# ANALYSIS OF THE IMPACT OF DECISION TIME ON THE SYSTEM PERFORMANCE IN DISTRIBUTED SYSTEMS

by

Kamer Özgün

B.S., Mechanical Engineering, Boğaziçi University, 2004

M.S., Industrial Engineering, Boğaziçi University, 2007

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Industrial Engineering

Boğaziçi University

2012

# ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my supervisor, Assoc. Prof. Ali Tamer Ünal for his guidance, consideration and support. It was a pleasure for me to realize my graduate studies under his supervision. His wide knowledge and his logical way of thinking have been of great value for me. I learned from him to believe in my work and myself.

I wish to express my special thanks to my thesis committee members, Prof. Gündüz Ulusoy, Prof. Tülin Aktin, Assist. Prof. Aybek Korugan, Assist. Prof. Zeki Caner Taşkın for their construcive comments and suggestions.

This thesis is dedicated to my husband Emrecan Özgün and my about to be born baby. I and my daughter defensed this thesis together when I was 30 weeks pregnant. My deepest gratitude goes to my husband for his love, understanding, care and encouragement. He is the one.

## ABSTRACT

## ANALYSIS OF THE IMPACT OF DECISION TIME ON THE SYSTEM PERFORMANCE IN DISTRIBUTED SYSTEMS

In this study, we investigate the effect of the time it takes to generate a schedule on the performance of a stochastic dynamic scheduling system. To isolate the impact of the decision time on the system performance, we devise a single machine stochastic scheduling environment where the performance of the system is measured by average earliness - tardiness cost. Our study is composed of two phases. In the first phase, we construct a centralized scheduling system. We explicitly model the decision time. We test the trade off between spending more time for the scheduling process by employing more sophisticated scheduling algorithms and using simple fast heuristic algorithm. In the second phase, we construct a distributed scheduling system. We test the trade off between spending more time by including detailed global information to achieve global optimality under a centralized control structure and using timely accessible local information under distributed control. We simulated the system under various scheduling environments controlled by due date tightness, urgent job ratios, operation time variability and utilization using different centralized control polices and distributed control policies. Our experiments show that under certain shop conditions and control policies, the shop may operate more efficiently if a simple fast heuristic is used instead of a slow optimum algorithm to solve the scheduling problems. We have been able to also show that, again under some specific operating conditions, the dynamic production system will run more efficiently when we use fast distributed schedulers instead of a relatively slow centralized scheduler.

# ÖZET

## DAĞITIK SİSTEMLERDE KARAR VERME SÜRESİNİN SİSTEM PERFORMANSINA ETKİSİNİN ANALİZİ

Biz bu çalışmada bir çizelge üretmek için geçen zamanın stokastik dinamik çizelgeleme sistemi performansına etkisini inceledik. Karar verme süresinin sistem performansı üzerindeki etkisini ayırdebilmek için tek makineli dinamik ve rastsal bir çizelgeleme ortamı tasarladık. İşlerin ortalama erkenliği ve geçliği sistem performansı ölçütü olarak kullanıldı. Çalışmamız iki aşamadan oluşuyor. İlk aşamada merkezi bir çizelgeleme sistemi kurduk. Karar verme süresini modelledik. Çizelgeleme sürecinde daha fazla zaman alan gelişmiş çizelgeleme algoritmaları kullanımını, basit hızlı sezgisel algoritmaların kullanımı ile kıyasladık. İkinci aşamada dağıtık bir çizelgeleme sistemi kurduk. Daha fazla zaman harcayarak ayrıntılı global bilgiyi dahil edip global eniyiliğe ulaşan merkezi kontrol yapısını, çabuk ulaşılan lokal bilginin kullanıldığı dağıtık kontrol yapısı ile kıyasladık. Benzetim modelimizde termin tarihlerinin sıkılığı, acil işlerin gelme oranı, üretim zamanlarındaki varyasyon, makina doluluğu parametreleri ile çeşitli atölye koşulları yarattık ve sistemi alternatif merkezi ve dağıtık kontrol politikaları altında çalıştırdık. Deneylerimizde, belirli atölye ortamlarında ve kontrol politikaları altında, basit ve hızlı bir sezgisel yöntemin yavaş bir eniyileme algoritmasından daha iyi sonuç verdiğini gösteriyoruz. Ayrıca, deneylerimizde yine belirli operasyon koşullarında, nispeten yavaş merkezi çizelgeleyici yerine hızlı dağıtık çizelgeleyicileri kullandığımızda, dinamik üretim sisteminin daha etkili yönetildiğini gösteriyoruz.

# TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . . . . . . . . . . . . . . . . . . . . . . . . . . iii

ABSTRACT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . iv

ÖZET . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . v

LIST OF FIGURES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ix

LIST OF TABLES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xv

LIST OF SYMBOLS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xvi

LIST OF ACRONYMS/ABBREVIATIONS . . . . . . . . . . . . . . . . . . xviii

1. INTRODUCTION . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

2. LITERATURE REVIEW . . . . . . . . . . . . . . . . . . . . . . . . . . 6

  2.1. Static Deterministic Scheduling with Earliness/Tardiness Penalties . . . 6

    2.1.1. Scheduling without Machine Idle Times . . . . . . . . . . . . . . 6

    2.1.2. Scheduling with Inserted Machine Idle Times . . . . . . . . . . 6

  2.2. Deterioration of Static Schedules in Dynamic Stochastic Systems . . . . 7

  2.3. Dynamic Scheduling with Earliness/Tardiness Penalties . . . . . . . . . 8

  2.4. Distributed Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . 9

    2.4.1. Choice of the Decomposition Methodology . . . . . . . . . . . . 10

    2.4.2. Choice of the Control Structure . . . . . . . . . . . . . . . . . . 12

    2.4.3. Handling Mechanism for the Shared Resource Between Schedulers 13

  2.5. Scheduling Time Considerations . . . . . . . . . . . . . . . . . . . . . 14

3. SCHEDULING ENVIRONMENT . . . . . . . . . . . . . . . . . . . . . . 16

  3.1. The Centralized Scheduling System . . . . . . . . . . . . . . . . . . . 17

    3.1.1. The Centralized Scheduler . . . . . . . . . . . . . . . . . . . . . 18

      3.1.1.1. Instantaneous Mode of the Scheduler . . . . . . . . . . 19

      3.1.1.2. Busy Mode of the Scheduler . . . . . . . . . . . . . . . 20

      3.1.1.3. Available Mode of the Scheduler . . . . . . . . . . . . 20

    3.1.2. The Dispatcher in the Centralized Scheduling System . . . . . . 21

    3.1.3. Idle Times on the Machine . . . . . . . . . . . . . . . . . . . . . 22

      3.1.3.1. Queue Idleness (QI) . . . . . . . . . . . . . . . . . . . 22

      3.1.3.2. Forced Idleness (FI) . . . . . . . . . . . . . . . . . . . 22

3.1.3.3. Decision Idleness (DI) . . . . . . . . . . . . . . . . . . . 22
3.1.4. Example Run of the Centralized Scheduling System . . . . . . . 23
3.2. The Distributed Scheduling System . . . . . . . . . . . . . . . . . . . 27
3.2.1. The Distributed Schedulers . . . . . . . . . . . . . . . . . . . . 28
3.2.2. Issues in Distributed Systems . . . . . . . . . . . . . . . . . . . 30
3.2.3. The Dispatcher in the Distributed Scheduling System . . . . . . 31
3.2.4. Example Run of the Distributed Scheduling System . . . . . . . 34
4. EXPERIMENTATION . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.1. Scheduling Environment Settings . . . . . . . . . . . . . . . . . . . . 37
4.2. Control Policy Settings . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.2.1. Centralized Control Policy Settings . . . . . . . . . . . . . . . . 39
4.2.2. Distributed Control Policy Settings . . . . . . . . . . . . . . . . 41
5. RESULTS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.1. Effect of Decision Time on Scheduling System Performance . . . . . . . 43
5.1.1. General Observations . . . . . . . . . . . . . . . . . . . . . . . 44
5.1.2. Machine States . . . . . . . . . . . . . . . . . . . . . . . . . . 44
5.1.2.1. State P . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.1.2.2. FI and QI States . . . . . . . . . . . . . . . . . . . . . 49
5.1.2.3. DI State . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.1.3. Impact of Decision Time . . . . . . . . . . . . . . . . . . . . . 50
5.1.4. Responsiveness Versus Optimality . . . . . . . . . . . . . . . . 57
5.2. Effect of Synchronization on Distributed Scheduling System Performance 59
5.2.1. Dispatch Decision Percentages . . . . . . . . . . . . . . . . . . . 60
5.2.2. Centralized Versus Distributed Scheduling Systems . . . . . . . 67
6. CONCLUSION . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
APPENDIX A: DISCRETE EVENT SIMULATION . . . . . . . . . . . . . 73
A.1. Arrival Process . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
A.2. Scheduling Process . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
A.2.1. Instantaneous Mode . . . . . . . . . . . . . . . . . . . . . . . . 76
A.2.2. Busy Mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
A.2.3. Available Mode . . . . . . . . . . . . . . . . . . . . . . . . . . 77

A.3. Dispatch Process . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
A.4. Machine Process . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
APPENDIX B: ICRON IMPLEMENTATION . . . . . . . . . . . . . . . . . . 79
B.1. Event Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
B.1.1. NEWJOB Event for CSSS . . . . . . . . . . . . . . . . . . . . . 81
B.1.2. NEWJOB Event for DSSS . . . . . . . . . . . . . . . . . . . . . 82
B.1.3. OPERATIONCOMPLETED Event for CSSS . . . . . . . . . . 83
B.1.4. OPERATIONCOMPLETED Event for DSSS . . . . . . . . . . 85
B.1.5. SCHEDULINGCOMPLETED Event . . . . . . . . . . . . . . 86
B.1.6. WAKEUPTODISPATCH Event . . . . . . . . . . . . . . . . . 87
B.2. