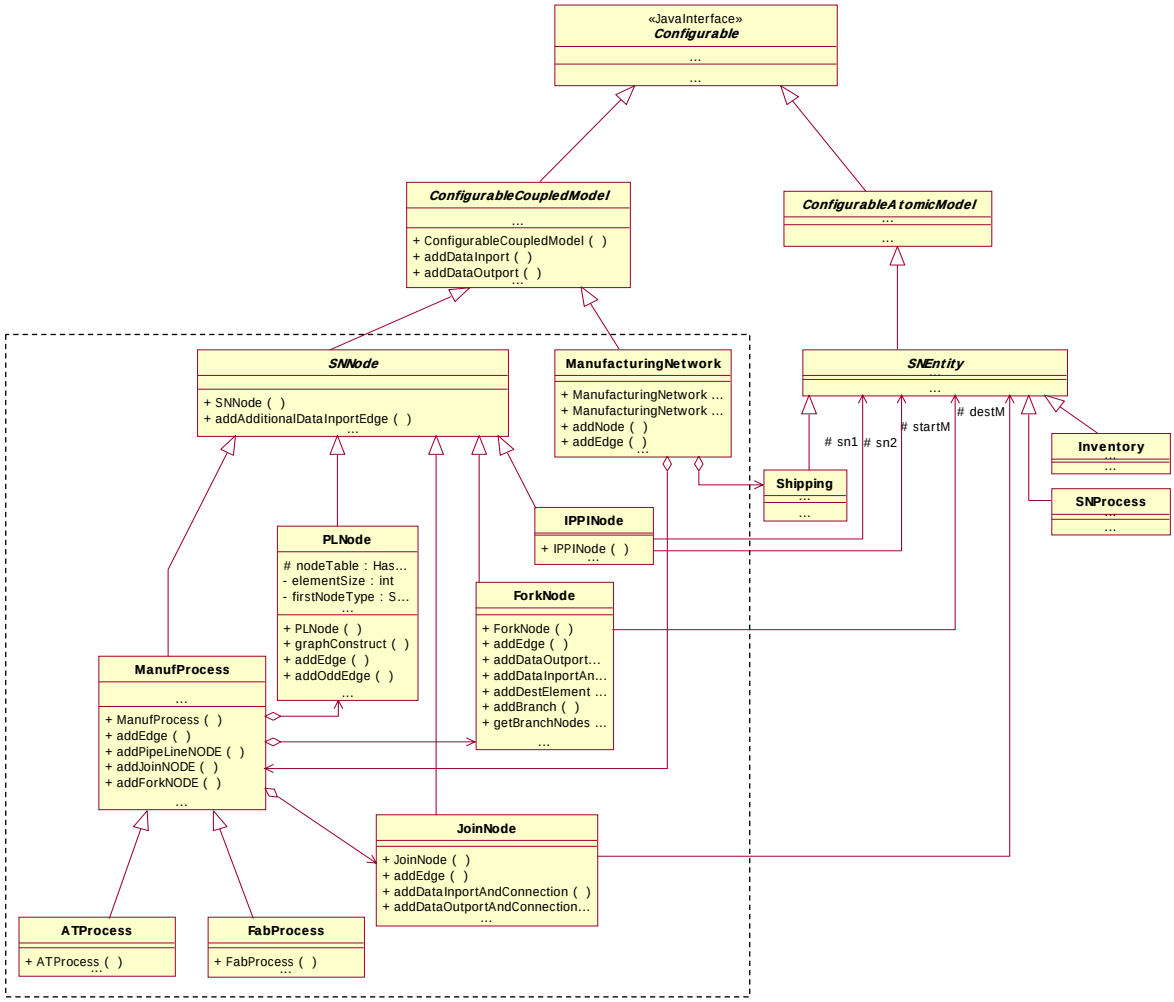

Figure 16: UML Design for the Template Models

CHAPTER 3

3. EXPERIMENTAL SETUP, RESULTS AND CONCLUSIONS

3.1. Experimental Setup and Results

To show the use of the template models presented in Section 2.3 and their roles in model configuration, two Semiconductor Manufacturing Process models are considered. One model called "SMP Model A" has a simple pipeline configuration (see Figure 17). This model serves as a *Reference model*. Another model has complex pipeline configuration as shown in Figure 18. This model called "SMP Model B" contains all the model templates and serves as a *Test model*. Despite their differences in their configurations, these two models are initialized and subjected to identical experimentation in order to examine the correctness of the implementation of the new SMP DEVSJAVA classes. This set up allows us to execute two sets of simulation experiments one for each of the models. We can compare the simulation results of SMP Model A and SMP Model B and validate the realization of the hierarchical model configurations and their auto-construction of the data flow couplings. In SMP Model A, all the model components are atomic and their dataIn and dataOut couplings are specified manually.

