SPECIFICATION AND AUTOMATIC GENERATION OF SIMULATION MODELS WITH APPLICATIONS IN SEMICONDUCTOR MANUFACTURING

A Thesis
Presented to
The Academic Faculty

by

Ralph Mueller

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Industrial and Systems Engineering

Georgia Institute of Technology
August 2007

Copyright © 2007 by Ralph Mueller

SPECIFICATION AND AUTOMATIC GENERATION OF SIMULATION MODELS WITH APPLICATIONS IN SEMICONDUCTOR MANUFACTURING

Approved by:

Dr. Christos Alexopoulos, Chair
School of Industrial and Systems Engineering
Georgia Institute of Technology

Dr. Leon McGinnis, Co-Chair
School of Industrial and Systems Engineering
Georgia Institute of Technology

Dr. Magnus Egerstedt
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Richard Fujimoto
College of Computing
Georgia Institute of Technology

Dr. David Goldsman
School of Industrial and Systems Engineering
Georgia Institute of Technology

Date Approved: 30 April 2007

Für meine Familie

ACKNOWLEDGEMENTS

I am very grateful to my advisers Drs. Christos Alexopoulos and Leon McGinnis for their unconditional support, guidance and understanding, which helped me to overcome many challenges during the PhD process. Their patience and the freedom they gave me, allowed me to pursue my research interests and made this thesis possible. I would also like to thank them for providing funding for this research. Special thanks go to Dr. Alexopoulos for his relentless efforts providing feedback to improve this manuscript.

I would also like to express my gratitude to the rest of the committee members, Drs. Egerstedt, Fujimoto, and Goldsman for serving on the committee and their helpful suggestions. Many other people deserve recognition for helping me to complete this degree. Some helped me remain determined to finish, and some simply made the PhD process tolerable.

TABLE OF CONTENTS

DEDICATION . . . iii

ACKNOWLEDGEMENTS . . . iv

LIST OF TABLES . . . x

LIST OF FIGURES . . . xi

SUMMARY . . . xiv

I INTRODUCTION . . . 1

1.1 Problem Description and Research Objective . . . 2

1.2 Overview of the Thesis . . . 6

II BACKGROUND, MOTIVATION AND LITERATURE REVIEW . . . 7

2.1 Motivation: Grand Challenges in Manufacturing Simulation . . . 7

2.1.1 Reduction in Problem Solving Cycle Times . . . 7

2.1.2 Development of Real-Time Simulation-Based Problem Solving Capabilities . . . 8

2.2 Definitions: System, State, Model, and Simulation . . . 10

2.2.1 Domain Definition and Description . . . 11

2.3 Existing Approaches for Improving Modeling Productivity . . . 12

2.3.1 Simulation Model Reuse . . . 12

2.3.2 Composable Simulation Models . . . 14

2.4 Challenges in Software Engineering Relating to Simulation Modeling . . . 14

2.4.1 Time . . . 14

2.4.2 Correctness . . . 14

2.4.3 Complexity of Simulation Models . . . 15

2.5 Principles of Simulation Modeling . . . 15

2.5.1 Conceptual Modeling . . . 16

2.5.2 Declarative Modeling . . . 16

2.5.3 World Views in Discrete-Event Simulation . . . 19

2.5.4 Traditional Simulation Languages . . . 26

2.5.5 Simulation Development Paradigms . . . 27

2.6 Simulation Model Specifications and Modeling Frameworks 29
2.6.1 Discrete-Event System Specification (DEVS) 30
2.6.2 Activity Cycle Diagrams 32
2.6.3 Condition Specification 33
2.6.4 Simulation Graphs and Simulation Graph Models 34
2.6.5 Critique of Exiting Simulation Model Specifications and Frameworks 37
2.7 Automatic Model Generation 37
2.8 Conclusions 38
III FRAMEWORK FOR SEMICONDUCTOR MANUFACTURING MODELING, CONTROL AND SIMULATION 39
3.1 Introduction 39
3.2 Overview of Petri Nets 40
3.2.1 Classical Petri Nets 40
3.2.2 Inhibitor Arcs 44
3.2.3 State Equations 44
3.2.4 Relation of Petri Nets to Other Formal Models for Discrete-Event Systems 45
3.2.5 Behavioral Properties of Petri Nets 45
3.2.6 Structural Properties of Petri Nets 47
3.2.7 Classical Analytical Methods for Petri nets 47
3.2.8 High-Level Petri Nets 48
3.2.9 Representation of Time in Petri Nets 49
3.3 Object-Oriented Petri Net Simulation Framework 51
3.3.1 Relationship to Time Colored Petri Nets 51
3.3.2 Advantages of Petri-Net-Based Formulation 52
3.3.3 Overview of Object-Oriented Programming 53
3.3.4 Core Elements of the Proposed Framework 54
3.3.5 Execution Mechanism 60
3.3.6 World View of the Proposed Framework 73
3.3.7 Advantages of the Proposed Framework 73
3.3.8 Limitations of the Proposed Framework 74

IV SEMICONDUCTOR MANUFACTURING SIMULATION DATA SPECIFICATION . . . 75
4.1 Semiconductor Wafer Fabrication . . . 75
4.2 Sematech Data Set . . . 76
4.3 Simulation Data Specification . . . 76
4.3.1 Fabmodel . . . 77
4.3.2 Product . . . 77
4.3.3 Process Route . . . 77
4.3.4 Process Step . . . 77
4.3.5 Tool Set . . . 78
4.3.6 Operator Set . . . 78
4.3.7 Rework Sequence . . . 78
4.3.8 Representation of Control Policies . . . 79
V SIMULATION MODEL GENERATION . . . 81
5.1 Considerations for the Generation of Simulation Models Based on Petri Nets 81
5.2 Mapping of Fabmodel Elements to Petri Net Simulation Model . . . 82
5.2.1 Tool Sets . . . 82
5.2.2 Operator Sets . . . 82
5.2.3 Process Routes . . . 83
5.2.4 Process Steps . . . 83
5.2.5 Rework Sequence . . . 98
5.2.6 Scrap Modeling . . . 98
5.2.7 Modeling of Breakdowns . . . 99
5.2.8 Dispatch Rules and Representation of Queues . . . 100
5.3 Generation of the PN Simulation Model . . . 104
5.3.1 Generation of the Petri Net . . . 105
5.3.2 Polymorphism of Process Steps . . . 106
5.3.3 Example . . . 107
VI ANALYSIS OF GENERATED PETRI NET . . . 111
6.1 Validity of Classical Analysis Techniques . . . 111
6.2 Synthesis and Reduction Techniques for Petri nets . . . 112

6.2.1 Synthesis Techniques for Petri nets 112
6.2.2 Reduction Methods for Petri Nets 113
6.2.3 Relationship between Reduction and Synthesis Methods for Petri Nets 116
6.3 Application of Reduction Rules 117
6.3.1 Basic Process Step 117
6.3.2 Process Step with Setup 121
6.4 Mutual Exclusion 121
6.4.1 Simple Mutual Exclusion 123
6.4.2 Analysis of Simple Mutual Exclusions 123
6.4.3 Nested Mutual Exclusion 125
6.4.4 Analysis of Nested Mutual Exclusions 125
6.4.5 Mutual Exclusion with Resource Setup States 127
6.4.6 Analysis of Mutual Exclusions with Resource Setup States 128
6.5 Simple Batch Process Step 130
6.6 Analysis of Batch Process Step 131
6.7 Analysis of Synthesized Petri Net Simulation Model 138
6.7.1 Analysis of Serial Coupling of Process Step Modules 138
6.7.2 Analysis of Parallel Coupling of Process Step Modules 143
6.8 Analysis of Rework and Scrap Modeling 145
6.9 Legitimacy 147
6.10 Conclusions 148
VII COMPLEXITY ANALYSIS 149
7.1 Literature Review 149
7.2 Analysis of Execution Algorithms 150
7.2.1 Initialization Phase 150
7.2.2 Execution Phase 152
7.2.3 Memory Requirements 154
7.3 Measures of Complexity for the Framework 155
7.4 Experimental Results 157
7.5 Conclusions 159

VIII CONCLUSIONS AND FUTURE RESEARCH 160
8.1 Summary and Conclusions 160
8.2 Future Research 163
APPENDIX A SIMULATION DATA SPECIFICATION 164
APPENDIX B PETRI NET GENERATION ALGORITHMS 176
REFERENCES 195

LIST OF TABLES

1 Sematech Data Set Overview . . . 107
2 Size of PN Simulation Models . . . 110
3 Runtime for 100 Days of Simulation Time . . . 157
4 Runtime for 500 Days of Simulation Time . . . 157
5 Overview of Complexity Measures . . . 158
6 Description of Simulation Data Specification . . . 164

LIST OF FIGURES

1 Arena Example 3

2 Use of the Simulation Framework 5

3 A Deterministic Automaton 17

4 A Simple Event Graph 18

5 Event-Scheduling World View, Similar to [10] 20

6 Activity Scanning World View [10] 22

7 Process-Interaction World View [10] 24

8 Steps in a Simulation Study from Law and Kelton [24] 28

9 Sargents Circle [45] 29

10 Activity Cycle Diagram Elements 32

11 Activity Cycle Diagram for a Pub 33

12 Simulation Event Graph for a Single-Server Queue [56] 35

13 Comparison of Conventional Method with New Framework 41

14 Parallel Processes 44

15 Overview of Timed Petri Nets [51] 50

16 Class Diagram of Core Elements 55

17 Example 57

18 Subclasses of Place 58

19 Subclasses of Transition 59

20 Subclass of Token 60

21 Timing Example 62

22 Example for Updating Time 65

23 First FIFO Example 71

24 Second FIFO Example 71

25 Example with Fixed Assigned Priorities 72

26 Object Model of Manufacturing System Specification 80

27 Basic Process Step 84

28 Basic Process Step with In-Tool Travel Time 84

29 Basic Process Step with Partial Operator Processing 85

30 Basic Process Step with Operator Loading 85
31 Basic Process Step with Operator Loading and Unloading 85
32 Basic Process Step with Travel Time to Next Tool 86
33 Basic Process Step, Set Priority 86
34 Basic Process Step with all Options 87
35 Naive Batch Process Step 88
36 Batch Process Step 90
37 Batch Process Step with Two Process Routes 91
38 Batch Process Step with Loading 92
39 Batch Process Step with Loading and Unloading 93
40 Batch Process Step with Individual Wafers Modeled 94
41 Process Step with Setup 96
42 Process Step with Setup, Complete Example 97
43 Process Step with Setup and All Options except Travel 97
44 Single Step Rework Sequence 98
45 Scrapping of Lot 99
46 Breakdown Modeling 100
47 Breakdown Modeling 100
48 Queue 104
49 Overview of Simulation Model Generation 104
50 Polymorphism of a Process Step 106
51 Two Process Steps Joined 107
52 Simulation Model Data for Data Set 1 (abridged) 108
53 PN Simulation Model for Data Set 2 (abridged) 109
54 Fusion of Serial Places (FSP) 114
55 Fusion of Serial Transitions (FST) 114
56 Fusion of Parallel Places (FPP) 115
57 Fusion of Parallel Transitions (FPT) 115
58 Elimination of Self-Loop Places (ESP) 115
59 Elimination of Self-Loop Transitions (EST) 116
60 Reduction of Basic Process Step Module 117

61 Reduction of Basic Process Step Module with Tool Travel Time 118
62 Reduction of Basic Process Step Module with Partial Operator Processing . 118
63 Reduction of Basic Process Step with Operator Loading 119
64 Reduction of Basic Process Step Module with Op. Loading and Unloading . 119
65 Reduction of Basic Process Step Module with Travel Time 120
66 Reduction of Basic Process Step Module with All Options 120
67 Reduction of Setup Process Step Module 121
68 Reduction of Setup Process Step Module with All Possibilities 122
69 Simple Mutual Exclusion 123
70 Simple Mutual Exclusion Analysis 124
71 Nested Mutual Exclusion 125
72 Nested Mutual Exclusion Analysis 126
73 Mutual Exclusion with Resource Setup States 128
74 Analysis of Mutual Exclusion with Resource Setup States 129
75 Reachability/Coverability Graph for Batch Process Step (1) 133
76 Reachability/Coverability Graph for Batch Process Step (2) 134
77 Serial Coupling 139
78 Batch Process Steps in Series with Identical `batchId` 141
79 Batch Process Steps in Series with Different `batchId` 142
80 Parallel Coupling 144
81 Analysis of Rework Sequence 146
82 Analysis of Scrap Modeling 146
83 Neighborhood of a Firing Transition 152
84 Average Firing time vs. $\Psi(PN)$ 159

SUMMARY

The creation of large-scale simulation models is a difficult and time-consuming task. Yet simulation is one of the techniques most frequently used by practitioners in Operations Research and Industrial Engineering, as it is less limited by modeling assumptions than many analytical methods. The effective generation of simulation models is an important challenge. Due to the rapid increase in computing power, it is possible to simulate significantly larger systems than in the past. However, the verification and validation of these large-scale simulations is typically a very challenging task.

This thesis introduces a simulation framework that can generate a large variety of manufacturing simulation models. These models have to be described with a simulation data specification. This specification is then used to generate a simulation model which is described as a Petri net. This approach reduces the effort of model verification.

The proposed Petri net data structure has extensions for time and token priorities. Since it builds on existing theory for classical Petri nets, it is possible to make certain assertions about the behavior of the generated simulation model.

The elements of the proposed framework and the simulation execution mechanism are described in detail. Measures of complexity for simulation models that are built with the framework are also developed.

The applicability of the framework to real-world systems is demonstrated by means of a semiconductor manufacturing system simulation model.

CHAPTER I

INTRODUCTION

This research is concerned with the challenges of creating large-scale discrete-event (computer) simulation (DES) models for manufacturing systems. Simulation involves the imitation of the operation of a real-world process over time [11]. It can be used to analyze the behavior of a system and evaluate different scenarios, without having to change the actual system. It is also possible to analyze systems that do not yet exist. Discrete-event simulation is and has been a reliable technique that can be used for a wide range of problems. There are virtually no limits for the use of simulation. It is one of the methods most frequently used by industrial engineers and operations research analysts [24].

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. Although it is fairly easy for non-experts to create small-scale simulations using drag-and-drop features of standard simulation software, this convenience can become a problem for large models. There is no simple way to verify that a simulation model is executing correctly, except for going through the model step-by-step. This involves checking each graphical module for most applications, i.e., clicking on it and studying the parameters, and then checking the parameters of other modules that are referenced to the first one. This quickly becomes overwhelming for large models.

Despite these challenges, discrete-event simulation will remain an important tool as analytical models reach their limitations due to their underlying assumptions. There are simply no other tools available for analysis of large production systems, except for rough-cut calculations. In order for simulation to be useful, the modeler and user of a model have to be confident that it correctly reflects the behavior of the real system and executes

accordingly. Simulation modeling is largely considered an "art" and the development and implementation of an error-free simulation model is a difficult task [55]. Simulation still carries the label of an expensive and uncertain problem solving technique. The lack of a comprehensive modeling framework that can facilitate specification and implementation of discrete-event simulation models gives simulation a reputation of being overly complex.

1.1 Problem Description and Research Objective

The main goal of this research is to develop a comprehensive framework that can be used to generate simulation models for manufacturing systems and the associated supply chains in an effective and efficient way. The term *framework* implies a basic conceptual structure that can be used to create simulation models. This is similar to the use of this term in software engineering, where frameworks are used to build larger software modules from basic elements, and differs from the term *reference model*, which indicates a rather static, parameterized model.

Efficient simulation model generation will allow the user to simplify and accelerate the process of producing verifiable and credible simulation models. Two fundamental steps in the development of simulation models are verification and validation.

Verification is concerned with the "examination of the simulation program to ensure that the operational model accurately reflects the conceptual model" [11]. Even though an array of model verification techniques exist, there are currently no tools available that can perform the verification process automatically for a given simulation model. Verification usually involves an iterative process called "debugging," which is simply a trial-and-error approach.

Validation is the "determination that the conceptual model is an accurate representation of the real system" [11]. Since simulation modeling always involves an abstraction of the real-world system, the validation of a model is usually based on expert opinion. Since it is unlikely to have an automated procedure for validation, model validation is not part of the scope of this research.

Model credibility is attained when the end user accepts the simulation model as a correct

one. This does not necessarily mean that the model is valid. Improving the verification process can improve the model credibility, especially when the end user can retrace the verification process. The focus of this research is on verification. Questions that arise from this perspective are: “Do events execute in the right order (causality, time)? Will the simulation not end up in a deadlock?”

The proposed framework promotes a “bottom up” approach to simulation modeling. Small verifiable modules 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. This framework is not intended to model manufacturing systems where a deadlock situation can occur, as it would not make sense to build a large-scale simulation model of deadlocking subsystems.

The domain of the framework is discrete-event simulation of discrete-part manufacturing systems. The main purpose our development is to support the process of converting a given specification for a manufacturing system to an executable simulation model.

Figure 1 illustrates a common problem when building simulation models with the Arena™ simulation package [5]:

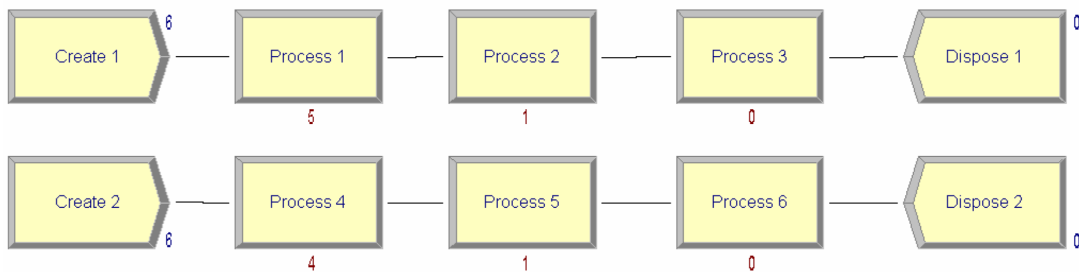

Figure 1: Arena Example

Two parallel process routes share two resources (R1, R2). The sequence for using resources for an entity following the first process route is:

1. Process 1: Seize R1, Delay Entity

2. Process 2: Seize R2, Delay Entity

3. Process 3: Release R1, Release R2

The sequence for entities going through the second process route is:

1. Process 4: Seize R2, Delay Entity
2. Process 5: Seize R1, Delay Entity
3. Process 6: Release R1, Release R2

For both process routes, entities are created with interarrival times following a statistical distribution with positive realizations. The capacity for each resource is one unit. The model is currently in a deadlock state. The digit below each `Process` module indicates the number of entities waiting for or receiving service, so technically the model can be executed, i.e., entities are released constantly into the system, but no more entities can reach the `Dispose` block. Eventually this will lead to a memory overflow. Arena's compiler will not indicate any errors in the model since there is no error in the model syntax. An experienced modeler knows the source of the problem: each process is waiting for the other to release a needed resource. The graphical Arena simulation model cannot represent this problem. The user has to go through each module/block and examine the parameters in order to identify the problem. Although the problem is obvious in this case, for larger models with hundreds of processing steps it is very difficult to identify similar problems.

Another approach for simulation is to use a generic high-level programming language or a specialized simulation language. For a custom programmed simulation model in such a language, a high level of trust in the programmer's ability is required. This is the most flexible approach and is preferable in many cases. However, a graphical representation of the simulation model is not usually available. The end user will only see the results of the simulation, but has no way to actually investigate the simulation model directly, making the establishment of credibility very difficult.

The aforementioned issues motivate the following research objective: "Given a manufacturing system specification, generate an appropriate simulation model of that manufacturing system without direct human interference." The term *manufacturing system specification*

means a description of a manufacturing system that contains all necessary resources and processes to produce all the products that are manufactured by this system. In Chapter 4 the requirements of such a specification will be described in more detail.

The simulation model is built in a the bottom up fashion with an automated procedure. The model is composed of elements that are put together in a way that important behavioral properties are maintained. The system specification completely determines the characteristics of the simulation model. This also connotes that verification of the final simulation model is reduced drastically or is no longer necessary. Of course, this requires a correct implementation of the procedure generating the simulation model. However, this needs to be done only once for each type of manufacturing system, and then this procedure can be used to generate a wide variety of simulation models. The advantage is that this avoids programming errors in the simulation model. Figure 2 explains the concept:

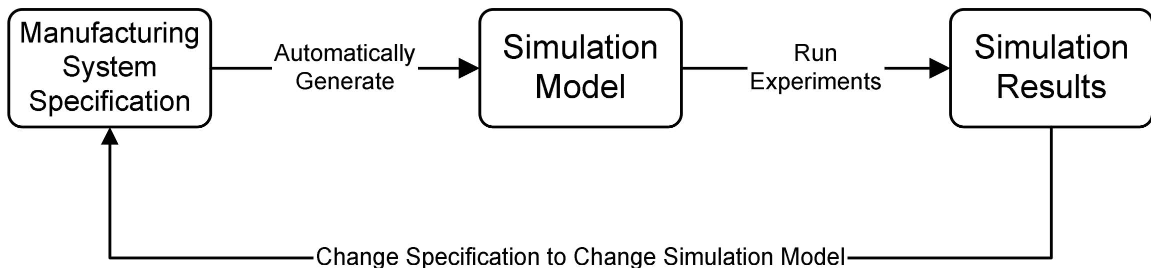

Figure 2: Use of the Simulation Framework

The use of the simulation framework is as follows: Based on a given manufacturing system specification, the simulation model is created. The specification is stored in a specific file format, in this case XML. The generation of the simulation is an automated process and occurs without interference by the user. Then the generated simulation model is used to run experiments. In order to create different scenarios, the user can change the manufacturing system specification and automatically create a new simulation model. The experimental frame is not part of the simulation model and will not be further addressed here. The manufacturing system specification has to be done properly, i.e., it has to follow a certain

format that will be introduced in Chapter 4.

The simulation framework has to be capable of representing the state and flow of all relevant informational and physical entities through a manufacturing system on a detailed operational level. The system refers in this context to the inventory status as well as the status of jobs.

The simulation model is based on an object-oriented Petri net data structure. With this structure certain properties can be enforced, e.g., absence of deadlock and reachability of the final state. As discussed above, this is especially important for large-scale simulation models, where a manual verification is either impossible or is extremely time-consuming.

1.2 Overview of the Thesis

Chapter 2 elaborates on the grand challenges of simulation in manufacturing and supply chain systems. Hence it serves as a motivation for the development of a simulation framework. Further, the principles of simulation modeling are discussed in detail and existing simulation modeling frameworks are discussed. Chapter 3 introduces the elements of the proposed framework and the execution of a simulation model. It also discusses the fundamentals of Petri nets upon which the framework is built. Chapter 4 describes the object model, which is used for the manufacturing data description. Chapter 5 presents procedures that can generate simulation models based on a given specification and illustrates them with a real-world example. Chapter 6 analyzes the properties of the generated PN simulation model. Chapter 7 discusses measures of complexity for the proposed framework. This allows the modeler to estimate beforehand the amount of computational effort for running a simulation. Chapter 8 presents conclusions and directions for future research.

CHAPTER II

BACKGROUND, MOTIVATION AND LITERATURE REVIEW

2.1 Motivation: Grand Challenges in Manufacturing Simulation

In [18] Fowler discusses the grand challenges in modeling and simulation of complex manufacturing systems. A grand challenge is a problem that is difficult, with the solution requiring one or more orders-of-magnitude improvement in capability along one or more dimensions; further, the problem should not be provably insolvable. A solution to a grand challenge will have a significant economical and/or social impact; among others, the two following grand challenges in manufacturing simulation are stated:

- An order of magnitude reduction in problem solving cycles
- Development of real-time simulation-based problem solving capability

2.1.1 Reduction in Problem Solving Cycle Times

The biggest challenge in simulation modeling today is to reduce the time it takes to design, collect data, build, execute, and analyze simulation models to support decision making. A reduction of this time would lead to more analysis cycles than currently is possible. Areas for improvement for the simulation process for manufacturing systems analysis are:

- Model design
- Model development
- Model deployment

The primary goal in model design is to determine how much detail should be added to the model. Discrete-event simulation models can be arbitrarily accurate, but they can take a long time to build. In addition, the execution of discrete-event simulation models can be slow. High-level continuous simulation models for a supply chain can be built more

quickly, since they use less detail and execution is usually faster than discrete-event models. However, since these models are less detailed, the level of accuracy will be lower.

The first step in model development is choosing a modeling approach. This could be one of the three main simulation world views, such as the event-scheduling approach, process-interaction, or activity scanning. The next step involves building the actual simulation model. The earliest simulation models were built using assembly code or programming languages such as FORTRAN. These approaches allowed for efficient execution but little reusability of the simulation model. In later years, simulation languages such as GPSS, Simscript, GASP, SLAM, or SIMAN were introduced; those included many elements that support the simulation process, such as statistical routines and random number generation [18].

Another way to build a simulation model is to use a simulator, where the simulation model is already coded; the user only supplies the data for the model. AutoSched AP™ and Factory Explorer™ are examples that use this approach [6, 7].

These languages and packages described above have reduced significantly the time and effort to build simulation models, yet there is still considerable room for improvement. Another way to reduce the time to build and verify a simulation model is to generate the model automatically. This would significantly reduce the time to debug the code and can even make debugging unnecessary.

The execution of a simulation model is another area requiring improvement. Complex simulation models can run for a prohibitively long time, especially when detailed material handling is modeled; hence decreasing the run time of a simulation model helps to increase the number of problem solving cycles.

2.1.2 Development of Real-Time Simulation-Based Problem Solving Capabilities

Currently most simulation models are used in single projects to support decisions with long-term horizons, e.g., equipment purchases. Less is known about how to use simulation for real-time decisions in manufacturing or material handling. Real-time simulation-based problem solving offers new opportunities to evaluate abrupt changes in the status of a

manufacturing system. If the state of a manufacturing system changes abruptly (e.g., due to equipment failure), a simulation run could be used to support decisions. In order to use simulation for real-time problem solving, it is necessary that the time to built simulation models and the time to collect the relevant data are very short. In addition, the runtime of the model has to be sufficiently short.

There are two different ways discussed in the literature for implementing such systems [18]:

- Use of a simulation model that is permanently running synchronized to the factory
- Automated building of a model from the factory databases

Permanent, always-on, synchronized factory models would be continuously updated and synchronized with factory data. This requires that the factory state is clearly defined and the data exist. Currently, in most cases the relevant data are not available or the quality of the data is not good enough. This persistent, constantly synchronized simulation model is then the master copy for a clone. A clone of a simulation model is simply an exact copy of the original simulation model. This clone can then be used to start a new simulation run.

Another approach is to automatically build factory models on demand. The experimenter generates the model directly from the factory databases. The required data are retrieved from the databases and transformed into a simulation model. This approach is probably slower than cloning but offers more flexibility.

Technological advances are also supporting the trend of real-time simulation based decision-making: IP networks are now found everywhere, which makes real-time information available at very low cost. Manufacturing Execution System (MES), Enterprise Resource Planning Systems (ERP), and Supply Chain Management Systems (SCM) are in use at many companies nowadays. This means that a basic information structure is usually already in place that can be used to access data. Since computing power is constantly increasing (e.g., due to Moore's Law), the execution of large simulation models will become more economical and faster over time.

The proposed framework has capabilities that support both of theses approaches, that is, generation of simulation models on demand as well as synchronization with the factory floor.

2.2 Definitions: System, State, Model, and Simulation

A *system* is a collection of entities that act and interact to accomplish some logical goal [24]. This term can mean many different things depending on the goal of a study. The *state* of a system is a collection of variables necessary to describe it at a particular time. Discrete systems have state variables that change instantaneously at certain points in time, while continuous systems have state variables that change continuously with respect to time. A *model* is a representation of the system under scrutiny. It is a surrogate for the actual system. A model can be a *physical model* (i.e., a scaled down model of a real system) or a *mathematical model* (i.e., a model that is described with logical and quantitative relationships). Here the focus is on mathematical models, as this is the typical domain of operations research. A modeler has to decide which elements are to be included into the model. This decision is mainly determined by the purpose of the model. Also, the system boundaries have to be defined clearly. Ideally, once a mathematical model is formulated, one would like to be able to use analytical methods to answer questions of interest. Since, usually, there are no analytical solutions available for many real-world problems, simulation remains the only viable option. *Simulation* can be described as "the imitation of a system over time" [11]. An artificial history of the system is generated; from this history conclusions concerning the operational characteristics are drawn. Simulation can be used to analyze the behavior and address "what if" questions about real systems as well as conceptual systems. The following types of simulation can be distinguished:

- Discrete simulation models
  - Discrete-time models
  - Discrete-event models
- Continuous simulation models

Continuous simulation models describe continuous systems with state variables that change continuously over time. These changes are usually expressed via differential equations. *Discrete* simulations involve discrete-time models or discrete-event models. Discrete-time models have state changes in fixed time intervals, whereas discrete-event models can have state changes at any time. The focus here will be on discrete-event models.

2.2.1 Domain Definition and Description

The proposed simulation framework is intended for use in the domain of discrete part manufacturing. This is a very broad domain containing most manufacturing systems. The domain of discrete manufacturing systems consists of systems in which materials flowing through the system are countable objects, as opposed to process manufacturing where a continuous stream such as a fluid is going through a number of processing steps. The following classes of manufacturing systems can be distinguished based on their process structure [20]:

- Job shops
- Disconnected flow lines
- Connected flow lines
- Continuous flow processes

These classes are ordered according to product variety and production volume that they are capable of handling. Job shops generally have the highest possible range of product variety, whereas continuous-flow processes systems are the least flexible and usually will be capable of producing only one type of product. Job shops have flexible routings and use manufacturing equipment that performs many different tasks. Production is in small lot sizes or even lot sizes of one. Disconnected flow lines have a limited number of product routings and production in small to medium lot sizes. Connected flow lines, such as assembly lines, have rigid routings and are used for high-volume production. Continuous flow lines are usually specialized production systems such as refineries, where product moves along a fixed routing and is processed automatically.

It has been estimated that more than 75% of manufacturing occurs in batches of 50 units or less [9]. This means that most of manufacturing occurs in job shops or disconnected flow lines, as these systems are able to handle low volumes with great product variety. The proposed framework also targets these types of manufacturing systems.

2.3 Existing Approaches for Improving Modeling Productivity

Different approaches for improving the modeling productivity exist, among them are simulation model reuse and composable simulation models.

2.3.1 Simulation Model Reuse

In order to avoid the costly development cycle of simulation models, researchers proposed the idea of simulation model reuse [35]. The idea of model reuse is to avoid the cost of model specification, simulation model coding, verification, and validation. With a drastically reduced development time, it is possible to construct, nearly instantaneously, new simulations. The improved quality of (reused) simulation models is based on trusted and efficient components that were previously developed. Simulation model reuse can be accomplished by:

- Reuse of basic modeling components (approach used by COTS)
- Reuse of subsystem models
- Reuse of a similar model (adaptation of previously developed model)

Despite the promising advantages of reusing simulation models, this approach is difficult in practice and it has been the focus of much research in the simulation community. It is also one of the grand challenges in simulation. An area in simulation where reuse of components is very common is the infrastructure to support model execution and development. These include (among others) statistical routines, graphical generation tools, and random number generators. Some of the key issues of simulation model reuse are given in [35]:

- Determining how to locate potentially reusable components

- Recognizing objective incompatibilities among model components
- Recognizing assumption incompatibilities among model components
- Building components that enhance reuse
- Determining the level of granularity of each reusable component
- Capturing the objectives and constraints of each component
- Representing the objectives, assumptions, and constraints
- Specifying the level of fidelity of each component, i.e., the level of accuracy of the component
- Determining the modifiability of a reusable component
- Determining the interoperability of the reused components
- Determining if constraints (such as execution speed) will be satisfied with the selected objects
- When a new simulation is constructed entirely from verified and validated components, what can be said about the newly composed simulation?

If the components are specified at a very low level, their reuse requires much the same effort as coding from scratch. If the components are high-level aggregates, then their reuse may be limited by their predefined behavior. Although model reuse has been a goal in simulation modeling for a long time, it has not been used effectively, except for infrastructure components, e.g., random number and random variate generators. Simulation model reuse must consider original and new objectives; valid reuse requires consistency between the two sets of objectives. All these issues make a general automated solution to the reuse problem unlikely. Thus, model reuse for many problems will probably stay an unreachable goal for the future.

2.3.2 Composable Simulation Models

A closely related approach for building simulating models is to use (standard) components and create a simulation model based on these predefined components. Page and Opper [39] showed that deciding whether a set of objectives $O = \{o_1, o_2, \ldots, o_n\}$ can be satisfied by a (sub)set of components $C = \{c_1, c_2, \ldots, c_m\}$ is most likely a NP-complete problem. They showed this for a specific (sub-)problem, but it is not clear if it is applicable to the general problem of composition, since the mechanism of composition is not precisely defined. The notion of satisfying a set of objectives is difficult to interpret, as it is possible that two components could individually satisfy a certain subset of the objectives but not when used in a composition. Although Page and Opper use an abstract point of view for the modeling objective, they indicate that it is may be unlikely to find a satisfactory general solution for composable simulation modeling.

2.4 Challenges in Software Engineering Relating to Simulation Modeling

Since every simulation study can also be seen as a software engineering project, many of the challenges in software engineering are also applicable to simulation modeling. For example, some of the goals in software engineering, such as reliability and maintainability, are certainly important for simulation as well. However there are certain aspects of simulation that are unique.

2.4.1 Time

The notion of time is one of the central characteristics of simulation. Time clearly establishes an order in the processing of events and is also the limiting factor for parallel or distributed execution of a simulation model. This characteristic is responsible for a large body of research in the area of parallel and distributed simulation.

2.4.2 Correctness

Another characteristic, as already noted, is the very constrained interpretation of correctness as a project objective. Correctness assumes a high priority in simulation projects and refers

to the verification and validation issues. A simulation model is not very useful if there are doubts about its validity or its verifiability.

Verification tries to establish that the relationship between the simulator and the underlying model holds [57]. In other words, verification tries to determine if the computer implementation of the conceptual model is correct, i.e., if the computer code represents the model that has been formulated. Therefore, the verification process is often described as the "debugging" of the simulation code. There are two general approaches to verification:

- Formal proofs of correctness
- Testing

Validation is the process of determining if the conceptual model is a reasonable representation of the real-world system. Therefore, it often relies on the opinion of experts, who can use statistical tests and other methods to establish validity.

2.4.3 Complexity of Simulation Models

Another important issue is the computational complexity of the simulation model. While model development time and cost are considerable in simulation, the necessity for repetitive sample generation for statistical analysis and the testing of numerous alternatives requires that the simulation model is executable in an efficient way. A more detailed discussion is presented in Chapter 7.

2.5 Principles of Simulation Modeling

There are no established principles of modeling. Simulation modeling is considered an "art and a creative activity" [11]. This is a very unsatisfactory position as it perpetuates the notion that simulation is very hard and always requires highly skilled personal. According to Webster's dictionary [3] a model is "a schematic description of a system, theory, or phenomenon that accounts for its known or inferred properties and may be used for further study of its characteristics." A model is an abstraction of a system. A modeler has to decide which elements are to be included in the model. This decision is mainly determined by the purpose for modeling.

The common paradigm in simulation modeling is to use a minimalist approach. This means that only the necessary details for the planned simulation study will be considered. In other words, the simplest, minimal model that can fulfill the requirements is preferred. Thus, models should be just barely good enough to meet objectives. This general scientific principle dates back to the 14th-century English logician and Franciscan friar, William of Ockham (Ockham's razor) who stated that "entities should not be multiplied without necessity" or "it is vain to do by more what can be done by fewer" [12]. Advantages of this approach include ease of understanding, quick implementation, run-time efficiency, and a preference for simple and elegant models. However, this is a myopic view of the modeling problem. Often it is not clear what the model will be used for in the future, and it can be difficult to implement capabilities that were not anticipated at the onset of the modeling effort. This view of modeling comes from a time when computing power was expensive, as it allows for simulation models with less coding effort and better execution performance. A focus on model minimalism also makes the reuse of simulation models more difficult.

2.5.1 Conceptual Modeling

A conceptual model is a description of the target system in natural language and/or a pictorial description of the system; often the term "model assumptions" is used [24]. Since there are no formal methods for creating a conceptual model, the main problem associated with using such a model is that many ambiguities will be present in the model. Two different programmers will most likely create two different simulation models based on the same conceptual model. Nevertheless, this approach is usually taught in a simulation curriculum, as it can be used for any system.

2.5.2 Declarative Modeling

The two primary components of declarative models are states and events [17]. The dynamic behavior of the system under investigation is represented as a sequence of changes in states, or state transitions.

2.5.2.1 State-Based Models

A *Deterministic Automaton* is a six-tuple $G = \{X, E, f, \Gamma, x_0, X_m\}$, where X is the set of states, E is a finite set of events associated with the transition of states in X, f is the transition function, Γ is the active event function, x_0 is the initial state, and $X_m \in X$ is the set of marked states [14]. By designating certain states as marked, the modeler can indicate that these are desired states, for example, they represent the completion of certain tasks. This object is also known as state machine or generator. If X is a finite set, the object is called a finite-state automaton or finite-state machine.

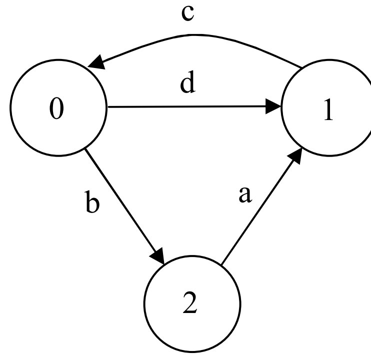

Figure 3: A Deterministic Automaton

Figure 3 depicts a simple finite-state machine with three states $\{0, 1, 2\}$ and four events $\{a, b, c, d\}$. Each arc is associated with a certain event and indicates the next state the system will assume after the occurrence of the event.

A *Nondeterministic Automaton* is defined in a similar way as the deterministic version. However, the transition function is replaced with $f_{\text{nd}} : X \times E \cup \{\varepsilon\} \rightarrow 2^X$ and the initial state may be a set of states, i.e., the transition function has a different domain $X \times E \cup \{\varepsilon\}$ and co-domain 2^X, where 2^X is the set of all subsets of X (power set) and ε is the empty string, which denotes that no event occurred. This means that an event can trigger the transition to a set of possible new states as opposed to one specific state. It can be shown that a nondeterministic automaton can always be transformed to a deterministic automaton [14].

There is a broad theory on automata, especially in combination with the theory of formal languages, yet the use of automata for large-scale simulation is very limited, as all states have to be expressed explicitly. For large simulation models, the number of distinct states can be extremely large, making the representation and manipulation of such models overwhelming. Another drawback of these models is that they do not have a concept for time, although extensions for time do exit (timed automata [14]).

2.5.2.2 Event-Based Models

Finite-event automata or event graphs can be interpreted as dual to deterministic automata [48]. Nodes in the graph represent events and transitions or arcs describe the change in the state variables. A formal method of transforming a finite-state automaton to a finite-event automaton is presented by Fishwick [17]. Relationships between events are represented as arcs in the event graph. Each arc is associated with a set of logical expressions that express state changes and time.

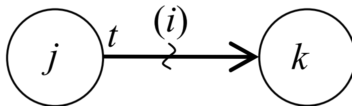

Figure 4: A Simple Event Graph

Figure 4 shows a simple event graph. The interpretation is as follows: after the occurrence of event j, event k will be scheduled after t time units if condition i is true. The advantage of event graphs is that there is a close relationship with the event-scheduling simulation world view. Each node corresponds to a routine in the simulation program. It is also not necessary to explicitly model all states, only the changes in states, e.g., increase of counters, etc. A disadvantage of this approach is that it is difficult to use with large simulation models involving many different events, as the graphs will become very large. In addition, entities cannot be modeled explicitly. They can only be modeled indirectly via variables. This creates another layer of abstraction that can be hard to understand, as it is not possible to directly follow an entity through a system but only a sequence of events in the event graph.

2.5.2.3 Hybrid Models

Petri nets can be seen as hybrids of state-based and event-based models, as they represent events in the form of transitions and state in the form of markings of places. A more detailed discussion about Petri nets will follow in Chapter 3. The aforementioned declarative models are usually not applied to manufacturing simulations, as they are too cumbersome to handle real-world models.

2.5.3 World Views in Discrete-Event Simulation

An important concept in simulation modeling is the notion of world views (conceptual framework, simulation scheme) [16, 37]. Over the last three decades, a set of conceptual frameworks for implementing discrete-event simulations have been established. The idea of a world view is traced back to the early days of discrete-event simulation and emerged from the development of simulation programming languages and simulators. The three most common simulation paradigms or world views employed in discrete-event simulation languages and packages are frequently categorized as [11, 34, 57]:

- Event-scheduling world view
- Activity scanning world view
- Process-interaction / process-oriented world view

Event-Scheduling (Figure 5) is the most basic approach for discrete-event simulation. Events with timestamps are put in an event list and are processed in the order of the timestamps. After an event is processed, it can generate new events, which are also inserted in the future event list (FEL). New events will have timestamps equal or larger than the current simulation clock. When the event list becomes empty or a certain event occurs, the simulation will end.

The event-scheduling world view can be implemented in any programming language and is therefore useable for many purposes. Events are executed at discrete points in time, which causes the change of state variables. No time elapses when an event executes. Therefore,

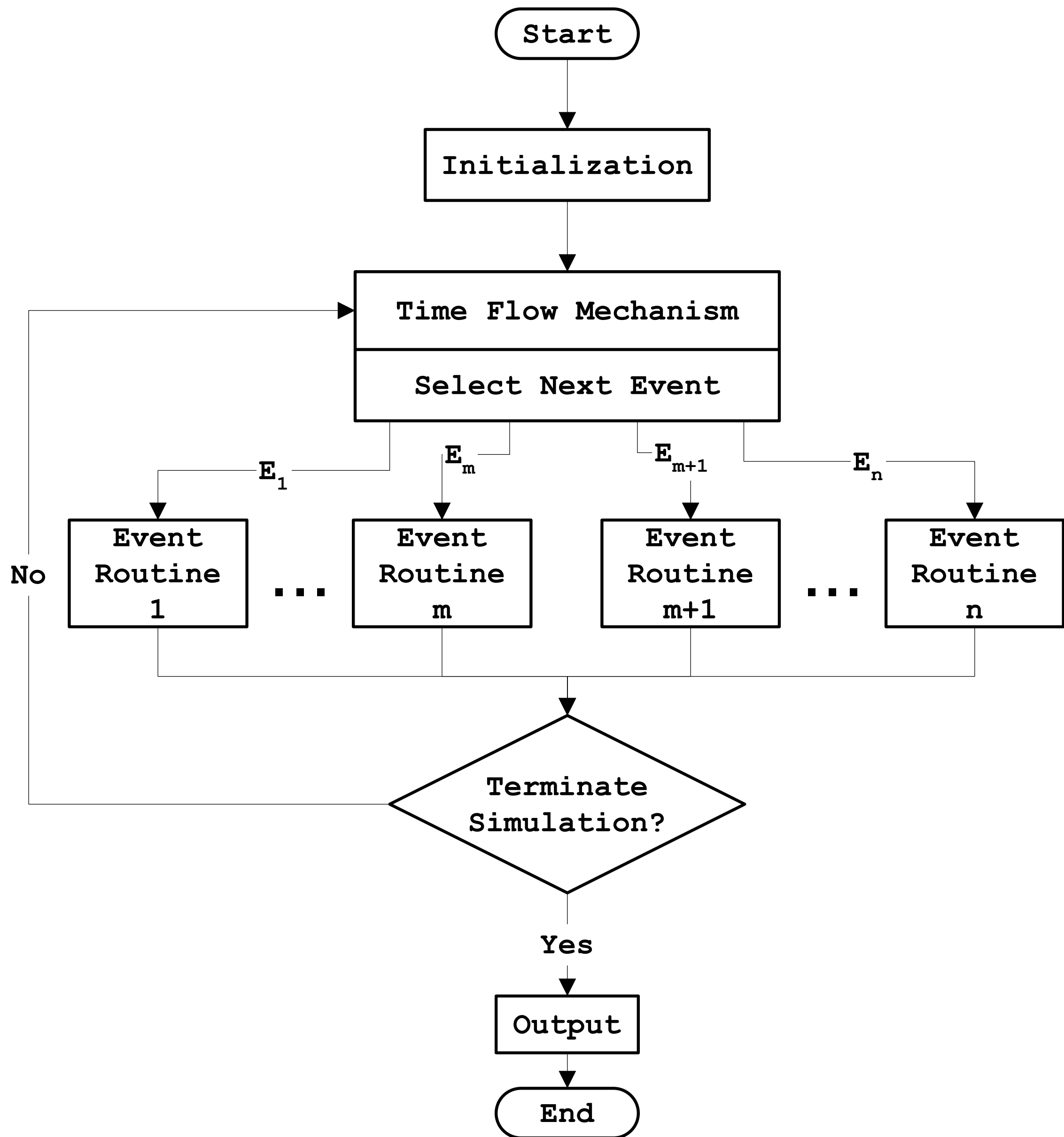

Figure 5: Event-Scheduling World View, Similar to [10]

if two or more events can occur at the exact same time, precedence has to be preassigned to ensure a well-defined transition function. Under this approach, the modeler first has to identify all events and their effect on system state [13]. The execution of an event can also trigger the generation of a new event based on the current system state. This event can be scheduled to occur at the same instance in time as a direct consequence of the triggering event or a future point in time. After all events that are scheduled for one specific execution time are processed, the simulation time can be advanced to the next scheduled event. With this simulation world view fast executing simulations can be realized.

Activity Scanning (Figure 6) is also known as the two-phase, state-based approach [10]. The modeler describes an activity in two parts. First, the condition that causes an action is specified as a logical expression, and then the action that will be performed if this condition is true is described. Simulations that follow this approach often use a timing mechanism with fixed time increments. Testing the conditions and performing the corresponding actions represent a single iteration of the execution algorithm. It may not be sufficient to perform only one scan, as some actions may lead to additional conditions. Hence, all conditions must be scanned repeatedly, until no condition is satisfied at the current simulation time. At this point in time, the simulation clock can be advanced. Activities need to be prioritized for the order of condition testing. With the activity scanning approach modular simulation programs can be easily implemented that consist of independent modules waiting to be executed.