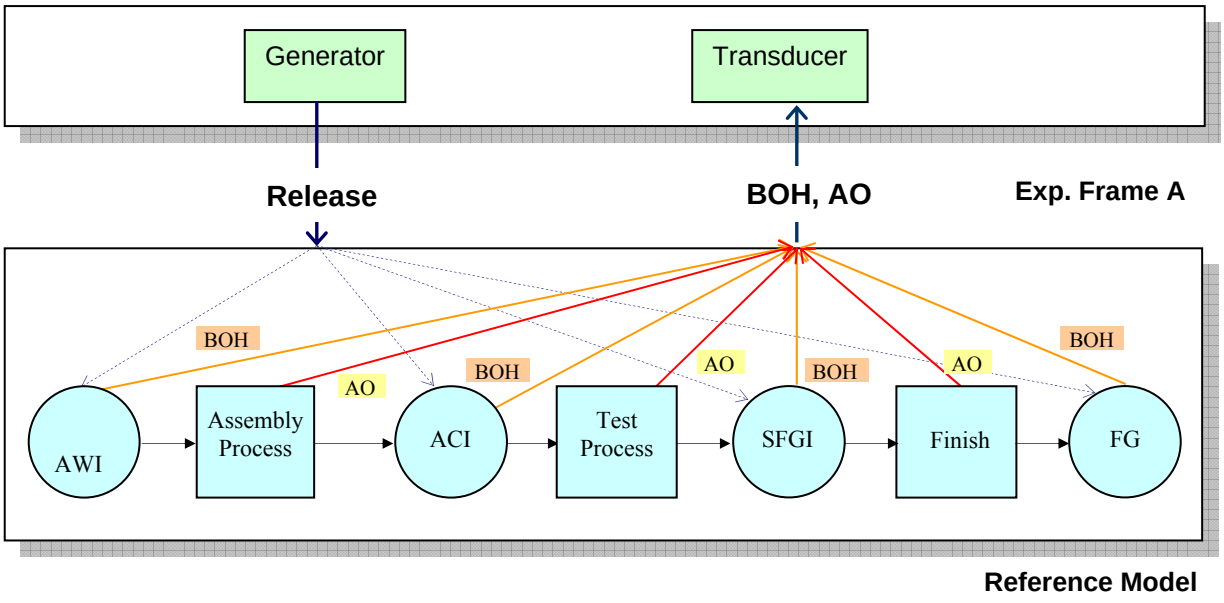

Figure 17: SMP Model A (Reference Model)