Due to the repeated scanning and fixed time increments, the execution of simulation programs of this type is usually quite slow [10]. Another problem is that the fixed time increments can be a source of error, since the time resolution is limited to the size of the increments.

The *Process-Interaction* world view (Figure 7) emulates the flow of an entity through a system. An entity moves as far as possible until it is delayed, starts an activity, or leaves the system. Hence, the simulation model describes the sequence of all states that an entity can attain in the system. An object or entity can be classified into two types: dynamic or static [10]. A dynamic entity enters the model, logically moves through some processes, and

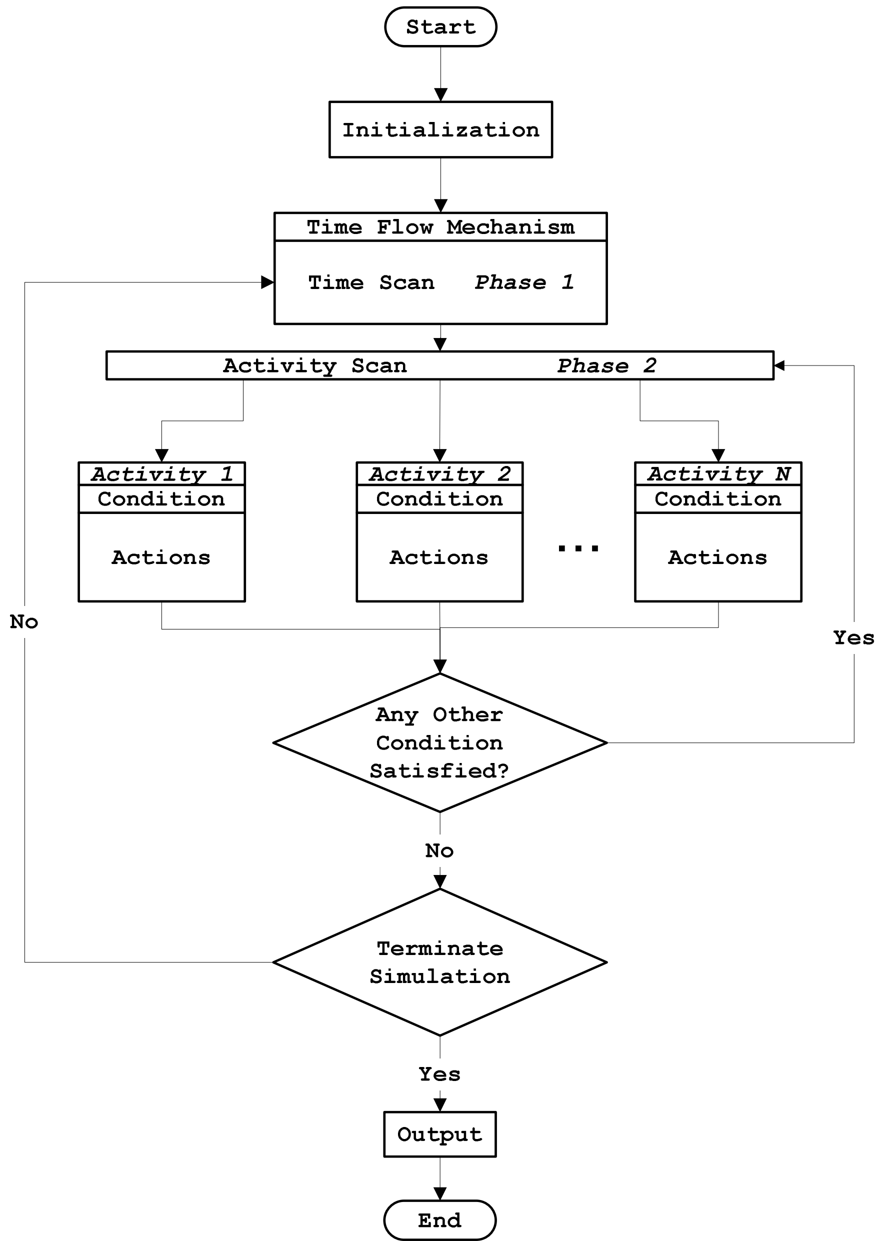

Figure 6: Activity Scanning World View [10]

leaves the model. A process is a time-ordered sequence of events, activities, and delays that describe the flow of a dynamic entity through a system. A static entity, such as a resource, does not move on its own. Under this approach simulation is conducted by going back and fourth between a clock update phase and scan phase. During the clock update phase, time is advanced to the first object in the future object list (FOL). All objects with move-times equal to the current simulation time are transferred to the current object list (COL). During the scan phase, all objects on the COL are moved one-by-one through as many processes as possible. The movement of an object can be interrupted by an unsatisfied condition, a deliberate delay, departure from the system, or stopping for some reason, such as waiting in a queue. Since the movement of objects can cause changes in state variables, the COL has to be repeatedly scanned until no more objects can be moved. Then the simulation can continue with the clock update phase.

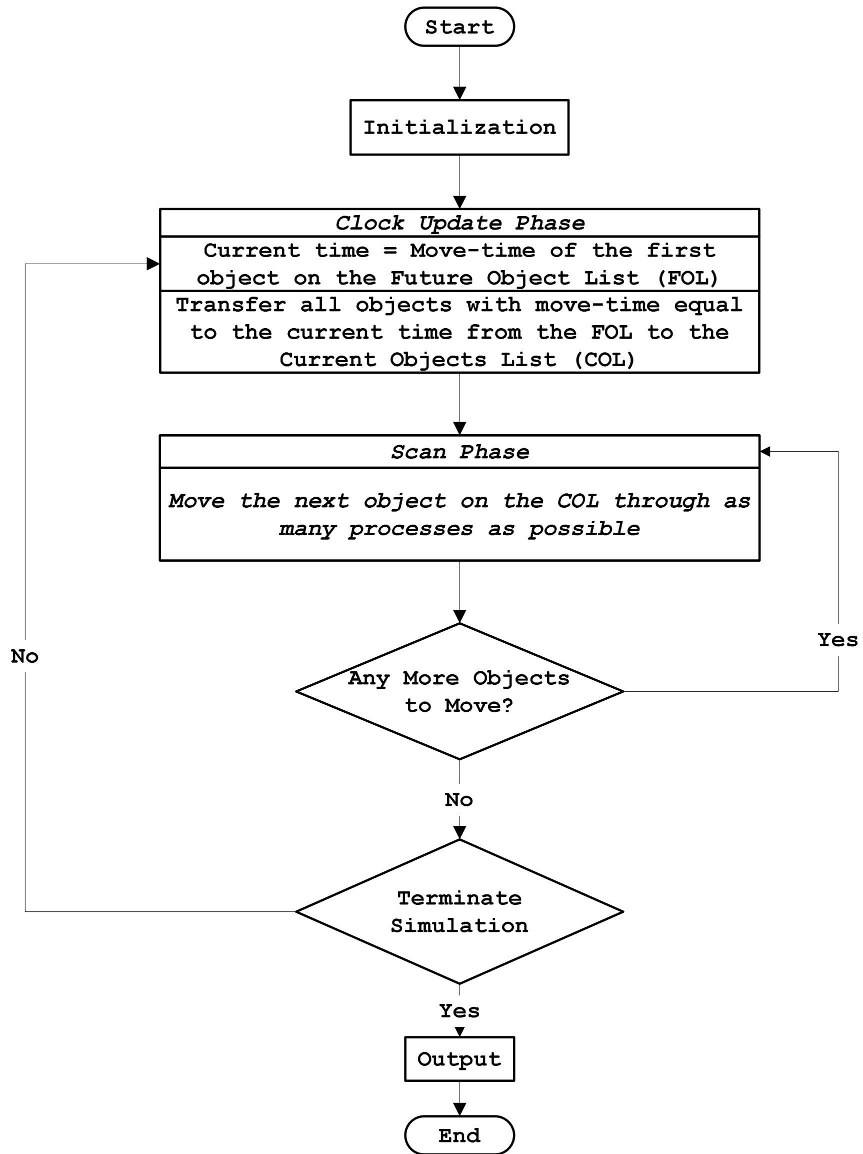

Figure 7: Process-Interaction World View [10]

2.5.3.1 *Critique of World Views*

The three simulation world views mentioned above are the most cited in the literature. There is not a single rigorous description of the world views; they somewhat differ from author to author. There are usually inconsistencies when world views are described. For example, Law and Kelton [24] point out that the process-interaction approach is actually executed as an event-scheduling approach. On the contrary, Zeigler et al. [57] describe the process-interaction approach as a combination of the event-scheduling and activity scanning approaches. Further, they state that the event-scheduling approach does not allow for conditional events that can be activated based on the global state. This distinction is usually not made by other authors. The process-interaction world view is implemented in various different ways, which leads to different behavior how resources are claimed and released by competing entities [47].

There are also other simulation world views discussed in the literature, e.g., the three-phase approach [11]. Since the descriptions of the different simulation world views are rather informal in the literature, there is a need for a more formal approach to simulation modeling.

Many researchers suggest that there is a choice between these world views. However, these world views are not directly comparable as they describe system behavior at different levels. They merely describe the approach for implementing a simulation program. In addition, they do not have a strict formal description, with event-scheduling being an exception, since it can be directly mapped to event graphs.

The world view that is fundamental to all world views is event-scheduling, which describes each type of event with regard to subsequent events and state changes. The process-interaction world view on the other hand is describing how entities travel through the system and the resources they require, but the underlying execution mechanism is still based on the event-scheduling approach.

The activity scanning method is somewhat related to the event-scheduling method, but it uses fixed time increments, which can introduce errors. Further this method is not suitable for manufacturing simulations. However, it could be convenient for inventory systems with

periodic review.

2.5.4 Traditional Simulation Languages

Traditional simulation programming languages are tied to the above mentioned simulation world view. Most simulation languages, such as GPSS, SIMULA, SIMAN or SLX [8], adopt the process-interaction world view [24]. Most of these languages have similar characteristics, and their main components or objects are [16]:

- Entities: Objects requiring services (e.g., parts, jobs, or customers)
- Attributes: Information characterizing a particular entity
- Process Functions: Instantaneous or time delays experienced by entities
- Resources: Objects providing services (i.e., performing process functions)
- Queues: Sets of entities (e.g., entities waiting for service)

Almost all COTS packages with a graphical user interface employ the process-interaction world view. The graphical modules and connectors between them show the logical flow of entities through the system. Underneath the graphical user interface, a simulation language is used, such as SIMAN for Arena.

SIMULA was the first simulation language to employ object-oriented concepts for programming. Several languages followed (e.g., SIMULA 67) that served as drivers in the development of modern object-oriented programming languages.

Simulation languages usually have special list processing capabilities that are needed for managing the FEL. The most common task during a simulation is inserting events in to the FEL, removing events with the smallest timestamps while keeping the FEL in proper order at all times. Overall, simulation languages must provide [34]:

- Generation of random variates
- List processing capability, so that entities and events can be manipulated, created and deleted

- Statistical analysis routines
- Report generation
- Mechanisms to provide an explicit representation of time

Before the widespread use of object-oriented programming languages, the use of these traditional simulation languages was justified. However these languages are becoming obsolete, as there are more and more frameworks or toolkits that build upon existing object-oriented languages such as JAVA (e.g., DSOL [21] and SSJ [25]), which provide efficient data structures to manage the FEL. These toolkits allow a more flexible approach to building simulation models as they allow the user to implement customized components with the convenience of existing components such as random number generators

2.5.5 Simulation Development Paradigms

A simulation study consists of a series of different steps, such as data collection, verification, validation, experimental design, and output data analysis. The temporal relations of these steps can be described in the flow diagram in Figure 8 taken from Law and Kelton [24].

Many authors point out that the steps in a simulation study are not just simple sequential process steps, but often require revisiting of previous steps. A similar diagram can be found in [11]. These descriptions have the introduction of a conceptual model in common. There is no precise definition in the literature of what a conceptual model is. According to [11], the conceptual model is created in the mind of the modeler as a series of mathematical and logical relationships concerning the components and structure of the system under study. A conceptual model is a non-formal description of the simulation model; the term "assumptions document" is also used [24].

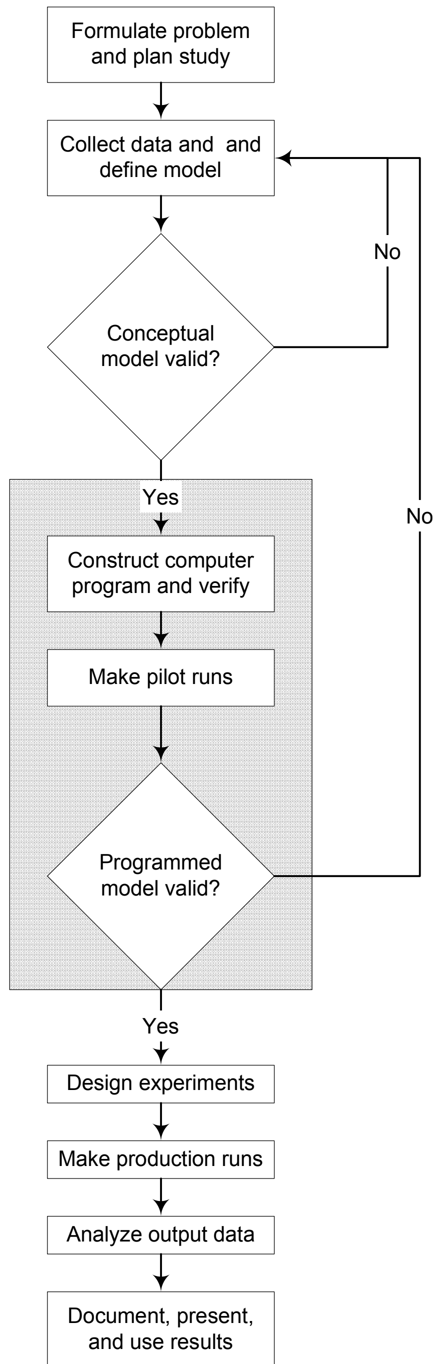

Figure 8: Steps in a Simulation Study from Law and Kelton [24]

2.5.5.1 Sargent's Circle

Another paradigm for simulation development is Sargent's circle (Figure 9). It is not a sequential diagram, but it shows the relationships between the system, conceptual model, and computerized model.

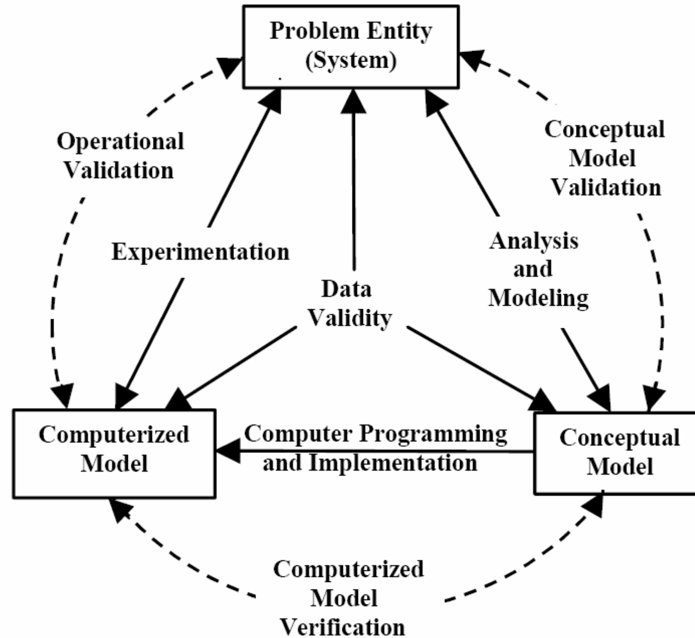

Figure 9: Sargents Circle [45]

These simulation paradigms are all lacking a formal foundation. They are rough descriptions of steps that one would perform when conducting a simulation study. The lack of formal description of the conceptual model introduces uncertainty into the modeling process. Many authors describe the simulation modeling process as inherently iterative, but one can argue that this is a direct consequence of the lack of formal models.

2.6 Simulation Model Specifications and Modeling Frameworks

The fundamental problem of simulation modeling is the necessity to describe the dynamics of a system in static terms. As mentioned before, the use of a conceptual model leaves many details of the implementation open or ambiguous.

A *specification* is usually defined as a set of requirements. Here it will be used to mean a detailed description of the system to be modeled. A *simulation model specification* is

the detailed description of system states, state transitions, and the conditions that cause these transitions. The related term *conceptual model* will not be used as it can have many different meanings.

In a sense, each simulation model can be seen as a specification of the modeled system. This is because a valid simulation model has to be able to emulate all the relevant events and data of interest. Since a simulation model is a computer program, there is no room for ambiguities when the model is executed: each event of interest has to be specified in detail in the code. This can make it difficult for the user to understand how the simulation code relates to the real-world model, and is especially true for simulation models that are written in a general programming language. In order to simplify the implementation of simulation models, different simulation models, specifications, or modeling frameworks have been defined. These specifications provide a more domain-specific problem view.

2.6.1 Discrete-Event System Specification (DEVS)

Zeigler [57] developed the Discrete-Event System Specification, which is a formal basis for the low-level representation of discrete-event models and their simulators. This specification defines a language that expresses the inputs, outputs, states, and transition functions. DEVS is part of a larger framework that tries to unify discrete-event and continuous dynamic systems. It is also the most comprehensive simulation modeling framework currently available in the literature, and is based on a rigorous formal mathematical description of the states and transition functions of the system. Hierarchical components are the main building block in the DEVS. Low-level components can be combined to form aggregate components. Coupling relations describe how these subcomponents are combined and how they interact.

The DEVS allows specifying a simulation model without any ambiguities. One clear advantage is that the simulation model and simulator are clearly separated. Hence, a single simulator can be used for all models that follow DEVS specifications.

An *atomic model* in the DEVS specification is given by

$$M = (X, S, Y, \delta_{\text{ext}}, \delta_{\text{int}}, \lambda, ta) \tag{1}$$

where X is the set of input values; S is the set of states; Y is the set of output values; $\delta_{\text{int}} : S \rightarrow S$ is the internal transition function; $ta : S \rightarrow R^+$ is the time advance function; $\delta_{\text{ext}} : Q \times X \rightarrow S$ is the external transition function, where $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ is the total state set and e is the time elapsed since the last transition; and $\lambda : S \rightarrow Y$ is the output function.

The interpretation of this atomic model is as follows: Let s be the current system state. If there are no external events, the system will stay in state s for $ta(s)$ time units. If the value for $ta(s)$ is zero, the system will immediately move into another state, i.e., this state is transitory. If the value is ∞, the system will stay in the same state forever unless an external event will occur. If an external event occurs and the system is in the total state (s, e) with $e \leq ta(s)$, the system will move to state $\delta_{\text{ext}}(s, e, x)$. In other words, the internal transition function determines the state of the system if no external event occurs since the last transition. The external transition function determines the new state if an external event occurs.

Zeigler defines a DEVS to be *legitimate* if for each $s \in S$,

$$\lim_{n \to \infty} \sum (s, n) \rightarrow \infty \tag{2}$$

where $\sum (s, n)$ is defined recursively by $\sum (s, 0) = 0$ and $\sum (s, n) = \sum_{i=0}^{n-1} ta(\delta^{+}_{\text{int}}(s, i))$, and $\delta^{+}_{\text{int}}(s, n)$ is the state reached after n iterations starting at state $s \in S$ if no external events intervene. In other words, the function $\sum (s, n)$ accumulates the time needed to make these n transitions. A legitimate DEVS model will always be able to advance the simulation clock and will not loop through a cycle of non-transitory states. Zeigler introduces the following conditions for a legitimate DEVS:

(a) If M is finite (S is a finite set): Every cycle in the state diagram of δ_{int} contains a non-transitory state (necessary and sufficient condition)

(b) If M is infinite: There is a positive lower bound on the time advances, that is, $\exists b > 0$ such that $\forall s \in S$, $ta(s) > b$ (sufficient condition)

Condition (a) means that the simulation model cannot go through an infinite cycle of

transitory states, which would prevent the simulation model from advancing the simulation clock. Condition (b) means that if all time advances have a lower bound, then there are no transitory states in the simulation model and therefore time will always be able to advance. The DEVS formalism includes the means to build coupled models from components. All components have to be DEVS models themselves, and can consist of other DEVS components.

2.6.1.1 Limitations of DEVS

There are several limitations in the DEVS framework. The main emphasis of the framework is on states. If there are simultaneous internal and external events, then mechanisms have to be provided to prioritize these events. Further, there are problems inherent in coupled DEVS systems [54]: it is difficult to identify components that violate required interaction constraints, and the evolution and substitution of new components in coupled models can lead to unanticipated behavioral conflicts. DEVS does not provide direct support for verification and validation, because it provides no means to analyze the dynamic behavior of the system. Although it is fairly easy to verify and validate small components, it is typically difficult to verify the coupled model.

2.6.2 Activity Cycle Diagrams

Activity cycle diagrams [41] (also called entity cycle diagrams) conceptualize the problem in terms of the logical flow of objects in the system. They can model interactions of system objects and are particularly useful for systems with natural queuing structure. These diagrams use only two symbols, which describe the life cycle of the system's objects or entities.

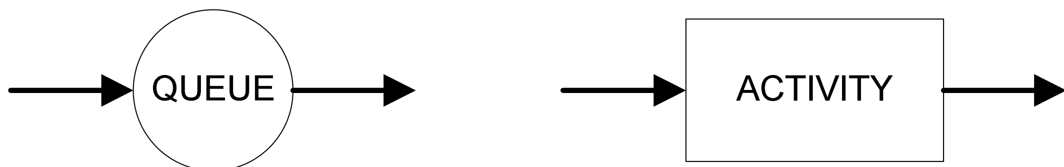

Figure 10: Activity Cycle Diagram Elements

An entity is any object of the model that can keep its identity through time. An entity

is either idle, in queue or active. When active, an entity can be engaged with other entities in time-consuming activities. The cycles of different entities are joined by their common activities. Figure 11 gives a simple example.

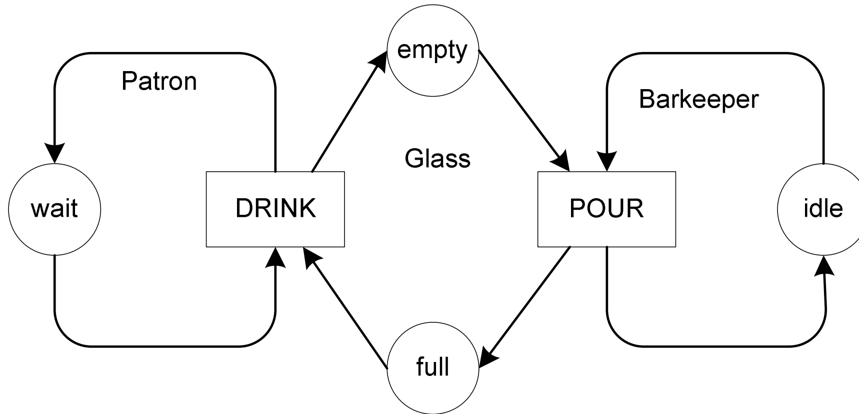

Figure 11: Activity Cycle Diagram for a Pub

There are three entities in this diagram: a patron, a glass, and a barkeeper. Each entity goes through certain cycles, e.g., wait and drink. The cycles are joined where the entities share an activity; for example, the activity `POUR` requires the glass and the barkeeper who fills the glass. A big disadvantage of activity cycle diagrams is that they cannot model complex routing decisions. They are also not capable of representing any kind of decision logic. Large graphs also become quickly confusing for the user. Because of these reasons, they are rarely used in practice.

2.6.3 Condition Specification

Condition specification is a specification language for discrete-event simulation models [36]. It is independent of traditional simulation world views; a model is defined in terms that do not prescribe any particular implementation techniques. Condition specification is not intended to be used as a language with which the modeler works directly when creating a simulation. It has many similarities with DEVS. A model is specified via a quintuple $(\Phi, \Omega, \Gamma, \tau, \Theta)$. Here Φ is the input specification (i.e., the information that the model receives,) and Ω is the output specification (i.e., the information that the environment receives from the model) [38]. Γ is the object definition set consisting of ordered pairs $(O, A(O))$,

where O is an object and $A(O)$ is the attribute set of the object. The state of an object at time t is defined by the values of its attributes at that time. τ is the system time and provides a partial ordering of actions during a simulation run. Finally, Θ is the transition specification that contains an initial state for the model, a termination condition, and the definition of the dynamic behavior of the model. The transition specification is a set of ordered pairs called condition-action pairs. A condition is a Boolean expression consisting of model attributes. When the condition is true, an action is performed; this could be a value change of an attribute, a time sequencing action, object generation or destruction, environment communication, or simulation termination.

Condition Specification also uses inputs, outputs and transition functions similar to the DEVS. The DEVS, though, focuses on components that are assembled, whereas the Condition Specification describes the simulation model in its entirety.

2.6.4 Simulation Graphs and Simulation Graph Models

Yücesan and Schruben [56] define simulation graphs and simulation graph models, which are a mathematical formalization of the event graphs introduced by Schruben [48] and described in Section 2.5.2.2. Events are represented on the graph as vertices and each vertex is associated with a set of state variable changes. The focus is on system events while entities are represented implicitly by variables. Relationships between events are represented as arcs between pairs of vertices. Each edge is associated with a set of logical conditions, which determine if the event that the arc is pointing to will be scheduled for execution. A *simulation graph* is defined as an ordered quadruple $G = (V(G), E_S(G), E_C(G), \Psi_G)$, where $V(G)$ is the set of event vertices, $E_S(G)$ is the set of scheduling edges, $E_C(G)$ is the set of canceling edges, and Ψ_G is the incidence function. The incidence function associates with each edge in $E_S(G) \cup E_C(G)$ an ordered pair of event vertices in G.

A *simulation graph model* is defined as $S = (F, C, T, \Gamma, G)$, where F is the set of state transition functions, C is the set of edge conditions, T is the set of edge delay times, and Γ is the set of event execution priorities.

Figure 12 shows a simulation graph for a single-server queue. The state variables are as

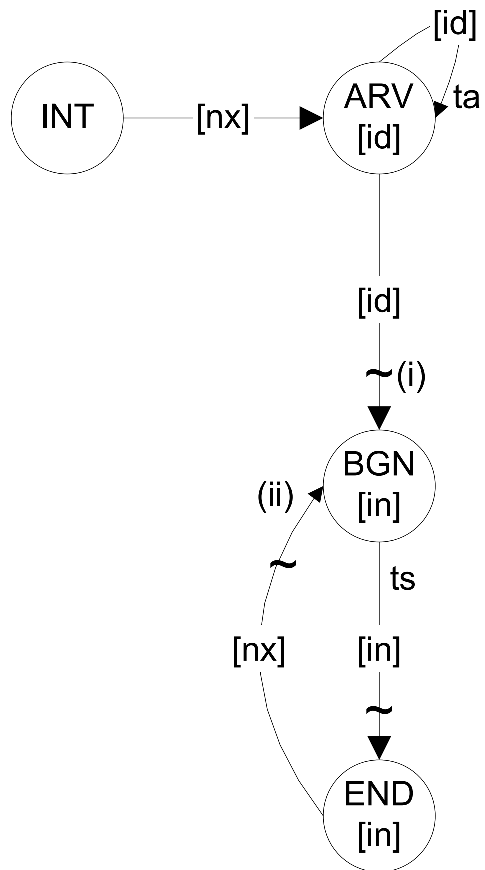

Figure 12: Simulation Event Graph for a Single-Server Queue [56]

follows: Q represents the number of customers waiting for service; S denotes the status of the server, with $0 \equiv busy$ and $1 \equiv idle$; id denotes the customer identification number; in is the identification number of the customer currently in service; nx represents the identification number of the customer next in line; $W[i]$ is the total time customer i spends in the system; and CLK represents the value of the simulation clock.

The *transition function* for this example is defined as

$$\begin{gathered} F = \{f_{\text{init}}; f_{\text{arv}}; f_{\text{bgn}}; f_{\text{end}}\} = \\ \{Q \leftarrow 0, S \leftarrow 1, nx \leftarrow 1; Q \leftarrow Q+1, id \leftarrow id+1, W[id] \leftarrow CLK; \\ Q \leftarrow Q+1, id \leftarrow id+1, W[id] \leftarrow CLK; S \leftarrow 1, W[in] \leftarrow CLK - W[in]\} \end{gathered} \tag{3}$$

The set of *edge conditions* is defined as

$$C = \{C_{\text{arv,bgn}}; C_{\text{end,bgn}}\} = \{S = 1; Q > 0\} \tag{4}$$

The set of *edge delay times* is

$$T = \{t_{\text{arv,bgn}}; t_{\text{bgn,end}}\} = \{t_a; t_s\} \tag{5}$$

Finally, the set of *event execution priorities* is

$$\Gamma = \{\gamma_{\text{init,arv}}; \gamma_{\text{arv,arv}}; \gamma_{\text{arv,bgn}}; \gamma_{\text{bgn,end}}; \gamma_{\text{end,bgn}}\} = \{2; 2; 1; 2; 1\} \tag{6}$$

These expressions together with Figure 12 represent the respective simulation graph model. Yücesan and Schruben [56] use simulation graph models to define structural and behavioral equivalence of simulation models.

Two simulation graphs models are *structurally equivalent* if they have elementary simulation graph that are *isomorphic*, that is, a bijection between the vertices of the graphs exists.

An elementary simulation graph model contains only simple event vertices (only one state variable changes at any time) and simple edge conditions, i.e., conditions that have only two arithmetic expressions connected by a relational operator ($<, \leq, =, \neq, \geq, >$). An elementary simulation graph can always be created by a process called expansion of the general simulation graph model. Hence, the problem of determining structural equivalence

of simulation graph models can be reduced to the problem of checking if the equivalent elementary simulation graphs are isomorphic.

Two simulation models A and B are *behaviorally equivalent* with respect to a subset of state variables if $T(E, A) = T(E, B)$ and $S(E, A) = S(E, B)$, where $T(E, A)$ is the partially ordered set of event times for the execution of model A within the experimental frame E. An experimental frame specifies a limited set of circumstances under which a system (real system or model) is to be observed or subjected to experimentation [56]. $S(E, A) = \{S_1, S_2, \ldots\}$ is the ordered set of state variables for the execution of model A within the experimental frame E. Behavioral equivalence relates the state variables and event occurrences of the simulation run; hence it is directly related to performance measures one usually tries to estimate via simulation. Yücesan and Schruben show that structural equivalence is a sufficient but not necessary condition for behavioral equivalence.

Simulation graph models are one of the most complex and comprehensive frameworks. In comparison to DEVS, simulation graphs have the ability to graphically display the precedence relationships of events in a single graph. There is no direct support of the process-interaction world view, because entities have to be modeled implicitly by means of state variables. This increases the number of variables for large models. Further, there is no object orientation, i.e., no encapsulation of variables through entities.

2.6.5 Critique of Exiting Simulation Model Specifications and Frameworks

All frameworks reviewed in this chapter have a commonality in that their use in practice is not widespread. One reason might be that users have difficulties understanding and using them. Some of these frameworks have a solid theoretical foundation and are potential candidates to build a framework for large-scale simulations, but none can address the verification problem directly.

2.7 Automatic Model Generation

A very different approach to building simulation models is automatic generation. A "semi-automatic" approach is based on graphical user interfaces that allow the user to parameterize modules; the simulation package then will generate code based on that input. Another

approach is to use a simulator that reads input from files, which specify the details of the system.

According to Mathewson [30], 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 model's behavior. One of the earliest examples for simulation model generator is presented in [30], which is based on entity cycle diagrams and in [31], an early PC implementation of the former. In [19] a simulation code generator software for an automated guided vehicle system is presented. In [26] a WITNESS^{TM} simulation model for shop floor control systems is generated automatically from graph-based process plans.

2.8 Conclusions

This chapter gave an overview of the current challenges in manufacturing simulation. Specifically, it discussed the principles of simulation modeling and approaches for improving modeling productivity. It also reviewed several simulation model specifications and modeling frameworks. The most potent frameworks, such as DEVS and Simulation Graph Models, have great modeling power, but are not very easy to understand and implement. These facts motivate the development of the proposed framework in the remaining chapters.

CHAPTER III

FRAMEWORK FOR SEMICONDUCTOR MANUFACTURING MODELING, CONTROL AND SIMULATION

3.1 Introduction

Modern simulation software often allows the user to drag and drop simulation modules and modify them for his/her needs. This often saves time during model development. Relying on this feature alone limits the diversity of models that can be formulated since the behavior of these modules is predefined. This is of particular interest when formulating large-scale models. The verification of large models also can be very difficult.

COTS software typically only allows for checking the syntax and some basic structure of the generated code. For example, the software can check if a modeling block is missing required input parameters. Another way of creating a simulation is to code it manually with a general purpose programming language or a special simulation language. This is the most flexible option. However, this option requires an even more rigid approach to verification. Without graphical support, it is the programmer's job to verify if the simulation model actually corresponds to the conceptual model. According to the American Heritage Dictionary [4], a framework is defined as:

1. A structure for supporting or enclosing something else, especially a skeletal support used as the basis for something being constructed

2. An external work platform; a scaffold

3. A fundamental structure, as for a written work

4. A set of assumptions, concepts, values, and practices that constitutes a way of viewing reality

A unique feature of the proposed simulation framework is that all relevant state information is contained within the model. The FEL can be reconstructed from the current state

of the model. This means that changes to the model can be made during a simulation run and the FEL can be updated accordingly. This simplifies greatly the usage of simulation for real-time decision-making. For example, it is possible to let the simulation model run in parallel with the real system. Then it can be used to immediately instantiate a simulation run for due date quoting or evaluation of different control strategies.

Figure 13 clarifies the intended use of the simulation framework in comparison with the traditional steps of simulation modeling. The framework replaces the traditional steps of programming, making pilot runs, and validation with a single step. The simulation model is formulated with a simulation data specification, which will be introduced in Chapter 4. In other words, the simulation framework can be described as "specification-based simulation modeling."

3.2 Overview of Petri Nets

The proposed framework is based on a Petri net (PN) data structure. The following subsection gives an overview of Petri nets. These nets can combine a number of different important characteristics. They can serve as a graphical tool to display the state and behavior of a system, yet they are based on a rigorous mathematical foundation and they possess simulation capabilities. PNs incorporate the concept of distributed system state with rules that define how state changes occur. They are currently being successfully employed to support many stages of the development of complex systems: rapid prototyping, formal specification, verification of correctness, performance evaluation, and documentation [59]. Overall, PNs represent a well-known, powerful, and widely used analytical formalism. Various applications in the areas of system modeling and control use PNs as a simulation and analysis tool for different engineering systems, such as robots, processing plants and batch systems.

3.2.1 Classical Petri Nets

Carl Adam Petri developed the original Petri net in 1962 [43]. A PN is a general-purpose mathematical tool for describing relations between conditions and events. Specifically, a PN is a bipartite graph with places and transitions (a bipartite graph has a set of vertices, which can be divided into two disjoint sets so that no two vertices of the same set share an

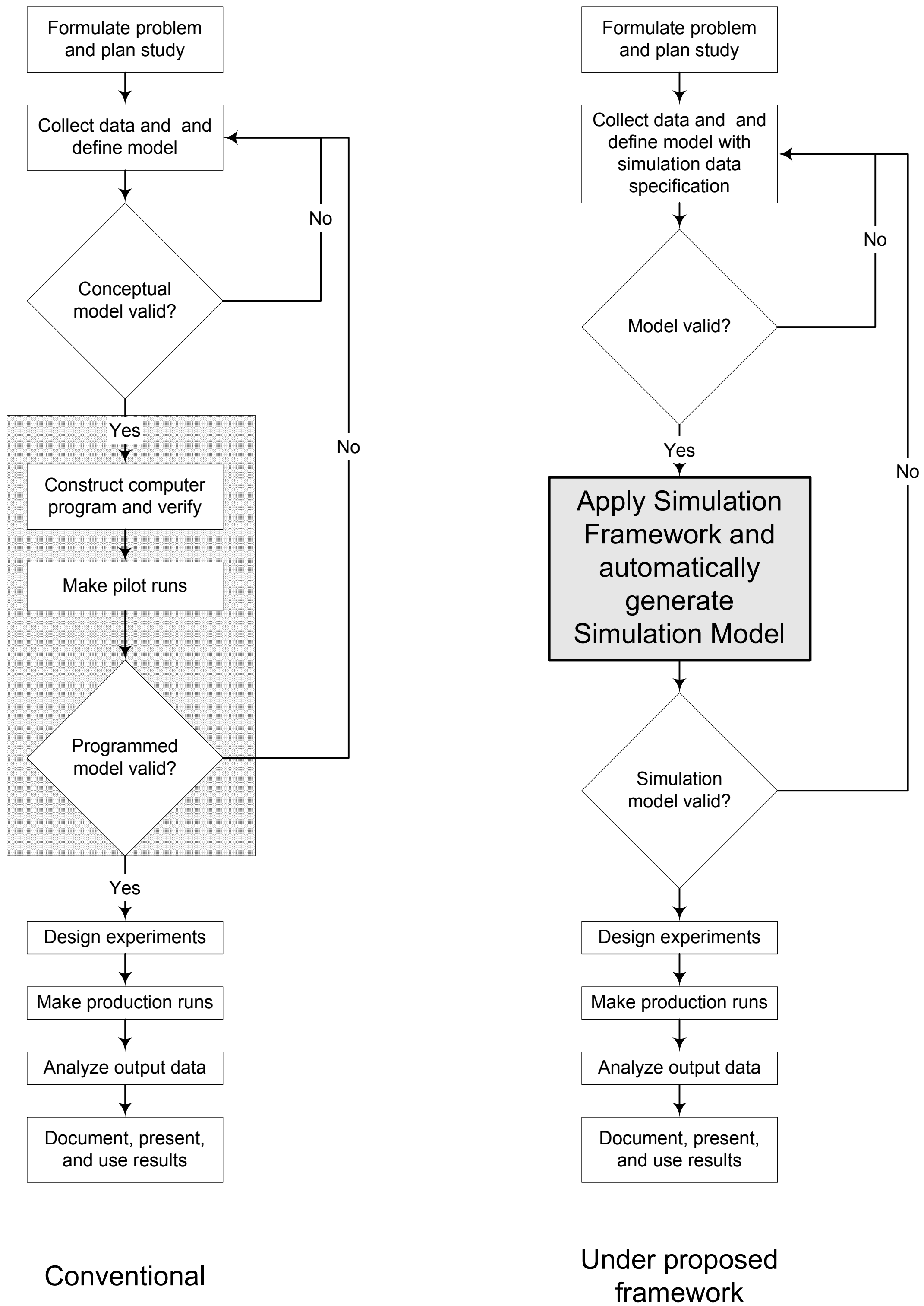

Figure 13: Comparison of Conventional Method with New Framework

edge).

Places are connected to transitions and transitions to places. Clearly, there will never be a connecting arc between any two places or any two transitions. Events are associated with transitions. Places and transitions are represented by circles and bars respectively. A transition can “fire” or is live when every input place to the transition has at least one token. Every finite-state machine can be represented by a PN [14], but the opposite is not true in general. Since every computer simulation model can be interpreted as a finite-state machine or automaton, it is in principle possible to describe any computer simulation as a PN. A *Petri net graph* is a weighted bipartite graph (P, T, A, w), where P is the finite set of places and T is the 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 1. $A \subseteq (P \times T) \cup (T \times P)$ is the set of arcs from places to transitions and transitions to places. $w : A \rightarrow \{1, 2, 3, \ldots\}$ is the weight function on the arcs, i.e., a positive integer is assigned to each arc as a weight. Similarly, ${}^{\bullet}t = \{p|(p, t) \in A\}$ is the set of input places of t and $t^{\bullet} = \{p|(t, p) \in A\}$ is the set of output places of t. ${}^{\bullet}p = \{t|(t, p) \in A\}$ is the set of input transitions of p and $p^{\bullet} = \{t|(p, t) \in A\}$ is the set of output transitions of p.

These definitions describe only the structure of the graph. Another important aspect is the state and dynamic behavior of a discrete-event system represented by the marking of the PN and the state transition function. A *marked Petri net* a five-tuple (P, T, A, w, m) where (P, T, A, w) is a PN graph and m is a row vector representing the marking of the set of places $P = P_1, \ldots, P_n$; $m = [m(p_1), m(p_2), m(p_x), \ldots, m(p_n)] \in N^n$. The number of tokens in place p_i is indicated by the ith entry $m(p_i)$ of this vector. The marking of a PN represents its state.

The *state transition function*, $f : N^n \times T \rightarrow N^n$ of a marked PN (P, T, A, w, m) is defined for transition $t_j \in T$ if and only if $m(p_i) \geq w(p_i, t_j)$ for all $p_i \in {}^{\bullet}t_j$. If $f(m, t_j)$ is defined, then the new marking of the PN is $m' = f(m, t_j)$, where $m'(p_i) = m(p_i) - w(p_i, t_j) + w(t_j, p_i)$.

A PN is called ordinary if all arcs have weight 1. A pair of transitions is said to be in *conflict* if the firing of one transition will disable the other transition [51]. A pair of transitions is said to be *concurrent* if the firing of one transition will not disable the other

transition. The following subclasses of PNs have been defined in the literature [33]:

A *state machine* is an ordinary PN such that each transition has exactly one input place and one output place: $|^\bullet t| = |t^\bullet| = 1$, for all $t \in T$. This means there can be no concurrency, but there can be conflicts between transitions.

A *marked graph* is an ordinary PN such that each place has exactly one input transition and one output transition: $|^\bullet p| = |p^\bullet| = 1$ for all $p \in P$. This means there can be no conflict, but there can be concurrency between transitions.

A *free-choice net* is an ordinary PN, where every arc from every place is either a unique outgoing arc or a unique incoming arc to a transition: $|p^\bullet| \leq 1$ or $^\bullet(p^\bullet) = \{p\}$, for all $p \in P$. This means there can be concurrency or conflict between a pair of transitions, but not both. An *extended free-choice net* is an ordinary PN, such that $p_1^\bullet \cap p_2^\bullet \neq \emptyset \Rightarrow p_1^\bullet = p_2^\bullet$ for all $p_1, p_2 \in P$. An *asymmetric choice net* is an ordinary PN, such that $p_1^\bullet \cap p_2^\bullet \neq \emptyset \Rightarrow p_1^\bullet \subseteq p_2^\bullet$ or $p_1^\bullet \supseteq p_2^\bullet$, for all $p_1, p_2 \in P$.

Many theorems exit for these subclasses of PNs that describe behavior and structural properties. However, these classes are very restrictive and not very useful for manufacturing system modeling. The following example illustrates this.

Example 1: Figure 14 shows three processes that share two resources. P_1, P_2, P_3 represent the start places for the three different processes. Process 1 requires resources R_1 and R_2, process 2 requires resource R_1, and process 3 requires resource R_2. Clearly this PN is not a state machine as $|^\bullet T_1| > 1$. The net does not belong to the class of marked graphs because $|R_1^\bullet| > 1$. Further, it is not a free-choice net as $|R_1^\bullet| > 1$ and $^\bullet(R_1^\bullet) \neq \{R_1\}$. Since $R_1^\bullet \cap R_2^\bullet \neq \emptyset$ and $R_1^\bullet \neq R_2^\bullet$, it is not an extended free-choice net; and since $R_1^\bullet \cap R_2^\bullet \neq \emptyset$, $R_1^\bullet \not\subset R_2^\bullet$, and $R_2^\bullet \not\subset R_1^\bullet$, it is not an asymmetric choice net. However, it is an ordinary PN, because all arc weights are equal to one. This is a fairly simple net representing a common situation in manufacturing. Unfortunately, many existing theoretical results in PN theory are not applicable, as they often are restricted to the above classes. $\diamondsuit$

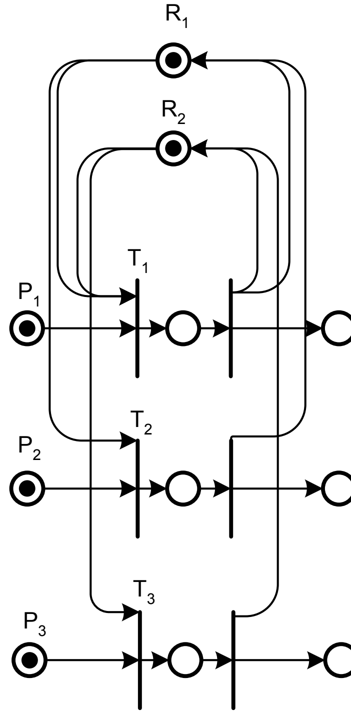

Figure 14: Parallel Processes

3.2.2 Inhibitor Arcs

Inhibitor arcs were introduced to give PNs the computational power of a Turing machine [33] . Turing machines are the most basic model of computation in computer science, and can model all computational capabilities of modern computer software. Hence, inhibitor arcs enhance the modeling power of PNs to the highest degree that is known.

Inhibitor arcs are similar to normal arcs, except that instead of checking for the presence of a token in a place, they check for the absence of tokens. In general, a PN with inhibitor arcs cannot be transformed to an ordinary PN. Nonetheless, a bounded inhibitor arc net can always be transformed into an ordinary PN.

3.2.3 State Equations

It is also possible to describe the dynamic behavior of classical PNs using linear algebra. The $m \times n$ matrix A of a PN with elements $a_{ij} = w(t_j, p_i) - w(p_i, t_j)$ is called the *incidence matrix* [14]. The m-dimensional unit firing row vector $u_j = [0, \ldots, 0, 1, 0, \ldots, 0]$ with a 1 appearing at the jth position indicates that the jth transition is firing. The vector state

equation can then be written as $m' = m + u_jA$ where m' is the new state after firing of the jth transition. In other words, the state transition function is $f(m, t_j) = m + u_jA$. A similar approach in encoding a PN is to use two matrices, one for the input functions and one for the output functions of the transitions. Every possible firing sequence has to fulfill the state equation of the PN, but the existence of a solution does not guarantee the existence of a valid firing sequence [14].

3.2.4 Relation of Petri Nets to Other Formal Models for Discrete-Event Systems

As mentioned earlier, it can be shown that any automaton (finite-state machine) can be transformed into a PN but not vice versa [14]. Hence PNs have larger modeling power than automata. Event graph models also have been shown to have the modeling power of Turing machines [46]. This gives them the same expressiveness as PNs with inhibitor arcs. Schruben [49] also shows that a stochastic PN can be mapped to an event graph model. However, when using event graphs one cannot take advantage of the rich literature that exists on PNs. In summary, PNs can be seen as one of the most versatile modeling tools for discrete-event systems.

PNs have been applied to many industrial applications. For example, they have been used to model real-time fault tolerant and safety critical systems, flexible manufacturing systems and controllers for such systems, communication protocols, computer systems with parallel processors, and resource allocation systems [59].

3.2.5 Behavioral Properties of Petri Nets

When modeling a system with a PN, one is interested in determining if the system is able to exhibit certain behavior. Two types of properties can be distinguished: properties that depend on the initial marking and properties that are independent of the initial marking [33]. Below we discuss the most common cited properties of PNs.

Boundedness: A PN is bounded if the number of tokens in each place for any marking does not exceed a finite number. It is called *k-bounded* if the number in all places does not exceed k. An 1-bounded PN is called safe. A PN is called *structurally bounded* if it is

bounded for any initial marking.

Reachability: A marking m in a PN is reachable if there exists a firing sequence that transforms the initial marking m_0 to m. The set of reachable states of a PN N with initial marking m_0 is denoted by $R(N, m_0)$.

Coverability: A marking m is said to be coverable if there exists a marking m' in $R(N, m_0)$ such that $m'(p) \geq m(p)$.

Liveness: The concept of liveness is closely related to the concept of deadlock. A *deadlock* in a PN is a set of transitions that cannot fire permanently. A main source of deadlocks is the existence of shared resources. A transition is *live* if it is not deadlocked. This does not mean that it is enabled but rather that there must exist a firing sequence that will eventually enable the transition to fire.

A PN N with initial marking m_0 is live if there is always some sample path (i.e., a sequence of transition firings) such that any transition can eventually fire from any state reachable from m_0. A PN contains a deadlock if there is a marking $R(N, m_0)$ such that no transition is enabled. Hence a live PN does not contain a deadlock. This is a very strong property. In [14] and [42] the following levels of liveness for a transition are defined:

- $L0$-live or *dead*: A transition is L0-live or dead, if it can never be fired
- $L1$-live: A transition is L1-live, if there is a firing sequence from the initial marking such that the transition can fire at least once
- $L2$-live: A transition is L2-live, if there is a positive integer k such that the transition can fire at least k times
- $L3$-live: A transition is L3-live, if there is some (infinite) firing sequence in which the transition appears infinitely often
- $L4$-live: A transition is L4-live if it is L1-live for every state reached from m_0

Reversibility: A PN is said to be reversible if for each marking m in $R(N, m_0)$ the initial state m_0 is reachable. In other words, the PN can always go back to the initial state after firing some sequence of transitions. The properties boundedness, liveness, and reversibility

are independent of each other; this means the existence of one of these properties does not imply the existence of an other.

Persistence: A PN is persistent if for any two enabled transitions the firing of one will not disable the other, i.e., no transitions are in conflict. Therefore, an enabled transition in a persistent net will stay enabled until it fires. All marked graphs are persistent.

3.2.6 Structural Properties of Petri Nets

Structural properties are determined based on the structural properties of the PN graph. Therefore, these properties do not depend on the initial markings of the PN.

Conservation: A PN is called *strictly conservative* if the total number of tokens remains constant for all reachable states $R(N, m_0)$. A PN is called conservative with respect to a weight vector $w = (w_1, w_2, \ldots, w_n)$, $n = |P|$, $w_i > 0$, if for all $m \in R(N, m_0)$, $\sum_i w_i\, m(p_i)$ is constant.

Structural boundedness: A PN is structurally bounded, if it is bounded for any initial marking.

Structural liveness: A PN is structurally live if there is an initial marking that makes the net live.

3.2.7 Classical Analytical Methods for Petri nets

We start with a few additional definitions: The *reachablility tree* (also occurrence graph) of a PN is the graph that contains a node for each reachable marking and an arc for each possible transition occurrence [14]. The root node is the initial marking of the net. If the net is unbounded, the reachability tree will grow infinitely.

The *coverability tree* is closely related to the reachability tree. The symbol ω is used at the jth position of a marking m', if $m'(p_j) \geq m''(p_j)$, where $m''(p_j)$ is the marking of a node at a higher level in the tree. In other words, the state is "covered" for all following markings, i.e., the token count of p_j will not decrease.

The coverability tree can be used to check for safeness, boundedness, conservation, and coverability. For a bounded PN, the coverability tree is the same as the reachability tree. A general problem arising from the use of these methods is state space explosion since the

coverability or reachability tree can get very large for relatively small PNs. Hence these methods are impractical for large manufacturing systems that can contain thousands of places and transitions.

Invariant analysis is another approach to study PNs. A *place invariant* (also S-invariant) is a solution y to the equation $Ay = 0$, where A is the incidence matrix. The non-zero entries in y represent weights associated with the corresponding places, so that the weighted sum of tokens over these places is constant for all markings reachable from the initial marking [51]. These places are said to be covered by a place invariant. If each place is covered by a place invariant, then the PN is bounded.

A *transition invariant* (T-invariant) is a solution m of the equation $A^T m = 0$, where "T" indicates the transpose. A T-invariant specifies a firing count vector that leads back to the initial marking. However, it does not identify the exact order of the firings. Recall that a firing vector only represents the number of firings for each transition. In many cases, this means that it is possible for a firing vector to represent different firing sequences. Hence the existence of a transition invariant is only a necessary condition for a PN to be reversible.

The PN representations of real-world models are very large. Hence, the classic property checking methods, such as coverability tree analysis, invariant analysis, and algebraic analysis hardly apply to such models [52].

3.2.8 High-Level Petri Nets

The classic PN formalism only encodes the state of the system in terms of a vector, which indicates the number of tokens in each place. Therefore, it is difficult to model complex systems where entities carry additional information. Also, the notion of time does not exist in the classical formulation of PNs. Many extensions of the basic PN formalism exist that try to incorporate time and other additional information into the net. These are called high-level PNs, and have additional information attached to the tokens in the net.

Colored Petri nets are the most commonly cited high-level PNs. They were introduced by Jensen [23]. These PNs use colored tokens; as a result, it is no longer possible to represent the state only by the count of tokens for each place. The enabling of a transition depends

not only on the presence of tokens in the input pace of the transition, but also on the color of those tokens. The transition function can be very complex, as it can generate tokens of different colors in the output places. Colored PNs can have a very compact representation as they hold much more information, such as token colors and transition functions, than classical PNs. Although the classical analytical techniques are not applicable directly, it is possible to use net unfolding techniques that convert a colored PN to a standard PN.

Other types of high-level PNs include nets with abstract data types, well-formed (colored) nets, and regular nets. Many different types and extensions exist, each having specific firing rules and data types. Extensions of PNs with times are discussed in the next section.

3.2.9 Representation of Time in Petri Nets

As discussed in Section 3.2.8, the classic PN formulation does not have any notion of time. Over time, a great number of extensions of PNs to capture time have evolved. The two main categories are *deterministic timed Petri Nets* and *stochastic timed Petri nets*.

Deterministic timed transition nets associate deterministic firing delays to individual transitions. A transition can only fire after it has been enabled for the respective deterministic time interval.

Deterministic timed places nets are the "dual" version of deterministic timed transition nets; they associate firing delays with places. *Deterministic timed arc nets* have delays associated with the arcs; in other words, there is a "travel" time along an arc for tokens.

Time Petri nets associate two time values with each transition; the earliest firing time and the latest firing time.

Stochastic Petri nets associate exponential firing times with transitions. It can be shown that a stochastic PN is essentially a compact representation of a continuous-time Markov chain [51].

Generalized stochastic Petri nets are extensions of stochastic PNs that allow immediate transitions in addition to timed transitions. This makes their analysis more complex than the analysis of standard stochastic PNs.

High-level stochastic Petri nets are extensions of high-level PNs that associate exponential firing times with transitions, similar to stochastic PNs.

Semi-Markovian stochastic Petri nets are extensions of stochastic PNs that allow a non-exponential distributions for delay times. Figure 15 gives a hierarchical overview of timed PNs.

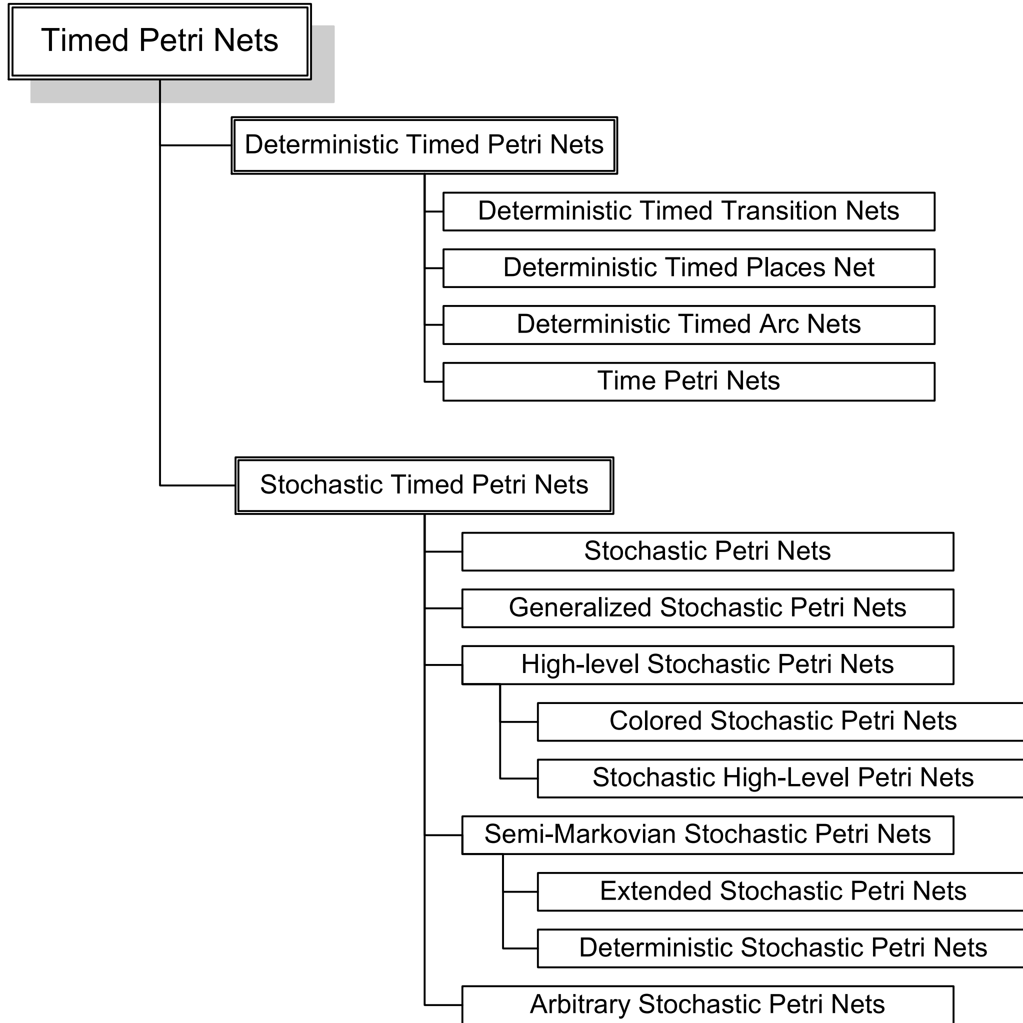

Figure 15: Overview of Timed Petri Nets [51]

Deterministic timed Petri nets use fixed times that are usually associated with transitions, places, or arcs. The firing rules can be quite variable. For example, in one version of deterministic timed transition nets the enabling tokens have to be present during the

delay time before the transition can fire. If the transition becomes disabled during this time interval, the timer is reset. The transition can only fire when it becomes enabled again and the enabling tokens are present during the entire delay time. In another version of deterministic timed transition nets, there is no reset of the timer, so the transition can fire if it is enabled at the scheduled firing time.

Another problem with the approaches above is the interpretation of firing delays because it is often unclear when the firing of a transition actually starts. The firing of a transition often represents time durations, but in classical PNs it represents the instantaneous transition of a discrete-event system to a new state.

A different approach, which is not mentioned above is interval timed colored Petri nets [50]. They are based on colored PNs, but use timestamps on tokens to capture time. A similar but simplified approach is used in the proposed simulation framework.

3.3 Object-Oriented Petri Net Simulation Framework

Fidelity refers to the accuracy of a simulation model in representing the behavior of the real-world system [57]. It is closely related to validity, but also describes the level of detail of the simulation model. Thus, a high-fidelity model is a valid and very detailed simulation model. Unfortunately, there are not any established measures of fidelity in practice.

The focus for the application of the proposed simulation framework is on the production scheduling level within a manufacturing system. Hence, this framework does not intend to create high-fidelity simulations for complex material handling systems. The implicit assumption is that a material handling system is already appropriately designed and therefore does not present a bottleneck. The framework is capable of modeling capacitated transportation systems, but is not intended to be used for modeling complex transportation routing systems such as automated guided vehicle systems.

3.3.1 Relationship to Time Colored Petri Nets

The PN simulation framework has similarities with time colored Petri nets. For example, it also uses tokens that have additional data assigned. These data are mainly times, priorities, and attributes. However, the enabling rule for transitions is exactly the same as for classical

PNs, i.e., only the required amount of tokens have to be available in each input place. Attributes can be added to tokens, but they do not influence the enabling conditions for transitions. Colored Petri nets can use complex enabling rules that involve tokens with different colors. Here no colored tokens (in a traditional sense) are used, as the attributes are not involved in enabling of transitions.

In classical and colored Petri nets the firing of enabled transitions is undetermined. This means that all enabled transitions can fire in any order. For simulation of manufacturing systems, this is not suitable, because all events need to have a defined ordering. This is especially true for transitions that are in conflict, as different firing sequences will lead to different simulation results. In classical or colored PNs, specific ordering of simultaneously enabled transitions can only be implemented with inhibitor arcs, which assign a fixed order between transitions through the capability of zero testing, i.e., a inhibitor arc can test if there are zero tokens in a given place. However, this is "hard-coded" within the Petri net and the only way to change the ordering is by changing the PN. Further, in order to implement this, all transitions pairs that are in conflict have to be considered and an ordering between them has to be provided.

The proposed simulation mechanism on the other hand provides a convenient way to model the ordering of transition firings. It is possible to implement different dispatch rules that require different firing sequences. The structure of the PN does not have to be changed for this, only the dispatch rule that is assigned to a transition has to be altered. Further, the mechanism also ensures that the appropriate job tokens are moved through the PN. This means that if multiple job tokens are available in a place, the simulation mechanism will remove the appropriate token with the highest priority. On the contrary, colored PNs consider all tokens of the same type to be equivalent and no specific order is established between them.

3.3.2 Advantages of Petri-Net-Based Formulation

The graphical nature and firm mathematical foundation of PNs offer a unique advantage. Numerous analytical methods are available, which can prove, for example, certain structural

properties. This is important during the model building process, so that there will be no deadlock situations. A PN-based model is very adaptable; the model can be augmented with additional places and transitions to address, for example, the addition of new machines. Further, a PN combines the information of state and flow in one model. The flow of goods, information, and resources can be modeled in a unified way. Distributed and parallel processes can be modeled as well as synchronization of two or more processes. This allows modeling of an operation in a very elegant and transparent way. It is also possible to model finite capacities or shared resources.

The generation of the simulation model is very flexible yet rigorous. It can be decoupled from execution, e.g., different parts of the model can be built independently and then put together for execution. The underlying PN can also define the control structure of the system. It is ideally suited to model pull systems. Transitions that remove tokens from a storage place can also serve as "ordering" transitions, i.e., they can trigger the production of new parts to replace the removed material.

The physical layout of a manufacturing system does not correspond directly to the graphical structure of the respective PN. Manufacturing jobs may share the same physical route through the manufacturing system. However, they are represented as different routes in the PN.

This is in contrast to commercial simulation tools. The graph represented by the PN will also become very large. Hence, it may not be possible to display the complete PN graph for a large-scale model. This is, however, not a problem as the complete simulation model is generated from verifiable sub-PNs; therefore, it is not really required to be able to inspect the entire PN at once.

3.3.3 Overview of Object-Oriented Programming

Object-Oriented Programming (OOP) has revolutionized the way computer programming is done. It is very different from the traditional functional programming. OOP is based on the idea of *objects*. An object represents an abstraction of some physical object or just some idea or concept that may be represented by an internal state [53]. Objects have fields

or attributes that define their state and can be modified by invoking *methods* of the object.

The behavior of an object is specified in a class description. The object is said to be an instance of the *class* that describes its behavior. The class description specifies the internal state space of the object and defines the types of messages that may be sent to all its instances. Hence, a class can be seen as a “blueprint” for an object.

OOP is based on the notion of sending *messages* to objects. Messages can either modify or return information about the internal state of an object. A message that is sent to an object will invoke the corresponding method of that object. *Encapsulation* is another important concept in OOP because it allows hiding details of an object, so that other objects can only use the public methods on the target object. This can significantly simplify the software development process. *Inheritance* is another key concept in OOP. Existing classes can be reused and extended in a way that they inherit characteristics from the ancestor (super) class, while more specialized characteristics can be introduced. *Polymorphism* is closely related to the concept of inheritance: it allows the use of objects in expressions without knowing the specific type of the objects. During runtime, polymorphism ensures that the right methods on the target object are invoked. An *interface* is a contract that specifies which methods a class has to specify.

3.3.4 Core Elements of the Proposed Framework

As with the classical definition of a PN, the core elements of the proposed 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 is representing 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 16 shows how these elements relate to each other.

A `Place` is an object that represents a place in the PN. Each `Place` has an `id` field, which uniquely identifies it in the complete 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 corresponds to the set ${}^{\bullet}p$ of a place p in a PN. The field `outTransitions` is a `List` of `Transition` objects, which represent the output transitions of the `Place`. This is corresponds to the set $p^{\bullet}$ of a place p in a PN. There are subclasses of `Place`, which can represent resources or operators; these will be introduced in Section 3.3.4.1.

The field `tokens` is a `SortedSet` of `Token` objects that represent the marking of the place. A `SortedSet` is a set of objects, which is ordered according to a specific criterion, in this case the timestamp of the tokens in the set.

The method `addToken()` is used during a simulation to add a token object to the tokens set of the place. It uses a `Token` object as a parameter, which refers to the token to add. The method `removeToken()` is used to remove a token from the place. The token to be removed is a parameter of the method.

The methods `addInTransition()` and `addOutTransition()` are used when building the PN. They add a Transition object to the `inTransition` list or the `outTransition` list respectively. There are more fields and methods in the Place class, which are of auxiliary

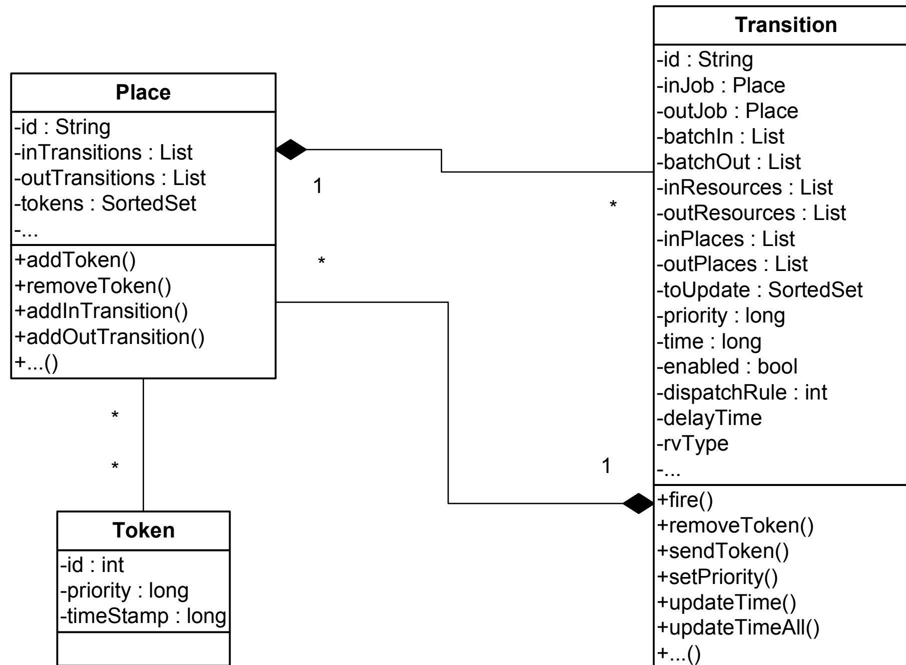

Figure 16: Class Diagram of Core Elements

nature and are not shown in Figure 16.

A `Token` is an object that represents a token in a classical PN. However, it has extension for time and priority, which are represented by the fields `timeStamp` and `priority`.

A `Transition` is an object that corresponds to a transition in a classical PN. Each `Transition` object has a unique identifier stored in the field `id`. The fields `inJob`, `batchIn`, `inResources`, and `inPlaces` are used to hold places that represent the set of input places of the Transition object, i.e., the set ${}^{\bullet}t$ of a transition t in a PN. 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 $w > 1$. The field `inResources` corresponds to places that represent resources. The field `inPlaces` is used for all other places that do not belong to any of the previous types.

The fields `outJob`, `batchOut`, `outResources`, `outPlaces` are used to hold places that represent the set of output places of the `Transition` object, i.e., the set $t^{\bullet}$ of the transition t. Their usage is analogous to the input places.

The field `time` is used to indicate the firing time of the respective 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 `dispatchRule` is used to store the kind of queuing discipline used; the default is FIFO. 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, the default value is `DETERMINISTIC`. Any distribution can be implemented, as long the appropriate generator is available to the framework.

The field `toUpdate` is a list of `Transition` objects that have to be checked when the transition fires. A detailed algorithm to determine this set will be presented in Section 3.3.5.

The method `fire()` will remove the appropriate `Token` objects from the input places and will place the appropriate tokens in the output places. The methods `removeToken()` and `sendToken()` are used for that. The `fire()` 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. An algorithm will be presented in Section 3.3.5.5. The method `updateTime()` will calculate the firing time of the transition. The method `updateTimeAll()` will calculate the firing times and check enabling conditions of the transitions in the `toUpdate` List.

The core elements above have a direct mapping to a classical PN $N = (P, T, A, w)$. 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 other.

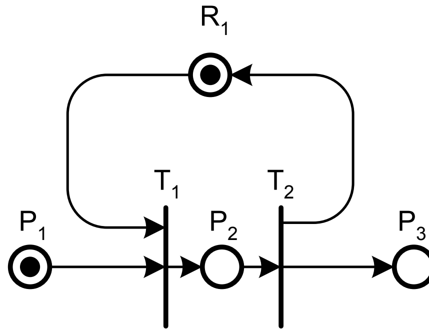

Figure 17: Example

Example 2: Figure 17 shows a PN $N = (P, T, A, w)$ with $P = \{P_1, P_2, P_3, R_1\}$, $T = \{T_1, T_2\}$, $A = \{(P_1, T_1), (P_2, T_2), (R_1, T_1), (T_1, P_2), (T_2, R_1), (T_2, P_3)\}$, and unit weights. The current marking of the net is $m = \{1, 0, 0, 1\}$. Each place is represented by an object. Place P_1 has the following field values: `inTransitions` = `null`, `outTransitions` = $\{T_1\}$. Place P_2 has `inTransitions` = $\{T_1\}$, `outTransitions` = $\{T_2\}$. P_3 has `inTransitions` = $\{T_2\}$, and `outTransitions` = `null`. The resource place R_1 has the field values `inTransitions` = $\{T_2\}$ and `outTransitions` = $\{T_1\}$. The field values for transition T_1 are `inJob` = $\{P_1\}$, `outJob` = $\{P_2\}$, and `inResources` = $\{R_1\}$. The field values for T_2 are `inJob` = $\{P_2\}$, `outJob` = $\{P_3\}$, and `outResources` = $\{R_1\}$. All other field values for the input and output places are `null`. In this fashion the entire structure of the PN graph is stored as references from places to transitions and vice versa. These references are also important for the execution of the simulation model. $\diamondsuit$

The simulation model is not coded in a traditional way. The model is generated "online," meaning that the simulator will create the PN structure from an input file, which is used for the detailed specification of the system under study.

3.3.4.1 Subclasses of the Core Elements

For better organization of the model the aforementioned classes are extended with subclasses. For example, the class `Place` has subclasses `Resource`, `Operator`, `ProcessPlace`, `ControlPlace`, and `BatchPlace` as shown in Figure 18. These objects are primarily used to simplify the model generation.

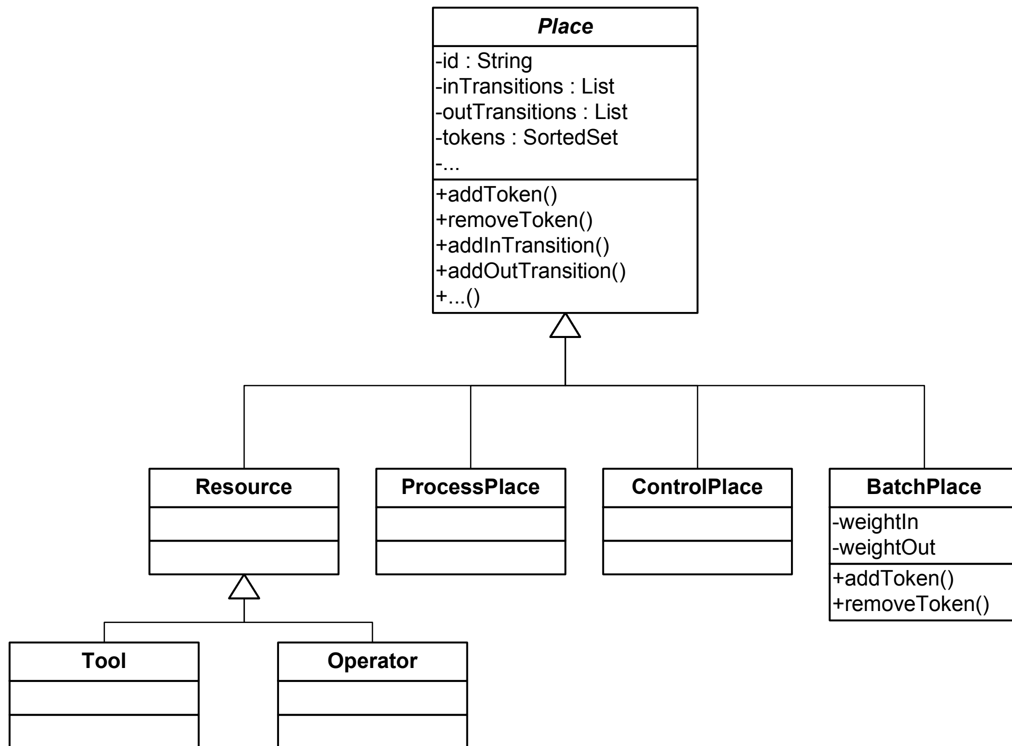

Figure 18: Subclasses of Place

The `Transition` class has the subclasses `FixedMaxPriority`, `FixedMinPriority`, and `SwitchTransition`. The class `FixedMaxPriority` has the subclass `TriggerTransition` (see Figure 19). In most cases only the normal `Transition` class is used.

The class `FixedMaxPriority` will assume the maximum possible priority. Hence, when

a transition of this type is enabled, it will be always ordered before other types of transition in the FEL and will fire before them. The class `TriggerTransition` is used for transitions with no input place, and therefore it is always enabled. The purpose of this type of object is to create tokens in (potentially random) time intervals, for example, to release jobs at a certain rate to the system or to trigger machine failures.

The class `FixedMinPriority` will assume the minimum possible priority. This will assure that, when enabled, any other transition types with the same time stamp will fire first.

The class `SwitchTransition` is used to route tokens to another place with a certain probability. Usually the job token will be sent to the place that is specified in the field `outJob`. For this class the `JobToken` will be sent to an alternative place with probability P; otherwise,it will be sent to the usual place. This type of transition is used to model rework and scrapping of lots.

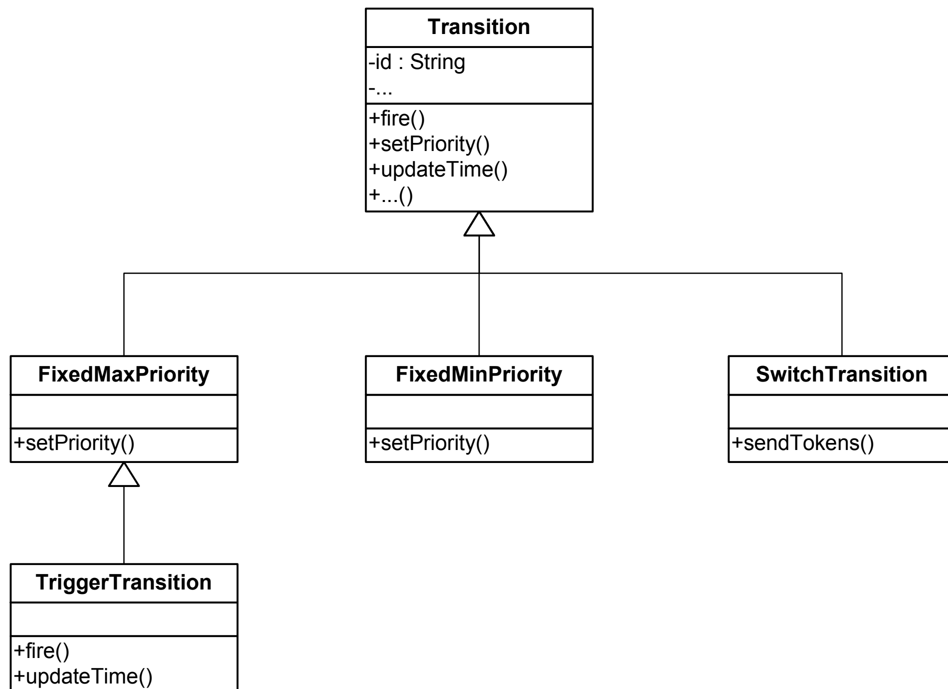

Figure 19: Subclasses of Transition

The `Token` class is extended with the subclass `JobToken`. This type of object represents manufacturing jobs that go through the system. Objects of the normal `Token` class are used for all other purposes.

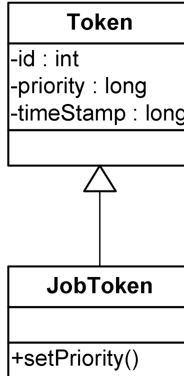

Figure 20: Subclass of Token

3.3.5 Execution Mechanism

After the PN is generated, the simulation model can be executed. Events correspond to firing of transitions. In Figure 17, the event "start processing" corresponds to the firing of transition T_1. The event "end processing" corresponds to the firing of transition T_2. To simulate PNs, 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.

3.3.5.1 Enabling Rule

A transition is enabled if there are enough tokens in each of the input places. The pseudo-code for checking whether a transition t is enabled is as follows (comments are marked with $\triangleright$):

CHECK IF ENABLED (TRANSITION t)

1 $t.enabled \leftarrow true$

2 **for** each place $p \in {}^{\bullet}t$

3 **do if** $|p.tokens| < w(p,t)$

4 **then** $t.enabled \leftarrow false$

5 break $\triangleright$ transition is not enabled

If there is a place that has fewer tokens than the required amount $w(p,t)$, the loop will be abandoned. In most cases the arc weights are equal to 1 except for objects of type `BatchPlace`, which is used for batch operations.

3.3.5.2 Timing Mechanism

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. In order to save computing steps, the algorithm for updating time is combined with the algorithm CHECK IF ENABLED for enabling transitions. 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.

Example 3: Consider the PN in Figure 21. 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 $\{\langle 5 \rangle\}$, P_1 has two tokens with timestamps $\{\langle 11 \rangle, \langle 25 \rangle\}$, and P_2 has three tokens with timestamps $\{\langle 15 \rangle, \langle 35 \rangle, \langle 40 \rangle\}$. Note that these tokens are all sorted according to their values. 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 and since the value of the timestamp in R_1 is 5. The enabling time for T_2 is 15 since that is the value of the smallest timestamp in P_2. $\diamondsuit$

Algorithm UPDATE TIME is used to determine if a transition t is enabled and when it is eligible to fire.

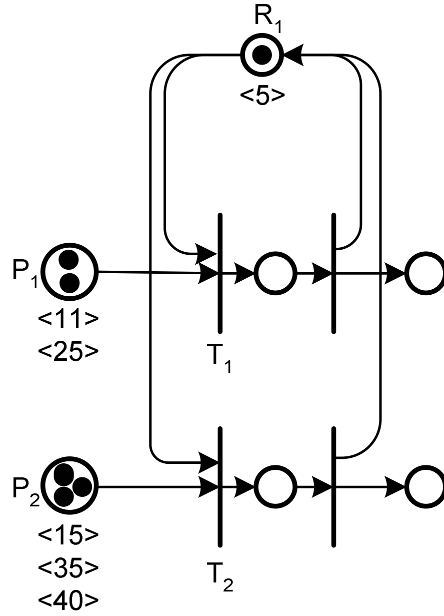

Figure 21: Timing Example

First the variable *t.enabled* is set to true. Then the algorithm iterates over the set of input places of the transition. For each place p in that set there are two cases that the algorithm has to check: $w(p,t) = 1$ (the arc weight from the input place p to transition t is 1) and $w(p,t) > 1$. The tokens in *p.tokens* are ordered according to their timestamps. The first element will have the smallest timestamp. If $w(p,t) = 1$, the value of this timestamp is read and stored temporarily in the local variable *timeStamp*. If $w(p,t) > 1$, the algorithm will iterate over the first $w(p,t)$ tokens in *p.tokens*. If $|p.tokens| < w(p,t)$, the transition cannot be enabled; hence, the loop will be aborted with *t.enabled* being set to *false*. Clearly, this algorithm is sufficient if there are no (conflicting) transitions, i.e., transitions that share an input place, or if the firing times are all at distinct points in time. Since this cannot be guaranteed for most systems, a conflict resolution mechanism needs to be implemented, as is discussed in Section 3.3.5.5.

Update Time (Transition t)

```
 1  t.enabled ← true
 2  maxTimeStamp ← 0
 3  timeStamp ← 0
 4  for each place p ∈ •t
 5        do
 6            if w(p, t) = 1
 7               then if |p.tokens| > 0
 8                         then timeStamp = timestamp of first token in p.tokens
 9                         else  t.enabled ← false
10                               break ▷ transition not enabled
11               else  if |p.tokens| ≥ w(p, t)
12                         then count ← 0
13                               for each token T ∈ p.tokens
14                                     do
15                                         timestamp = T.timestamp
16                                         count ← count + 1
17                                         if count = w(p, t)
18                                            then break ▷ leave inner loop
19                         else  t.enabled = false
20                               break ▷ leave outer loop, transition not enabled
21            if timeStamp ≥ maxTimeStamp ▷ find maximum timestamp
22               then maxTimeStamp ← timeStamp
23  if t.enabled = true
24     then t.time ← maxTimeStamp
25            add t to FEL
```

3.3.5.3 Transition Firing

The previous algorithms determine the eligibility and time of a transition to fire. When a transition t actually fires the following steps are undertaken:

- Remove tokens from 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 because their attributes contain 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 amount 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. Recall that the delay time can be either deterministic or it can be a random variate generated according to the distribution specified in the field `rvType` of transition t. The `JobToken` object is then added to the place that is specified in the field $t.outJob$. Further, 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 will be generated and added to the place.

After a transition fires, all affected transitions have to be updated since the firing transition removes tokens from its input places and deposits new tokens in its output places. 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, in accordance with the enabling rule. Places that received tokens can enable their output transitions, and

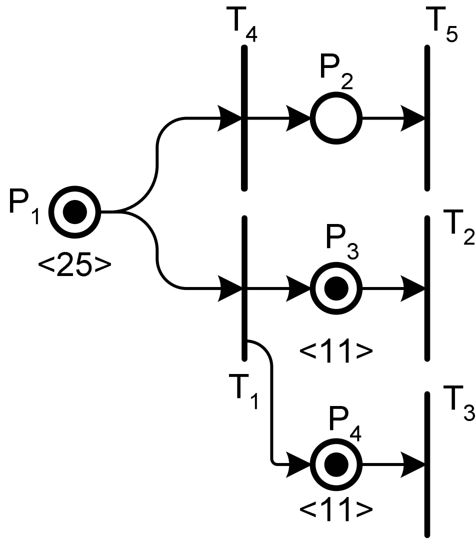

Figure 22: Example for Updating Time

places that have tokens removed might disable their output transitions.

Example 4: The PN in Figure 22 depicts a situation where transition T_1 has just fired. The current simulation time is 11, since this is the timestamp value of the tokens in the output places P_3 and P_4. Transitions T_2 and T_3 each have an input place that is also an output place of transition T_1. Therefore, they need to be updated. T_4 also needs to be checked, as it is in conflict with transition T_1 (i.e., it shares the input place P_1). Note that it is not necessary to check transition T_5 since it cannot possibly become enabled or disabled when T_1 fires. Finally, the firing transition itself obviously needs to be updated as it removes a token from its input place. $\diamondsuit$

The set of transitions that need to be updated each time a transition fires is stored in the field `toUpdate`. For transition T_1 in Example 4, this field has the transitions $\{T_2, T_3, T_4\}$. The algorithm to determine this set is as follows:

DETERMINE SET TOUPDATE (TRANSITION t)

1 **for** each place $p \in {}^{\bullet}t$

2   **do for** each transition $t' \in p^{\bullet}$

3   add t' to $t.toUpdate$

4 **for** each place $p \in t^{\bullet}$

5   **do for** each transition $t' \in p^{\bullet}$

6   add t' to $t.toUpdate$

When updating the time after a transition t fires, all transitions in the set $t.toUpdate$ will be updated according to Algorithm UPDATE TIME. Prior to that instance, the transition is removed from the FEL if it is enabled. Once the time is updated and the transition enabled, it will be added to the FEL again in order to ensure proper ordering of the list.

Under this timing scheme, new events (i.e., newly enabled transitions) will always have enabling times that are greater than or equal to the last firing time. This will ensure the proper temporal firing sequence of transitions. The above mechanism will not enable transitions with timestamps in the past. This can be expressed with the following proposition:

Proposition 1. *The algorithms for updating time and firing transitions ensure that no transitions will become enabled with enabling time less than the current simulation time.*

Proof. The current simulation time t_{now} is equal to the firing time of the last transition t that fired. As the FEL is ordered according to enabling time of the transitions, all other enabled transitions have a timestamp greater than or equal to t_{now}. Transition t might enable other transitions. All tokens that are generated by the firing of t will have a timestamp with value at least t_{now}. When one of the tokens that were sent by t is received by a place p, the place can be either empty or it can already contain other tokens. If it is empty and a token is added, this might lead to enabling of a transition in its output transition set. The firing time of this transition can never be smaller than the timestamp t_{now} of that token because the update time algorithm checks every place for the timestamp of the tokens and determines the maximum timestamp. On the other hand, if the place already contained a token, then some or all of the transitions in its output set might already have been enabled. Transitions

that already were enabled will stay enabled, whereas transitions that have not been enabled might become enabled, i.e., if the place was of the type `BatchPlace` (as this type is used to model arc weights that are greater than one). If a transition becomes enabled, the algorithm UPDATE TIME determines the maximum timestamp value of the enabling token. This value will be the enabling time. Therefore, it is always ensured that transitions will always fire in a non-decreasing order of their enabling times. □

3.3.5.4 Execution of Petri Net Model

The execution of the PN model proceeds as follows. After an initial scan of all transitions, the FEL will contain all currently enabled transitions. Then the simulation will enter a loop that will remove the first transition from the FEL at every iteration until the FEL is empty or the termination time has been reached. The main loop looks as follows:

RUN SIMULATION

1 **while** FEL $= \emptyset$

2 **do** t = first element of FEL

3 **if** $t.time > simulationEndTime$

4 **then** break ▷ simulation end

5 **else** $t.fire()$

This algorithm is very similar to the standard timing routine for any discrete-event simulation. Note that it is not necessary to have a global timing variable and time advance mechanism for this variable, as the current time can always be read from the firing time of the last transition.

3.3.5.5 Conflict Resolution

If the firing times of all transitions are unique, there will not be conflicts amongst transitions. Depending on the network structure, the processing sequence of transitions with exactly the same enabling time might not be important. However, the processing sequence will be important for transitions that are in conflict with each other. In that case, the order can have an important effect on subsequent events, as a transition that fires will usually

enable or disable other transitions. The classical approach to ensure a certain order of firing transitions that are in conflict is to use inhibitor arcs. This can be done for each pair of transitions that are in conflict. This is a static solution that is permanently encoded in the PN.

In order to have unique ordering in or setting, transitions have a field `priority`, which takes on integer values. Transitions in the FEL are then ordered first according to their firing time and secondly according to their priority. If the priorities are also equal, then they will be sorted according to their identifier in the field `id`, to ensure an unambiguous ordering.

Timing Mechanism The timing mechanism is essentially the same as before. If a transition is enabled and the firing time has been determined, then the priority of that transition also has to be determined. For this purpose the method `setPriority` is called.

Set Priority The priority is set based on the token field `priority` of the `JobToken` object. This allows specifying higher priorities to certain jobs in order to allow them to seize resources before lower priority jobs. Priorities are only determined by the places that hold objects of type `JobToken`. These places are stored in the field `inJob` of the `Transition` object. The algorithm for calculating the priority of a transition t is as follows:

SET PRIORITY (TRANSITION)

1 $priority \leftarrow Long.MIN_VALUE$ ▷ set priority to the lowest possible value first
2 **for** each token $T \in inJob.tokens$
3   **do if** $(T.timeStamp \leq t.time) \wedge (T.priority > priority)$
4   **then** $priority \leftarrow T.priority$

The object `inJob` refers to the input place of transition t that holds the tokens representing jobs. The algorithm iterates over all tokens in that place and finds the token with the maximum priority and a timestamp less than or equal to the firing time of the transition.

Remove Token When a transition fires, it removes tokens from the respective input places. For a classical PN, tokens are just represented as an integer count, and hence, removing a token means simply decreasing a counter by one. This is not applicable to the proposed framework as every token is represented as an object. This also means that tokens that are removed from a place need to be uniquely determined. The algorithm for removing a single token from a place p works as follows:

Remove Single Token from Place p

1 $S = \emptyset$
2 **for** each token $T \in p.tokens$
3 **do if** $T.timestamp \leq$ firing time of transition
4 **then** add T to S
5 find token T' with highest priority in S
6 remove T' from $p.tokens$

There are two parts to the algorithm. First all tokens with timestamps that are less or equal to the firing time of the transition are added to a set S. Then the token with the highest priority in that set is chosen. This token is removed from the original set of tokens of place p. For places that are connected to transition with arc weight n, the algorithm for removing these tokens is:

REMOVE n TOKENS FROM PLACE P

```
1   S = ∅
2   for each token T ∈ p.tokens
3       do if T.timestamp ≤ firing time of transition
4           then add T to S
5   sort S according to priority of token
6   count ← 0
7   for each Token T ∈ S
8       do count ← count + 1
9          remove T′ from p.tokens
10         remove T′ from S
11         if count = n
12           then break
```

The first part is identical to algorithm REMOVE SINGLE TOKEN FROM PLACE P. The set S is ordered according to the priorities of the tokens. The second part iterates n times over the set S, each time removing a token from *p.tokens*. This in effect will remove n tokens with the highest priorities from the place.

3.3.5.6 Implementation of Dispatch Rules

A classical PN cannot model directly any dispatch rules since tokens are only represented as integer values. (Even the simplest FIFO dispatch rule cannot be modeled.) The use of the concept of priority allows the implementations of different dispatch rules. The FIFO rule is implemented by assigning a priority equal to the negative of the execution time to the `JobToken` object when it is removed from the input place. When a transition t fires the FIFO dispatch rule is applied as follows:

FIFO DISPATCH RULE

```
1  remove token T from t.inJob
2  T.priority = – firing time of transition t
3  add T to t.outjob
```

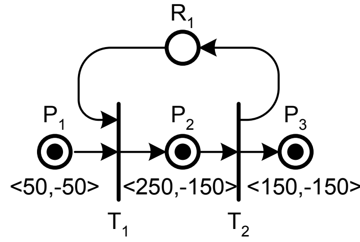

Figure 23: First FIFO Example

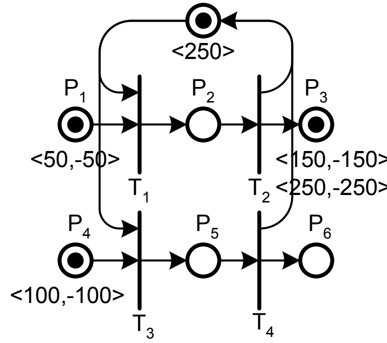

Figure 24: Second FIFO Example

Example 5: Figure 23 illustrates a simple processing step that seizes a single resource. The processing time is a 100 time units; hence, the delay time at transition T_1 is 100. The values for `timestamp` and `priority` are given as $\langle timestamp, priority \rangle$. Place P_1 represents the queue that is waiting for resource R_1. Place P_2 represents the processing state, and place P_3 represents the buffer after processing is finished. The current simulation time is 100. The token in P_1 has `timestamp` $= 50$ and `priority` $= -50$, indicating that this job arrived to P_1 at time 50. Place P_2 holds a token with `timestamp` $= 250$ and `priority` $= -150$. This corresponds to a job whose processing was started at time 150 and will finish at time 250. The priority is set to -150 since that was the firing time of transition T_1. $\diamondsuit$

Example 6: Figure 24 shows two parallel process steps that compete for the same resource, R_1. P_1 and P_2 both hold one token, which means that there are two jobs waiting for R_1. The transitions T_1 and T_3 are currently enabled at time 250. Since they are in conflict which each other, their ordering in the event list is such that T_1 will fire before T_3, because the token priority in P_1 is -50 versus -100 for the token in P_4. This enforces the FIFO

rule as the token in P_1 arrived before the token in P_4. $\diamondsuit$

Another possibility is to assign fixed priorities to job tokens. With fixed priorities, it is possible to implement a dispatch rule such as earliest due date (EDD). Priorities are assigned according to the following rule when a transition t fires:

EDD Dispatch Rule

1 remove token T from $t.inJob$

2 $T.priority = -$due date

3 add T to $t.outjob$

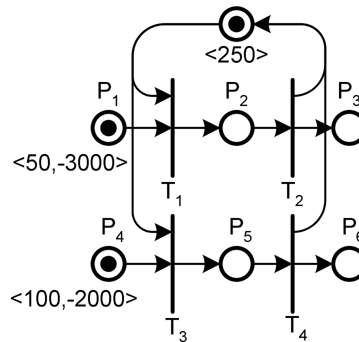

Figure 25: Example with Fixed Assigned Priorities

Example 7: Figure 25 shows the same PN as Figure 24, but now the job in place P_1 has a due date of 3000 and the job in P_2 has a due date of 2000. Both transitions T_1 and T_2 are currently enabled. As the due date of the job in P_4 is earlier, transition T_3 will fire first. Priorities do not have to be manipulated when a token is removed from a place as they will stay constant in this case. $\diamondsuit$

Other dispatch rules that require updating the priorities each time a transition fires are also possible to model. The shortest processing time remaining (SPTR) dispatch rule can be applied as follows when transition t fires:

SPTR Dispatch Rule

1 remove token T from $t.inJob$

2 $T.priority = -$processing time remaining

3 add T to $t.outjob$

Further, combined rules can be implemented. For example, jobs that follow a FIFO rule will always have a negative priority assigned. This can be combined with high priority jobs, by assigning them a positive priority value. This way these jobs will always be preferred, yet the FIFO order of the other jobs will not be altered.