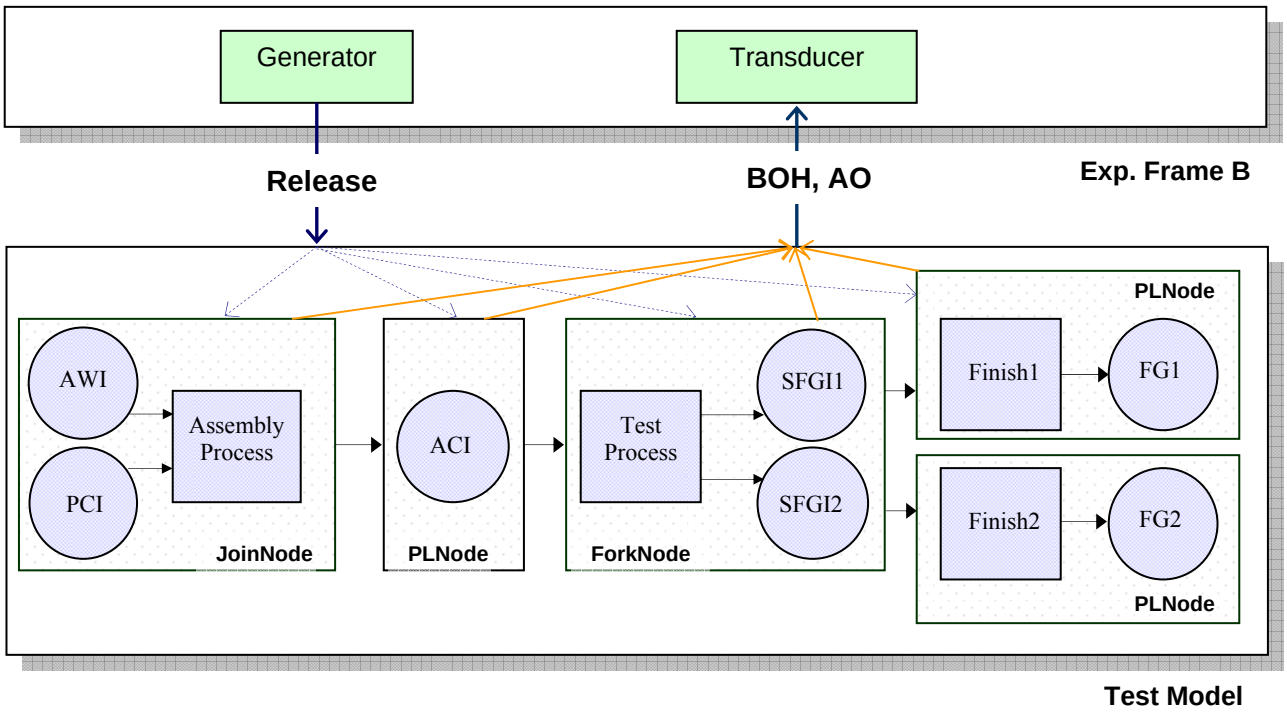

Figure 18: SMP Model B (Test Model)

The experimental frame is set up and used for both Reference and Test models. The Experimental Frame A (EFA) for the Reference model consists of Generator and Transducer. The Experimental Frame B (EFB) for the Test model consists of the same Generator and Transducer as in the EFA.

The Generator will send the release commands to the processing model at every 0.75 time period.) As shown in Figures 17 and 18, Transducer will monitor the Actual Out (AO) messages sent by Assembly Process, Test Process, and Finish process (or Finish1 and Finish2 processes. The inventory levels, BoH (beginning on hand), of AWI, ACI, SFGI, and FG of the Reference model are also measured against the inventory levels of AWI, PCI, ACI, SFGI1, SFGI2, FG1, and FG2 of the Test model at the beginning of each time period. The Generator and Transducer input variables, output variables and state variables are listed in Table 5. The observation time for Transducer is set to 30 days.

Atomic Model Name	Variable type	Variable Name
Generator	State	phase
		sigma
		commandProduced
	Output	release
Transducer	Input	actualOut
		BoH
		release
	State	phase
		sigma
		total_ta
		SNEntities_act
		SNEntities_boh
		SNEntities_res
	Output	finish

Table 5: Input, Output and State Variables for Generator and Transducer

Although the Test model has slight difference from the Reference model in the structure, we make them behavioral equivalent by setting the initial values and inputs. So that easy for us to compare and validate the model. The initial inventories and processing time of the components in the two models for four experiments are shown in Table 6 and 7.

Components		State Variable	
Type	Name	Name	Initial Value
INVENTORY	AWI	Inventory (units)	24000
	ACI		20000
	SFGI		20000
	FG		20000
	ALL Inventory Components In Reference Model	Process Time (days)	1
PROCESS	Assembly Process	Process Time (days)	2
	Test Process		

Table 6: Impacted State Variables in Reference Model

Components		State Variable				
Type	Name	Name	Initial Value			
			Experiment 1	2	3	4
INVENTORY	AWI	Inventory (units)	12000	2500	12000	12000
	PCI		12000	21500	12000	12000
	ACI		20000	20000	20000	20000
	SFGI1		10000	10000	1000	10000
	SFGI2		10000	10000	19000	10000
	FG1		10000	10000	10000	1000
	FG2		10000	10000	10000	19000
	ALL Inventory Components In Test Model	Process Time (days)	1			
PROCESS	Assembly Process	Process Time (days)	2			
	Test Process					
	Finish1					
	Finish2					

Table 7: Impacted State Variables in Test Model

The release commands are generated by Generator, which are the inputs to the manufacturing process layer and sent to each inventory model through "ctrlIn" ports. The release command for the four experiments to base and test model are shown in Figure 19.

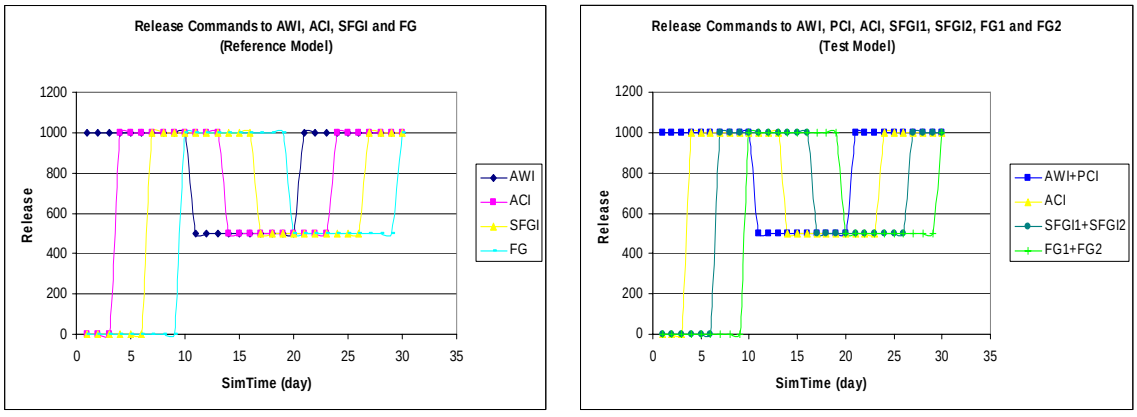

Figure 19: Release Commands to Reference and Test Models for Four Experiments

The Actual Out which is the material actually built and sent out by each process model, and the BoH message, which is the starting inventory at each simulation cycle and reported by each inventory model, are observed and recorded by the Transducer. As shown in Figure 20, the AO values of Assembly Process and Test Process are the same for the Reference and Test models. The sum of the Finish1 and Finish2 AO values is also the same as the one of Finish in Reference model.

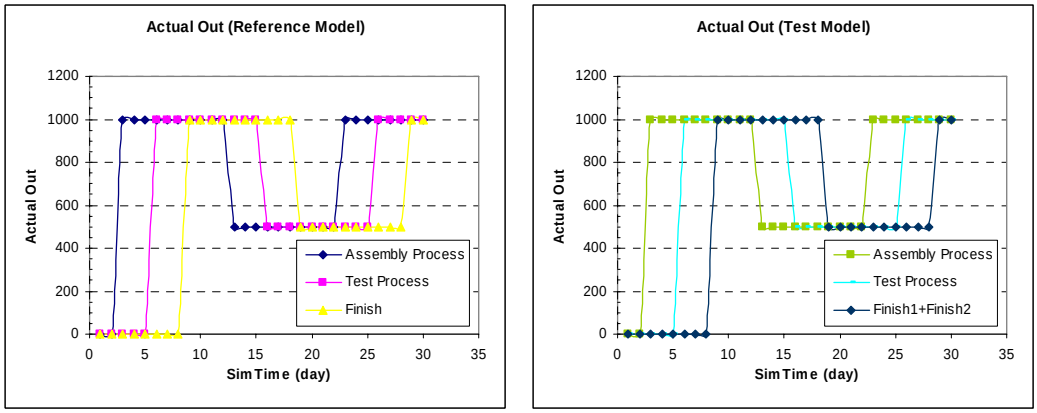

Figure 20: Actual Out

As shown in Figure 21, the sum of BoH of AWI and PCI in test model is the same as the BoH of AWI in the Reference model. The BoH for SFGI1, SFGI2 and FG1 and FG2 of the Test model are the same with respect to their counterparts in the Reference model.