3.3.6 World View of the Proposed Framework

The proposed framework does not follow any specific simulation world view. As with every discrete-event simulation, it also follows the event-scheduling approach at a low level. Firing transitions correspond to event executions. After a transition fires, other transitions can become enabled, i.e., new events will be scheduled. The framework also has aspects of existing major world views. For instance, it can describe the movement of an entity through a manufacturing system similar to the process-interaction world view. The fields `inJob` and `outJob` of the transition class are used for this purpose. Also, elements of activity scanning are present as the execution or firing of a transition will cause a state change of the system; this state change causes a scanning of all the affected transitions.

3.3.7 Advantages of the Proposed Framework

One of the main advantages of the framework is that it is not necessary to code a simulation model directly in a simulation language. The simulation is modeled in terms of a PN that resides as a data structure in main memory. Coding is only necessary to specify the scripts that will generate the PN. This PN is also the conceptual model of the simulated system; therefore, the translation step from the conceptual model to the computer model is eliminated reducing potential implementation errors. Only a few core elements are used in the framework, which also reduces potential programming errors. Another significant advantage is that the behavior of the simulation model can be changed during a simulation run between the executions of events. For instance, it is possible to augment the PN with

additional transitions and places in order to model new process routes. In other simulation languages the simulation model is coded and compiled or interpreted. This means that any changes in the simulation model have to be coded first and then compiled. Hence it is not possible to alter a simulation model once it is loaded into memory.

3.3.8 Limitations of the Proposed Framework

The limitations of this framework are essentially the same as for all discrete-event systems. Only certain points in time can be captured, usually when the system state changes, i.e., event occurrences. Hence the framework does not provide any mechanism to model any continuous-time systems. This is usually not a problem as most systems in the IE domain are discrete-event systems. This might be different for the simulation of the actual physical manufacturing process, such as milling. However, the continuous nature of time can be captured, as the base time unit can be chosen arbitrarily small. As we stated earlier, the framework has limited support for complex material handling systems such as AGVs or conveyors.

CHAPTER IV

SEMICONDUCTOR MANUFACTURING SIMULATION DATA SPECIFICATION

This chapter introduces a specification for the data that is used to generate the PN-based simulation model. According to the Webster's Dictionary [3] a specification is "a detailed precise presentation of something or of a plan or proposal for something." In this case it refers to the detailed description of a manufacturing system for the purpose of simulation.

Here the term *Simulation Data Specification* will refer to the precise description of the data that will be used to generate the simulation model. There exist very few data specifications for discrete-event simulations in the literature. One of the few examples can be found in [28, 27], where the NIST XML simulation interface specification is used. This specification was first introduced first by [32], and is still under development.

The actual Sematech data set represents already 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.

4.1 Semiconductor Wafer Fabrication

Semiconductor wafer fabrication is considered to be one of the most complex and capital intensive manufacturing processes [44]. It involves several hundred processing steps. The number of operations that have to be carried out exceeds the number of available machines. This forces wafers to visit the same machines more than one time (re-entrant lines). In addition, some wafers have to go through rework processes, which adds even more complexity.

Wafer fabs can be described as complex job shops [29]. They contain job-specific re-entrant flows across a number of unique tool groups with multiple, identical machines operating in parallel. A re-entrant flow is characterized by jobs that require often repeated processing by the same tool along their process route. Some tools process jobs in batches, while others require sequence-dependent setups. Wafers are grouped in lots, which follow

specific process routes. Each wafer in a lot will follow the same sequence. However, there can be different lots in the wafer fab, which follow different routes through the system. Some tools in the wafer fab can be loaded with a single lot, whereas others will accept batches of several lots. For example, several wafer lots will be batched in a furnace that can hold 200 wafers [44].

4.2 Sematech Data Set

The Sematech data set describes several semiconductor fabrication facilities and is available from the Modeling and Analysis for Semiconductor Manufacturing Laboratory website [1]. It is available to the public in order to assist the evaluation and comparison of simulation tools, analytical tools, and control strategies. The Sematech data set provides researchers and practitioners with actual data that can be used to benchmark control strategies and software.

The data set consist of seven sets, each describing a different semiconductor facility. Every set consists of several files describing the process routes, rework sequence, tool sets, operator sets, and release rates. The *process route* and *rework sequence* files describe each processing step and all required tools and operators for each step. Further, all relevant time information is given: loading times, unloading times, setup times, and processing times. Batch sizes, scrap probabilities, and rework probabilities are also provided. The *tool set* file describes each tool set and available quantity as well as down times for each tool set. The *operator set* file specifies the operators, available quantities, and their break times. The *release* file specifies the release rate and the lot size for each product of the facility.

4.3 Simulation Data Specification

This simulation data specification describes how the elements of the semiconductor manufacturing systems are represented. This approach differs from what commercial simulation software uses. Simulation models that are created in commercial software can only be saved in a format specific to the simulation software package. This format or data specification is merely a way to store the elements and modules that are used in the simulation model. The data specification that is presented here refers to the actual physical simulation system.

The simulation model is automatically generated from this specification.

This approach has several advantages because the data is represented in the manufacturing 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, and Operator Set.

4.3.1 Fabmodel

This class represents the root for all the other elements, i.e., it contains all the other objects for the simulation model. There is exactly one instance of this type of object for each data set.

4.3.2 Product

This class is used to represent product data, such as `id`, product name, and release rate.

4.3.3 Process Route

This class holds all the information of one specific process route. It contains all the process steps that the route consists of. There is exactly one process route for each product produced by the wafer fab.

4.3.4 Process Step

This 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 models 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 steps that can batch lots together and process them at the same time. Lots from different process routes can be batched together if the

`batchId` field is 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 simultaneously.

The Process Step with Setup is used to model steps that require a setup of a 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 belongs to a different setup group.

4.3.5 Tool Set

The tool set 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 is 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 setup states. Downtimes are also listed in the field downtimes with description, duration, and times between failures.

4.3.6 Operator Set

The operator set class describes the operators that are needed to operate the tool sets in the waferfab. It has fields for `id`, `description`, and `quantity`. If `quantity` is greater than one, all of the operators in the set are considered identical.

4.3.7 Rework Sequence

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, then 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 and enters it at a step that can be located after or before the process step where the lot left the original route.

4.3.8 Representation of Control Policies

The Sematech data set does not contain any data for control policies. It merely provides release rates for each product, i.e., a pure push policy is assumed.

The data is stored as an XML file. Appendix A.3 lists the complete XML schema. The advantage of XML is that it has become an industry standard and parsers are available that can automatically read such files. The following class diagram shows the relationships between all the elements. Some objects implement the `Comparable` interface. This is required to provide a unique order for the objects that are stored in sets and lists, and is mentioned here only to describe the object model completely.

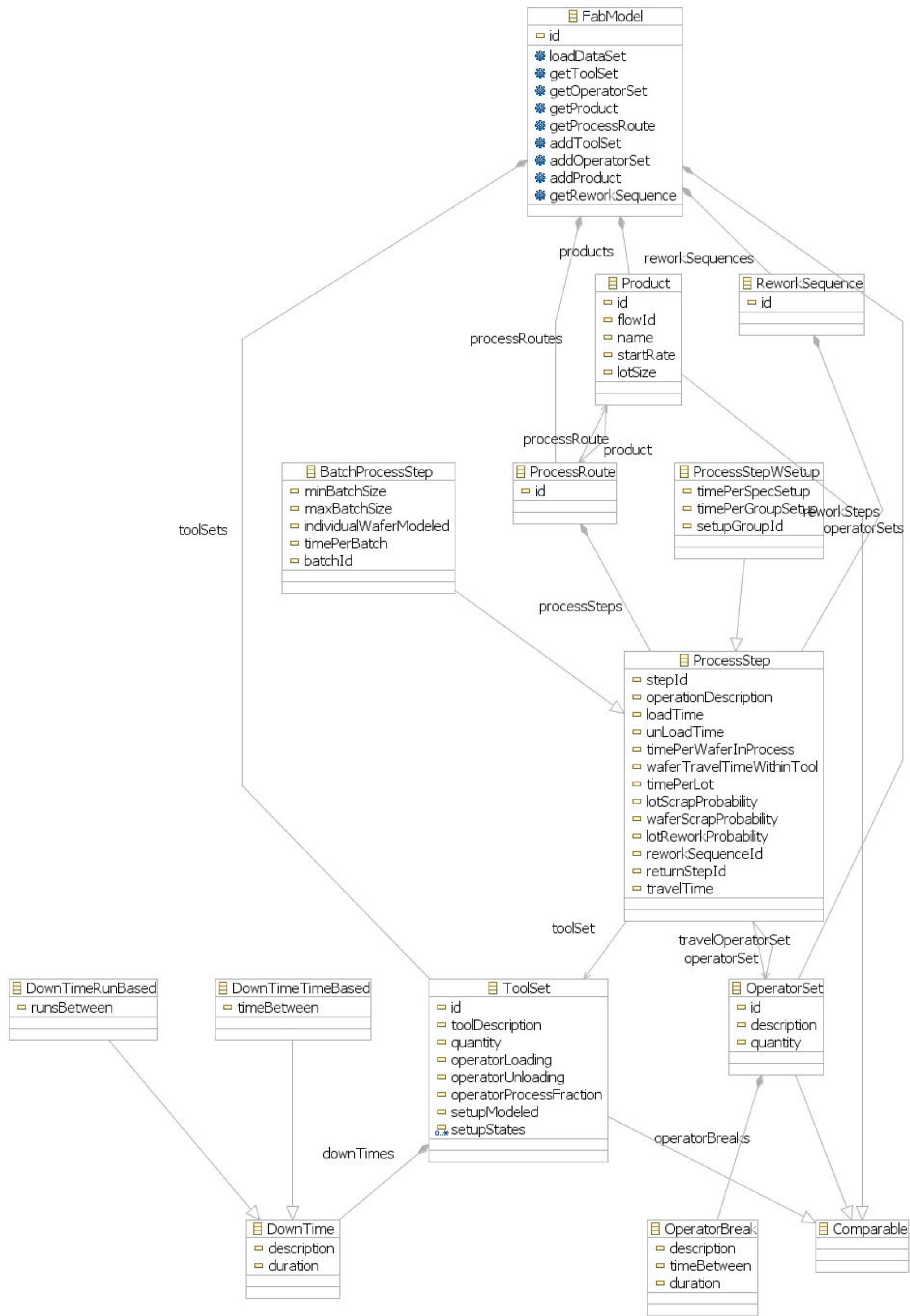

Figure 26: Object Model of Manufacturing System Specification

CHAPTER V

SIMULATION MODEL GENERATION

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

This chapter deals with the fundamental problem of generating a simulation model based on a given specification for a semiconductor manufacturing system. The resulting simulation model should have certain properties, e.g., absence of deadlocks and boundedness. The dynamic behavior of the simulation model is based on a PN, as described in Section 3.3.

5.1 Considerations for the Generation of Simulation Models Based on Petri Nets

Traditionally, a PN is first built for a specific system, and then traditional analytical methods are used to analyze its properties such as boundedness and liveness. This approach can work well for small manufacturing systems. For larger systems the state space explosion problem complicates substantially the analysis of the respective PNs via traditional methods. For certain special classes of PNs, polynomial time algorithms for deadlock avoidance exist. The analysis of certain classes of resource allocation problems are examples of this approach [40].

The previous discussion motivates the following fundamental problem for this research: *Given the specifications of a manufacturing system, model the system as a PN such that its*

structure and initial markings make it live, bounded, and reversible.

Liveness is important for the modeling of shared resources, since they are a potential source of deadlocks. A simple example can illustrate this: A resource that is seized by one process cannot be released until another resource is released by another process and vice versa. Hence, the proper modeling of mutual exclusion becomes essential. For liveness, it will be required that every transition in the PN will be L4-live. Recall that an L4-live transition can potentially fire infinitely often (Section 3.2.5). A lower level of liveness means that the transition might fire but can become eventually dead. Hereafter, the term “live” will be used in place of “L4-live”.

Boundedness is desirable because every finite system can only have a finite number of entities. It may not be required in a strict sense as a standard queuing systems are unbounded. Reversibility is required because the system should be able to return to the initial state.

5.2 Mapping of Fabmodel Elements to Petri Net Simulation Model

This section discusses how each of the elements of the fabmodel described in Chapter 4 are represented within the PN. This representation forms the basis for the algorithms that generate the PN simulation model.

5.2.1 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 represent 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.

5.2.2 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 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.

5.2.3 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. For the detailed description of how the process steps are generated, see Section 5.2.4.1.

5.2.4 Process Steps

There are three main types of process steps (see Section 4.3.4): the Basic Process Step, the Batch Process Step, and the 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. The following subsections will describe these mappings in detail.

5.2.4.1 Basic Process Step

Figure 27 shows the Basic Process Step in its simplest form. P_1 is the input place. A `jobToken` will first arrive here. P_2 represents processing and 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. Transition 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 28 shows the Basic Process Step with an extension to model transportation within the tool. The first part is identical to Figure 27. After processing, the wafer lot has to be transported within the tool. Since the transportation process does not require any

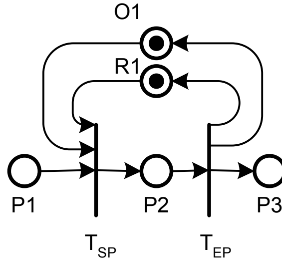

Figure 27: Basic Process Step

resources, it can be simplified as a simple time delay. Transition T_M will add this delay to the token in P_3 and deposit it in P_4.

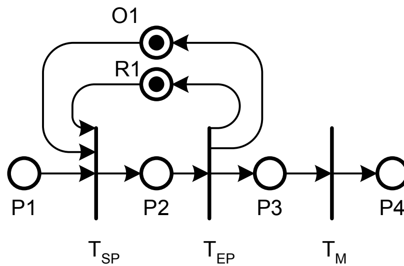

Figure 28: Basic Process Step with In-Tool Travel Time

Figure 29 shows the Basic Process Step with partial operator processing. The Sematech data set specifies an operator process fraction, which is the fraction of time the operator is needed for lot processing. A fraction of one means that the operator is needed for the entire time. If the fraction is greater then zero and less then one, processing is split into two parts: first, the lot is processed with the operator for the proper fraction of time and then the lot is processed without him/her. Transition T_{SP} represents the start of lot processing with the operator. One operator and one tool are required in order to start. Transition T_{EPO} represents the end of processing with the operator. At that time the operator is released by putting a token back into the operator set place.

Figure 30 shows the Basic Process Step with operator loading. Transition T_{SL} represents the start of the loading process. Transition T_{SP} marks the end of the loading and the beginning of the processing of the lot. In this example, the operator is used throughout until processing has finished.

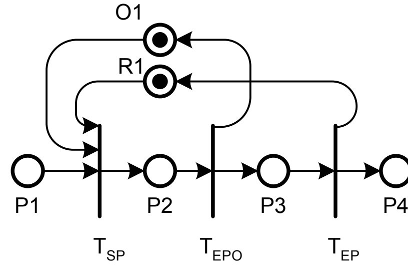

Figure 29: Basic Process Step with Partial Operator Processing

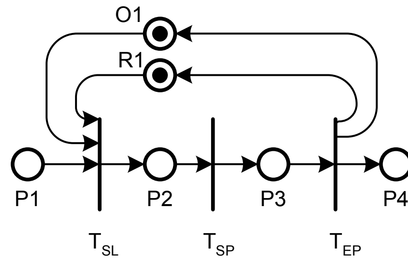

Figure 30: Basic Process Step with Operator Loading

Figure 31 shows the Basic Process Step with operator loading and unloading. This is similar to Figure 30 with the addition of Transition T_{EL}. Transition T_{EP} marks the end of processing and the beginning of the unloading step, and T_{EL} marks the end of unloading. The operator is used for entire time for loading, processing, and unloading.

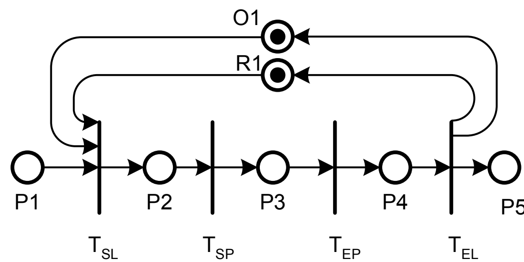

Figure 31: Basic Process Step with Operator Loading and Unloading

Figure 32 shows the Basic Process Step with travel time to next tool. For some process steps, transportation to the next tool is explicitly modeled. After the processing phase is finished, the job token will be in place P_3. Transition T_{ST} then seizes the required operator token for transportation and Transition T_{ET} releases this operator token at the end of the

transport.

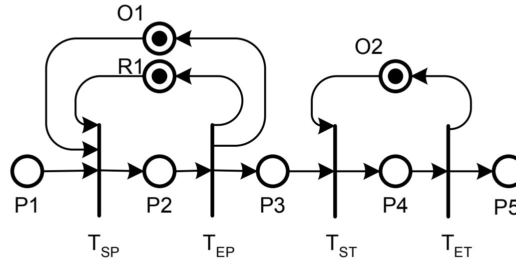

Figure 32: Basic Process Step with Travel Time to Next Tool

5.2.4.2 Set Priority

Figure 33 shows the Basic Process Step with a set priory transition. For each process step, there is a transition at the beginning of the step that will set the priority for the token to the appropriate value. Different dispatch rules (see Section 5.2.8) can be implemented by assigning the appropriate value to the job token at T_P. This transition is present at the start of each process step, but is omitted in most illustrations. The other transitions of the process step will keep the assigned priority value constant.

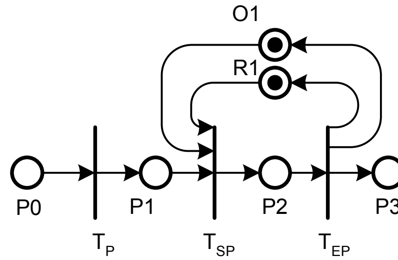

Figure 33: Basic Process Step, Set Priority

Figure 34 shows the Basic Process Step with all options discussed above. Place P_0 represents the input place for the process step. Transition T_P will set the priority of the job token. Transition T_{SL} marks the start of the loading process, and Transition T_{SP} represents the end of the loading and the beginning of the processing with operator. Transition T_{EPO} corresponds to the end of the processing with operator, and releases the operator seized by Transition T_{SL}. Transition T_{EP} denotes the end of processing and the beginning of the

unloading process — it seizes the operator again. Transition T_{EU} represents the end of the unloading process and releases all resources. Transition T_M represents the travel time within the tool. Transition 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_M, and T_{ST} all add the appropriate times to the token timestamp, to represent the respective time delays.

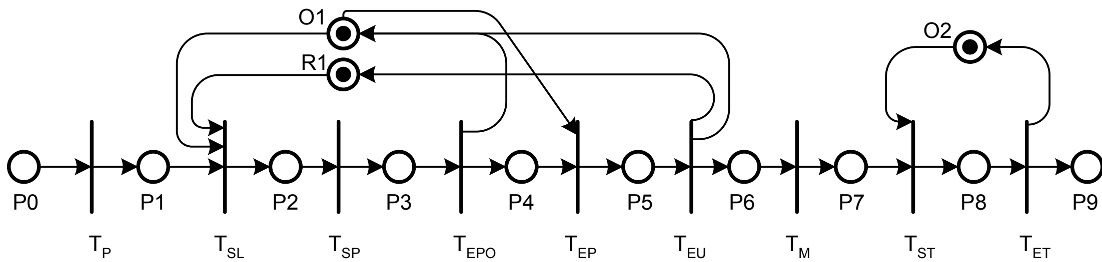

Figure 34: Basic Process Step with all Options

5.2.4.3 Batch Process Step

This section will introduce the Batch Process Step, which 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 (see also Appendix A.1). For a given resource, all lots from process steps that use this resource and have identical `batchId` fields, can be batched together.

A naive version of modeling a Batch Process Step could look like Figure 35, which shows a Batch Process Step with a batch size of four wafer lots. Whenever four job tokens are present in P_1, they are consumed at once by T_{SP} when processing starts. At the end of processing, T_{EP} puts four job tokens back into P_3.

The problem with this approach is that it only allows for modeling Batch Process Steps with a fixed size. However, the data sets specify minimum and maximum batch sizes. This makes it very hard to model it in a straightforward manner. This reason for this is as

follows: when the minimum number of wafer lots is present, processing can begin. However, 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 sizes. Further, the number of events that start the processing of the lots can be different.

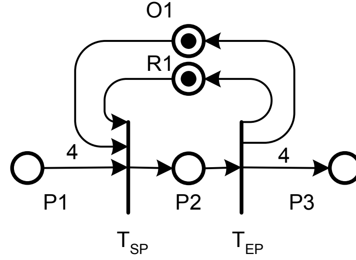

Figure 35: Naive Batch Process Step

Figure 36 shows a rather simple Batch Process Step that addresses the most basic version of this problem. 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 $\{P_1, T_{S_1}, P_2, T_{S_2}, P_3, T_{EP}, P_4\}$, where P_1 is the arrival place and P_4 is the end of processing place. P_3 represents the processing stage and P_2 is an intermediate place needed to model the batch mechanism. Initially, the control place C holds two tokens. This represents the minimum batch size. Lots that arrive in P_1 will trigger the firing of T_{S_1}. After two firings, T_{S_1} will be disabled, as the two tokens in C will be consumed. These firings of T_{S_1} will place two tokens in B_1. B_1 then triggers T_{B_1}, which will seize the resource R_1. When T_{B_1} fires, it will consume the two tokens in B_1 and will place two tokens back into C and B_{10}. Further, it will place three tokens in L and one in B_2. This represents the "leftover" capacity, i.e., the difference between maximum and minimum capacity. These tokens enable T_{P_1} and allow additional lots to enter the processing stage P_3. If there are no lots waiting in P_1, the tokens in place L are consumed by T_{D_2}. The tokens in B_{10} enable T_{S_2}, so that the tokens in P_2 can enter P_3. At this stage all lots that are processed are in P_3.

The token in B_2 enables transition T_{B_2}, which is responsible to add the proper delay time representing the processing time and then move the token to B_3. This in turn enables T_{B_3}, which will add a token back to R_1 (thereby releasing the tool), and then it will add

a number of tokens equal to the maximum batch size to B_{20}. These tokens will have timestamps equal to the release time of the resource, and will enable T_{EP}. Transition T_{EP} will then move the token to P_4, which marks the end of processing. If the number of lots processed was less than the maximum batch size, there would be tokens leftover in B_{20}, which will be removed by T_{D_1}.

In order for this mechanism to work, it is important to follow a specific firing order of the enabled transitions. Figures 75 and 76 in Chapter 6 show a detailed analysis. The transitions T_{B_1}, T_{S_1}, and T_{P_1} are of the normal type `Transition`. Their priority is determined by the `JobTokens` that are waiting to be processed. Transition T_{P_1} will always fire before T_{S_1} in order to avoid "taking over" of lots: lots are prevented by entering P_2 as long as there are tokens available in place L. Both transitions will always have the same priority when enabled, as they share the same `inJob` place, which holds the `JobToken` that determines the priority. The ordering is archived by assigning an identifier to T_{P_1} in such a way that it will occur before T_{S_1}, as time and priority of both transitions will be identical and the `id` field will determine the order.

The transitions T_{B_2}, T_{B_3}, T_{S_2}, and T_{EP} are of the type `FixedMaxPriorityTransition`. This means that when enabled, they take precedence over all other types of transitions with the same timestamp. As these transitions are not in conflict with each other, their firing order is not important.

The transitions T_{D_1} and T_{D_2} are of the type `FixedMinPriorityTransition`, which means that they will only fire after all other enabled transitions with the same timestamp have fired. These transitions "clean up" leftover tokens in L and B_{20} after the processing has started. Otherwise, these tokens would let more lots into the system than the available capacity.

Figure 37 (an augmented version of Figure 36) shows two parallel Batch Process Steps that share the same resource and `batchId`. This means that lots are batched together from different process steps, as they need exactly the same treatment. The additional places are P'_1, P'_2, P'_3, P'_4, and the extra transitions are T'_{S_1}, T'_{S_2}, T'_{EP}, and T'_{P_1}. These places and transitions have the same meaning as before, they only refer to the parallel process step.

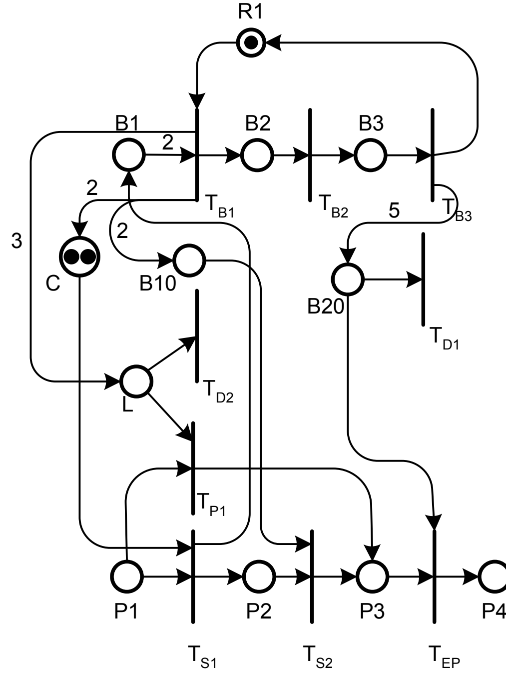

Figure 36: Batch Process Step

The mechanism works in the same way as before. The difference is that lots arriving in P_1 and P'_1 will trigger this mechanism. T_{S_1} and T'_{S_1} each puts a token in B_1 when firing. When the number of tokens in B_1 reaches the minimum batch size, T_{B_1} will be able to fire. At this point in time, the total number of tokens or lots in P_2 and P'_2 will be equal to the minimum batch size. After T_{B_1} fires, a number of tokens equal to the minimum batch size will be placed into B_{10} and a number of tokens equal to the difference between the minimum and maximum batch size is placed into L. This enables transitions T_{S_2} and T'_{S_2}, which will move the tokens in P_2 and P'_2 to the processing stage P_3. The tokens in L then also enable T_{P_1} and T'_{P_1} to move jobs that are waiting in P_1 and P'_1 to P_3 and P'_3, respectively. If there are more jobs waiting then tokens available in L, the ones with the highest priorities are chosen according to the mechanism described previously in Section 3.3.5.5.

This basic structure is used in the same way for n parallel process steps, which share the same `batchId`. Each parallel process step will connect to the places B_1, C, L, B_{10}, and B_{20}. Further, the whole structure is duplicated for process steps with a different `batchId`, but are using the same tool sets and operator sets.

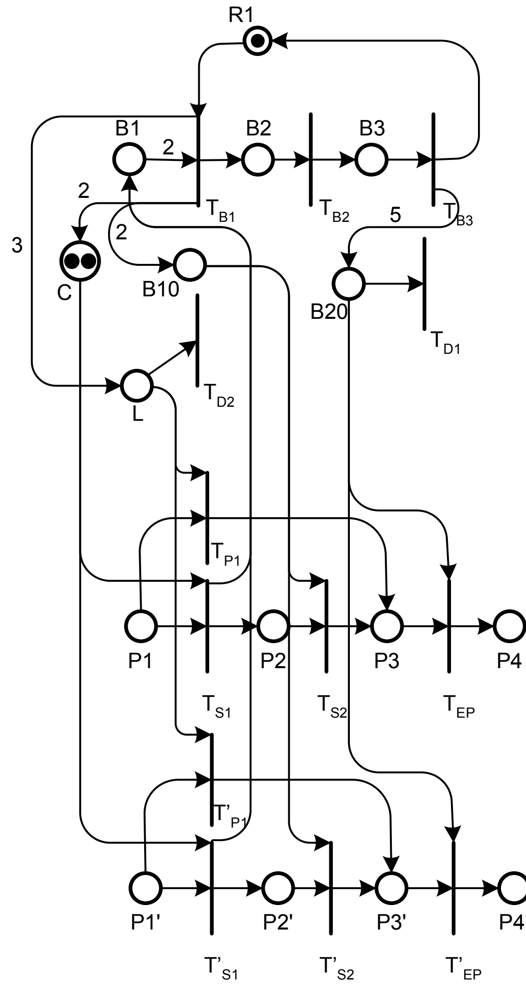

Figure 37: Batch Process Step with Two Process Routes

Figure 38 shows the extension of the basic Batch Process Step for a loading operation before processing. It uses the same basic structure as Figure 36, but is augmented with the places B_4, B_5, B_{30}, P_5, and O_1 and transitions T_{EL}, T_{B_4}, T_{B_5}, and T_{D_3}. The triggering mechanism works exactly the same way as before. The difference is that T_{B_1} represents the onset of the loading operation by operator O_1. Transition T_{B_1} removes a token from O_1 and R_1. The end of loading is represented by transition T_{B_3}, which will release the operator by putting a token back into O_1 and a number of tokens equal to the maximum batch size into B_{20}. This enables transition T_{EL}, which moves the lots to the processing stage, represented by place P_4. Transition T_{B_4} now adds the proper delay that represents the processing time to the token in B_4 and moves it to B_5. Transition T_{B_5} represents the end of processing. It

will put back a token to R_1 and a number of tokens equal to the maximum batch size into B_{30}. This enables T_{EP}, which will move the token from P_4 to P_5.

The transitions on the path $\{B_2, T_{B_2}, B_3, T_{B_3}, B_4, T_{B_4}, B_5, T_{B_5}\}$ have no other input places but the ones on the path. This means that a token in B_2 will trigger all the subsequent transitions that represent the end of loading, start of processing, and end of processing.

Parallel process steps that include loading are modeled in the same way as in Figure 37. The set of places P_1, P_2, P_3, P_4, and P_5 and the set of transitions T_{S_1}, T_{S_2}, T_{P_1}, T_{EL}, and T_{EP} are duplicated and are connected to B_1, C, L, B_1, and B_{20} for each parallel process step.

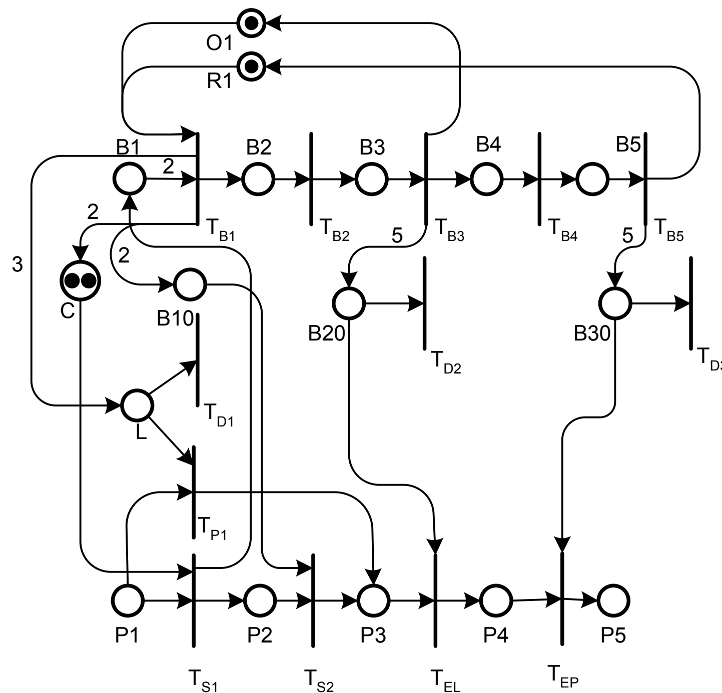

Figure 38: Batch Process Step with Loading

Figure 39 shows the extension of the Basic Process Step for loading before processing and unloading after processing. It is based on Figure 38 with the extension of the places B_6, B_7, B_{40}, and P_6 and transitions T_{B_6}, T_{B_7}, T_{D_4}, and T_{EU}. The meaning of the places is the same as in Figure 38, except for the unloading step, represented by P_5.

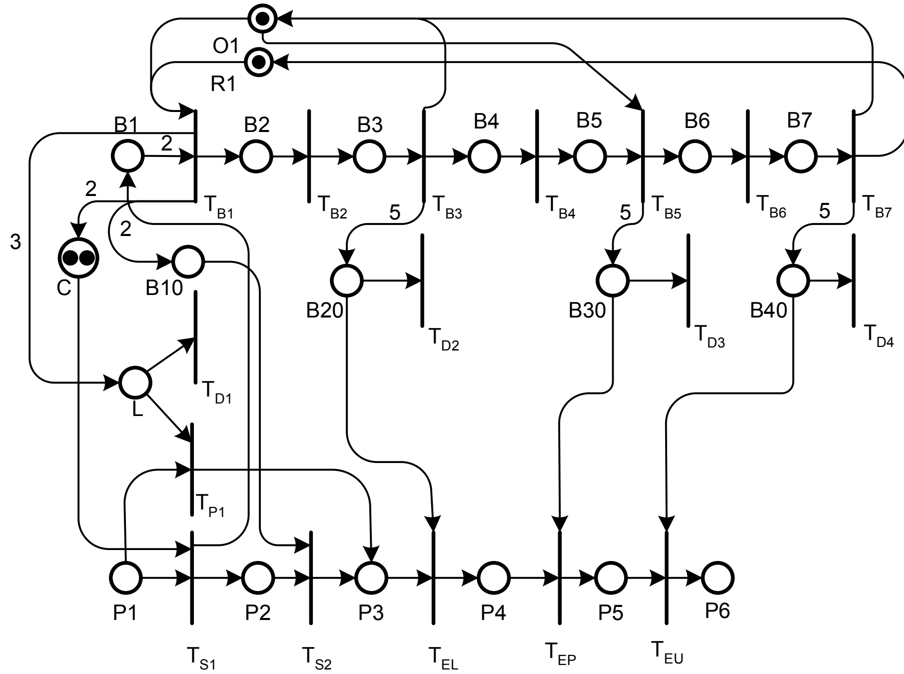

Figure 39: Batch Process Step with Loading and Unloading

5.2.4.4 Modeling of Individual Wafers

Some Batch Process Steps require that each wafer in a lot is modeled individually. This is because the minimum and maximum batch sizes are given as the number of wafers that can be processed. If the number of wafers is not a multiple of the lot size, lots have to be split in order to be processed. In some data sets the difference between maximum and minimum batch size is equal to the $(lotsize)k - 1$, k integer. If this condition is true, individual wafers do not have to be modeled. For example, when the minimum batch size is one, the maximum batch size is 50 and the lot size is always 25, the minimum number of lots that will be processed is one lot consisting of 25 wafers and the maximum lot size is two lots, with 25 wafers each. When the maximum batch size in wafers is less than the lot size, the lot has to be split and therefore the individual wafers have to be modeled. Figure 40 shows an example where lots with 25 wafers are "disassembled" by transition T_{S_B} when entering the Batch Process Step. At the end of the Batch Process Step, the lot is put together again.

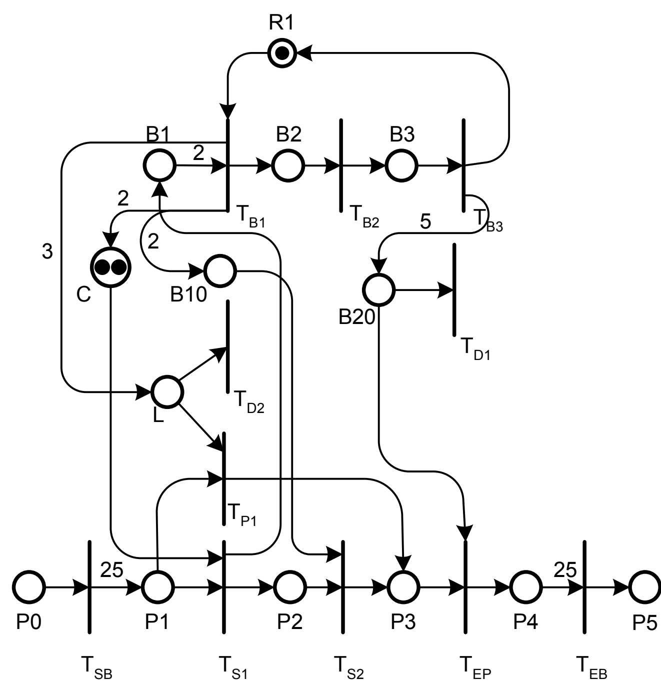

Figure 40: Batch Process Step with Individual Wafers Modeled

5.2.4.5 Process Step with Setup

The last type of process step involves resources that require a setup. These resources have to be modeled in such a way that it is possible to determine the setup state of the resource when processing starts. Figure 41 shows an example for this type of process step. For each setup state there is a resource place, as described in Section 5.2.1, here R_1S_1, R_1S_2, and R_1S_3. The Sematech data sets specify times for specific setups as well as times for group setups. Each setup group represents a setup state. The setup starts with either of T_{S_1}, T_{S_2}, or T_{S_3}. Transition T_{S_i} is used for setup state i, $i = 1, 2, 3$. The setup times can differ for each setup state. If the resource is in the same setup state as required, the setup time is given as `timePerSpecSetup` in the `ProcessStepWithSetup` object. This means that no group setup is required. If the resource is in a different setup state than needed, the setup time is given as `timePerGroupSetup` + `timePerSpecSetup`.

In Figure 41 the resource is currently in setup state 1. This means that the process can only start with T_{S_1}. The delay time for T_{S_1} is `timePerSpecSetup`, and the delay time for each of T_{S_2} and T_{S_3} is `timePerGroupSetup` + `timePerSpecSetup`. P_2S_1, P_2S_2, and P_2S_3 represent the setup operation. The dots under P_2S_3 indicate that there can be more than three setup states. Transition T_{E_1}, T_{E_2}, and T_{E_3} represent the end of the setup operation. These transitions will move the tokens to P_3. From place P_3 on, the PN is equivalent to the Basic Process Step. The only difference is that the resource and operator are already seized for the setup process. Actual processing will start with transition T_{S_P} and end with T_{EP}, which will put back a token into R_1S_1 because the resource is in setup state 1. Note that there are arcs from other process steps to R_1S_2 and R_1S_3, which are not shown in this example.

Figure 42 shows a complete example with two process steps that require different setup states. The upper process step requires setup state 1 and the lower process step requires setup state 2. To start the setup process, transition T_{S_1} or T_{S_2} is used for the upper process step and transition T'_{S_1} or T'_{S_2} is used for the lower process step. For the lower process step, T'_{S_2} represents the shorter setup time, as it uses the token from R_1S_2. At the end of processing, T_{EP} puts back a token into R_1S_1 and T'_{EP} puts back a token into R_1S_2. If there

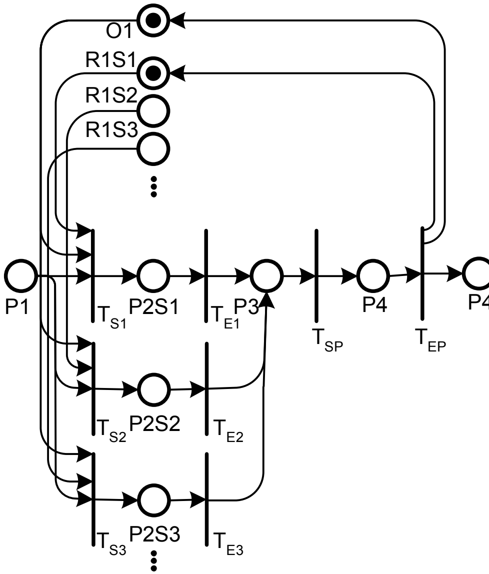

Figure 41: Process Step with Setup

are multiple resources available, it is possible that they are in different setup states. If there is a token available in more than one resource place (e.g., R_1S_1 and R_1S_2), it is possible to start the setup with either one of them. However, it is better to start with a token that corresponds to the required setup state in order to minimize setup time. To implement the rule Shortest Setup First, the transitions have to be ordered accordingly. If two or more transitions that start the setup process are enabled, their priority and timestamps will have the same value. Therefore, they have to be ordered in such a way that the transition that uses the shortest setup time will always come first. This is achieved by assigning an identifier to that transition which guarantees the proper ordering.

Figure 43 shows a Process Step with Setup, loading, unloading, processing, and travel time in a tool. It is essentially identical with the Basic Process Step after the setup is completed in place P_3. For data sets that do not model operators, these process steps are identical except that the operator places are omitted along with all arcs into them.

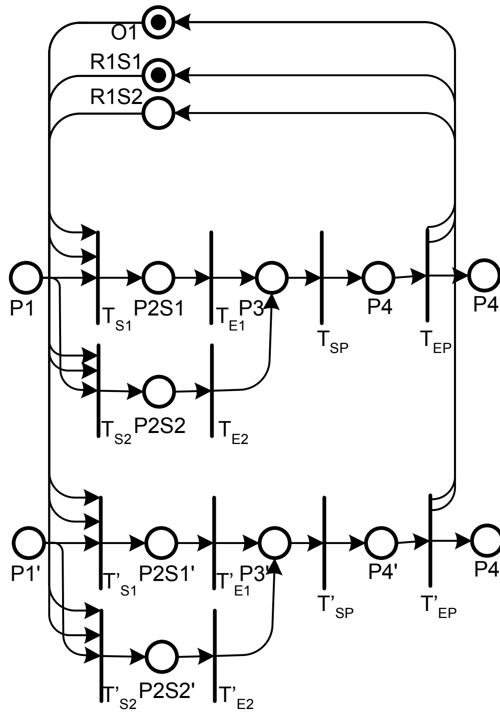

Figure 42: Process Step with Setup, Complete Example

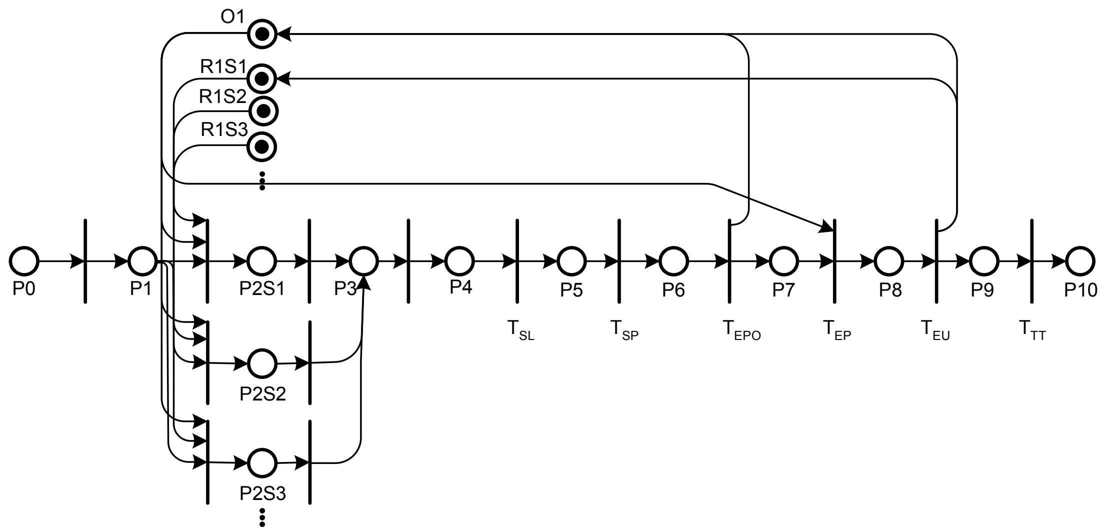

Figure 43: Process Step with Setup and All Options except Travel

5.2.5 Rework Sequence

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 data set specifies for some process steps the probability that a lot has to go through a rework sequence. In Figure 44, T_{SW} represents a "switch" that can route the lot through the rework sequence. The dashed arc from T_{SW} to P_9 is used to indicate this behavior. When the transition fires, it sends the token that represents the lot through the rework sequence, which starts with P_9. In this example, the rework sequence consists of only one step. At the end of processing, the lot is sent back to the original sequence, here place P_6. Depending on what the data set specifies, the point where the lot enters the original process route could be either before or after the process step, where the lot left the original process route.