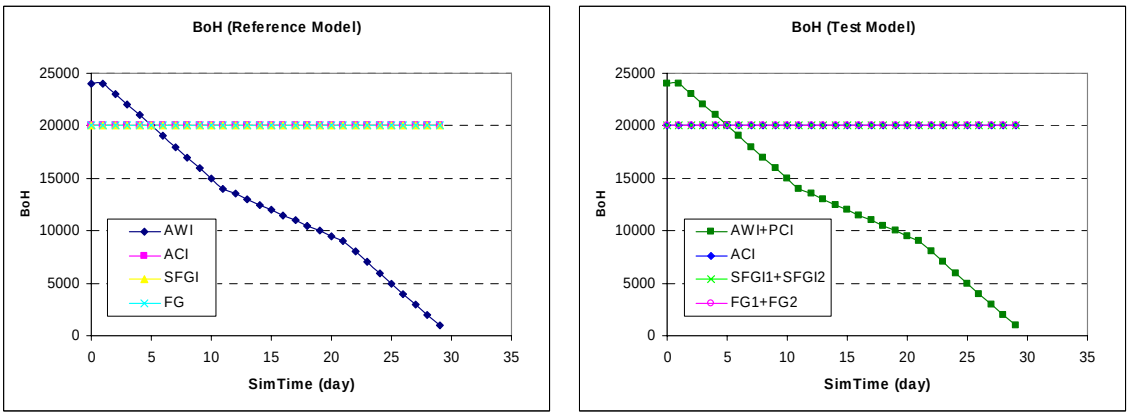

Figure 21: Beginning on Hand

3.2. Conclusions

In this project, we have implemented a set of DEVSJAVA template models which are useful for developing simulation models of semiconductor manufacturing processes. Simulation models of a representative physical semiconductor manufacturing processes are built from these template models. The results of the simulation executions demonstrate the implementation of the template models is correct. The availability of these model templates can help with the development of large-scale semiconductor supply-chain models. This work contributes to the ongoing research for composing semiconductor manufacturing process simulation models with decision models.

REFERENCES

[1] R. E. Nance and R.G. Sargent. Perspectives on the evolution of simulation. *Operations Research*, Vol. 50, No. 1, pp. 1593-1601.

[2] B. P. Zeigler. Theory of Modeling and Simulation. New York, 1976, Wiley.

[3] B. P. Zeigler, H. S. Sarjoughian. Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-based Simulation Models. Sept., 2003.

[4] ACIMS, DEVSJAVA Software, http://www.acims.arizona.edu/SOFTWARE/software.shtml, 2004.

[5] System-Modeling, ARENA. 2004, http://www.arenasimulation.com.

[6] G. Godding, H. S. Sarjoughian and K.G. Kempf. Semiconductor Supply Network Simulation. *In Proceedings of 2003 the Winter Simulation Conference*, New Orleans, LA, 2003, pp. 1593-1601.

[7] R. K. Singh, H. S. Sarjoughian and G. Godding. Design of Scalable Simulation Models for Semiconductor Manufacturing Processes. *In Proceedings of 2004 the Summer Simulation Conference*, San Jose, CA, 2004, pp. 1593-1601.

APPENDIX A

Acronyms and Definitions

Acronyms

ENGLISH

ACI	Assembled Components Inventory
AWI	Assembly Wafer Inventory
EOF	End of Factory Inventory
Fab	Fabrication
FG	Finished Goods Inventory
IPPI	Inventory ->Process/Process->Inventory
PCI	Package Components Inventory
PLNode	Pipeline type node
S	Shipping
SFGI	Semi-finished Goods Inventory

Definitions

1. Wafer inventory is the inventory to store raw silicon wafers.
2. Fab Process is the process where integrated circuits (die) are developed on silicon wafer through several hundred process steps.
3. Fab sort inventory is the inventory holding point to store the finished Fab product.
4. AWI is the inventory to store the assembly wafer.
5. PCI is the inventory to store package component.
6. Assembly process is the process where good die are separated and packaged for specific purpose (e.g. desktop or mobile package.).
7. ACI is the inventory to store the assembled components.
8. Test process is the process to test the maximum speed a die can be used for.
9. SFGI is the inventory to store semi-finished product.
10. Finish is the process where semi-finished product is fused to a specific speed (e.g. 2.5 GHz) and stamped.
11. FG is the inventory to store finished product which is ready to customer.

APPENDIX B

Sample Source Code

```
public class ForkNode extends SNNode{
    protected SNEntity startM;
    private Queue elementQ;
    private Queue branchQ;

    public ForkNode(String name, SNEntity startComponent ){
      super(name);
      this.startM = startComponent;
      elementQ = new Queue();
      branchQ = new Queue();
      add(startM);
      finalizeConfiguration();
    }

    public void addEdge(){
      addDataInportAndConnection();
      addDataOutportAndConnection();
      super.finalizeConfiguration();
    }

    public void addDataOutportAndConnection(){
      String portName;
      SNEntity destElementsFirst = (SNEntity)elementQ.get(0);
      add(destElementsFirst);
      //connect the "dataOut" outport of start Atomic Model to the "dataIn" outport of the
        first dest-element of ForkNode
      addCoupling(startM,DATA_OUTPUT_PORT,destElementsFirst,DATA_INPUT_PORT);
      addCoupling(destElementsFirst,DATA_OUTPUT_PORT,this,DATA_OUTPUT_PORT);

      //add additional Inports and links
      for (int i=0; i<elementQ.size()-1;i++){
        portName = DATA_OUTPUT_PORT + (i+2);
        super.addOutport(portName);
        SNEntity sn = (SNEntity)elementQ.get(i+1);
        add(sn);
        addCoupling(startM,portName,sn,DATA_INPUT_PORT);
        addCoupling(sn,DATA_OUTPUT_PORT,this,portName);
      }
    }

    public void addDataInportAndConnection(){
      List InportsList = (List)startM.getInportNames();
      addCoupling(this,DATA_INPUT_PORT,startM,DATA_INPUT_PORT);
      for (int i=0; i<InportsList.size();i++){
        String InportName = (String)InportsList.get(i);
        if (!InportName.equals(DATA_INPUT_PORT)){
          super.addInport(InportName);
          addCoupling(this,InportName,startM,InportName);
        }
      }
    }

    public void addDestElement(SNEntity sn){
      elementQ.add(sn);
    }

    public void addBranch(SNNode sn){
      branchQ.add(sn);
    }

    public Queue getBranchNodes(){
          return branchQ;
    }
...
}
```

```
public class SMP_EXP extends ViewableDigraph{
    …
    public SMP_EXP(){
       super("SMP_EXP");
       SNExpFrameLP ef = new SNExpFrameLP("EXP", 1, 30, 54);
       ViewableAtomic t = new TransducerLP("Transducer",30);

       //JoinNode
       Inventory awi  = new Inventory("AWI", 1);
       awi.initializeInventoryLevel(new Lot("Initial", "prod1", 12000));
       Inventory pci  = new Inventory("PCI", 1);
       pci.initializeInventoryLevel(new Lot("Initial", "prod1", 12000));
       SNProcess assemblyProcess = new SNProcess("AssemblyProcess", 2, 1000,1);
       JoinNode jn1 = new JoinNode("JN1", assemblyProcess);
       jn1.addStartElement(awi);
       jn1.addStartElement(pci);
       jn1.addEdge();

       //PipeLineNode
       Inventory aci  = new Inventory("ACI", 1);
       aci.initializeInventoryLevel(new Lot("Initial", "prod1", 20000));
       PLNode plCS = new PLNode("PLCS", 1, PLNode.INVENTORY);
       plCS.addElement(aci);
       plCS.graphConstruct();

       //ForkNode
       SNProcess2pt testProcess = new SNProcess2pt("TestProcess", 2, 1000,1);
       Inventory sfgi1  = new Inventory("SFGI1", 1);
       sfgi1.initializeInventoryLevel(new Lot("Initial", "prod1", 10000));
       Inventory sfgi2  = new Inventory("SFGI2", 1);
       sfgi2.initializeInventoryLevel(new Lot("Initial", "prod1", 10000));
       ForkNode fn1 = new ForkNode("FN1", testProcess);
       fn1.addDestElement(sfgi1);
       fn1.addDestElement(sfgi2);
       SNProcess finish1 = new SNProcess("Finish1", 2, 1000,1);
       Inventory fg1  = new Inventory("FG1", 1);
       fg1.initializeInventoryLevel(new Lot("Initial", "prod1", 10000));
       PLNode pL1 = new PLNode("PL1", 2, PLNode.PROCESS);
       pL1.addElement(finish1);
       pL1.addElement(fg1);
       pL1.graphConstruct();
       SNProcess finish2 = new SNProcess("Finish2", 2, 1000,1);
       Inventory fg2  = new Inventory("FG2", 1);
       fg2.initializeInventoryLevel(new Lot("Initial", "prod1", 10000));
       PLNode pL2 = new PLNode("PL2", 2, PLNode.PROCESS);
       pL2.addElement(finish2);
       pL2.addElement(fg2);
       pL2.graphConstruct();
       fn1.addBranch(pL1);
       fn1.addBranch(pL2);
       fn1.addEdge();

       ManufProcess manuPro = new ManufProcess("ManufacturingProc");
       manuPro.addJoinNODE(jn1);
       manuPro.addPipeLineNODE(plCS);
       manuPro.addForkNODE(fn1);
       manuPro.addEdge();
       this.add(manuPro);
       this.add(ef);
       this.add(t);
    …
  }
…
}
```