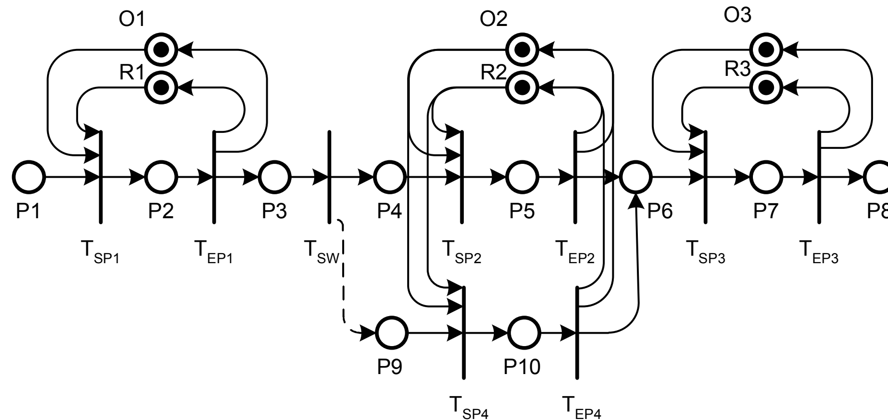

Figure 44: Single Step Rework Sequence

5.2.6 Scrap Modeling

The Sematech data set also contains scrap probabilities for some process steps. This means that it is possible that the processed lot has to be scrapped at the end of a process step. Figure 45 models the mechanism. In this case transition T_{SW} sends a token to P_5 with the scrap probability. P_5 represents a place where all the scrapped lots are accumulated. At the end of a simulation run, all places that hold the scrapped lots can be examined to analyze

the scrap rate. The implementation assumes that a lot has a chance to be scrapped only if it was not routed to a rework sequence. That means that the switch transition has to be at the very end of the process step.

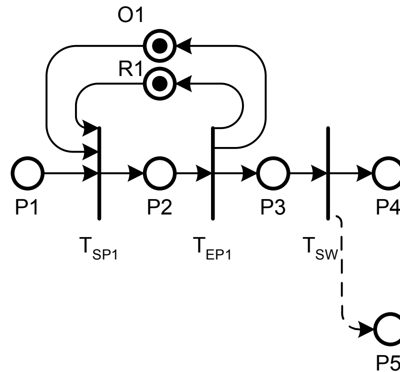

Figure 45: Scrapping of Lot

5.2.7 Modeling of Breakdowns

Breakdowns of tools are modeled as shown in Figure 46. Transition T_B is a transition that will generate tokens according to a Poisson process. Each generated token represents a breakdown of the tool. The token will have a timestamp equal to the onset of the breakdown. The token in B enables transition T_D, which will remove a token from place R. Transition T_D is adding the time duration of the breakdown to the timespamp of the token. This effectively makes the token unavailable until the simulation clock has advanced to the end of the breakdown period.

5.2.7.1 Breakdown Modeling for Resources with Setup States

The modeling of breakdowns for resources that have different setup states is slightly different. Figure 47 illustrates this concept. Note that only the setup states and the breakdown mechanism are displayed. All arcs to process steps that use the resource are not displayed. R_1S_1 and R_1S_2 are the two setup states of resource 1. If there are tokens in place B, which represents the start of a breakdown, they can be consumed by either T_{D_1} or T_{D_2}, which

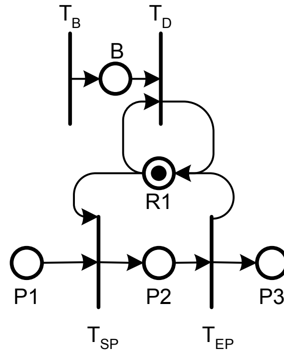

Figure 46: Breakdown Modeling

will generate a breakdown for the resource in the corresponding setup state. This works the same way as the standard breakdown mechanism.

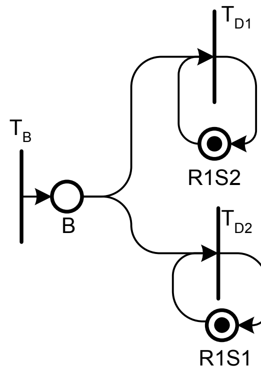

Figure 47: Breakdown Modeling

5.2.8 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. For this simulation framework two cases can be distinguished: only one job type is waiting or several job types are waiting to be processed. Jobs of the same type will wait at the same place in the PN as they follow the same process route. Hence, for these jobs only one transition that tries to seize the needed resource will be enabled. This means that only the job token with the highest priority will

be removed from that place and sent to the processing place. The algorithm that selects the correct token was described in Section 3.3.5.3. If more than one type of job is waiting to be processed, more than one transition is enabled. For each job type there is a process route, and hence a transition that will be in conflict with the other transitions. If the resource is available, all these transitions will be enabled at the same moment in time, necessitating the establishment of an order. This uses the mechanism described in Section 3.3.5.5. The following dispatch rules are implemented within the framework.

- First-In-First-Out (FIFO): the longest waiting job will be processed first
- Shortest Remaining Processing Time: the sum of the processing times of the process steps along the process route that still have to be performed
- Shortest Remaining Processing Time as a Percentage of Total Processing Time: shortest remaining processing time divided by total processing time
- Largest Number of Operations: the job with the largest number of total process steps will be processed first
- Largest Number of Operations Remaining: the job with the largest number of process steps to finish will be processed first
- Shortest Processing Time: the job that has the shortest processing time for the next step will be processed first

Dispatch rules are assigned to specific transitions in the PN simulation model. These transitions are the first of each process step, and will set the job token priority to the appropriate value. The other transitions will not change the value of the priority of the job token. The dispatch rules above should not be mixed, as it does not make sense to assign one dispatch rule to a transition and a different one to another transition. Below we discuss the implementation of these dispatch rules.

FIFO Dispatch Rule The assignment for the job token priority is $jobtoken.priority = -currentTime$. As time progresses, older jobs will automatically have higher priorities as

new jobs. Therefore the FIFO order is guaranteed.

Shortest Remaining Processing Time The assignment for the job token priority is $jobtoken.priority = -jobToken.PROCESSING_TIME_REMAINING$. The job with the shortest remaining processing time will be selected first, as the priority will have the highest value.

Shortest Remaining Processing Time in Percent of Total Processing Time The assignment for the job token priority is $jobtoken.priority = -\frac{PROCESSING_TIME_REMAINING}{TOTAL_PROCESSING_TIME}$. The priority is assigned using the same principle as before, except it is normalized by the total processing time.

Number Of Operations The assignment for the job token priority is $jobtoken.priority = NO_OF_OPERATIONS$. The job with the greatest number of process steps will be processed first.

Number Of Operations Remaining The assignment for the job token priority is $jobtoken.priority = NO_OF_OPERATIONS_REMAINING$. The job with the greatest number of operations remaining will be processed first.

Shortest Processing Time The assignment for the job token priority is: $jobtoken.priority = -PROCESSING_TIME$. The job with the shortest processing time will be processed first.

In general, any dispatch rule that is a function of the attributes or timestamp of the job tokens can be implemented.

5.2.8.1 Attributes of Job Tokens

These dispatch rules require that the job token have the following attributes:

- Release Time
- Number of Operations

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

The first three attributes 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. This is usually done when the processing at the process step is finished; for example, the attribute "Processing Time Remaining" is reduced by the process time after the processing has finished.

It is possible to extend the dispatch rules as long as the priority that will be assigned to the job tokens is a function of these attribute values. Further, it is possible to extend the set of attributes of a job token and create dispatch rules based on these.

5.2.8.2 Representation of Queues

There is no explicit representation of queues in the framework. A queue is represented either by a 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 tokens that are waiting to be processed. If there is more than one job type, the queue is represented by the set of places that hold the respective job tokens. Figure 48 shows an example where the queue is represented by places P_1 and P_2.

A standard queue follows the FIFO dispatch rule. This means that transitions T_1 and T_2 in Figure 48 use the FIFO dispatch rule for assigning token priorities. This will ensure that the token with the oldest timestamp, i.e., the highest priority will be processed first.

Under a FIFO dispatch rule, it can be guaranteed that no job token will be left unprocessed because the priorities of new job tokens will always be less than the priorities of existing job tokens. As time progresses, there can never be a job token with a higher job priority that will be added to the queue.

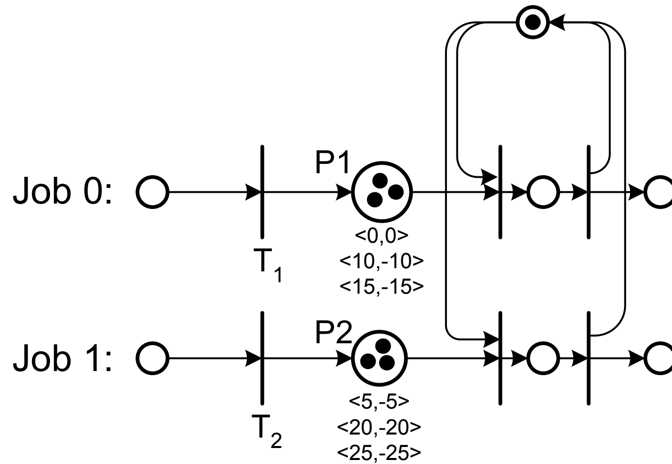

Figure 48: Queue

However, it is not possible to guarantee this for every dispatch rule: there are some dispatch rules that can assign token priorities that are higher than the priorities of the tokens that are present in the queue, which could cause older jobs to be in queue indefinitely. Obviously, this is not a problem if there is a finite number of jobs and no new jobs are entering the system.

5.3 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 model is loaded from an XML file. This will instantiate a `FabModel` object, which was described in Section 4.3. 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.

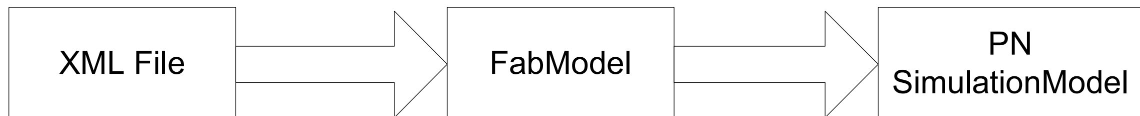

Figure 49: Overview of Simulation Model Generation

The preceding sections described the mapping from the data specification to the PN.

The overall procedure is as follows:

1. Create Tool Sets

2. Create Operator Sets

3. Create Process Routes

4. Create Rework Routes

5. Create Input Transitions

5.3.1 Generation of the Petri Net

5.3.1.1 Create Tool Sets

There are two cases to distinguish when generating the places in the PN that represent the tool sets. For each tool set that does not involve a setup state, a place of type `Tool` is created. For tool sets that model setup states, a place of type `Tool` is created for each setup state of the tool.

If the tool set is subject to breakdowns, a `TriggerTransition` is created with a firing interval corresponding to the breakdown rate. The detailed algorithm is in Appendix B.1.

5.3.1.2 Create Operator Sets

For each operator set, a place is created and the appropriate number of tokens are added. The detailed algorithm can be found in Appendix B.2.

5.3.1.3 Create Process Routes

The process steps for each process route are created sequentially. The detailed algorithm can be is described in Appendix B.3.

5.3.1.4 Create Rework Sequence

Rework sequences are created when they are encountered during the creating of a process route. If the return step of the rework sequence has not been created yet, the rework sequence will be generated after all process steps are created.

5.3.1.5 *Create Input Transitions*

The input transitions are transitions of type `TriggerTransition`, which release jobs (see Appendix B.4).

5.3.2 Polymorphism of Process Steps

Polymorphism is a fundamental element of Object Oriented Programming. It refers to the concept that a data type can refer to multiple actual implementations. Here polymorphism means that a process step can have many different actual implementations, as long as it satisfies a basic structure.

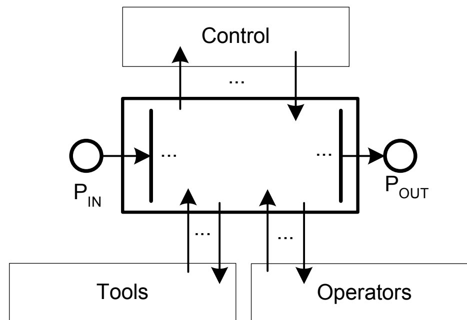

Figure 50: Polymorphism of a Process Step

Figure 50 shows the basic structure that represents an abstraction of a process step. 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 box contains all transitions that model the different process stages of the process step (not shown in detail). P_{IN} is the place that receives job tokens that have to be processed. P_{OUT} is the place that holds the processed job tokens. These places serve as interfaces between the process steps. P_{OUT} is also the receiving place for the next process step. The P_{IN} place coincides with the P_{OUT} place of the previous 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. Figure 50 does not show exactly how they are connected to the transitions, as it is specific for each process step type. Figure 51 shows an example of two process steps joined at places P_{OUT}

and P'_{IN}.

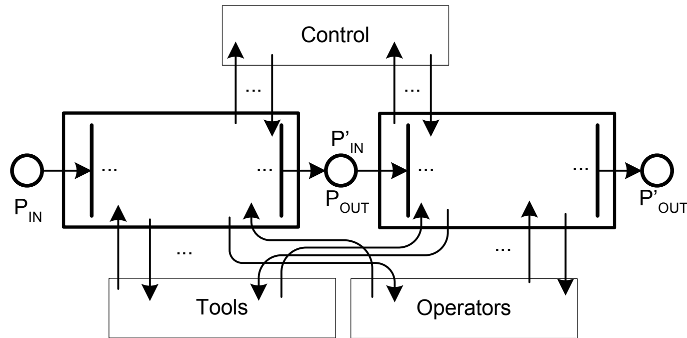

Figure 51: Two Process Steps Joined

5.3.3 Example

Table 1 gives an overview of the aforementions Sematech data set. A graphical user interface 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.

Table 1: Sematech Data Set Overview

Data Set	# Process Routes	Avg. # Process Routes	# Rework Routes	# Tools	# Operators
Set 1	2	228	1	83	32
Set 2	7	229	106	97	97
Set 3	11	376	105	73	0
Set 4	2	56	0	35	0
Set 5	24	174	0	85	4
Set 6	8	318	0	104	7

Figure 52 shows a small portion of the information in the XML file. The horizontal rows represent the process routes for each product. The rework sequences are displayed at the bottom. The simulation model generation described in Section 5.3 can be triggered with a command in a pull-down menu. A small portion of the generated PN is shown in Figure 53. Some arcs are not displayed for better readability (the arcs to the tool places and operator places have been omitted).

Process Routes

Route	Step 1	Step 2	Step 3	Step 4	Step 5
1 Bipolar Arrays SRAM #1: Start rate: 57.0 /day Lotsize 25	BatchProcessStep 1 Lift Off	BatchProcessStep 2 Clean	BasicProcessStep 3 Ash	BatchProcessStep 4 Clean	BasicProcessStep 5 Apply Resist
2 Fine Line Logic #1: Start rate: 32.0 /day Lotsize 18	BasicProcessStep 1 Inspect	BasicProcessStep 2 Measure	BatchProcessStep 3 Chem/Mech Polis	BasicProcessStep 4 Clean	BasicProcessStep 5 Inspect
3 Conventional Logic #1: Start rate: 21.0 /day Lotsize 18	BasicProcessStep 1 Inspect	BasicProcessStep 2 Measure	BatchProcessStep 3 Chem/Mech Polis	BasicProcessStep 4 Clean	BasicProcessStep 5 Inspect
4 Bipolar Arrays SRAM #2: Start rate: 76.0 /day Lotsize 25	BatchProcessStep 1 Lift Off	BatchProcessStep 2 Clean	BasicProcessStep 3 Ash	BatchProcessStep 4 Clean	BasicProcessStep 5 Apply Resist
5 Fine Line Logic #2: Start rate: 71.0 /day Lotsize 18	BatchProcessStep 1 Lift Off	BatchProcessStep 2 Clean	BasicProcessStep 3 Ash	BatchProcessStep 4 Clean	BasicProcessStep 5 Apply Resist
6 Single Level Test: Start rate: 53.0 /day Lotsize 42	BatchProcessStep 1 Lift Off	BatchProcessStep 2 Clean	BasicProcessStep 3 Ash	BatchProcessStep 4 Clean	BasicProcessStep 5 Apply Resist
7 Conventional Logic #2: Start rate: 28.0 /day Lotsize 18	BatchProcessStep 1 Lift Off	BatchProcessStep 2 Clean	BasicProcessStep 3 Ash	BatchProcessStep 4 Clean	BasicProcessStep 5 Apply Resist

Rework Routes

Route	Step 1	Step 2	Step 3
1:	BasicProcessStep 1 Expose UV	BatchProcessStep 2 Develop	BatchProcessStep 3 Bake

Figure 52: Simulation Model Data for Data Set 1 (abridged)

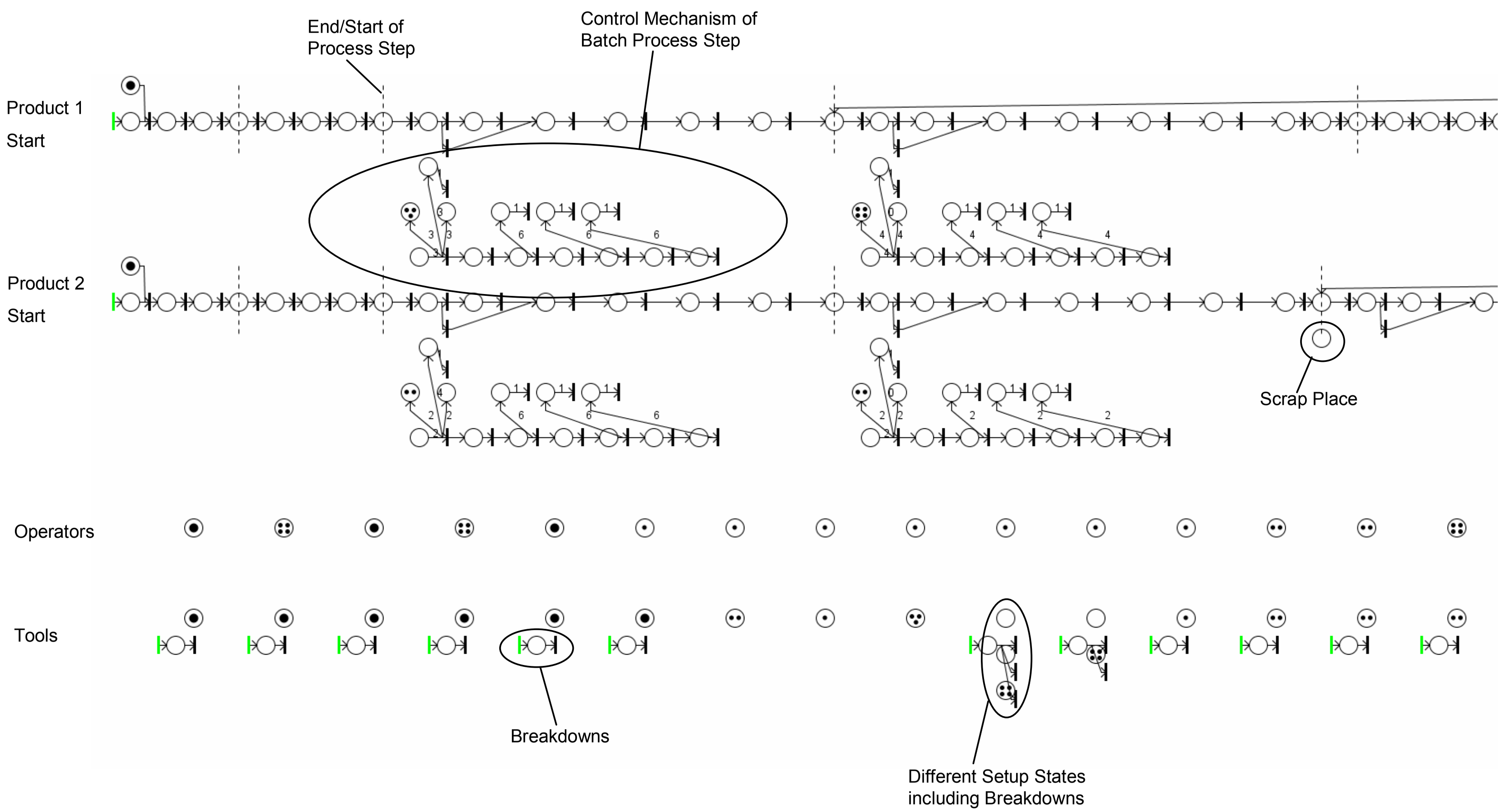

Figure 53: PN Simulation Model for Data Set 2 (abridged)

The PN simulation model in Figure 53 has 6,363 places and 3,751 transitions. Most of the places are of type `ProcessPlace` (the number of `Resource` places is 127). An overview of the size of the other simulation models generated from the Sematech data sets is given in Table 2.

Table 2: Size of PN Simulation Models

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

CHAPTER VI

ANALYSIS OF GENERATED PETRI NET

This chapter analyzes the properties of the generated PN simulation model (PNSM). First the scope and validity of the classical analysis techniques are explored. Then reduction rules are introduced, which are the basis of the analysis. Applying these reduction rules to the process steps in Section 5.2.4.1, leads to a set of distinct PN constructs. These constructs are analyzed in lieu of the original process steps, and it is shown that they are live, bounded, and that the final processing stage is reachable. Based on these results, it is shown that the generated PN will also maintain the same properties. Finally, conditions are introduced for the legitimacy of the simulation model, which are important for the timing mechanism of the simulation model.

6.1 Validity of Classical Analysis Techniques

In order to show that the generated PNSM will have certain properties, the model can be analyzed just like a classical PN. The question remains if the analytical techniques for classical PNs are applicable for the generated PNSM. We start with two propositions.

Proposition 2. *If the underlying PN is live, then the timing mechanism and the priorities introduced in Section 3.3.5 preserve liveness of the PNSM.*

Proof. The reachability set of a live PN with initial marking m_0 is $R(PN, m_0)$. This set will not contain any dead states by the definition of liveness, i.e., there are no states where no transition is enabled. The enabling rule of the execution mechanism in Section 3.3.5 for a transition is identical to the enabling rule of classical PNs. However, the introducing of the execution mechanism from Section 3.3.5 will order the enabled transitions in the FEL. The enabled transitions are ordered first according to their enabling times (i.e., the time when they are eligible to be fired) and second based on their priorities. This means that if there is a set of transitions T_E which is enabled in some state, only one of the transitions will fire

first. This effectively cuts off the paths in the reachability graph that are formed by the other enabled transitions. Therefore, the reachability graph of the simulation PN will be a subgraph of $R(PN, m_0)$. Different dispatch rules will lead to different subgraphs. As there are no dead states in the underlying PN, there are also no dead states in any subset of the original reachability set. Therefore liveness is preserved. □

Proposition 3. *If the underlying PN is bounded, then the timing mechanism and the priorities introduced in Section 3.3.5 preserve boundedness of the PNSM.*

Proof. The proof is analogous to the proof of Proposition 2. Since a bounded PN will have a finite reachability set, the reachability graph will also be finite. The introduction of the execution mechanism from Section 3.3.5 will reduce the state space (i.e., the reachability graph will be a subgraph of the underlying reachability graph). Since the reduced state space is a subset of the original state space, boundedness is preserved. □

In general, reversibility and reachability cannot be maintained when introducing timing and priorities. This is because certain paths in the reachability graph are removed, which obviously reduces the state space and therefore certain states remain unreachable. This is not a problem, as only some states are of interest and not the entire reachability set. Here we are mostly interested in showing that the state that indicates the completion of a process step can be reached. Reversibility means that the system can assume its initial state. Hence, reversibility can simply be proved by a sequence of transition firings that will lead to the initial state, it does not rely on the reachability graph of the underlying PN.

Propositions 2 and 3 form the basis for the analysis of the PNSM. The next section discusses synthesis techniques and reduction rules for PNs. The mappings from the data specification to the PN are analyzed using theses rules.

6.2 Synthesis and Reduction Techniques for Petri nets

6.2.1 Synthesis Techniques for Petri nets

Jeng [22] distinguishes between bottom-up and top-down synthesis techniques. Bottom-up synthesis methods use sub-PNs, which are modeled separately ignoring interactions

between the subsystems. At each step, the interactions between the subsystems are considered and the corresponding PNs are combined through merging of places or transitions. These bottom-up techniques focus on place invariant analysis, i.e., rules are derived for obtaining place invariants of the synthesized system from the invariants of the subsystems. However, invariants do not convey all system properties, which can make the analysis of some properties, such as liveness, difficult.

Top-down techniques on the other hand start with an aggregate model, which neglects details at first. Through refinement of transitions or places, the level of detail is increased. Places or transitions are replaced with subnets, which increases the level of detail to the desired level. However, these techniques are difficult to use for systems with shared resources.

6.2.2 Reduction Methods for Petri Nets

Reduction methods are primarily used to simplify the analysis of PNs. Most reduction steps do not change important properties of the PN. This also means that the same techniques may be used to augment an existing PN without changing the same properties. The following reduction rules preserve liveness, safeness, and boundedness [33].

Fusion of Serial Places (FSP) is depicted in Figure 54. Two places p_1 and p_2 that are in series and are only connected through a single transition t can be fused together; i.e., if $p_1^\bullet = t$, $t^\bullet = p_2$, and $t \subseteq {}^\bullet p_2$, the places p_1 and p_2 can be replaced by a single place p_3, such that ${}^\bullet p_3 = {}^\bullet p_1 \cup {}^\bullet p_2 \backslash \{t\}$ and $p_3^\bullet = p_2^\bullet$. Each token that arrives in p_1 will automatically be transferred to p_2, as t will always be enabled if a token is present in p_1. Since firing of t does not change the overall token count, boundedness and safeness are maintained with this transformation. Liveness is maintained as well, as any marking $m(p_1)$ will be transferred to p_2. The merging of the two places will result in the same marking for p_3. As $p_3^\bullet = p_2^\bullet$, the same output transitions will be enabled as in the original net.

Fusion of Serial Transitions (FST) is similar to FSP except that transitions are fused together. Figure 55 shows two transitions t_1 and t_2 in series and connected through a single place p. If $t_1^\bullet \subseteq p$, ${}^\bullet p = t_1$, $p^\bullet = t_2$, and ${}^\bullet t_2 = p$, then these two transitions can be fused to a single transition t_3, so that ${}^\bullet t_3 = {}^\bullet t_3$ and $t_3^\bullet = t_2^\bullet \cup t_1^\bullet \backslash \{p\}$. If the initial marking of p is

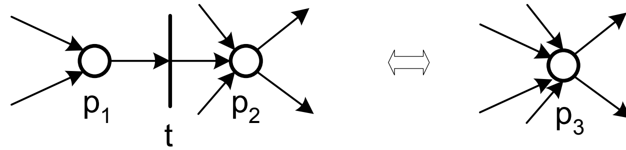

Figure 54: Fusion of Serial Places (FSP)

zero, this transition also maintains safeness and boundedness, as the total token count will not be affected by the transformation. Since all tokens that are consumed by t_1 will also be consumed by t_3, and all tokens generated by t_2 will also be generated by t_3, safeness and boundedness will not be affected. If t_1 fires, it will automatically enable t_2. Since every firing of t_1 will trigger a firing of t_2, liveness will also be maintained by this transformation.

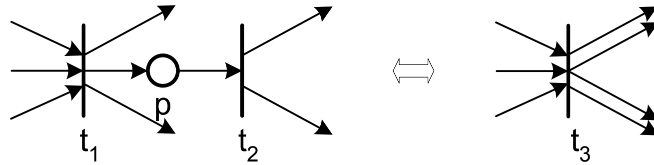

Figure 55: Fusion of Serial Transitions (FST)

Fusion of Parallel Places (FPP) is depicted in Figure 56. Two places p_1 and p_2 in parallel can be fused together, i.e., they can be replaced by a single place p_3 with ${}^\bullet p_3 = t_1$ and $p_3^\bullet = t_2$, if ${}^\bullet p_1 = {}^\bullet p_2 = t_1$ and $p_1^\bullet = p_2^\bullet = t_2$. This transformation will maintain liveness as the firing of t_1 will always enable t_2. Safeness and boundedness are also unaffected. All the tokens that t_1 consumes are also consumed by t_3, and all tokens that t_2 produces are also produced by t_3.

Fusion of Parallel Transitions (FPT) is shown in Figure 57. Two transitions t_1 and t_2 that are in parallel can be fused together to a single transition t_3 when ${}^\bullet t_1 = {}^\bullet t_2 = p_1$ and $t_1^\bullet = t_2^\bullet = p_2$. The token count remains unchanged in both cases, which implies that safeness and boundedness are maintained. Since any marking that enables t_1 or t_2 will also enable t_3, liveness is also preserved.

Elimination of Self-Loop Places (ESP) is illustrated in Figure 58. Place p_1 can be

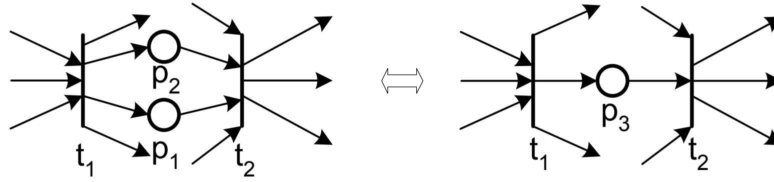

Figure 56: Fusion of Parallel Places (FPP)

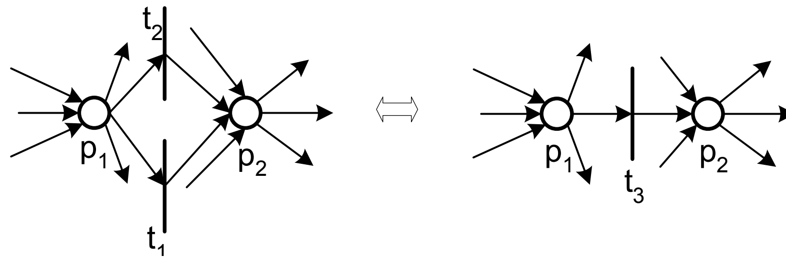

Figure 57: Fusion of Parallel Transitions (FPT)

eliminated if ${}^{\bullet}p_1 = p_1{}^{\bullet} = t_1$. Since firing of t_1 will not change the marking of p_1, the liveness of t_1 will not be influenced by p_1. Obviously, the marking of p_1 has to be $\geqslant 1$ or else t_1 is not live. Safeness and boundedness are also unaffected by this transformation as ${}^{\bullet}t_1\backslash\{p_1\}$ and $t_1^{\bullet}\backslash\{p_1\}$ remains unchanged.

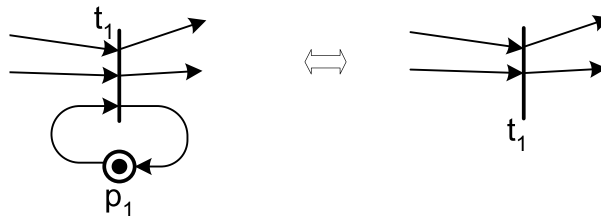

Figure 58: Elimination of Self-Loop Places (ESP)

Elimination of Self-Loop Transitions (EST) is depicted in Figure 59. If ${}^{\bullet}t_1 = t_1^{\bullet} = p_1$, clearly t_1 can be eliminated from the net. Obviously, firing of t_1 will not change the marking of the net. Hence boundedness, safeness, and liveness are not affected by this transformation.

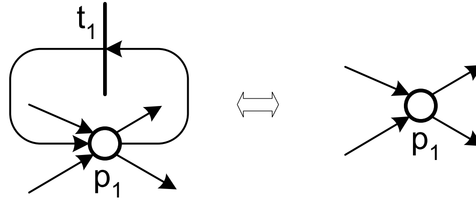

Figure 59: Elimination of Self-Loop Transitions (EST)

All these transformations can also be used sequentially in any order; however, one must take into consideration that every transformation will reduce the state space, and therefore the level of detail of the model is also reduced. This is not a problem as long as these transformations are only used to prove liveness, safeness, and boundedness of the PN.

6.2.3 Relationship between Reduction and Synthesis Methods for Petri Nets

The reduction and synthesis methods for PNs are closely related. Reduction methods can be seen as the inverse of synthesis methods and vice versa. When synthesis rules are used, the interpretation of the rule with respect to the target system has to be considered. The PN that is created using synthesis rules should have an interpretation in the target system. On the other hand, the interpretation of a reduction rule is not important. When a reduction rule is applied, detail is lost, but the primary goal is to simplify the analysis of the PN.

6.3 Application of Reduction Rules

This section uses the aforementioned reduction rules and applies it to the different process step types. This will lead to a set of constructs that are easier to analyze than the original process steps.

6.3.1 Basic Process Step

The basic process step module (Section 5.2.4.1) can be reduced to a set of very compact constructs using reduction rules from Section 6.2.2. Figure 60 shows the application of the FST reduction rule which results in the elimination of P_2 and the fusion of T_{SP} and T_{EP}.

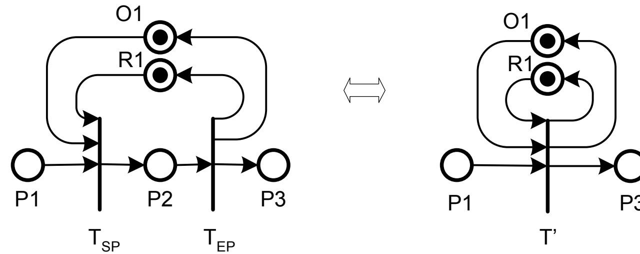

Figure 60: Reduction of Basic Process Step Module

Note that, in general, it is not possible to reduce this construct further. The ESP rule cannot be applied to the right side of Figure 60, as in general the resource places R_1 and O_1 will usually be connected to other process steps that require them as resources. Figure 61 shows the basic process step with tool travel time modeled. First the FSP is applied and T_M and P_3 are eliminated. Then the FST rule is applied as before.

Figure 62 shows the reduction of the basic process step with partial operator processing. The FST rule is applied twice. First the transitions T_{EPO} and T_{EP} are fused together. The resulting subnet, shown in Figure 60, is then reduced in the same way as in Figure 60.

Figure 63 shows the basic process step with operator loading. The FSP rule is used first to eliminate T_{SP} and merge P_2 and P_3. Then the FST rule is applied as in Figure 60.

Figure 64 shows the basic process step with operator loading and operator unloading. The FSP rule is applied twice to merge P_2, P_3, and P_4 and eliminate T_{SP} and T_{EP}. Then

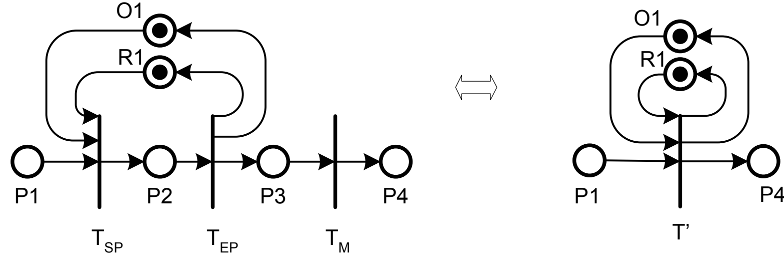

Figure 61: Reduction of Basic Process Step Module with Tool Travel Time

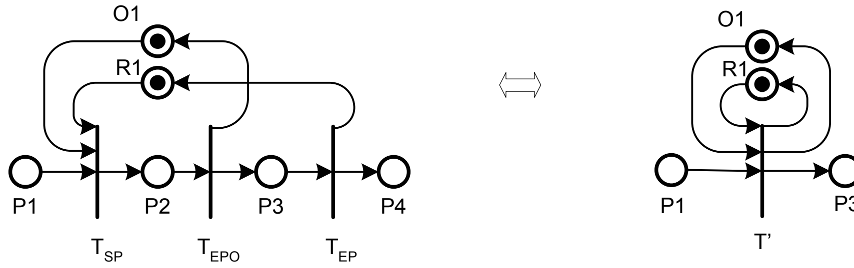

Figure 62: Reduction of Basic Process Step Module with Partial Operator Processing

the FST rule is applied as in Figure 60.

Figure 65 shows the basic process step with operator travel time modeled. The left portion is identical to Figure 60. The right portion models the transportation to the next tool that requires an operator. The FST rule is used twice to first fuse T_{SP} and T_{EP} and eliminate P_2, and then to fuse T_{ST} and T_{ET} and eliminate P_4.

All previous examples lead to the same basic construct. This is not the case when a process step does not require an operator during the entire processing time and releases the operator before finish processing.

Figure 66 shows a basic process step with all modeling possibilities. The FSP rule eliminates T_P, T_{SP}, and T_M and fuses each of the pairs $\{P_0, P_1\}$, $\{P_2, P_3\}$ and $\{P_6, P_7\}$. Then the FST rule can be applied sequentially to first fuse T_{SL} and T_{EPO} and, finally, T_{EP} and T_{EU}. This scenario cannot be simplified to the same level as in the previous cases (Figures 60-65). This is because T_{EPO} and T_{EP} cannot simply be fused together as

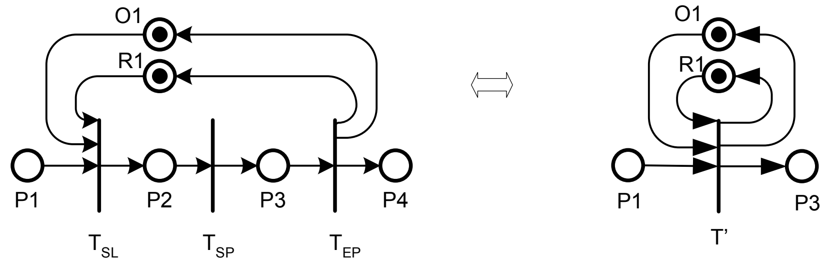

Figure 63: Reduction of Basic Process Step with Operator Loading

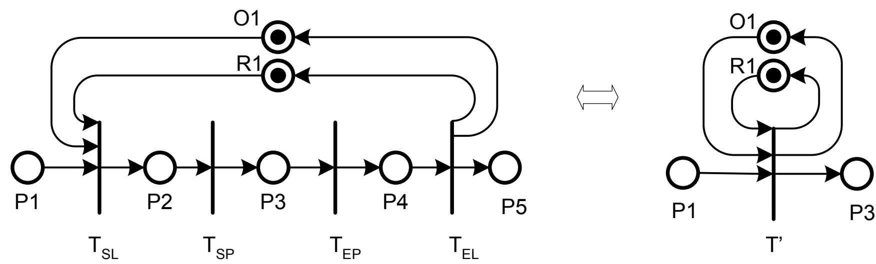

Figure 64: Reduction of Basic Process Step Module with Op. Loading and Unloading

$P_4^{\bullet} = T_{EP}$ but ${}^{\bullet}T_{EP} = \{P_4, O_1\}$.

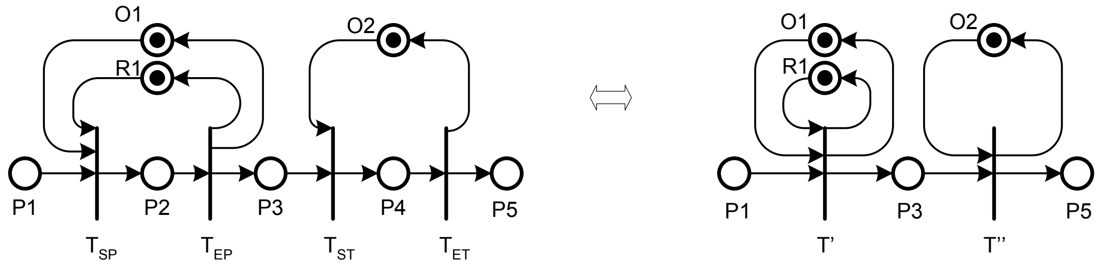

Figure 65: Reduction of Basic Process Step Module with Travel Time

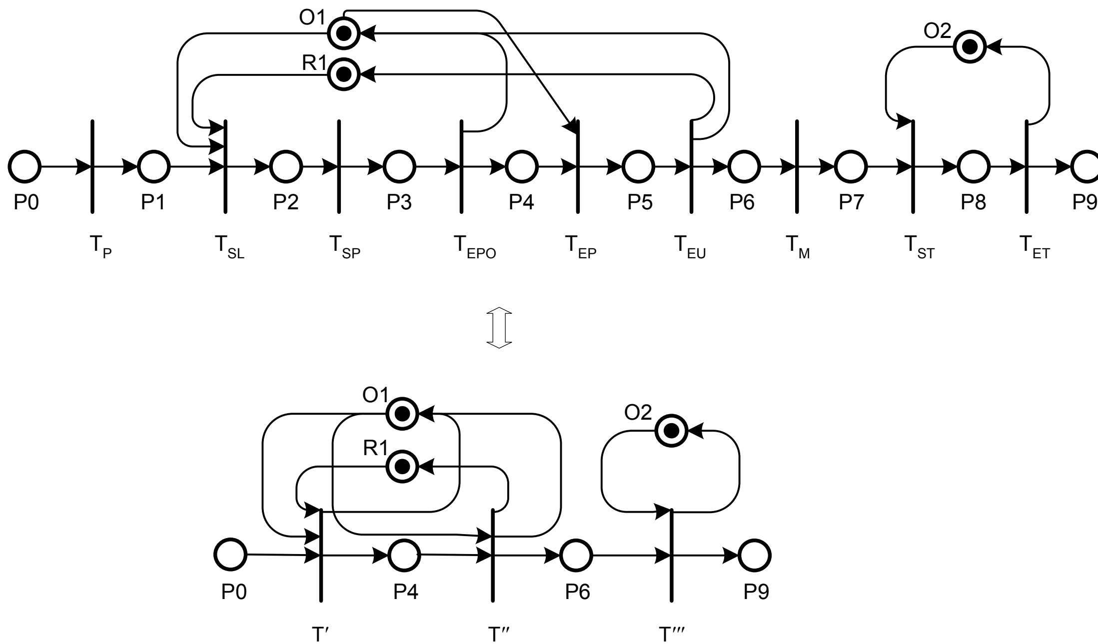

Figure 66: Reduction of Basic Process Step Module with All Options

6.3.2 Process Step with Setup

The process step with setup can be reduced similar to the constructs in Section 6.3.1 constructs. The main difference is the modeling of the setup step before actual processing starts. Figure 67 shows how the setup process step is reduced. The FST rule is applied to fuse T_{S_1} and T_{E_1} to T'_{S_1}, T_{S_2} and T_{E_2} to T'_{E_2}, and T_{S_3} and T_{S_3} to T'_{S_3}. Further, T_{SP} and T_{EP} can be fused together.

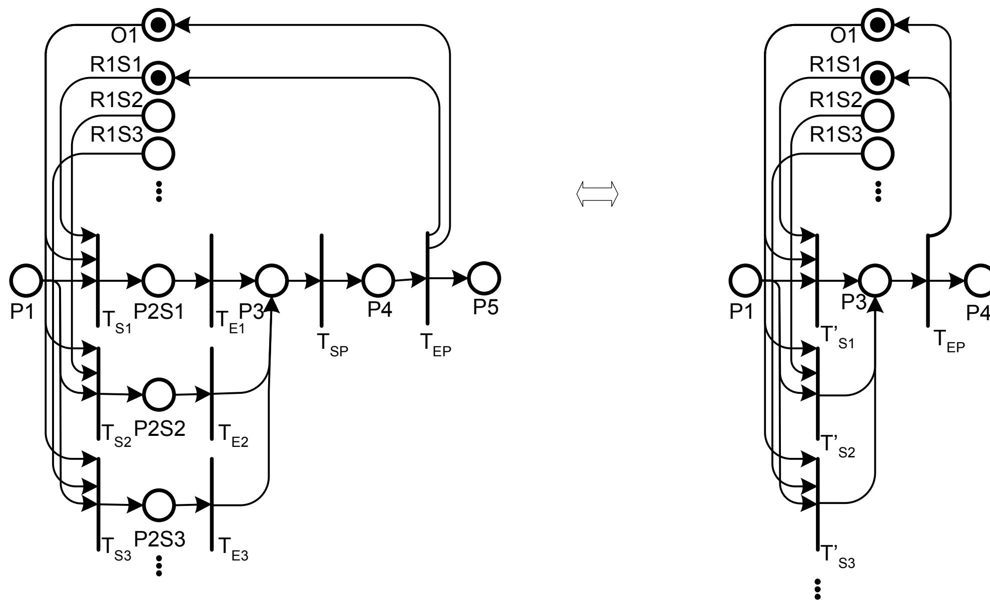

Figure 67: Reduction of Setup Process Step Module

Figure 68 depicts the setup process step with all possible options except for the travel step to the next tool. The setup process step can be reduced to T'_{S_1}, T'_{S_2}, and T'_{S_3} as before. The resulting net is very similar to Figure 66 and can be reduced in the same manner. The net at the bottom of Figure 68 cannot be reduced further as the operator is released during processing and later seized again for the unloading operation.

6.4 Mutual Exclusion

This section introduces several mutual exclusion concepts. These are used in Section 6.7 for the analysis of the synthesized PN. Zhou et al. [58] introduce a complex theory of mutual

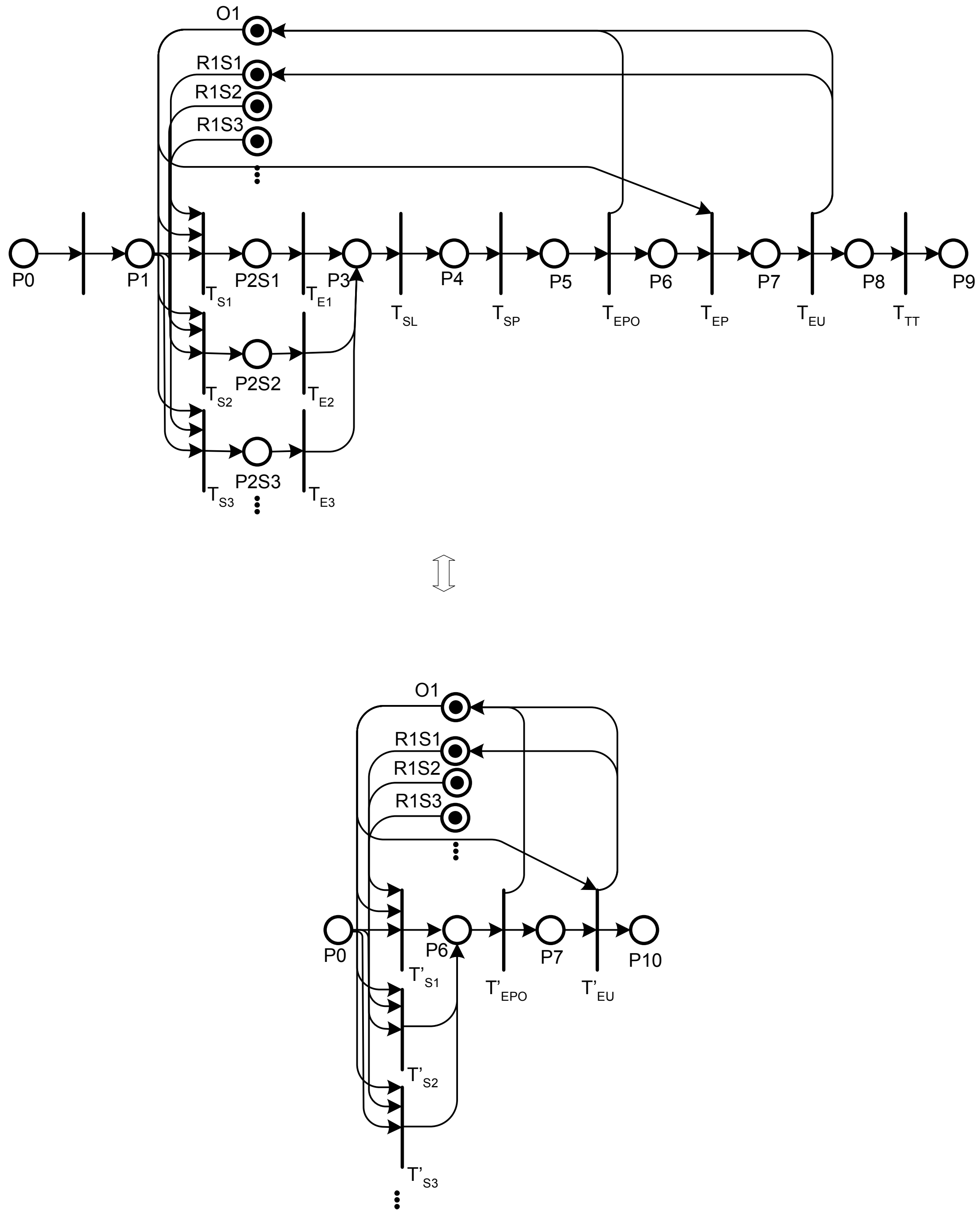

Figure 68: Reduction of Setup Process Step Module with All Possibilities

exclusion concepts for PNs, which is limited to single resources (i.e., resource places that are safe). Parallel mutual exclusions are used to represent independent processes that share the same resources, whereas serial mutual exclusion refers to resources that are used by the same process at different times. The Sematech data set includes resources with multiple units, often more than 20 units are quoted.

In this section we introduce a simplified mutual exclusion concept, which is based on the constructs that have been derived in Sections 6.3.1 and 6.3.2. For the following discussion the following sets of places are used: the set of operator places $\mathcal{O}$, the set of tool places $\mathcal{R}$, and the set of process places Π. The tool set and operator set places are used to model available tools and operators, respectively. The process places are used to model the different processing stages of a wafer lot traveling through the wafer fab.

6.4.1 Simple Mutual Exclusion

Definition 1. A *simple mutual exclusion* consists of a tool set place R_1 and/or an operator place O_1, process places $P_{\mathrm{IN}}, P_{\mathrm{OUT}}$, and a transition T, such that ${}^\bullet T = \{O_1, R_1, P_{\mathrm{IN}}\}$ and $T^\bullet = \{O_1, R_1, P_{\mathrm{OUT}}\}$, with all arc weights equal to one and initial markings $m(R_1) \geqslant 1$ and $m(O_1) \geqslant 1$. In other words, T has the same tool place as input place and output places. If an operator is used, it has the operator place as an input place and output place as well.

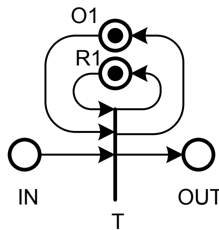

Figure 69: Simple Mutual Exclusion

6.4.2 Analysis of Simple Mutual Exclusions

The simple mutual exclusion presented in Figure 69 is not live by itself. In order to analyze it, we make the following augmentation: A transition T_R is added with ${}^\bullet T_R = P_{\mathrm{OUT}}$ and

$T_R^\bullet = P_{\text{IN}}$. Further $k \geqslant 1$ tokens are added to P_{IN}. T_R is added to ensure a steady supply of tokens to P_{IN}. The k tokens in P_{IN} represents k jobs in the system. T_R is enabled whenever there are tokens in P_{OUT}, that is, for each finished job a new job can be released by T_R.

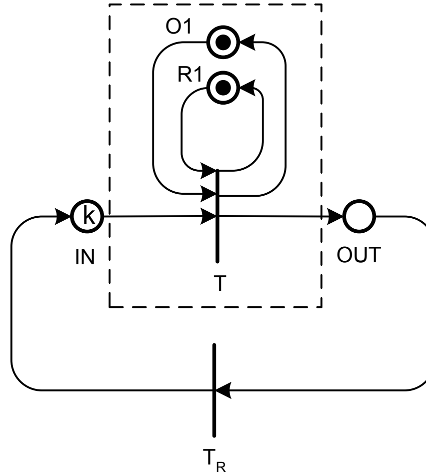

Figure 70: Simple Mutual Exclusion Analysis

6.4.2.1 Liveness

Initially T will be live, as there are tokens in all input places. T will be able to fire at least k times. The firing of T will result in the same marking for R_1 and O_1, hence the liveness of T depends solely on the marking of P_{IN}. Each firing of T will also enable T_R. Each firing of T_R will enable T if it is not already enabled. Therefore, the augmented simple mutual exclusion module (Figure 70) is live for any initial marking with $m(O_1) \geqslant 1$ and $m(R_1) \geqslant 1$.

6.4.2.2 Reversibility

Reversibility requires that the system can assume the initial state (i.e., $m(O_1) \geqslant 1$, $m(R_1) \geqslant 1$, and $m(IN) = k$). Initially there are k tokens in P_{IN}. Firing each of T and T_R n times $(n \leq k)$ will lead to the same state as the initial state, hence reversibility is established.

6.4.2.3 Boundedness

Each firing of T and T_R will keep the token count constant. Therefore, boundedness is guaranteed for the construct in Figure 70. Safeness is not an issue in general, as the

markings of O_1 and R_1 can be greater than one.

6.4.2.4 Reachability

For the simple mutual exclusion, it is important to show that tokens arriving at the P_{IN} place will ultimately reach the P_{OUT} place. Recall that the P_{OUT} place represents the completion of a job or wafer lot. It is obvious in this case that every token in P_{IN} can reach P_{OUT} by the firing of T.

The analysis above was done for Case C in Figure 69, but the same arguments can be clearly applied for Cases A and B.

6.4.3 Nested Mutual Exclusion

Figure 71 shows a nested mutual exclusion that motivates the following definition.

Definition 2. A *nested mutual exclusion* consists of a tool set place R_1, and an operator place O_1, process places $P_{\mathrm{IN}}, P_{\mathrm{OUT}}$, and P_2, and transitions T_1 and T_2 such that ${}^{\bullet}T_1 = \{P_{\mathrm{IN}}, O_1, R_1\}$, $T_1^{\bullet} = \{P_2, O_1\}$, ${}^{\bullet}T_2 = \{P_2, O_1\}$, and $T_2^{\bullet} = \{R_1, O_1, OUT\}$, with all arc weights equal to one and initial markings $m(O_1) \geqslant 1$, and $m(R_1) \geqslant 1$. Note that if only the tool is modeled, this construct can be reduced to a simple mutual exclusion.

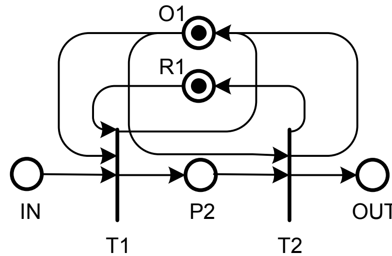

Figure 71: Nested Mutual Exclusion

6.4.4 Analysis of Nested Mutual Exclusions

The nested mutual exclusion in Figure 71 can be analyzed in the same fashion as before. A transition T_R is added, as shown in Figure 72.

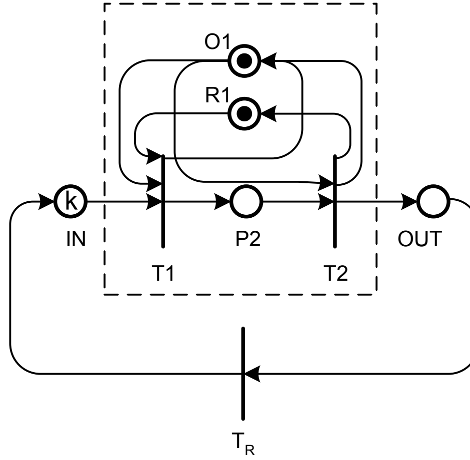

Figure 72: Nested Mutual Exclusion Analysis

6.4.4.1 *Liveness*

Initially, T_1 is live because there are tokens in all input places of T_1. Each firing of T_1 will enable T_2 since $T_1^\bullet = {}^\bullet T_2$. T_1 can fire up to n times, with $n = \min\{m(R_1), k\}$. When T_2 fires, a token will be put back into R_1. Hence, any sequence that contains equal numbers of T_1 and T_2 firings will lead to the same initial marking of O_1 and R_1. After each firing of T_2, there will be a token in P_{OUT}, which in turn will enable T_R. Each firing of T_R will put back a token to P_{IN}, which then enables T_1 again. Hence the augmented nested mutual exclusion module is live for any initial marking with $m(O_1) \geqslant 1$ and $m(R_1) \geqslant 1$.

6.4.4.2 *Reversibility*

Initially there are k tokens in P_{IN}. Firing each of T_1, T_2, and T_R n times with $n \leq \min\{m(R_1), k\}$, will lead to the same state as the initial state establishing reversibility.

6.4.4.3 *Boundedness*

The firing of the pair $\{T_1, T_2\}$ will keep the total token count constant. A firing by T_2 will increase the token count by one. On the other hand, a firing by T_1 will decrease the token count by one. As the number of T_2 firings can never exceed the number of T_1 firings because of $T_1^\bullet = {}^\bullet T_2$, boundedness is guaranteed.

Another way to formally prove boundedness is to inspect the place invariants (see also Section 3.2.7). It can be easily seen that all places are covered by a place invariant, i.e., $\{P_{\text{IN}}, P_2, P_{\text{OUT}}\}, \{R_1, P_2\}, \{O_1\}$, which is a necessary and sufficient condition for boundedness. Safeness is also not an issue, as the markings of O_1 and R_1 can be greater than one.

6.4.4.4 Reachability

As with a simple mutual exclusion, every token that arrives in P_{IN} is able to reach the completion stage P_{OUT}. In this case, firing of T_1 and T_2 will take a job token from P_{IN} to P_2 and then to P_{OUT}.

6.4.5 Mutual Exclusion with Resource Setup States

A more complex situation arises for the case when setup states have to be modeled. Figure 73 shows the basic construct for a mutual exclusion with setup states. There are two possible cases: Case A is the standard case similar to the simple mutual exclusion, and Case B is similar to the nested mutual exclusion. The standard case is defined as follows.

Definition 3. A *mutual exclusion with resource setup states* consists of a set of resource places $R_{S_1}, \dots, R_{S_k}$, each corresponding to a distinct setup state. Further, there is a set of process places $\{P_{\text{IN}}, P_{\text{OUT}}, P\}$, an operator place O, and transitions $T_{S1}, \dots, T_{S_k}$ and T_{EP}, such that ${}^\bullet T_{S_j} = \{P_{\text{IN}}, R_{S_j}\}$ and $T_{S_j}^\bullet = \{P\}$ for all $j \in \{1, \dots, k\}$, ${}^\bullet T_{EP} = \{P\}$, and $T_{EP}^\bullet = \{P_{\text{OUT}}, O, R_{S_l}\}$, with l indicating the setup state required for the process step. Further the initial markings are $m(O) \geqslant 1$ and $\sum_{j=1}^{k} m(P_{S_j}) \geqslant 1$, i.e., there is at least one token in the set of places that represent the resource. The transitions $T_{S1}, \dots, T_{S_k}$ mark the beginning of the setup. At the end of processing, the resource is released by T_{EP} to the appropriate setup state.

The nested case is defined as follows.

Definition 4. A *nested mutual exclusion with resource setup states* consists of a set of resource places $R_{S_1}, \dots, R_{S_k}$, each corresponding to a distinct setup state. There is a set of process places $\{P_{\text{IN}}, P_{\text{OUT}}, P, P_2\}$, an operator place O, and transitions $T_{S1}, \dots, T_{S_k}$, T_{EPO},

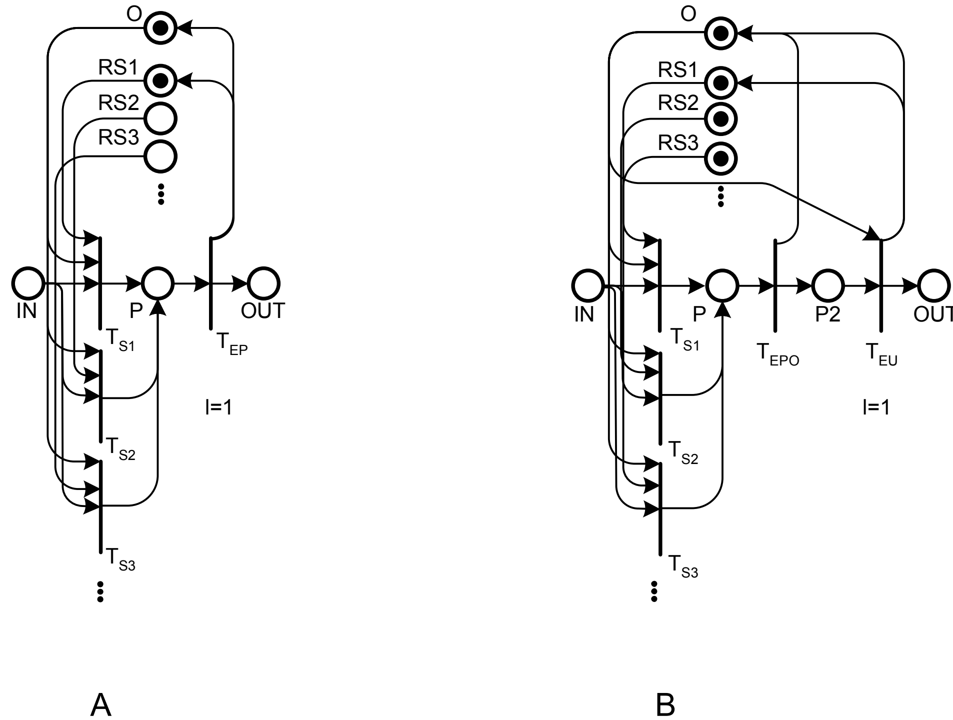

Figure 73: Mutual Exclusion with Resource Setup States

and T_{EU}, such that ${}^{\bullet}T_{S_j} = \{P_{\text{IN}}, R_{S_j}\}$ and $T_{S_j}^{\bullet} = \{P\}$ for all $j \in \{1, \ldots, k\}$, ${}^{\bullet}T_{EPO} = \{P\}$, $T_{EPO}^{\bullet} = \{O\}$, ${}^{\bullet}T_{EU} = \{P_2, O\}$, and $T_{EU}^{\bullet} = \{O, R_{S_l}\}$, with l indicating the setup state required for the process step. The transitions $T_{S1}, \ldots, T_{S_k}$ mark the beginning of the setup as in the standard case. Finally, T_{EPO} represents the end of processing with and operator, and T_{EU} represents the end of unloading.

6.4.6 Analysis of Mutual Exclusions with Resource Setup States

The mutual exclusion with resource setup states in Figure 73 can be analyzed in the same fashion as before. A transition T_R is also added (see Figure 74) and k tokens are placed into P_{IN}.

6.4.6.1 Liveness

Initially one of $T_{S_1}, \ldots, T_{S_k}$ will be live, as there will be k tokens in P_{IN} and at least one token will be present in the set $\{R_{S_1}, \ldots, R_{S_k}\}$ because of the condition $\sum_{j=1}^{k} m(P_{S_j}) \geqslant 1$.

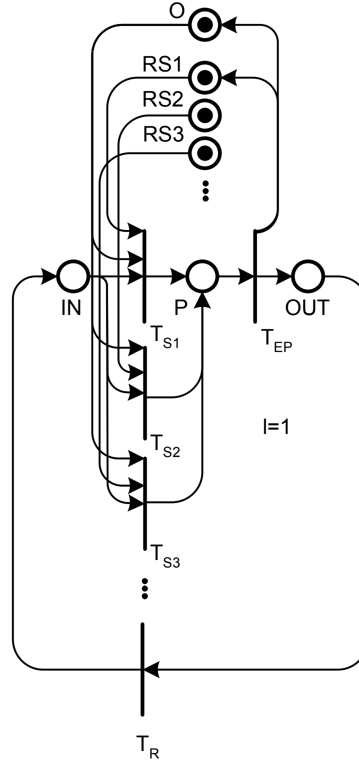

Figure 74: Analysis of Mutual Exclusion with Resource Setup States

After a transition T_{S_l}, $l \in \{1, \ldots, k\}$ fires, T_{EP} will be enabled. After T_{EP} fires, the marking of O_1 will be equal to the initial marking. However, T_{EP} will always place a token into R_{S_l}, with l being the required setup state for the process step (in this case $l = 1$). This means that the token could originate from any place R_{S_j}, $j \in \{1, \ldots, k\}$. This is not a problem, as $\sum_{j=1}^{k} m(P_{iS_j})$ will stay constant. Note that the arcs to R_{S_j}, $j \in \{1, \ldots, k\} \setminus \{l\}$, with l being the required setup state, are not displayed, as they will come from other process steps. See also Figure 42 in Chapter 5.

6.4.6.2 Reversibility

Reversibility can only be established for the case where all tokens of the resource are in setup state l, where l is the required setup state of the process step. Intuitively, this means that the required resource has to be in the required setup state to guarantee reversibility. The firing sequence comprised of n firings by T_{S_j}, n firings by T_{EP}, and n firings by T_R with

$n = \min\{k, \min_{l \leq j \leq k} R_{S_j}\}$ establishes reversibility. The latter requirement is not necessary when analyzing jointly all process steps that use a particular resource with setup states. With an appropriate sequence of setups, it is possible to return the system to the initial state.

6.4.6.3 Boundedness

Boundedness can be established with the place invariants $\{IN, P, OUT\}$, $\{O, P\}$, and $\{R_{S_1}, \ldots, R_{S_k}, P\}$. Intuitively it can be seen that by firing any pair from T_{S_j}, $j \in \{1, \ldots, k\}$ and T_{EP} the token count will stay constant. Firing T_R does not change the token count. The firing of T_{S_j}, $j \in \{1, \ldots, k\}$ will decrease the token count by one and the firing of T_{EP} will increase the token count by one. The number of times T_{EP} can fire is limited by the number firings of T_{S_j}, $j \in \{1, \ldots, k\}$. Hence the token count will not increase, establishing boundedness for a mutual exclusion with setup.

6.4.6.4 Reachability

Every token that arrives in P_{IN} is clearly able to reach the completion place P_{OUT} by firing of T_{S_j}, $j \in \{1, \ldots, k\}$ and T_{EP}.

6.5 Simple Batch Process Step

The most complex construct is the batch process step. As discussed in Section 5.2.4.3, the basic problem is that the size of the batches that are to be processed is not known in advance and can vary. A batch process step with minimum batch size a and maximum batch size b is defined as follows: Given is the (sub-)PN with $P = \{P_1, P_2, P_3, P_4, B_1, B_2, B_3, B_{10}, B_{20}, C,$ $L, R_1\}$ and transitions $T = \{T_{S_1}, T_{S_2}, T_{EP}, T_P, T_{D_2}, T_{B_1}, T_{B_2}, T_{B_3}, T_{D_1}\}$. The arc weights are $w(T_{B_1}, C) = a$, $w(T_{B_1}, L) = b - a$, $w(B_1, T_{B_1}) = a$, $w(T_{B_3}, B_20) = b$, with all other arc weights equal to 1. The initial markings are $m(C) = b - a$, $m(R_1) \geqslant 1$, with all others being zero. The transition input and output arcs are: ${}^\bullet T_{S_1} = \{P_1, C\}$, $T_{S_1}^\bullet = \{P_2, B_1\}$, ${}^\bullet T_{S_2} = \{P_2, B_{10}\}$, $T_{S_2}^\bullet = \{P_3\}$, ${}^\bullet T_{EP} = \{P_3, B_{20}\}$, $T_{EP}^\bullet = \{P_4\}$, ${}^\bullet T_{P_1} = \{C, L\}$, $T_{P_1}^\bullet =$ $\{P_3\}$, ${}^\bullet T_{B_1} = \{B_1, R_1\}$, $T_{B_1}^\bullet = \{B_2, C, L, B_{10}\}$, ${}^\bullet T_{B_2} = \{B_2\}$, $T_{B_2}^\bullet = \{B_3\}$, ${}^\bullet T_{B_3} = \{B_3\}$, $T_{B_3}^\bullet = \{R_1, B_{20}\}$, ${}^\bullet T_{D_1} = \{B_{20}\}$, $T_{D_1}^\bullet = \{\emptyset\}$, ${}^\bullet T_{D_2} = \{L\}$, $T_{D_2}^\bullet = \{\emptyset\}$, ${}^\bullet T_{P_1} = \{P_1, L\}$, and

$T^{\bullet}_{P_1} = \{P_2\}$. Another way to express the above construct is the incidence matrix:

$$A = \begin{bmatrix} -1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & -a & 1 & 0 & a & 0 & a & b-a & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & b & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{bmatrix}$$

Each row represents the state change when the associated transition fires. The rows and columns are ordered according to the sets T and P, respectively. This matrix is helpful to calculate the state changes for the analysis in the next section.

6.6 Analysis of Batch Process Step

Figures 75 and 76 are used for the analysis of the batch process step. This graph is not a reachability graph in the traditional sense. A reachability graph for a classical PN enumerates all possible transitions firings from a given state. Here we consider only the transitions that are eligible to fire according to the execution mechanism. This means that the ordering of the enabled transitions in the FEL is considered. There are two reasons for this consideration: (1) the resulting graph will be much more compact than the traditional reachability graph; and (2) due to the execution mechanism, some of the states in the traditional graph cannot be reached (this prioritizes transitions according to time and priority). The right-hand column of both graphs shows the firing time of the events.

The graph starts with Figure 75 and continues with Figure 76. The row vectors represent the state of the system, and the labeled arrows indicate an event, i.e., the firing of the corresponding transition. A series of firings of the same transition is compressed into one arrow with a label that indicates the number of firings. A transition T that fires a times is

denoted by $a \times T$. Events that can be executed independently can be compressed to one arrow, with the labels listing the transitions that can fire independently.

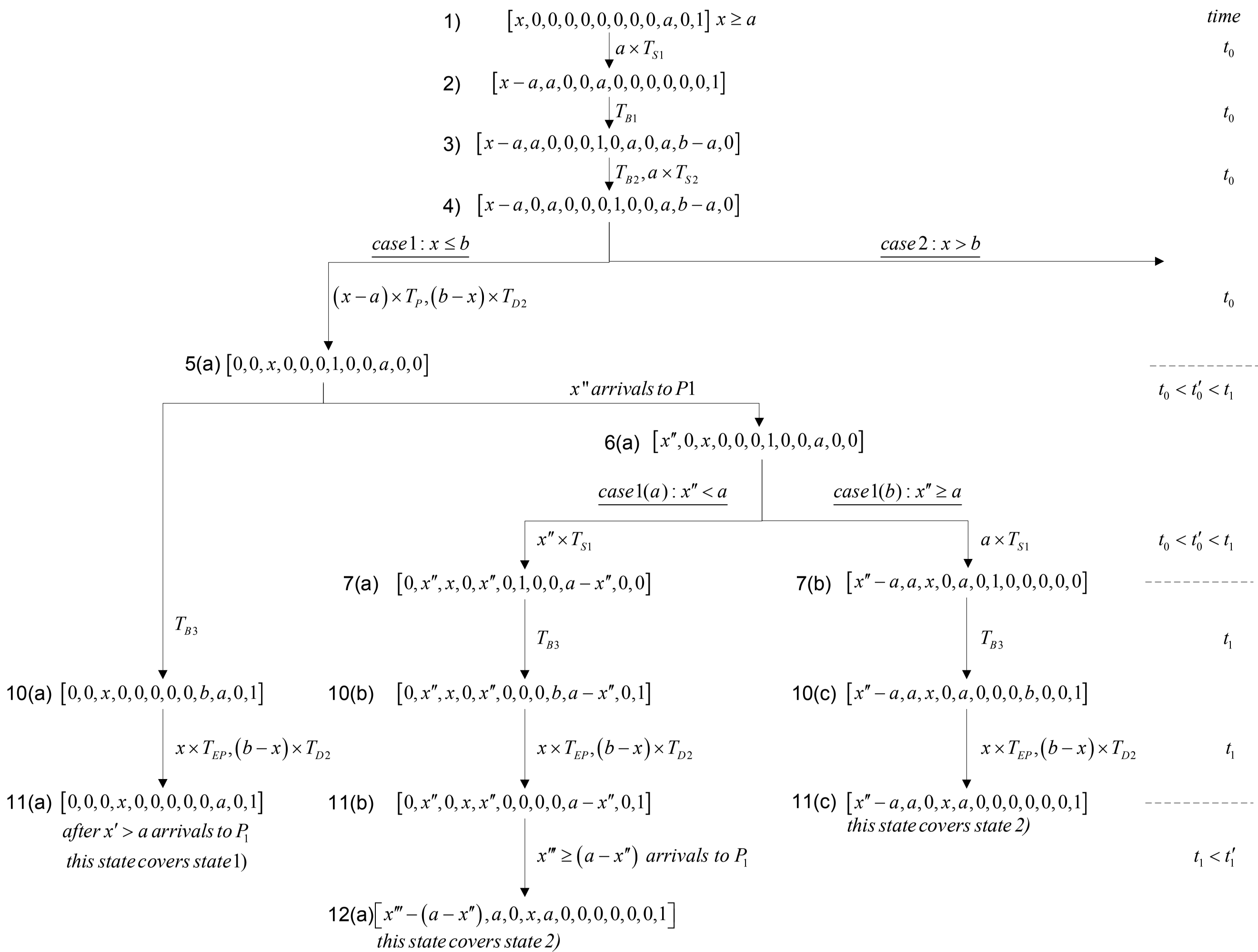

Figure 75: Reachability/Coverability Graph for Batch Process Step (1)

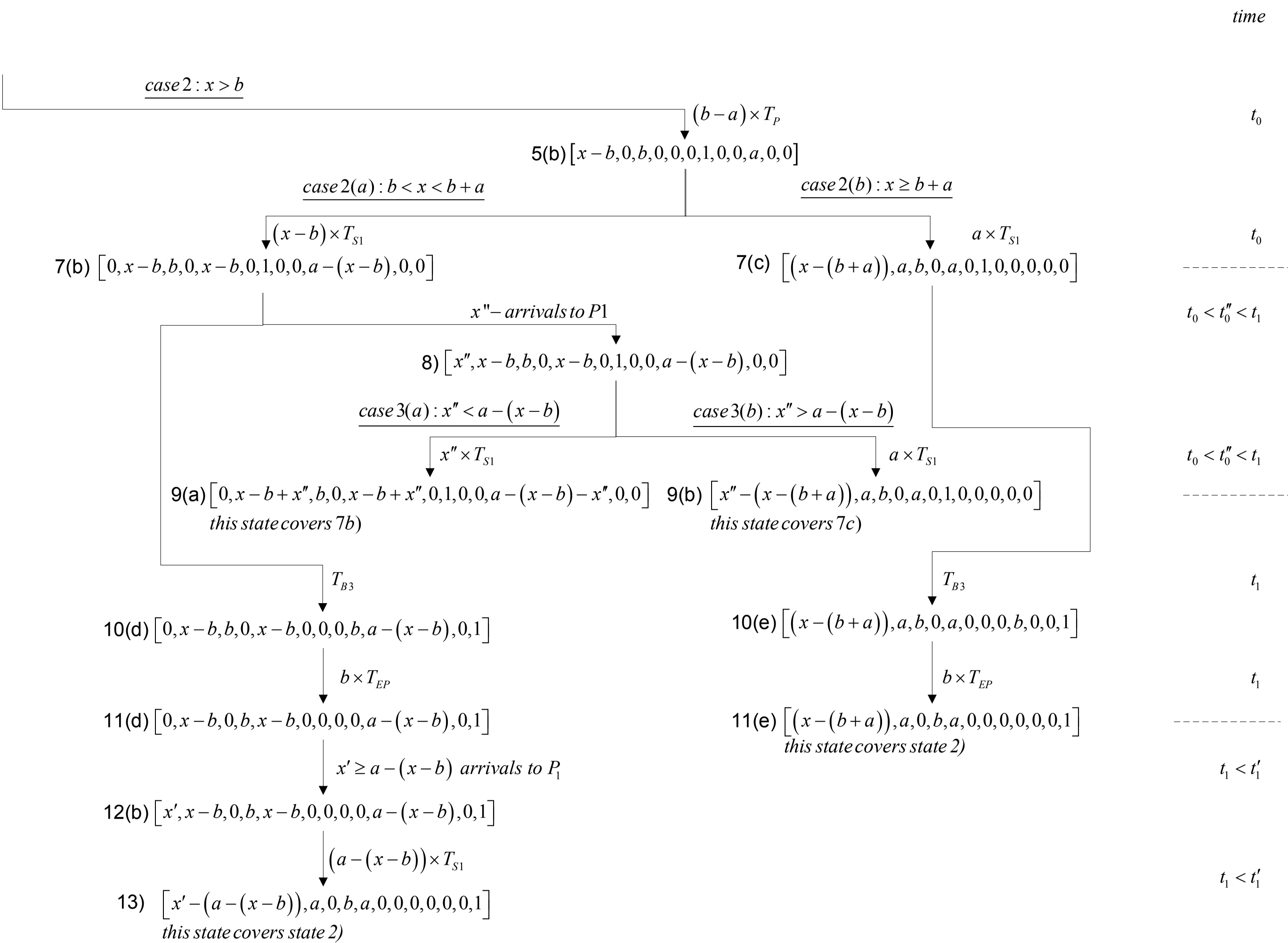

Figure 76: Reachability/Coverability Graph for Batch Process Step (2)

The analysis assumes that jobs will always be able to arrive at place P_1. This assumption ensures that there will be at least a tokens after a finite amount of time. When $x \geqslant a$ jobs have arrived there are x tokens in P_1 (state 1 in Figure 75). At this instance T_{S_1} can fire a times, leading to state 2. Then T_{B_1} can fire, bringing the system to state 3. In this state transitions T_{B_2}, T_{S_2}, and T_{D_2} are enabled. However, T_{D_2} has the lowest possible priority assigned, as described in Section 5.2.4.3. Hence T_{B_2} and T_{S_2} can fire in any order, as ${}^\bullet T_{B_2} \cap {}^\bullet T_{S_2} = \emptyset$ and T_{S_2} does not enable any other transition; this will lead to state 4. In this state T_{B_3}, T_P and T_{D_2} are enabled. The enabling time of T_{B_3} is t_1 (note that the time has not advanced yet). At this point there are two cases to distinguish. Case 1 corresponds to $x \leqslant b$ and Case 2 corresponds to $x > b$. Case 2 will be analyzed with the help of Figure 76. For Case 1, T_P will fire $x - a$ times and then T_{D_2} will fire $b - a$ times, as the priority of T_P will be higher. This will lead to state 5a; in this state only T_{B_3} will be enabled. If there are no arrivals during the time interval (t_0, t_1), T_{B_3} will fire leading to state 10a. If there are x'' arrivals during the time interval (t_0, t_1), it will lead to state 6a. In state 10a time has advanced to t_1 and T_{EP} and T_{D_2} are enabled. First, T_{EP} will fire x times and then T_{D_2} will fire $b - x$ times, leading to state 11a. This state is basically identical to state 1 after x' arrivals to P_1, except for the tokens in P_4, which do not influence the liveness. In state 6a, x'' tokens have arrived at place P_1. Two cases have to be distinguished: Case 1(a) with $x'' < a$ and Case 1(b) with $x'' \geqslant a$.

For Case 1(a), T_{S_1} will fire x'' times and lead to state 7a. At time t_1, T_{B_3} will fire and lead to state 10b. From here T_{EP} will fire x times before T_{D_2} fires $b - x$ times leading to state 11b. After x''' arrivals, the system will be in state 12a, which is identical to state 2, except for the tokens in P_4.

For Case 1(b), T_{S_1} will fire a times and lead to state 7b. Transition T_{B_3} will fire at time t_1 and lead to state 10c. In this state T_{EP} will fire x times before T_{D_2} fires $b - x$ times, which brings the system to state 11c, which is also identical to state 2, except for the tokens in P_4.

Case 2 is shown in Figure 76. Firing $b - a$ times by T_P leads to state 5b. Here again two cases can be distinguished: Case 2(a) with $b < x < b + a$ and Case 2(b) with $x \geqslant b + a$

(recall that x was the initial number of tokens in place P_1).

For Case 2(a) T_{S_1} will fire $x - b$ times, which leads to state 7b. If there are no arrivals until t_1, T_{B_3} will fire at time t_1 and lead to state 10d. If there are x'' arrivals during the interval (t_0, t_1) at state 7b, the system will assume state 8. From state 10d, T_{EP} will fire b times and the system will be in state 11d. After x' arrivals to P_1, with $x' \geqslant a - (x - b)$ the system will move to state 12b and transition T_{S_1} is enabled. After T_{S_1} fires $a - (x - b)$ times, the system assumes state, 13 which is identical to state 2, except for the tokens in P_4. In state 8 there are two cases to consider, Case 3(a) with $x'' < a - (x - b)$ and Case 3(b) with $x'' \geqslant a - (x - b)$. For Case 3(a), T_{S_1} will fire x'' times, leading to state 9a, which is identical to state 7b. For Case 3(b) T_{S_1} will fire a times and lead to state 9b, which is identical to state 7c.

For Case 2(b) in state 5b, T_{S_1} will fire a times, which will lead to state 7c. Then T_{B_3} will fire at time t_1 and lead to state 10e. Then after T_{EP} fires b times state 11e is reached, which is identical to state 2, except for the tokens in P_4.

6.6.0.5 Liveness

The graph has no states in which no transition will be enabled; hence all leaf nodes in the graph cover a previous state. Hence the batch process step is live.

6.6.0.6 Reversibility

Reversibility cannot be proved with the graphs in Figures 75 and 76. After all jobs are processed, the system will be in almost the same state as the initial state, except that the job tokens are in P_4 instead of P_1. However, when adding a hypothetical transition T_R with ${}^{\bullet}T_R = \{P_4\}$ and $T_R^{\bullet} = \{P_1\}$ to the batch process module (just like in the previous cases), one can see that when all the tokens are transferred from P_4 to P_1, the system will be in the same state, establishing reversibility.

6.6.0.7 Boundedness

The batch process step is not bounded by itself, as there are states that cover other states. This is because tokens can always arrive in place P_1. More importantly, it can be seen in

the graph that the tokens in P_4 cannot exceed the number of tokens that arrived in P_1, as they are on the same path. When no additional tokens arrive in P_1, it can be seen from the analysis above that the system will be bounded.

6.6.0.8 Reachability

It is important that tokens can arrive in place P_4, i.e., jobs can finish. Clearly, all the tokens that arrive in P_1 can reach the final stage in P_4, establishing the reachability of the finished processing stage.

The preceding analysis is for a simple batch process step, where no loading and unloading is modeled. Further, the graph in Figures 75 and 75 is for a single tool set and no operator. Adding an operator will not change the sequence of events, and making the analysis essentially identical. The analysis is also very similar for multiple tool sets having more than one resource available (i.e., the number of tokens in R_1 is greater than one). The difference is that the quantity that enters the processing stage may exceed the maximum batch size. If there are n tools available, n times the maximum batch size can be processed simultaneously. This makes the analysis more complex, but the basic mechanism is the same.

It can easily be seen that the addition of a loading and unloading process, as modeled in Figures 38 and 39, will not change the basic sequence of events analyzed before. The difference is that there are more events, and the token in B_1 will go through the additional places B_4, B_5, B_6, and B_7 for the case where loading and unloading is modeled. After transition T_{B_1} has fired, the token will be moved along the direct path connecting T_{B_2}, B_2, T_{B_3}, B_3, T_{B_4}, B_5, T_{B_5}, B_6, T_{B_6}, B_7, and T_{B_7} (Figure 39). All transitions along this path will automatically become enabled because they have only one input place, which is the output place of the previous transition. The only exception is T_{B_5}, which requires an operator for the unloading process. This situation is analogous to the nested mutual exclusion in Section 6.4.3. The transitions T_{B_1}, T_{B_2}, T_{B_3}, T_{B_4}, T_5, and T_{B_6} are in series and can be fused using the FST rule in Section 6.2.2. This will lead to an almost identical construct as in Figure 71. The input places for the transitions will be identical, the only difference being that

there will be additional outgoing arcs to the places B_{20}, B_{30}, and B_{40}, which control the movement of the job tokens.

For batch process steps that only model loading or unloading, the situation is analogous to the simple mutual exclusion in Section 6.4.1.

6.7 Analysis of Synthesized Petri Net Simulation Model

The previous sections establish the properties of the individual modules. However, this is not sufficient for the properties of the synthesized PN. The complete simulation model is synthesized using the modules that were introduced earlier. This section will establish the properties for the completed PNSM. Each module represents a type of process step, which was reduced from its original structure. These reductions did not influence liveness and boundedness. Each of these modules has an input place P_{IN} and output place P_{OUT}, which represent the waiting for processing and the end of processing. Further, each module is connected to resource places and other control places in a specific way. Now it remains to show that these connections will guarantee the liveness, boundedness, reversibility, and reachability of the entire PN.

6.7.1 Analysis of Serial Coupling of Process Step Modules

As mentioned in Section 5.3.2, every process step module has a place P_{IN} that receives job tokens and a place P_{OUT} that holds the job token(s) when processing has finished. The previous sections have shown that all process step modules are live, bounded, reversible, and that the finished processing state is reachable. According to the PN generation algorithm in Section 5.3, process step modules are joined in such a way that place P_{OUT} of the preceding module coincides with place P_{IN} of the following module. Figure 77 shows two process step modules coupled in series. The transition T_R is added for the same reasons as before to ensure an infinite supply of tokens as well as boundedness.

There are four basic types of process step modules:

I. Simple mutual exclusion

II. Nested mutual exclusion

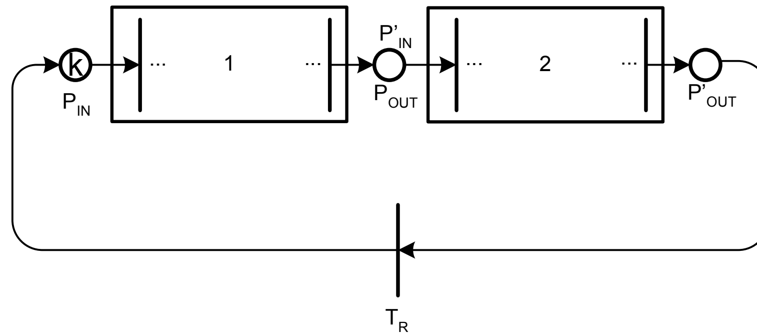

Figure 77: Serial Coupling

III. Mutual exclusion with resource setup state

IV. Batch process

Based on these basic types the following cases of combinations of modules need to be analyzed:

(a) both modules are of type I

(b) both modules are of type II or one module is of type I and one module is of type II

(c) both modules are of type III

(d) both modules are of type IV

(e) both modules are of a different type, except for Case (b)

6.7.1.1 Liveness

It has been shown that all process modules are live when analyzed as standalone modules. Hence it can be assumed that module 1 is also live by itself. We consider five cases.

Case (a) We consider two sub-cases: module 2 uses different resources than module 1 or module 2 uses the same resources as module 1. When different resources are used, module 2 will be live as no places are shared with module 1, except for P'_{IN}, and P_{OUT}. If both

modules share resources, then both modules also share the corresponding places. Initially no transition in module 2 will be able to fire, as there are no job tokens in the module. This also means that none of the tokens of the resources has been used within module 2. Therefore, the liveness of module 1 is not affected. All resources are released automatically after processing finishes, i.e., the marking of the resource places will be the same as before processing began. Therefore the resources will be available in module 2, making this module live.

Case (b) Again two sub-cases are considered: module 1 and module 2 use different resources, and both modules use the same resources. In the first case, liveness is not affected by the same arguments as in Case (a) above. If the modules share resources, module 1 will be live and can process the tokens in P_{IN}. If module 1 is a nested mutual exclusion consisting of two transitions (see also Figure 71), then the first transition will seize the tool and will seize and release the operator. This means that the marking of the operator places will be unchanged after firing. The second transition will only seize the operator, hence it will be able to fire. After the firings, all the resource markings will also be the same. Hence module 2 will also be live, as all resources will be available.

Case (c) Again two sub-cases can be distinguished. If both modules use different resources, liveness is not affected for the same reasons as for Case (a). If the resources are shared, then module 1 will be live at first. After firing of the appropriate transition T_{S_j}, $j \in \{1, \ldots, k\}$ (see Figure 73), T_{EP} will release the operator and the tool. Since the sum of tokens $\sum_{j=1}^{k} m(R_{S_j})$ will remain constant, module 2 will be able to start processing with one of the transitions T'_{S_j}, $j \in \{1, \ldots, k'\}$.

Case (d) Again we consider two cases. Liveness is not affected when both modules use different resources. When the resources are shared, two sub-cases have to be considered: the value for `batchId` is identical for both modules and the value for `batchId` is different. The PN will look like Figure 78 if both batch process steps use the same `batchId` value. Specifically the resource is only seized by T_{B_1} and released by T_{B_3}. The places B_{10} and

B_{20} control the progress of the tokens in the process places P_1, P_2, P_3, P_4, P_2', P_3', and P_4'. Tokens that arrive in P_1 and P_1' will trigger processing if the minimum batch size has been reached. After T_{S_1} and T_{S_1}' have fired a times (a is the minimum batch size), the sum of tokens in P_2 and P_2' will be a. Further, $b - a$ tokens will be in place L after T_{B_1} has fired (recall that b is the maximum batch size). These tokens allow T_{P_1} and T_{P_1}' to fire at most $b - a$ times. Therefore, the maximum number of tokens in P_3 and P_3' will be at most b. At the end of processing, there will be b tokens in place B_{20}, which are sufficient to move the tokens in P_3 and P_3' to places P_4 and P_4', respectively.

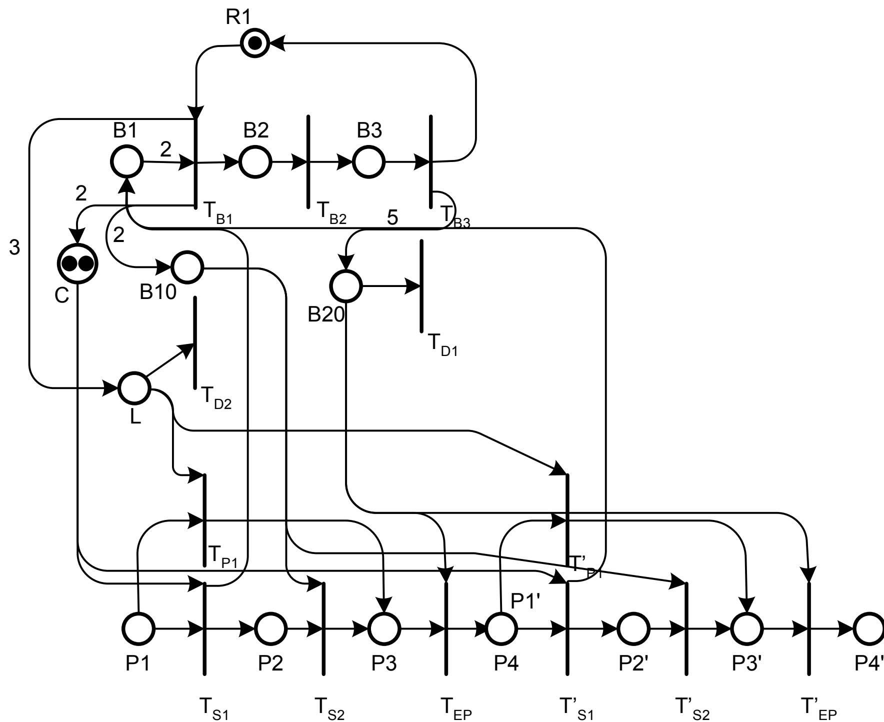

Figure 78: Batch Process Steps in Series with Identical `batchId`

If the process steps have different `batchId` values, the PN will look like Figure 79. In this case T_{B_1}, B_1, T_{B_2}, B_2, T_{B_3}, B_3, T_{B_1}', B_1', T_{B_2}', B_2', T_{B_3}', B_3', and R_1 form a simple mutual exclusion. The only difference is that there are arcs to C, C', B_{10}, B_{20}, B_{10}', and B_{20}'. These places are used to control the movement of the job tokens. T_{B_1} and T_{B_1}' are both competing for the same resource. If one seizes the resource first, then the other has to

wait until processing finishes. Other than that, both batch process steps operate as in the standalone version.

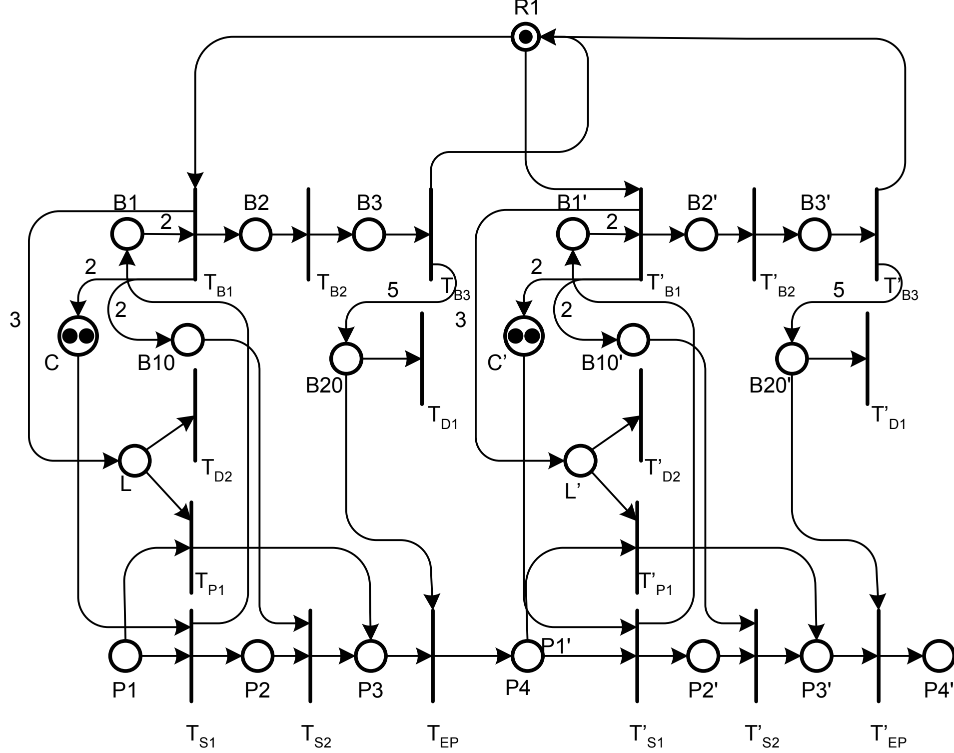

Figure 79: Batch Process Steps in Series with Different `batchId`

Case (e) If both modules are of a different type, they do not share any resources. Therefore liveness cannot be affected, and will be preserved.

6.7.1.2 Reversibility and Reachability

Reversibility and reachability can be analyzed simultaneously. If all tokens can reach P'_{OUT}, then firing T_R as many times will establish reversibility. Hence it is sufficient to show that all tokens can reach P'_{OUT}. It has been shown that it is possible to reach the P_{OUT} place of each individual module. Initially module 1 will be live, hence it will be able to process at least one of the tokens in P_{IN}. Either all tokens are processed first in module 1, or only some tokens are processed by module 1 and some by module 2. If all tokens are processed by module 1 first, they will be in P'_{IN} after processing. Then all tokens can be processed by

module 2. If only some tokens are processed by module 1, these tokens will be in P'_{IN} after processing, where they can be processed by module 2. When module 2 has no more tokens to process, then any leftover token in P_{IN} will be processed. When all tokens in P_{IN} have been processed, the remaining of the tokens in P'_{IN} can be processed in module 2. Hence all tokens will eventually be able to reach P'_{OUT}.

6.7.1.3 Boundedness

None of the transitions for the basic type I, II, or III will change the total token count when fired. Hence adding these basic types to an existing series of modules of type I, II, or III does not affect boundedness. The batch process step has been analyzed individually in Section 6.6. When there are no shared resources, it is clear that two batch process steps in series will also be bounded. On the other hand, if the two batch process steps use the same resources and the same `batchId` value, the situation is similar to a single batch process step, with the exception that the places C, L, B_{10}, and B_{20} in Figure 78 are connected to the transitions in both batch process steps. However, transitions T_{B_1}, T_{B_2}, T_{B_3}, T_{D_1}, T_{D_2}, B_1, B_2, B_3, B_{10}, B_{20}, C, L, and R_1 form the same structure as in the standalone module, which is bounded. The transitions T_{S_1}, T_{S_2}, T_{EP}, T'_{S_1}, T'_{S_2}, and T'_{EP} control the movement of a job token. Each of these transitions does not increase the token count when firing. Therefore, boundedness is preserved. It can be shown by repetitive usage of these arguments that adding another module in series will also preserve the aforementioned properties.

6.7.2 Analysis of Parallel Coupling of Process Step Modules

Section 6.7.1 showed that it is possible to couple process step modules in series. Each series of process step modules represents a process route in the manufacturing system. This section will analyze the case when modules are coupled in parallel. Figure 80 shows such a configuration. The analysis is very similar to the serial case.

6.7.2.1 Liveness

Parallel modules can process in any order. After a module has finished processing, the resources used are available for any other module. Hence liveness is not affected.

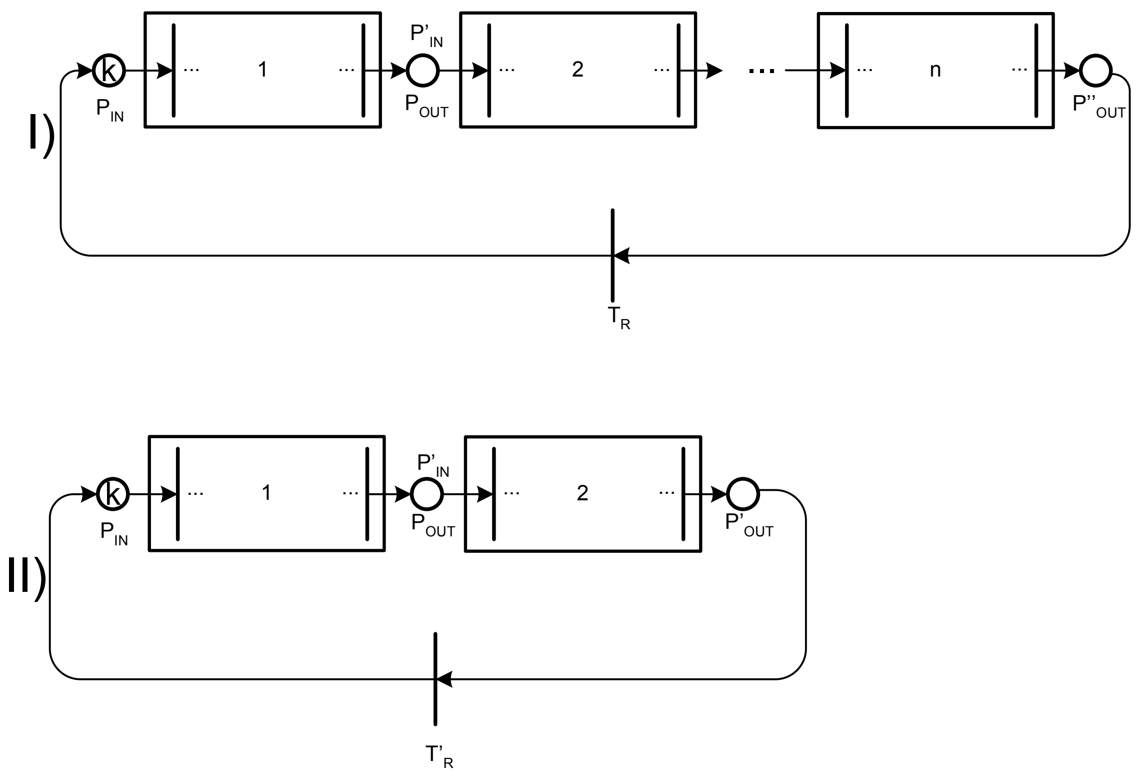

Figure 80: Parallel Coupling

6.7.2.2 Reachability and Reversibility

In general, reversibility cannot be guaranteed. It will depend on the dispatch rule that is used, because it is possible that the transitions of a module will always have a higher priority assigned then the transitions of another parallel module.

In general, reachability of the final state can only be guaranteed by dispatch rules that assign priorities to transitions such that *priority* of $T_1 >$ *priority* of T_2, if the firing time of T_1 is earlier than the firing time of T_2. This will guarantee that the older enabled transitions will fire before newer enabled transitions. Hence, there cannot be a transition that has to wait indefinitely to be able to fire. An example of this would be the FIFO rule, as the assigned priorities are the negative values of the firing times.

6.7.2.3 Boundedness

Clearly, the argument for the serial case can be used in a similar fashion. Therefore boundedness is maintained. Further, it can be shown that adding another parallel process module will not change the properties.

6.8 Analysis of Rework and Scrap Modeling

Rework sequences are modeled in the same way as normal process steps. A rework sequence is "forked off" the original process route. A switch transition will send the job token to the rework sequence instead of the scheduled next process step. After the rework sequence is executed, the job token will reenter the original sequence before, at, or after the leaving process step.

Figure 81 shows a process route with process steps 1, 2, 3, and 4. After process step 1, there is a chance that the rework process sequence has to be entered. This is indicated by the dashed arc. A job token will either travel to process step 2 or will be diverted to the rework sequence. This would change the original sequence to 1, R_1, R_2, 2, 3, 4. Rework steps are modeled in the same way as normal process steps; therefore, the situation is equivalent to serial coupling of process steps. There is a slight chance that a job token has to go through several rework steps; however, the probability that this will happen indefinitely approaches

zero, as there is a positive probability that a job token will not have to go through a rework sequence. Since rework sequences simply extend the normal process routes with additional process steps, the previous analysis is still valid.

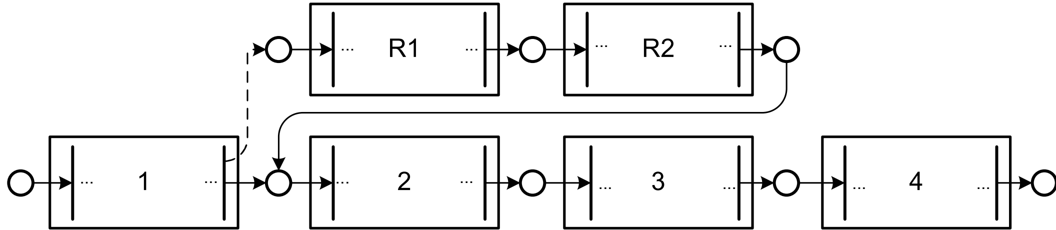

Figure 81: Analysis of Rework Sequence

The modeling of scrap is shown in Figure 82. After process step 1, a "scrapped" job token is sent to place S_1. This will effectively end all the future processing for that lot.

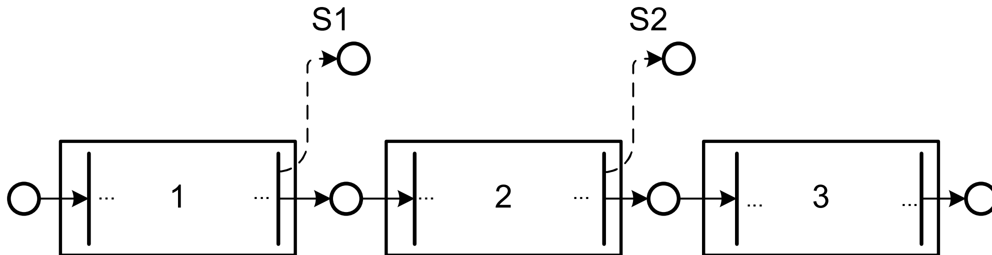

Figure 82: Analysis of Scrap Modeling

6.9 Legitimacy

Legitimacy was introduced in Section 2.6.1. It is an important property, as it guarantees that the simulation time will be able to advance. The simulation time will not be able to advance if there is an infinite series of events, which will occur at the same instance in time. For example, consider the case where an event A at time t causes event B and event B in turn causes event A at the same time. The following definitions are used in the following discussion.

Definition 5. (Elementary Path) An elementary path, or dipath is a sequence of distinct nodes (i.e., transitions or places): $x_1, x_2, \ldots, x_n$, $n > 1$, with $(x_i, x_{i+1}) \in A$.

Definition 6. (Elementary Circuit, Directed Cycle) An elementary circuit or directed cycle is a sequence of nodes $x_1, x_2, \ldots, x_n$, $n > 1$, with $(x_i, x_{i+1}) \in A$ such that $x_1, x_2, \ldots, x_{n-1}$ is a elementary path and $x_1 = x_n$.

Proposition 4. *Sufficient conditions for the generated PN to be legitimate are: (1) For any process module, every path between P_{IN} and P_{OUT} has at least one transition with a positive delay; (2) every transition of the process module must be on such a path; and (3) there must not be any circuit with zero delay transitions.*

Proof. An infinite sequence of transition firings that is repeated indefinitely will be represented as a circuit in the PN. If there is at least one transition with a positive delay, there cannot be an infinite sequence with no time advancement. For a token to travel from P_{IN} to P_{OUT}, there has to be a path between these places. If all possible paths have at least one delay transition, the token timestamp will be guaranteed to be greater in P_{OUT} than P_{IN}, which means that time will advance. □

Not all conditions above are true for the batch process step. There are no circuits with any delay transitions, but there are paths from P_{IN} to P_{OUT} with no delay transitions. However, all job tokens that go through a batch process step will have a greater timestamp after processing. The analysis of the batch process step in Section 6.6 showed that there cannot be any infinite sequences with no time advances.

6.10 Conclusions

This chapter analyzed the PNSM. It was shown that liveness and boundedness are preserved for the simulation model, when the underlying PN has these properties. The different types of process steps of Chapter 5 were transformed with reduction rules to four distinct types of constructs, which were analyzed an shown to be live and bounded. With the help of these constructs it was shown that the generated PN will also maintain these properties.

The next chapter introduces a complexity measure, that allows us to describe differences in execution speed of different PNSM.

CHAPTER VII

COMPLEXITY ANALYSIS

7.1 Literature Review

There is not a universally accepted definition for the complexity of simulation models. Zeigler [57] defines three kinds of complexity: analytic complexity, simulation complexity, and exploratory complexity. Analytic complexity is directly related to the number of states of the simulation model. Simulation complexity is the time and space required to simulate the model. Exploratory complexity refers to the resources that are required to explore the state space of simulation model. It can be very expensive to explore the state space for a coupled simulation model due to state space explosion. Measures of analytic complexity or exploratory complexity are often impractical because the state space may not be known explicitly. For a practitioner, the simulation complexity is of primary interest as it is tied to the resources and time needed to run a simulation model. However, Zeigler does not introduce a quantitative measure for simulation model complexity.

Schruben [48] defines complexity only as a measure that reflects the requirements of computational resources. He introduces and compares several measures such as the cardinality of the vertex set, the edge-to-vertex ratio, and the cyclomatic complexity, which is based on the cyclomatic number of the simulation graph. All these measures are graph theoretic approaches based on simulation graphs. Therefore, they can only be applied if a description in the form of a simulation graph exists.

In this chapter we will introduce a measure, related to the PN graph, that can capture differences in the execution speed of PNSM. We will also discuss, a simple measure that can capture the memory requirements for the simulation model. The first step is to analyze how execution algorithms work. This will provide insight to which are the driving forces that influence the runtime.

7.2 *Analysis of Execution Algorithms*

The execution of the PNSM goes through two main phases. First the PNSM is initialized during the initialization phase. The execution phase follows.

7.2.1 Initialization Phase

The initialization phase is the first task that has to be performed before the simulation can start, and consists of two steps. First, the set `toUpdate` for each transition has to be determined. This is the set of transitions that need to be updated after the transition fires. This will allow for a more efficient time update mechanism, as it does not require scanning the entire PN each time a transition is firing. Then for each transition `updateTime()` has to be called, which will determine if the transition is enabled and at what time.

The algorithm in Section 3.3.5.3 that determines the set `toUpdate` for each transition has two parts. The outer loop of the first part of the algorithm loops over $|{}^\bullet t|$ places, and the inner loop over $|p^\bullet|$ transitions for each place $p \in {}^\bullet t$. Similarly, for the second part of the algorithm the outer loop iterates over $|t^\bullet|$ places, and the inner loop over $|p^\bullet|$ transitions for each place $p \in t^\bullet$. The total number of iterations therefore is $\sum_{p \in {}^\bullet t \cup t^\bullet} |p^\bullet|$. In Figure 83 the field $T_1.toUpdate$ consists of the transitions $\{T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9\}$.

The `toUpdate` field of a `Transition` object is implemented as a `TreeSet`, which implements the `Set` interface. Both are standard types in the Java API. The `Set` interface is used to ensure that each transition is unique. Otherwise, the algorithm might add duplicates, which will slower its performance. Duplicates are created when some of the places in ${}^\bullet t$ and $t^\bullet$ share some output transitions. The `TreeSet` is implemented as a red-black tree in Java, thus insertion and deletion take $O(\log n)$ time [15], with n being the number of nodes in the tree. In most cases n is usually not very large. An upper bound on n is the upper bound on the number of transitions in `toUpdate` for transition t, which is $\sum_{p \in {}^\bullet t \cup t^\bullet} |p^\bullet|$. Assuming a linear runtime for insertion for simplicity, an upper bound for the initialization time of a transition is given by $K = C \cdot \left(\sum_{p \in {}^\bullet t_k \cup t_k^\bullet} |p^\bullet| \right)^2$, where C is a constant, k is the index of the transition with the most transitions in the field `toUpdate`, i.e., $k = \arg\max_{j \in \{1,\ldots,n\}} \sum_{p \in {}^\bullet t_j \cup t_j^\bullet} |p^\bullet|$, with $t_j \in T, j \in \{1, \ldots, n\}$, and T being the set of transitions in the PN. Therefore, the

initialization of every transition can be limited by a constant, independently of the size of the PN. Since every transition has to be initialized, the initialization of all transitions can be performed in $O(|T|)$ time. Hence, the running time requirement appears to grow linearly with the number of transitions.

The second step of the initialization phase is to call `updateTime()` for each transitions in the PN. This requires $|{}^{\bullet}t|$ calls to `updateTime()`. The UPDATE TIME algorithm (Section 3.3.5.2) iterates over all input places of the transition t to be updated. The operation inside the loop that iterates over the input places of the transition checks if the number of tokens is greater than zero and then finds the token with the smallest timestamp. Checking if the number of tokens is greater than zero requires a constant amount of time. The tokens are stored in a red-black tree, which is ordered according to the timestamps. Hence, the time to find the token with the smallest timestamp is of the order $O(\log n)$. To simplify the analysis; this time is approximated by a constant; this means that all operations inside the loop can be also approximated by a constant. The only exception occurs if the arc weight w is greater than one. This is rarely the case and will only increase the runtime to $O(w)$. Hence, the runtime can be approximated for most transitions to be of the order $O(1)$. This means that for a transition t the runtime for UPDATE TIME is $O(|{}^{\bullet}t|)$, hence it grows linearly with the size of the set $|{}^{\bullet}t|$. An upper bound on the size of this set can be obtained by $K = |{}^{\bullet}t_k|$ and $k = \underset{j \in \{1,\ldots,n\}}{\arg\max} |{}^{\bullet}t_j|$, $t_j \in T$,$j = \{1, \ldots, n\}$. Therefore, the runtime for UPDATE TIME for each transition can be considered as constant. As the UPDATE TIME algorithm is called for each transition in the PN, the total time to update the time for all transitions in the PN is of the order $O(|T|)$. In summary the runtime of the algorithms for the initialization phase is typically of the order $O(|T|)$. This is an important as it indicates that the time for the initialization phase will approximately grow linearly with the number of transitions in the PN.

As the initialization phase requires calling the initialization of each transition only once, the total time for the initialization will typically be substantially smaller compared to the time required by the execution phase. The next section will investigate the time requirements of the execution phase.

7.2.2 Execution Phase

The simulation of discrete-event models usually consists of processing events, scheduling new events, and, if necessary, canceling events. Under the proposed framework, the execution phase of the simulation consists entirely of the firing of transitions. Each firing usually corresponds to a real-world event; however, some firings of transitions are "artificial". Each firing of a transition consists of three phases:

1. Remove tokens
2. Send tokens
3. Update time

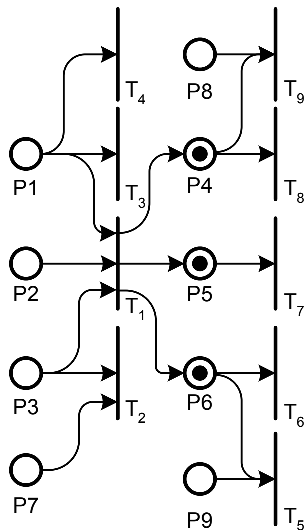

Figure 83: Neighborhood of a Firing Transition

7.2.2.1 Token Removal

Algorithm Remove Tokens was introduced in Section 3.3.5.5. For each place $p \in {}^{\bullet}t$, with t being the firing transition, the method `removeTokens()` is called. In Figure 83, T_1 is the firing transition and these places are $\{P_1, P_2, P_3\}$.

Method `removeTokens()` will identify all tokens with a timestamp less or equal to the current simulation time. Tokens with timestamps greater than the current simulation time are not yet available for the transition and therefore cannot be removed. Among the identified tokens, the method will choose the token with the highest priority. If the arc weight between the place p and the transition t is $w(p,t)$, the first $w(p,t)$ tokens with the highest priority will be removed by this method.

The algorithm first needs to traverse through the set of tokens until it reaches the token with the appropriate timestamp. The object that holds the tokens is also a red-black tree. Assuming there are n tokens in the place, the traversal will take of the order $O(n)$ time. Each of the tokens is then added to a new temporary red-black tree that holds all the tokens with appropriate timestamps. This operation will take $O(\log n)$ time for each token. When the tokens are added, they are inserted according to their priority. The first k tokens in this temporary set represent the k tokens to be removed from the place. Removing each of these tokens from the original place also takes $O(\log n)$ time. Hence, the runtime to remove a token from one place will be of the order $O(n)$ time. As discussed in Chapter 6, most places are bounded in the PN, which means that there is an upper bound for the possible number of tokens and therefore for the runtime of their removal. As a token removal has to be executed for each place in ${}^\bullet t$, the total run time for this phase will be of the order $O({}^\bullet t)$.

7.2.2.2 Sending Tokens

The next step is to send tokens to the places in $t^\bullet$. This simply means to adding one or more tokens depending on the arc weight to every place in $t^\bullet$. In Figure 83 these places are $\{P_4, P_5, P_6\}$. Each operation can also be performed in $O(\log(n))$ time (given the red-black tree). The total number of add operations is $|t^\bullet|$. Hence the total runtime of `sendTokens()` will also be $O(|t^\bullet|)$, with the approximation of each add operation as a constant.

7.2.2.3 Update time

Finally, the method `updateTime()` (analyzed in Section 7.2.1) has to be called on each of the transitions in the field `toUpdate`, which was analyzed in Section 7.2.1. After this, the FEL will contain all the current enabled transitions, with the first transition in the list

being the next firing transition.

7.2.3 Memory Requirements

Another important consideration is the memory requirements to store the simulation model. This section explores the driving forces for the memory requirements.

The PN simulation model consists of a set of transitions and a set of places. These form the static elements of the model. Tokens are the dynamic elements that are created, removed from places, and added to places. There are also other objects, mainly the simulation data objects that hold the simulation data specification. These are of auxiliary character and will not be included in the analysis as they are only needed during the generation phase.

Each transition requires a certain amount of memory to be stored as it has fields that hold references to the places it is connected to as well as fields for identifier, times, etc. (see also Section 3.3.4). Once the PN simulation model is created, the amount of memory to hold transitions will be constant over time. Each place also requires a certain amount of memory. Each place has fields for identifier and references to the transitions that the place is connected to. These fields will also require a constant amount of memory. In addition to that, each place has a field that holds the token objects that are currently in that place. The memory requirements for this will fluctuate over time as the number of tokens will fluctuate.

In summary, there are fixed memory requirements to store the PN structure and variable memory requirements depending on the total number of tokens in the system. Therefore, a simple measure to compare the memory requirements of the PN simulation models is the sum of the number of places and transitions, i.e., $|T|+|P|$. Here, it will only be used to compare the memory requirements of the simulation models. In order to determine the actual memory usage, factors can be estimated by analyzing each of the objects. Unfortunately, as the simulation framework is implemented in Java, there is no function that directly allows the determination of the size of a given object.

A critical factor is the number of tokens in the system. A PN simulation model that is flooded with a large number of tokens can run out of memory. This is not necessary a

real problem, as this indicates that the corresponding real-world model will not be able to handle the desired throughput.

7.3 Measures of Complexity for the Framework

This section attempts to derive a measure that can be used to predict the execution speed of the simulation. A naive approach is to assume that the speed of a simulation model is always constant for all simulation models. This might be true for similar simulation models. However, it is unlikely to be true for all simulation models.

The execution time of a simulation model will be determined by the time to process events and the number of events that have to be processed. For the PN simulation model the time to process an event is equivalent to the time it will take to fire a transition, thus the total execution time can be written as $T_{\text{tot}} = \bar{t}_f \cdot K$, where T_{tot} is the total time to run the simulation, $\bar{t}_f$ is the average time to fire, and K is the total number of transitions to fire.

The number of transitions to be fired will depend on the number of process routes in the PN simulation model, the number of process steps, and the rate with which jobs are released into the system. Differences in the firing rate will indicate structural differences in the simulation model.

In order to explain the differences in the firing rates one has to look at the PN data structure. As discussed in Section 7.2.2, firing of a transition consists of three parts: removing tokens, adding tokens to the new places, and updating all affected transitions.

The time to remove the tokens during the first part of the firing of a transition is mainly dependent on the size of the set of input places of the transition. For each input place, the method `removeTokens()` has to be called. Figure 83 shows the neighborhood of places and transitions of a transition T_1 just after it has fired. The set of places $\{P_1, P_2, P_3\}$ form the set of input places for transition T_1, hence for each of these places the method `removeTokens()` has been called. It can be expected that the time will grow linearly with the size of ${}^{\bullet}T_1$, assuming that the removal of a token will take constant time. Note that the actual time will depend on the number of tokens in those places, which is not known in advance. If

the time for the `removeTokens()` operation for a given place can be approximated by a constant, the time for removing all tokens when a transition t fires will be $O(|^{\bullet}t|)$.

Similarly, the time to add tokens to the output places of T_1 is mostly determined by the size of the set of output places. Based on the same argument as before, one can expect this time to be $O(|t^{\bullet}|)$.

The most complex operation is the time update operation. The method `updateTime()` has to be called for each transition in the set `t.toUpdate` because these transitions could be affected by the firing of transition T_1 in Figure 83. This set is $\{T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9\}$. As shown in Section 7.2.1, the run time for each individual call to `updateTime()` for a single transition is of the order $O(|^{\bullet}t|)$.

Base on the aforementioned analysis, several measures are conceivable for the time required to fire a transition t. The time required for the token removal phase can be captured by $|^{\bullet}t|$, i.e., the size of the set of input places. The time required to add tokens to the places in the output set will be approximately proportional to $|t^{\bullet}|$. These measures, however cannot capture completely the time requirements for the more complex time update phase. A more appropriate measure for this phase is $\sum_{t' \in t.toUpdate} |^{\bullet}t'|$. This expression represents the summation of all input places of the transitions. It can also capture indirectly the time requirements for the remaining two phases. This is because one can expect the sizes of $|t^{\bullet}|$ and $|^{\bullet}t|$ to grow with the size of the set `t.toUdate` as the transitions in `t.toUdate` are connected to t through the places in $t^{\bullet}$ and $^{\bullet}t$.

Clearly, not every transition requires the same amount of computational time as discussed above. Some transitions that are connected to many places and transitions in their immediate neighborhood will certainly require more time than transitions with only few places and transitions in their neighborhood. Therefore, a weighing factor for each transition is used. Using the weight $w = \sum_{t' \in t.toUpdate} |^{\bullet}t'|$ for each transition t, the model complexity of the PN simulation model with the set of transitions T and set of places P will be defined as

$$\Psi(PN) = \frac{\sum_{t \in T} (\sum_{t' \in t.toUpdate} |^{\bullet}t'|)}{|T|}. \quad (7)$$

This measure does not depend on the actual size of the PN and will be used to explain differences in the firing rate of the PN simulation model.

7.4 Experimental Results

In this section the measure in Equation (7) is applied to some of the Sematech data sets. The runtimes were obtained on a PC with an Intel T2050 CPU @ 1.6 GHz and 1 GB main memory under Windows XP. Tables 3 and Table 4 display the results for each data set for 100 and 500 days of simulated time, respectively.

Table 3: Runtime for 100 Days of Simulation Time

Data Set	Runtime [s]	# Firings	Firing rate [$1/s$]	Avg. Firing Duration [μs]
1	9	257,742	28,638	34.92
2	308	8,259,534	26,817	37.29
3	265	4,117,077	15,536	64.37
4	2	140,438	70,219	14.24
5	149	925,153	6,209	161.05
6	400	794,678	1,986	503.35

Table 4: Runtime for 500 Days of Simulation Time

Data Set	Runtime [s]	# Firings	Firing rate [$1/s$]	Avg. Firing Duration [μs]
1	11	302,669	27,515	36.34
2	1,737	46,294,332	26,652	37.52
3	1,911	21,642,961	11,325	88.30
4	2	156,031	78,016	12.82
5	1218	7,312,023	6,003	166.57
6	1380	2,598,718	1,883	531.03

Clearly, the average firing rate can vary greatly between different simulations. The maximum and the minimum times differ by almost a factor of 30. For further analysis only the larger data sets are considered. The runtime for data sets 1 and 4 are fairly small, but the average firing rates are very high. In order to avoid the influence of caching of main memory, these data sets are not considered. Table 5 compares only data sets where $|P|+|T|$ is of the same magnitude. Caching can accelerate the simulation performance extremely for

small models that can be stored entirely in the CPU cache because the access time to this type of memory is an order of magnitude faster than the access times to RAM. Hence it is difficult to make any comparison between very small models and very large models.

Table 5: Overview of Complexity Measures

Data Set	$\lvert P\rvert$	$\lvert T\rvert$	$\lvert P\rvert + \lvert T\rvert$	$\Psi(PN)$
2	19,751	19,303	39,054	39.61
3	31,845	27,109	58,954	228.91
5	36,075	31,661	67,736	678.23
6	25,463	22,097	47,560	1,506.91

Table 5 gives an overview of the data sets. Sets 2, 3, 5, and 6 are of comparable size as $|P| + |T|$ is of the same magnitude. Set 6 has the highest value for $\Psi(PN)$, which indicates that it will have the longest mean firing duration. The total number of transitions is not as high as for other data sets. This data set has only eight product routes and an average number of process steps of over 300. This indicates that the average transition is connected to many other places and transitions, which will increase the average size of the `toUpdate` field of each transition, which will also increase the value of $\Psi(PN)$.

Figure 84 shows data points for simulation runs of length 100 days and 500 days. An almost linear relationship between the firing rate and $\Psi(PN)$ can be observed. The measure $\Psi(PN)$ does not take congestion of the system into account. A system with a low release rate will have fewer tokens. A very congested system will certainly require more time as the constant time approximations for removing and adding tokens do not hold when the number of tokens become large in some places.

The data points for the 500 day simulation runs are all below the data points of the 100 day runs, except for the leftmost data points that are almost identical (i.e., 37.29 μs for 100 day run and 37.52 μs for 500 day run). This can be expected, as the longer runtime would result in a higher congestion of the system. The system is empty at the onset of the simulation run, hence the average number of tokens in a buffer place will be low. Over time the system fills up with jobs that are waiting to be processed, which will lead to more tokens in the buffer places.

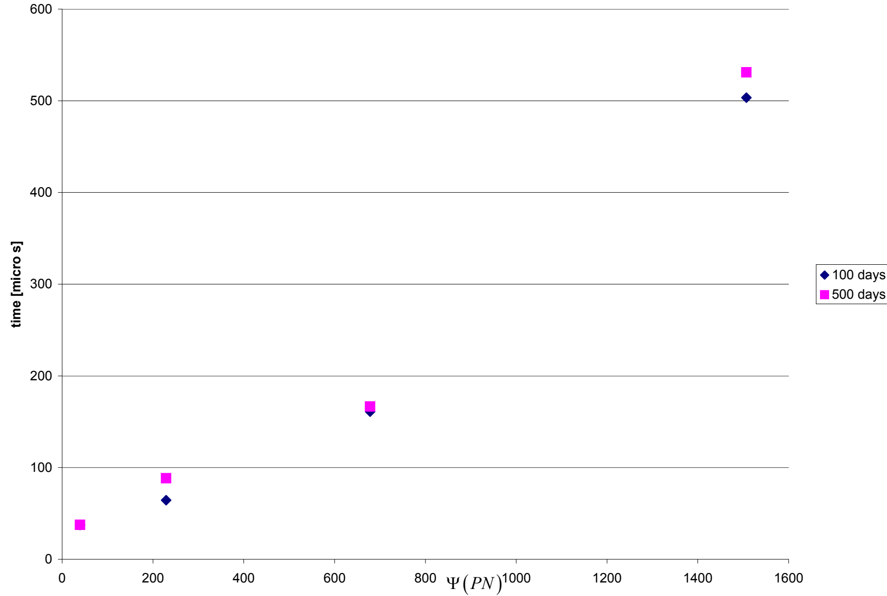

Figure 84: Average Firing time vs. $\Psi(PN)$

7.5 Conclusions

This chapter analyzed the execution algorithm that is used in the simulation framework. As the execution of the PN simulation model involves primarily transition firing, the analysis on a detailed level is possible. The analysis indicated a good scalability for large-scale simulation models. Further, the measure $\psi(PN)$ for simulation model complexity was introduced. Tests run on Sematech data sets of similar size showed significant differences in the average firing time. An almost linear relationship was observed between $\Psi(PN)$ and the average firing time of a transition. Hence the measure $\Psi(PN)$ appears to be a good indicator for the speed of a simulation model created with the proposed framework. The experiments also indicated good scalability for long simulation runs, as the firing rate remains roughly constant during a run.

CHAPTER VIII

CONCLUSIONS AND FUTURE RESEARCH

8.1 Summary and Conclusions

This thesis introduced a novel approach to simulation model generation. The grand challenges in manufacturing simulation provided the motivation for developing the proposed simulation framework. We started with a review of the principles of simulation modeling, reviewed existing simulation model specifications and modeling frameworks, and discussed their shortcomings.

The proposed simulation framework allows the effective generation of large-scale simulation models, based on the example of semiconductor manufacturing. The contributions of this approach are:

- A PN data structure (PNSM) that can be analyzed directly.
- The simulation model is not represented in a simulation language and does not have to be compiled.
- A simulation execution mechanism that is based on a PN with extension for time and priorities and allows flexible modeling of various dispatch rules.
- This data structure (PNSM) describes the behavior of the simulation model in its entirety, hence closing the gap between conceptual model and simulation model.
- This data structure (PNSM) can serve as a basis for the use of simulation for real-time decision making as it can be altered at any time.
- Introduction of an object model for a simulation data specification.
- Introduction of detailed mappings from this data specification to PN, with verifiable properties.
- Introduction of conditions for a legitimate simulation model.

- Introduction of a measure for simulation model complexity.

The core elements in the implementation of the PN framework are objects representing places, transitions, and tokens. Hence, the graph structure of the PN is directly modeled within the simulation model. The execution of the PNSM consists only of firing of transitions. The simulation model is not defined in terms of simulation code in a particular simulation language, and therefore no executable file is complied. An instance of the PNSM is created by populating a data structure that represents the simulation model. This opens new possibilities for creating online simulation models, i.e., simulation models that can represent the current state of a manufacturing system and can be used for real-time decision making. This is true because the simulation model can be altered at any time.

This framework has some unique novel features. It uses as a basis the same execution rules as classical PNs, yet it has extensions for time and priorities for firing of transitions. This allows the representation of time as well as the implementation of various dispatch rules. This is not directly possible with classical and colored Petri nets, as the firing of transitions is undetermined. In manufacturing simulation models, it is necessary to establish an order between enabled transitions. The only mechanism available for classical or colored PN is the use of inhibitor arcs. This is, however, an inflexible approach.

It has been shown that the simulation mechanism will not affect liveness and boundedness. Hence, it is possible to use standard analysis methods for liveness and boundedness directly on the generated PNSM.

The simulation model generation is based on an object model, which serves as 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 PNSM.

The 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, corresponds to parts in the PN. This mapping serves also as an unambiguous description of the simulation model.

This closes the gap between the conceptual model and implementation of a simulation model.

The mapping from the data specification to the PN follows a process-oriented view. Each process step is described in terms of the PN. The behavior of the process step is encapsulated within a sub-PN, yet these sub-nets are connected in specific ways to other parts in the PN that represent resources and control structures. Each of the sub-PNs are verifiable live, bounded, and reversible. These properties were also shown to be true for the complete generated PN simulation model.

Sufficient conditions for a legitimate simulation model have also been introduced. These conditions will ensure that the simulation model will be able to advance time.

Finally, we developed a measure to estimate simulation model complexity. This measure allows to measure simulation runtime speed. There are only a few small-scale examples in the literature for measures of this type.

In summary, the proposed simulation framework is intended to make the creation of large-scale discrete-event (computer) simulation (DES) models for manufacturing systems more manageable. It has several advantages: (1) There is an unambiguous description of the simulation model in the form of a PN. (2) The user can verify exactly how each component is working. (3) The user does not have to code 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 a finite state system can be modeled. However, there are some disadvantages. Due to the rigid control, it is not easily possible to make arbitrary changes in the behavior of the simulation model because this behavior is determined by the mapping from the object model to the PN simulation model. Hence, this mapping has to be altered accordingly.

Another practical concern is the modeling of material handling systems on a detailed

level. For example, in order to model a conveyor, the conveyor would have to be discretized in some way.

8.2 Future Research

There are many interesting possibilities for research is this area. Considering the steady increase of computing power, effective simulation modeling is the key for many new applications. For instance, the application domain can be extended to other types of manufacturing systems. One area that has not been treated here is assembly processes, which involve merging of different product streams. This can be easily modeled in terms of a PN. On a broader scale, this approach can be extended to modeling and simulating entire supply chains or networks.

Another area would be a tighter integration of the simulation framework with existing product and resource data. Here an object-oriented model for a semiconductor manufacturing system was developed, which contains all the necessary information for the simulation model generation. In the same fashion, it is possible to create a standard model that can specify a manufacturing model and supply chain. This model can then be used as a basis to generate a PN simulation model. If such modeling standards are available, they could lead to entirely new usages of simulation, e.g., integration of simulation for real-time decision making. This would obviously require a lot of groundwork.

All problems mentioned above are application-oriented. A more fundamental research area involves the use of parallel and distributed computing for PN simulation models. The research question here is how to take advantage of the graphical representation of the simulation model in form of a PN. It is clear that there are transitions in a PN that can fire independently from each other, as they are in different areas of the PN. The challenge is to find a suitable time advance mechanism such that causality is not violated. Further, an appropriate segmentation algorithm would have to be found that allows for a parallel execution of the PN simulation model.

APPENDIX A

SIMULATION DATA SPECIFICATION

A.1 Data Description

Table 6 explains in detail the meaning of each field for each object. The explanation is based on the information given in [1]. For a complete overview it, should be viewed together with Figure 26, the object model for the data specification.

Table 6: Description of Simulation Data Specification

Object	Field	Description
`Fabmodel`	`processRoutes`	List of all `ProcessRoute` objects in wafer fab
	`reworkSequence`	List of all `ReworkSequence` objects in wafer fab
	`toolSets`	List of all `ToolSet` objects in wafer fab
	`operatorSets`	List of all `OperatorSet` objects in wafer fab
	`products`	List of all `Product` objects in wafer fab
`ProcessRoute`	`processSteps`	List of all `ProcessStep` objects in process route
`ReworkSequence`	`reworkSteps`	List of all `ProcessStep` objects in rework route, process steps in the rework sequence have same format as normal process steps
`Product`	`id`	`String` representing the unique id of product
	`flowId`	`String` id of the process route of the product
	`name`	`String` name of the product
	`startRate`	`double` release rate to wafer fab in [wafers/day]
	`lotSize`	`int` lot size in wafers
	`processRoute`	`ProcessRoute` object representing the process route of the product

Continued on next page

Table 6 (continued).

Object	Field	Description
ToolSet	id	String representing the unique id of tool set
	toolDescritpion	String describing tool set
	quantity	int # of identical tools available
	operatorLoading	boolean indicating if operator is needed for loading
	operatorUnloading	boolean indicating if operator is needed for unloading
	operatorProcessFraction	double fraction of time operator is needed during processing
	setupModeled	boolean indicating if setup states are modeled
	downTimes	List of DownTime objects
	setupStates	List of String objects with setup state ids
DownTime	id	String representing the unique id of product
	description	String describing downtime
	duration	long duration [msec] of downtime
	timeBetween	long avg. time [msec] between downtimes (exp. distributed)
OperatorSet	id	String representing the unique id of operator set
	descritpion	String describing operator set
	quantity	int # of identical operators available
ProcessStep	stepId	String representing unique id for process step
	processRoute	ProcessRoute object that this object belongs to
	toolSet	ToolSet object that is needed for this process step
	operatorSet	OperatorSet object that is needed for this process step
	operationDescription	String that describes the process step
	loadTime	long time [msec] to load wafer lot into tool
	unloadTime	long time [msec] to unload wafer lot from tool

Continued on next page

Table 6 (continued).

Object	Field	Description
	`timePerWaferInProcess`	`long` time [msec] to process a single wafer in the tool
	`waferTravelTimeWithinTool`	`long` time [msec] wafer spends traveling inside tool, tool can process other lots during travel time
	`timePerLot`	`long` time [msec] to process a wafer lot
	`lotScrapProbability`	`double` probability that the entire lot is scrapped after this operation
	`waferScrapProbability`	`double` probability that wafer is scrapped after this operation
	`lotReworkProbability`	`double` probability that lot has to go through rework sequence
	`reworkSequenceId`	`String` id of rework sequence
	`returnStepId`	`String` id of step where reworked lot will enter the original sequence
	`travelTime`	travel time [msec] to next tool
	`travelOperatorSet`	`OperatorSet` object needed for travel to next tool
`BatchProcessStep` (subclass of `ProcessStep`)	`minBatchSize`	`int` minimum number of batches that must be present to start processing
	`maxBatchSize`	`int` maximum number of batches that can be processed at once
	`individualWaferModeled`	`boolean` indicating if lot is separated into single wafers for processing
	`timePerBatch`	`long` time [msec] to process single batch
	`batchID`	`String` id of batch, lots from other process steps with the same `batchID` can be batched together
`ProcessStepWithSetup` (subclass of `ProcessStep`)	`timePerSpecSetup`	`long` time [msec] for setup, has to be performed at the beginning of each processing
	`timePerGroupSetup`	`long` time [msec] for group

Continued on next page

Table 6 (continued).

Object	Field	Description
		specific setup, has to be performed each time the tool changes to a different setup group
	`setupGroupId`	`String` representing the setup group of this process step

A.2 Processing Times

The Sematech data sets specifies the following formulae for calculating processing time per lot (`pt`), time until tool becomes free (`tf`), and total lot cycle time through an operation (`ct`):

$$\begin{aligned}\texttt{pt} =& \texttt{timePerBatch} * \text{Number of batches required for the lot} \\ &+ \texttt{timePerLot} + \texttt{timePerWaferInProcess} * \texttt{lotSize} \\ &+ \texttt{timePerSpecSetup}(\text{if appropriate}) + \texttt{timePerGroupSetup}(\text{if appropriate})\end{aligned} \tag{8}$$

Note that in most cases only one of the three times will be non zero, i.e., the processing time is either given as time per batch, time per lot, or time per wafer. The number of batches required for the lot is usually one. It can be greater than one for batch process steps that have a capacity that is less than the lot size; then, the lot has to be split into several smaller batches. One has

$$\texttt{tf} = \texttt{loadTime} + \texttt{pt} + \texttt{unloadTime} \tag{9}$$

$$\texttt{ct} = \texttt{loadTime} + \texttt{pt} + \texttt{waferTravelTime} + \texttt{unloadTime}. \tag{10}$$

A.3 XML Schema for Data Specification

This XML schema is using the XMI standard, which is a standard for defining, interchanging, manipulating and integrating XML data and objects. XMI provides rules by which a schema can be generated for any valid XMI-transmissible MOF-based metamodel. Details are availabe in [2].

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:fabmodel="http:///fabmodel.ecore"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http:///fabmodel.ecore">
  <xsd:import namespace="http://www.omg.org/XMI"
```

```
schemaLocation="../../../plugin/org.eclipse.emf.ecore/model/XMI.xsd"/>
<xsd:complexType name="BatchProcessStep">
  <xsd:complexContent>
    <xsd:extension base="fabmodel:ProcessStep">
      <xsd:attribute name="minBatchSize" type="xsd:int"/>
      <xsd:attribute name="maxBatchSize" type="xsd:int"/>
      <xsd:attribute name="individualWaferModeled"
       type="xsd:boolean"/>
      <xsd:attribute name="timePerBatch" type="xsd:long"/>
      <xsd:attribute name="batchId" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="BatchProcessStep" type="fabmodel:BatchProcessStep"/>
<xsd:complexType name="DownTime">
  <xsd:choice maxOccurs="unbounded" minOccurs="0">
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="description" type="xsd:string"/>
  <xsd:attribute name="duration" type="xsd:long"/>
</xsd:complexType>
<xsd:element name="DownTime" type="fabmodel:DownTime"/>
<xsd:complexType name="DownTimeRunBased">
  <xsd:complexContent>
    <xsd:extension base="fabmodel:DownTime">
      <xsd:attribute name="runsBetween" type="xsd:int"/>
    </xsd:extension>
```

```
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DownTimeRunBased" type="fabmodel:DownTimeRunBased"/>
<xsd:complexType name="DownTimeTimeBased">
  <xsd:complexContent>
    <xsd:extension base="fabmodel:DownTime">
      <xsd:attribute name="timeBetween" type="xsd:long"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DownTimeTimeBased" type="fabmodel:DownTimeTimeBased"/>
<xsd:complexType name="FabModel">
  <xsd:choice maxOccurs="unbounded" minOccurs="0">
    <xsd:element name="toolSets" type="fabmodel:ToolSet"/>
    <xsd:element name="processRoutes" type="fabmodel:ProcessRoute"/>
    <xsd:element name="operatorSets" type="fabmodel:OperatorSet"/>
    <xsd:element name="products" type="fabmodel:Product"/>
    <xsd:element name="reworkSequences" type="fabmodel:ReworkSequence"/>
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="id" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="FabModel" type="fabmodel:FabModel"/>
<xsd:complexType name="OperatorBreak">
  <xsd:choice maxOccurs="unbounded" minOccurs="0">
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
```

```
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="description" type="xsd:string"/>
  <xsd:attribute name="timeBetween" type="xsd:long"/>
  <xsd:attribute name="duration" type="xsd:long"/>
</xsd:complexType>
<xsd:element name="OperatorBreak" type="fabmodel:OperatorBreak"/>
<xsd:complexType name="OperatorSet">
  <xsd:complexContent>
    <xsd:extension base="fabmodel:Comparable">
      <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="operatorBreaks"
            type="fabmodel:OperatorBreak"/>
      </xsd:choice>
      <xsd:attribute name="id" type="xsd:string"/>
      <xsd:attribute name="description" type="xsd:string"/>
      <xsd:attribute name="quantity" type="xsd:int"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="OperatorSet" type="fabmodel:OperatorSet"/>
<xsd:complexType name="ProcessRoute">
  <xsd:choice maxOccurs="unbounded" minOccurs="0">
    <xsd:element name="product" type="fabmodel:Product"/>
    <xsd:element name="processSteps" type="fabmodel:ProcessStep"/>
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
```

```
  <xsd:attribute name="id" type="xsd:string"/>
  <xsd:attribute name="product" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="ProcessRoute" type="fabmodel:ProcessRoute"/>
<xsd:complexType name="ProcessStep">
  <xsd:choice maxOccurs="unbounded" minOccurs="0">
    <xsd:element name="processRoute" type="fabmodel:ProcessRoute"/>
    <xsd:element name="toolSet" type="fabmodel:ToolSet"/>
    <xsd:element name="operatorSet" type="fabmodel:OperatorSet"/>
    <xsd:element name="travelOperatorSet" type="fabmodel:OperatorSet"/>
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="stepId" type="xsd:string"/>
  <xsd:attribute name="operationDescription" type="xsd:string"/>
  <xsd:attribute name="loadTime" type="xsd:long"/>
  <xsd:attribute name="unLoadTime" type="xsd:long"/>
  <xsd:attribute name="timePerWaferInProcess" type="xsd:long"/>
  <xsd:attribute name="waferTravelTimeWithinTool" type="xsd:long"/>
  <xsd:attribute name="timePerLot" type="xsd:long"/>
  <xsd:attribute name="lotScrapProbability" type="xsd:double"/>
  <xsd:attribute name="waferScrapProbability" type="xsd:double"/>
  <xsd:attribute name="lotReworkProbability" type="xsd:double"/>
  <xsd:attribute name="reworkSequenceId" type="xsd:string"/>
  <xsd:attribute name="returnStepId" type="xsd:string"/>
  <xsd:attribute name="travelTime" type="xsd:long"/>
  <xsd:attribute name="processRoute" type="xsd:string"/>
  <xsd:attribute name="toolSet" type="xsd:string"/>
```

```
  <xsd:attribute name="operatorSet" type="xsd:string"/>
  <xsd:attribute name="travelOperatorSet" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="ProcessStep" type="fabmodel:ProcessStep"/>
<xsd:complexType name="ProcessStepWSetup">
  <xsd:complexContent>
    <xsd:extension base="fabmodel:ProcessStep">
      <xsd:attribute name="timePerSpecSetup" type="xsd:long"/>
      <xsd:attribute name="timePerGroupSetup" type="xsd:long"/>
      <xsd:attribute name="setupGroupId" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ProcessStepWSetup" type="fabmodel:ProcessStepWSetup"/>
<xsd:complexType name="Product">
  <xsd:complexContent>
    <xsd:extension base="fabmodel:Comparable">
      <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="processRoute" type="fabmodel:ProcessRoute"/>
      </xsd:choice>
      <xsd:attribute name="id" type="xsd:string"/>
      <xsd:attribute name="flowId" type="xsd:string"/>
      <xsd:attribute name="name" type="xsd:string"/>
      <xsd:attribute name="startRate" type="xsd:double"/>
      <xsd:attribute name="lotSize" type="xsd:int"/>
      <xsd:attribute name="processRoute" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

```
<xsd:element name="Product" type="fabmodel:Product"/>
<xsd:complexType name="ReworkSequence">
  <xsd:choice maxOccurs="unbounded" minOccurs="0">
    <xsd:element name="reworkSteps" type="fabmodel:ProcessStep"/>
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="id" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="ReworkSequence" type="fabmodel:ReworkSequence"/>
<xsd:complexType name="ToolSet">
  <xsd:complexContent>
    <xsd:extension base="fabmodel:Comparable">
      <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="setupStates" nillable="true"
           type="xsd:string"/>
        <xsd:element name="downTimes" type="fabmodel:DownTime"/>
      </xsd:choice>
      <xsd:attribute name="id" type="xsd:string"/>
      <xsd:attribute name="toolDescription" type="xsd:string"/>
      <xsd:attribute name="quantity" type="xsd:int"/>
      <xsd:attribute name="operatorLoading" type="xsd:boolean"/>
      <xsd:attribute name="operatorUnloading" type="xsd:boolean"/>
      <xsd:attribute name="operatorProcessFraction" type="xsd:double"/>
      <xsd:attribute name="setupModeled" type="xsd:boolean"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

```
  <xsd:element name="ToolSet" type="fabmodel:ToolSet"/>
  <xsd:complexType abstract="true" name="Comparable">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  </xsd:complexType>
  <xsd:element name="Comparable" type="fabmodel:Comparable"/>
</xsd:schema>
```

APPENDIX B

PETRI NET GENERATION ALGORITHMS

The following pseudo-code is based on Java, but it is not in compilable form. Unimportant details have been omitted to improve readability. Comments start with "//".

B.1 Generate Tool Sets

Generate all tool sets for fabmodel:

```
for all tool sets ts in fabmodel{
    if (ts.setupModeled) {
        for each setup state s in ts.setupStates{
            t = new Tool();
            t.id = ts.getId()+"_S_"+setupStateId;
        }
        q = ts.quantity;
        for i = 0 to q{
            t.addToken(new Token());
        }
        createBreakDowns2();
    }else{
        t = new Tool();
        t.id = ts.id;
        q = ts.quantity ;
        for i = 0 to q{
            t.addToken(new Token());
        }
        createBreakDowns1();
    }
}
```

B.1.1 CreateBreakdowns1()

Create breakdowns for toolSet ts:

```
for each downtime d in toolset ts {
    timeBetwn = d.TimeBetween / toolSet.getQuantity();
    tr = new TriggerTransition();
    tr.setDelay(timeBetwn);
    breakDown = new ControlPlace();
    delay = new FixedMaxPriorityTransition();
```

```
    delay.setDelay(downTime.getDuration());
    tr.addOutPlace(breakDown);
    delay.addInPlace(breakDown);
    delay.addOutPlace(t);
    delay.addInPlace(t);
}
```

B.1.2 CreateBreakdowns2()

Create breakdowns for toolSet ts, with setup states:

```
for each downtime d in toolset ts {
    timeBetwn = d.TimeBetween / toolSet.getQuantity();
    tr = new TriggerTransition();
    tr.setDelay(timeBetwn);
    breakDown = new ControlPlace();
    tr.addOutPlace(breakDown);
    for each setup state s in toolSet.setupStates {
        delay = new FixedMaxPriorityTransition();
                delay.setDelay(downTime.duration);
        delay.addInPlace(breakDown);
        t = tool place corresponding to setup state s;
        delay.addOutPlace(t);
    }
}
```

B.2 Create Operator Sets

```
for each operatorSet os in fabModel.operatorSets{
    o = new Operator();
    for i = o to os.quantity{
        o.addToken(new Token());
    }
}
```

B.3 Create Process Routes

```
for each ProcessRoute pr in FabModel fm {
    flowId = pr.id;
    lotSize = pr.product.lotSize;
    startPlace = new ProcessPlace();
    lastPlace = startPlace;
    for each ProcessStep ps in pr{
        lastPlace = createProcesStep(...);
    }
}
```

The method createProcessStep(...) will call the appropriate method according to the type of process step.

B.3.1 Create Basic Process Step

Creates places and transitions for process step `ps`.

```
operatorModeled = ps.operatorSet != null ? true : false;
toolSeized = false;
operatorSeized = false;
prioritySet = new ProcessPlace();
setPriority = new Transition();
setPriority.setDispatchRule(dispatchRule);
setPriority.addInPlace(lastPlace);
setPriority.addOutPlace(prioritySet);
lastPlace = prioritySet;
ts = ps.toolSet;
tool = tool place of required tool;
operator = null;
if (operatorModeled) {
   operator = operator place of required operator;
}
delay = ps.timePerLot+ps.timePerWaferInProcess*lotSize;
// loading
if (processStep.getLoadTime() > 0) {
   loading = new ProcessPlace();
   startLoading = new Transition());
   startLoading.setDelay(processStep.loadTime);
   startLoading.addInPlace(lastPlace);
   startLoading.addInPlace(tool);
   startLoading.addOutPlace(loading);
   toolSeized = true;
   if (operatorModeled && ts.operatorLoading) {
      startLoading.addInPlace(operator);
      operatorSeized = true;
   }
   lastPlace = loading;
}
// processing with operator
if (operatorModeled) {
   inProcessWOperator = new ProcessPlace();
   startProcessingWOperator = new Transition();
   startProcessingWOperator.setDelay(
      delay*ts.operatorProcessFraction);
   if (lastPlace instanceof BatchPlace)
      startProcessingWOperator
         .addInJobBatchPlace( lastPlace);
   else
      startProcessingWOperator.addInPlace(lastPlace);
   startProcessingWOperator.addOutPlace(inProcessWOperator);
   if (not toolSeized) {
      startProcessingWOperator.addInPlace(tool);
```

```
        toolSeized = true;
    }
    if (not operatorSeized &&
        not (ts.operatorProcessFraction == 0)) {
        startProcessingWOperator.addInPlace(operator);
        operatorSeized = true;
    }
    lastPlace = inProcessWOperator;
}
// processing w/o operator
if (not operatorModeled || ts.operatorProcessFraction < 1.0) {
    inProcessWoutOperator = new ProcessPlace();
    startProcessingWoutOperator = new Transition();
    startProcessingWoutOperator.setDelay(
        delay*(1 - ts.operatorProcessFraction));
    if (lastPlace instanceof BatchPlace)
        startProcessingWoutOperator
                            .addInJobBatchPlace(lastPlace);
    else
        startProcessingWoutOperator.addInPlace(lastPlace);
    startProcessingWoutOperator.addOutPlace(inProcessWoutOperator);
    if (not toolSeized) {
        startProcessingWoutOperator.addInPlace(tool);
        toolSeized = true;
    }
    if (operatorSeized && operatorModeled) {
        startProcessingWoutOperator.addOutPlace(operator);
        operatorSeized = false;
    }
    lastPlace = inProcessWoutOperator;
}
// unloading
if (processStep.getUnLoadTime() > 0) {
    unloading = new ProcessPlace();
    startUnloadLoading = new Transition();
        startUnloadLoading.setDelay(processStep.unLoadTime);
    startUnloadLoading.addInPlace(lastPlace);
    startUnloadLoading.addOutPlace(unloading);
    if(not operatorSeized && operatorModeled &&
        ts.operatorUnloading){
        startUnloadLoading.addInPlace(operator);
        operatorSeized = true;
    }
    lastPlace = unloading;
}
// wafertraveltime
if (processStep.getWaferTravelTimeWithinTool() > 0) {
    inToolTransport = new ProcessPlace();
```

```
        startInToolTransport = new Transition();
        startInToolTransport.setDelay(ps.waferTravelTimeWithinTool);
        startInToolTransport.addInPlace(lastPlace);
        startInToolTransport.addOutPlace(inToolTransport);
        if (toolSeized) {
            startInToolTransport.addOutPlace(tool);
            toolSeized = false;
        }
        if (operatorSeized) {
            startInToolTransport.addOutPlace(operator);
            operatorSeized = false;
        }
        lastPlace = inToolTransport;
    }
    // travelling to next tool
    if (processStep.getTravelTime() > 0) {
        releaseOperator = new ProcessPlace();
        release = new Transition();
        release.addInPlace(lastPlace);
        release.addOutPlace(releaseOperator);
        lastPlace = releaseOperator;
        if (toolSeized) {
            release.addOutPlace(tool);
            toolSeized = false;
        }
        if (operatorSeized) {
            release.addOutPlace(operator);
            operatorSeized = false;
        }
        transportToNextTool = new ProcessPlace();
        startTransportToNextTool = new Transition();
        startTransportToNextTool.setProcessingTime(
            ps.timePerLot + ps.timePerWaferInProcess * lotSize);
        startTransportToNextTool.addInPlace(lastPlace);
        startTransportToNextTool.addOutPlace(transportToNextTool);
        if (processStep.travelOperatorSet != null) {
            travelOperator = operator place of required travel operator;
            startTransportToNextTool.addInPlace(travelOperator);
        }
        lastPlace = transportToNextTool;
    }
    // end
    end = new ProcessPlace();
    endT = new Transition();
    endT.addInPlace(lastPlace);
    endT.addOutPlace(end);
    if (toolSeized) {
        endT.addOutPlace(tool);
```

```
    toolSeized = false;
    }
if (operatorSeized) {
    endT.addOutPlace(operator);
    operatorSeized = false;
}
if (ps.travelOperatorSet != null) {
    travelOperator = operator place of required travel operator;
    endT.addOutPlace(travelOperator);
}
if (ps has a rework sequence) {
    switchP = new ProcessPlace();
    reworkStart = new ProcessPlace();
    reworkEnd = place of return step;
    switchT = new SwitchTransition(),
    switchT.setSwitchProbability(ps.lotReworkProbability);
    switchT.addInPlace(end);
    switchT.addOutPlace(switchP);
    if (reworkEnd == null) {
        reworkStepsToFinish.add(ps.reworkSequence);
    } else {
        createReworkSequence();
    }
    end = switchP;
}
if (ps.lotScrapProbability > 0) {
    scrapP = new ProcessPlace();
    scrapEnd = new ProcessPlace();
    scrapT = new SwitchTransition();
    scrapT.setSwitchProbability(ps.lotScrapProbability);
    scrapT.addInPlace(end);
    scrapT.addOutPlace(scrapEnd);
    end = scrapEnd;
}
return end;
```

B.3.2 Batch Process Step

Creates places and transitions for batch process step `bs`.

```
toolSet = place of required tool set;
operatorSet = place of required operator set;
operatorModeled = bs.operatorSet != null ? true : false;
minBatchSize = bs.minBatchSize;
maxBatchSize = bs.maxBatchSize;
if (not bs.isIndividualWaferModeled()) {
    minBatchSize = minBatchSize / lotSize +1;
    maxBatchSize = maxBatchSize / lotSize;
```

```
} else {
   t = new Transition();
   b = new BatchPlace();
   this.weightIn = lotSize;
   if (lastPlace instanceof BatchPlace)
      t.addInJobBatchPlace(lastPlace);
   else
      t.addInPlace(lastPlace);
   t.addOutJobBatchPlace(b);
   lastPlace = b;
}
operator = null;
if (bs.operatorSet != null)
   operator = place of required operator;
tool = place of required tool;
// sp is an object that represents a
// set of places and transitions representing
// the batch process place
sp = getSharedBatchProcesStep();
loadingModeled = bs.loadTime > 0 ? true : false;
unloadingModeled = bs.unLoadTime > 0 ? true : false;
//if the batch process step has not been created yet
if (sp == null) {
   if (loadingModeled && unloadingModeled)
      sp = createSharedBatchProcessingStepLoadingAndUnloading();
   if (not loadingModeled && unloadingModeled)
      sp = createSharedBatchProcessingStepUnLoading();
   if (loadingModeled && not unloadingModeled)
      sp = createSharedBatchProcessingStepLoading();
   if (not loadingModeled && not unloadingModeled)
      sp = createSharedBatchProcessingStep();
}
p0 = lastPlace;
p1 = new ProcessPlace();
p3 = new ProcessPlace();
p4 = null;
if (loadingModeled)
   p4 = new ProcessPlace();
p5 = null;
if (unloadingModeled)
   p5 = new ProcessPlace();
p6 = new ProcessPlace();
parallel = new Transition();
parallel.addInPlace(p1);
parallel.addOutPlace(p3);
parallel.addBatchInPlace(sp.c2);
t0 = new FixedMaxPriorityTransition();
t0.setDispatchRule(dispatchRule);
```

```
t1 = new Transition();
t2 = new FixedMaxPriorityTransition();
t3 = null;
if (loadingModeled) {
   t3 = new FixedMaxPriorityTransition();
}
t4 = new FixedMaxPriorityTransition();
t5 = null;
if (unloadingModeled)
   t5 = new FixedMaxPriorityTransition();
if (p0 instanceof BatchPlace)
   t0.addInJobBatchPlace(p0);
else
   t0.addInPlace(p0);
t0.addOutPlace(p1);
t1.addInPlace(p1);
t1.addOutJobBatchPlace(sp.b1);
t1.addBatchInPlace(sp.c1);
t2.addBatchInPlace(sp.b10);
if (loadingModeled) {
   t3.addBatchInPlace(sp.b20);
   t3.addInPlace(p3);
   t3.addOutPlace(p4);
   t4.addInPlace(p4);
} else {
   t4.addInPlace(p3);
}
t4.addBatchInPlace(sp.b30);
t1.addOutPlace(p2);
t2.addInPlace(p2);
t2.addOutPlace(p3);
t6 = new FixedMaxPriorityTransition();
t6.setProcessingTime(bs.timePerBatch);
if (unloadingModeled) {
   t4.addOutPlace(p5);
   t5.addInPlace(p5);
   t5.addBatchInPlace(sp.b40);
   t5.addOutPlace(p6);
} else {
   t4.addOutPlace(p6);
}
t6.addInPlace(p6);
end = null;
if (bs.isIndividualWaferModeled()) {
   startBatching = new FixedMaxPriorityTransition();
   b = new BatchPlace();
   t6.addOutJobBatchPlace(b);
   startBatching.addInJobBatchPlace(b);
```

```
    end1 = new ProcessPlace();
    startBatching.addOutPlace(end1);
    end = end1;
} else {
    end2 = new ProcessPlace();
    t6.addOutPlace(end2);
    end = end2;
}
if (bs.getTravelTime() > 0) {
    transportToNextTool = new ProcessPlace();
    TransstartTransportToNextTool = new Transition();
    startTransportToNextTool.addInPlace(end);
    startTransportToNextTool.addOutPlace(transportToNextTool);
    startTransportToNextTool.setDelay(bs.travelTime);
    if (bs.travelOperatorSet != null) {
        travelOperator = place or required travel operator;
        startTransportToNextTool.addInPlace(travelOperator);
    }
    last = new ProcessPlace();
    endT = new FixedMaxPriorityTransition();
    endT.addInPlace(transportToNextTool);
    endT.addOutPlace(last);
    if (bs.travelOperatorSet != null) {
        endT.addOutPlace(travelOperator);
    }
    end = last;
}
if (bs has rework sequence) {
    switchP = new ProcessPlace();
    reworkStart = new ProcessPlace();
    reworkEnd = getReworkEnd();
    switchT = new SwitchTransition();
    switchT.setSwitchProbability(bs.reworkProbability);
    switchT.addInPlace(end);
    switchT.addOutPlace(switchP);
    if (reworkEnd == null) {
        reworkStepsToFinish.add(bs.reworkSequence);
    } else {
        createReworkSequence();
    }
}
if (bs.lotScrapProbability > 0) {
    scrapP = new ProcessPlace();
    scrapEnd = new ProcessPlace();
    scrapT = new SwitchTransition();
    scrapT.addInPlace(end);
    scrapT.addOutPlace(scrapEnd);
    end = scrapEnd;
```

```
}
return end;
```

B.3.2.1 Create Shared Batch Process Step

Creates the control places and transitions for batch process step `bs` without loading and unloading.

```
toolSet = place of required tool set;
operatorSet = place of required operator set;
minBatchSize = bs.getMinBatchSize();
maxBatchSize = bs.getMaxBatchSize();
if not(bs.isIndividualWaferModeled()) {
   minBatchSize = minBatchSize / lotSize + 1;
   maxBatchSize = maxBatchSize / lotSize;
}
//this is a helper object
sp = new SharedBatchProcessStep();
processingTime = bs.timePerBatch;
b1 = new BatchPlace();
ProcessPlace b4 = new ProcessPlace();
ProcessPlace b5 = new ProcessPlace();
sp.b1 = b1;
tb1 = new Transition();
tb4 = new FixedMaxPriorityTransition();
tb4.setDelay(processingTime);
tb5 = new FixedMaxPriorityTransition();
tb1.addInJobBatchPlace(b1);
tb1.addInPlace(toolSet);
tb1.addOutPlace(b4);
tb4.addInPlace(b4);
tb4.addOutPlace(b5);
tb5.addInPlace(b5);
tb5.addOutPlace(toolSet);
b10 = new BatchPlace();
tb1.addBatchOutPlace(b10);
sp.b10 = b10;
b30 = new BatchPlace();
tb5.addBatchOutPlace(b30);
sp.b30 = b30;
dispose3 = new FixedMinPriorityTransition();
dispose3.addBatchInPlace(b30);
// this place limits the number of tokens in b1 place
c1 = new BatchPlace();
c1.addToken(new Token());
tb1.addBatchOutPlace(c1);
```

```
sp.c1 = c1;
c2 = new BatchPlace();
sp.c2 = c2;
dispose = new FixedMinPriorityTransition();
tb1.addBatchOutPlace(c2);
dispose.addBatchInPlace(c2);
return sp;
```

B.3.2.2 Create Shared Batch Process Step with Loading

Creates the control places and transitions for batch process step `bs` with loading.

```
toolSet = place of required tool set;
operatorSet = place of required operator set;
operatorModeled = bs.operatorSet != null ? true : false;
minBatchSize = bs.minBatchSize;
maxBatchSize = bs.maxBatchSize;
if not(batchProcessStep.individualWaferModeled) {
   minBatchSize = minBatchSize / lotSize +1;han
   maxBatchSize = maxBatchSize / lotSize;
}
// helper object
sp = new SharedBatchProcessStep();
loadingTime = batchProcessStep.getLoadTime();
processingTime = batchProcessStep.getTimePerBatch();
b1 = new BatchPlace();
b2 = new ProcessPlace();
b3 = new ProcessPlace();
b4 = new ProcessPlace();
b5 = new ProcessPlace();
sp.b1 = b1;
tb1 = new Transition();
tb2 = new FixedMaxPriorityTransition();
tb2.setDelay(loadingTime);// loading
tb3 = new FixedMaxPriorityTransition();
tb4 = new FixedMaxPriorityTransition();
tb4.setDelay(processingTime);// processing
tb5 = new FixedMaxPriorityTransition());
tb1.addInJobBatchPlace(b1);
tb1.addInPlace(toolSet);
tb1.addOutPlace(b2);
tb2.addInPlace(b2);
tb2.addOutPlace(b3);
tb3.addInPlace(b3);
tb3.addOutPlace(b4);
tb4.addInPlace(b4);
tb4.addOutPlace(b5);
tb5.addInPlace(b5);
```

```
if (operatorModeled) {
   tb1.addInPlace(operatorSet);
   tb3.addOutPlace(operatorSet);
}
tb5.addOutPlace(toolSet);
b10 = new BatchPlace();
tb1.addBatchOutPlace(b10);
sp.b10 = b10;
b20 = new BatchPlace();
tb3.addBatchOutPlace(b20);
sp.b20 = b20;
dispose2 = new FixedMinPriorityTransition();
dispose2.addBatchInPlace(b20);
b30 = new BatchPlace();
tb5.addBatchOutPlace(b30);
sp.b30 = b30;
dispose3 = new FixedMinPriorityTransition();
dispose3.addBatchInPlace(b30);
//this place limits the number of tokens in b1 place
c1 = new BatchPlace();
c1.addToken(new Token());
tb1.addBatchOutPlace(c1);
sp.c1 = c1;
c2 = new BatchPlace();
sp.c2 = c2;
dispose = new FixedMinPriorityTransition();
tb1.addBatchOutPlace(c2);
dispose.addBatchInPlace(c2);
return sp;
```

B.3.2.3 Create Shared Batch Process Step with Loading and Unloading

Creates the control places and transitions for batch process step `bs` with loading and unloading.

```
toolSet = place of required tool set;
operatorSet = place of required operator set;
operatorModeled = bs.getOperatorSet() != null ? true : false;
minBatchSize = bs.minBatchSize;
maxBatchSize = bs.maxBatchSize;
if (!bs.isIndividualWaferModeled()) {
   minBatchSize = minBatchSize / lotSize + 1;
   maxBatchSize = maxbatchSize / lotSize;
}
SharedBatchProcessStep sp = new SharedBatchProcessStep();
loadingTime = bs.loadTime;
```

```
unloadingTime = bs.unLoadTime;
processingTime = bs.getTimePerBatch;
b1 = new BatchPlace();
b2 = new ProcessPlace();
b3 = new ProcessPlace();
b4 = new ProcessPlace();
b5 = new ProcessPlace();
b6 = new ProcessPlace();
b7 = new ProcessPlace();
sp.b1 = b1;
tb1 = new Transition();
tb2 = new FixedMaxPriorityTransition();
tb2.setDelay(loadingTime);// loading
tb3 = new FixedMaxPriorityTransition();
tb4 = new FixedMaxPriorityTransition();
tb4.setDelay(processingTime);// processing
tb5 = new FixedMaxPriorityTransition();
tb6 = new FixedMaxPriorityTransition());
tb6.setDelay(unloadingTime);// unloading
tb7 = new FixedMaxPriorityTransition();
tb1.addInJobBatchPlace(b1);
tb1.addInPlace(toolSet);
tb1.addOutPlace(b2);
tb2.addInPlace(b2);
tb2.addOutPlace(b3);
tb3.addInPlace(b3);
tb3.addOutPlace(b4);
tb4.addInPlace(b4);
tb4.addOutPlace(b5);
tb5.addInPlace(b5);
tb5.addOutPlace(b6);
tb6.addInPlace(b6);
tb6.addOutPlace(b7);
tb7.addInPlace(b7);

if (operatorModeled) {
   tb1.addInPlace(operatorSet);
   tb3.addOutPlace(operatorSet);
   tb5.addInPlace(operatorSet);
   tb7.addOutPlace(operatorSet);
}
tb7.addOutPlace(toolSet);
b10 = new BatchPlace();
tb1.addBatchOutPlace(b10);
sp.b10 = b10;
b20 = new BatchPlace();
tb3.addBatchOutPlace(b20);
sp.b20 = b20;
```

```
dispose2 = new FixedMinPriorityTransition();
dispose2.addBatchInPlace(b20);
b30 = new BatchPlace();
tb5.addBatchOutPlace(b30);
sp.b30 = b30;
dispose3 = new FixedMinPriorityTransition();
dispose3.addBatchInPlace(b30);
b40 = new BatchPlace();
tb7.addBatchOutPlace(b40);
sp.b40 = b40;
dispose4 = new FixedMinPriorityTransition();
dispose4.addBatchInPlace(b40);
// this place limits the number of tokens in b1 place
c1 = new BatchPlace();
c1.addToken(new Token(0));
tb1.addBatchOutPlace(c1);
sp.c1 = c1;
c2 = new BatchPlace();
sp.c2 = c2;
dispose = new FixedMinPriorityTransition();
tb1.addBatchOutPlace(c2);
dispose.addBatchInPlace(c2);
return sp;
```

B.3.2.4 Create Shared Batch Process Step with Unloading

Creates the control places and transitions for batch process step `bs` with unloading.

```
toolSet = place of required tool set;
operatorSet = place of required operator set;
operatorModeled = bs.getOperatorSet() != null ? true : false;
minBatchSize = bs.getMinBatchSize();
maxBatchSize = bs.getMaxBatchSize();
if (!bs.isIndividualWaferModeled()) {
   minBatchSize = minBatchSize / lotSize + 1;
   maxBatchSize = maxbatchSize / lotSize;
}
//helper object
sp = new SharedBatchProcessStep();
unloadingTime = batchProcessStep.getUnLoadTime();
processingTime = batchProcessStep.getTimePerBatch();
b1 = new BatchPlace();
b4 = new ProcessPlace();
b5 = new ProcessPlace();
b6 = new ProcessPlace();
b7 = new ProcessPlace();
sp.b1 = b1;
tb1 = new Transition();
```

```
tb4 = new FixedMaxPriorityTransition();
tb4.setDelay(processingTime);// processing
tb5 = new FixedMaxPriorityTransition();
tb6 = new FixedMaxPriorityTransition();
tb6.setDelay(unloadingTime);// unloading
tb7 = new FixedMaxPriorityTransition();
tb1.addInJobBatchPlace(b1);
tb1.addInPlace(toolSet);
tb1.addOutPlace(b4);
tb4.addInPlace(b4);
tb4.addOutPlace(b5);
tb5.addInPlace(b5);
tb5.addOutPlace(b6);
tb6.addInPlace(b6);
tb6.addOutPlace(b7);
tb7.addInPlace(b7);
if (operatorModeled) {
   tb5.addInPlace(operatorSet);
   tb7.addOutPlace(operatorSet);
}
tb7.addOutPlace(toolSet);
b10 = new BatchPlace();
tb1.addBatchOutPlace(b10);
sp.b10 = b10;
b30 = new BatchPlace();
tb5.addBatchOutPlace(b30);
sp.b30 = b30;
dispose3 = new FixedMinPriorityTransition();
dispose3.addBatchInPlace(b30);
b40 = new BatchPlace();
tb7.addBatchOutPlace(b40);
sp.b40 = b40;
dispose4 = new FixedMinPriorityTransition());
dispose4.addBatchInPlace(b40);
// this place limits the number of tokens in b1 place
c1 = new BatchPlace();
c1.addToken(new Token(0));
tb1.addBatchOutPlace(c1);
sp.c1 = c1;
c2 = new BatchPlace();
sp.c2 = c2;
dispose = new FixedMinPriorityTransition();
tb7.addBatchOutPlace(c2);
dispose.addBatchInPlace(c2);
return sp;
```

B.3.3 Process Step with Setup

Creates places and transitions for process step with setup `ps`.

```
ts = place of required tool set;
operator = null;
if (ps.operatorSet != null) {
   operator = place of required operator set;
}
prioritySet = new ProcessPlace();
setPriority = new FixedMaxPriorityTransition();
setPriority.setDispatchRule(dispatchRule);
setPriority.addInPlace(lastPlace);
setPriority.addOutPlace(prioritySet);
lastPlace = prioritySet;
operatorModeled = ps.operatorSet != null ? true : false;
operatorSeized = false;
endSetup = new ProcessPlace();
for each setup state s in ts.setupStates{
   setupTime = ps.timePerSpecSetup;
   setup = new ProcessPlace();
   if (not (s == ps.getSetupGroupId)) {
      setupTime = setupTime + ps.timePerGroupSetup;
      setupStart = new Transition();
   } else {
      setupStart = new Transition();
   }
   tool = place of tool set for setup state s;
   setupStart.setDelay(setupTime);
   setupEnd = new Transition();
   setupStart.addInPlace(lastPlace);
   setupStart.addInPlace(tool);
   if (operatorModeled) {
      setupStart.addInPlace(operator);
      operatorSeized = true;
   }
   setupStart.addOutPlace(setup);
   setupEnd.addInPlace(setup);
   setupEnd.addOutPlace(endSetup);
}
lastPlace = endSetup;
delay = ps.timePerLot   + ps.timePerWaferInProcess * lotSize;
// loading
if (ps.getLoadTime > 0) {
   loading = new ProcessPlace();
   startLoading = new Transition();
   startLoading.setDelay(processStepWSetup.getLoadTime());
   if (lastPlace instanceof BatchPlace)
      startLoading.addInJobBatchPlace(lastPlace);
```

```
    else
      startLoading.addInPlace(lastPlace);
    startLoading.addOutPlace(loading);
    lastPlace = loading;
  }
  // processing with operator
  if (operatorModeled) {
    inProcessWOperator = new ProcessPlace();
    startProcessingWOperator = new Transition();
    startProcessingWOperator.setDelay(
      delay * ts.operatorProcessFraction);
    if (lastPlace instanceof BatchPlace)
      startProcessingWOperator.addInJobBatchPlace(lastPlace);
    else
      startProcessingWOperator.addInPlace(lastPlace);
    startProcessingWOperator.addOutPlace(inProcessWOperator);
    if (not operatorSeized) {
      startProcessingWOperator.addInPlace(operator);
      operatorSeized = true;
    }
    lastPlace = inProcessWOperator;
  }
  // processing w/o operator
  if ((not operatorModeled) || ts.operatorProcessFraction < 1.0) {
    inProcessWoutOperator = new ProcessPlace();
    startProcessingWoutOperator = new FixedMaxPriorityTransition();
    startProcessingWoutOperator.setDelay(
      delay * (1 - ts.operatorProcessFraction));
    if (lastPlace instanceof BatchPlace)
      startProcessingWoutOperator.addInJobBatchPlace(lastPlace);
    else
      startProcessingWoutOperator.addInPlace(lastPlace);
    startProcessingWoutOperator.addOutPlace(inProcessWoutOperator);
    if (operatorSeized && operatorModeled) {
      startProcessingWoutOperator.addOutPlace(operator);
      operatorSeized = false;
    }
  lastPlace = inProcessWoutOperator;
  }
  // unloading
  if (ps.unLoadTime > 0) {
    unloading = new ProcessPlace();
    startUnloadLoading = new Transition();
    startUnloadLoading.setDelay(ps.unLoadTime);
    startUnloadLoading.addInPlace(lastPlace);
    startUnloadLoading.addOutPlace(unloading);
    if (not operatorSeized && operatorModeled &&
      ts.operatorUnloading) {
```

```
      startUnloadLoading.addInPlace(operator);
      operatorSeized = true;
   }
   lastPlace = unloading;
}
// wafertraveltime
if (ps.waferTravelTimeWithinTool > 0) {
   inToolTransport = new ProcessPlace();
   startInToolTransport = new FixedMaxPriorityTransition();
   startInToolTransport.setDelay(ps.waferTravelTimeWithinTool);
   startInToolTransport.addInPlace(lastPlace);
   startInToolTransport.addOutPlace(inToolTransport);
   if (operatorSeized) {
      startInToolTransport.addOutPlace(operator);
      operatorSeized = false;
   }
   lastPlace = inToolTransport;
}
end = new ProcessPlace();
endT = new FixedMaxPriorityTransition();
endT.addInPlace(lastPlace);
endT.addOutPlace(end);
tool = place for tool in setup state ps.setupGroupId;
endT.addOutPlace(tool);
if (operatorSeized) {
   endT.addOutPlace(operator);
   operatorSeized = false;
   }
return end;
```

B.3.3.1 Create Rework Sequences

Creates the processing steps for rework sequence `rs`.

```
Place last = null;
for each process step s in rework sequence rs {
   //this is a call to same method that creates the
   //normal process steps
   reworkStart = createProcesStep();
   last = reworkStart;
}
Transition returnT = new Transition();
returnT.addInPlace(last);
reworkEnd = place of return process step;
returnT.addOutPlace(reworkEnd);
```

B.4 Create Input Transitions

```
for each process route pr {
   input = first place of pr;
   p = product of pr
   releaseRate = p.startRate;// start rate in wafers per day
   lotSize = p.lotSize;
   releaseRate = releaseRate / lotSize; // rate in lots per day
   //interval is the time between lot releases in msec;
   interval = (long) ((24 * 3600 * 1000l) / releaseRate);
   t1 = new TriggerTransition();
   t1.setDelay(interval);
   t1.stochastic = false;
   t1.addOutPlace(input);
}
```

REFERENCES

[1] *Modeling and Analysis for Semiconductor Manuafacturing Laboratory, Arizona State University.* http://www.eas.asu.edu/~masmlab (Date accessed: Oct 13, 2006).

[2] *OMG Modeling and Metadata Specifications.* http://www.omg.org/XMI (Date accessed: Nov 7, 2006).

[3] *Merriam-Webster's Collegiate Dictionary.* Springfield, MA: Merriam-Webster, 10th ed., 1993.

[4] *The American Heritage Dictionary of the English Language.* Boston, MA: Houghton Mifflin, 4th ed., 2000.

[5] *Arena - Version 11.* Milwaukee, WI: Rockwell Automation, 2007.

[6] *AutoSched AP.* Santa Clara, CA: Applied Materials, 2007.

[7] *Factory Explorer.* Pleasanton, CA: Wright Williams & Kelly, Inc., 2007.

[8] *SLX.* Alexandria, VA: Wolverine Software Corp., 2007.

[9] Askin, R. G. and Standridge, C. R., *Modeling and analysis of manufacturing systems.* New York, NY: John Wiley & Sons, 1993.

[10] Balci, O., "The implementation of four conceptual frameworks for simulation modeling in high-level languages," in *Proceedings of the 1988 Winter Simulation Conference*, (Piscataway, NJ), pp. 287–295, Institute of Electrical and Electronics Engineers, 1988.

[11] Banks, J., *Handbook of simulation: principles, methodology, advances, applications, and practice.* New York, NY: John Wiley & Sons, 1998.

[12] Brooks, R. J. and Tobias, A. M., "Choosing the best model: Level of detail, complexity, and model performance," *Mathematical and Computer Modelling (Oxford)*, vol. 24, no. 4, pp. 1–14, 1996.

[13] Carson, J., "Modeling and simulation worldviews," in *Proceedings of 1993 Winter Simulation Conference*, (Piscataway, NJ), pp. 18–23, Institute of Electrical and Electronics Engineers, 1993.

[14] Cassandras, C. G. and Lafortune, S., *Introduction to discrete event systems.* Norwell, MA: Kluwer Academic Publishers, 1999.

[15] Cormen, T. H., Leiseron, C. E., Rivest, R. L., and Stein, C., *Introduction to algorithms.* Cambridge, Mass.: MIT Press, 2nd ed., 2001.

[16] Fishman, G. S., *Discrete-event simulation: modeling, programming, and analysis.* New York, NY: Springer-Verlag, 2001.

[17] Fishwick, P. A., *Simulation model design and execution: building digital worlds.* Englewood Cliffs, NJ: Prentice Hall, 1995.

[18] Fowler, J. W. and Rose, O., "Grand challenges in modeling and simulation of complex manufacturing systems," *Simulation*, vol. 80, no. 9, pp. 469–476, 2004.

[19] Gong, D.-C. and McGinnis, L., "An AGVS simulation code generation for manufacturing applications," in *Proceedings of the 1990 Winter Simulation Conference*, (Piscataway, NJ), pp. 676–682, Institute of Electrical and Electronics Engineers, 1990.

[20] Hopp, W. J. and Spearman, M. L., *Factory physics: Foundations of manufacturing management.* Chicago, IL: Irwin, 1996.

[21] Jacobs, P. H., Lang, N. A., and Verbraeck, A., "D-SOL; a distributed Java based discrete event simulation architecture," in *Proceedings of the 2002 Winter Simulation Conference*, vol. 1, (Piscataway, NJ), pp. 793–800, Institute of Electrical and Electronics Engineers, 2002.

[22] Jeng, M. D. and DiCesare, F., "A review of synthesis techniques for Petri nets with applications to automated manufacturing systems," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 23, no. 1, pp. 301–312, 1993.

[23] Jensen, K., *Coloured Petri nets: Basic concepts, analysis methods, and practical use.* New York, NY: Springer-Verlag, 1992.

[24] Law, A. M. and Kelton, W. D., *Simulation modeling and analysis.* Boston, MA: McGraw-Hill, 3rd ed., 2000.

[25] L'Ecuyer, P. and Buist, E., "Simulation in Java with SSJ," in *Proceedings of the 2005 Winter Simulation Conference*, (Piscataway, NJ), pp. 611–621, Institute of Electrical and Electronics Engineers, 2005.

[26] Lee, S., Cho, H., and Jung, M., "A conceptual framework for the generation of simulation models from process plans and resource configuration," *International Journal of Production Research*, vol. 38, no. 4, pp. 811–828, 2000.

[27] Lee, Y. T. and Yan, L., "Data exchange for machine shop simulation," in *Proceedings of the 2005 Winter Simulation Conference*, (Piscataway, NJ), pp. 1446–1452, Institute of Electrical and Electronics Engineers, 2005.

[28] Lu, R. F., Qiao, G., and McLean, C., "NIST XML simulation interface specification at Boeing: A case study," in *Proceedings of the 2003 Winter Simulation Conference*, vol. 2, (Piscataway, NJ), pp. 1230–1237, Institute of Electrical and Electronics Engineers, 2003.

[29] Mason, S. and Fowler, J., "Maximizing delivery performance in semiconductor wafer fabrication facilities," in *Proceedings of the 2000 Winter Simulation Conference*, vol. 2, (Piscataway, NJ), pp. 1458–1463, Institute of Electrical and Electronics Engineers, 2000.

[30] Mathewson, S., "Program generators," in *Interactive Systems, Sept. 1975*, pp. 423–439, 1975.

[31] Mathewson, S., "Simulation program generators: Code and animation on a PC," *Journal of the Operational Research Society*, vol. 36, no. 7, pp. 583–589, 1985.

[32] McLean, C., Jones, A., Lee, T., and Riddick, F., "An architecture for a generic data-driven machine shop simulator," in *Proceedings of the 2002 Winter Simulation Conference*, vol. 2, (Piscataway, NJ), pp. 1108–1116, Institute of Electrical and Electronics Engineers, 2002.

[33] Murata, T., "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[34] Nance, R., "A history of discrete event simulation programming languages," *ACM SIGPLAN HOPL-II. 2nd ACM SIGPLAN History of Programming Languages Conference*, vol. 28, no. 3, pp. 149–175, 1993.

[35] Overstreet, C. M. and Nance, R., "Issues in enhancing model reuse," in *First International Conference on Grand Challenges for Modeling and Simulation*, 2002.

[36] Overstreet, C. M. and Nance, R. E., "Specification language to assist in analysis of discrete event simulation models," *Communications of the ACM*, vol. 28, no. 2, pp. 190–201, 1985.

[37] Overstreet, C. M. and Nance, R. E., "Characterizations and relationships of world views," in *Proceedings of the 2004 Winter Simulation Conference*, vol. 1, (Piscataway, NJ), pp. 279–287, Institute of Electrical and Electronics Engineers, 2004.

[38] Page, E. H. and Nance, R. E., "Incorporating support for model execution within the condition specification," *Transactions of the Society for Computer Simulation International*, vol. 16, no. 2, pp. 47–62, 1999.

[39] Page, E. and Opper, J., "Observations on the complexity of composable simulation," in *Proceedings of 1999 Winter Conference on Simulation*, vol. 1, (Piscataway, NJ), pp. 553–560, Institute of Electrical and Electronics Engineers, 1993.

[40] Park, J. and Reveliotis, S., "Deadlock avoidance in sequential resource allocation systems with multiple resource acquisitions and flexible routings," *IEEE Transactions on Automatic Control*, vol. 46, no. 10, pp. 1572–1583, 2001.

[41] Paul, R., "Activity cycle diagrams and the three phase method," in *Proceedings of 1993 Winter Simulation Conference*, (Piscataway, NJ), pp. 123–131, Institute of Electrical and Electronics Engineers, 1993.

[42] Peterson, J. L., *Petri net theory and the modeling of systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[43] Petri, C. A., *Kommunikation mit Automaten*. PhD thesis, Technische Hochschule Darmstadt, 1962.

[44] Ramirez-Hernandez, J. A., Heshan, L., Fernandez, E., McLean, C., and Swee, L., "A framework for standard modular simulation in semiconductor wafer fabrication systems," in *Proceedings of the 2005 Winter Simulation Conference*, (Piscataway, NJ), pp. 2162–2171, Institute of Electrical and Electronics Engineers, 2005.

[45] Sargent, R. G., "Validation and verification of simulation models," in *Proceedings of the 2004 Winter Simulation Conference*, vol. 1, (Piscataway, NJ), pp. 17–28, Institute of Electrical and Electronics Engineers, 2004.

[46] Savage, E. L., Schruben, L. W., and Yucesan, E., "On the generality of event-graph models," *INFORMS Journal on Computing*, vol. 17, no. 1, pp. 3–9, 2005.

[47] Schriber, T. and Brunner, D., "Inside discrete-event simulation software: How it works and why it matters," in *Proceedings of the 2005 Winter Simulation Conference*, (Piscataway, NJ), pp. 167–178, Institute of Electrical and Electronics Engineers, 2005.

[48] Schruben, L. W., "Simulation modeling with event graphs," *Communications of the ACM*, vol. 26, no. 11, pp. 957–963, 1983.

[49] Schruben, L. W. and Yucesan, E., "Transforming Petri nets into event graph models," in *Proceedings of the 1994 Winter Simulation Conference*, (Piscataway, NJ), pp. 560–565, Institute of Electrical and Electronics Engineers, 1994.

[50] van der Aalst, W., "Interval timed coloured Petri nets and their analysis," in *Applications and Theory of Petri Nets 1993. 14th International Conference Proceedings, 21-25 June 1993*, (New York, NY), pp. 453–472, Springer-Verlag, 1993.

[51] Wang, J., *Timed Petri nets: Theory and application.* Norwell, MA: Kluwer Academic Publishers, 1998.

[52] Wang, L. and Xie, X., "Modular modeling using Petri nets," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 5, pp. 800–809, 1996.

[53] Wiener, R. and Pinson, L. J., *Fundamentals of OOP and data structures in Java.* Norwood, MA: Cambridge University Press, 2000.

[54] Yilmaz, L., "Verifying collaborative behavior in component-based DEVS models," *Simulation*, vol. 80, no. 7-8, pp. 399–415, 2004.

[55] Yucesan, E. and Jacobson, S., "Building correct simulation models is difficult," in *Proceedings of the 1992 Winter Simulation Conference*, (Piscataway, NJ), pp. 783–790, Institute of Electrical and Electronics Engineers, 1992.

[56] Yucesan, E. and Schruben, L. W., "Structural and behavioral equivalence of simulation models," *ACM Transactions on Modeling and Computer Simulation*, vol. 2, no. 1, pp. 82–103, 1992.

[57] Zeigler, B. P., Kim, T. G., and Praehofer, H., *Theory of modeling and simulation: Integrating discrete event and continuous complex dynamic systems.* San Diego, CA: Academic Press, 2nd ed., 2000.

[58] Zhou, M. and DiCesare, F., *Petri net synthesis for discrete event control of manufacturing systems.* Norwell, MA: Kluwer Academic Publishers, 1993.

[59] Zurawski, R. and Zhou, M., "Petri nets and industrial applications: A tutorial," *IEEE Transactions on Industrial Electronics*, vol. 41, no. 6, pp. 567–583, 1994.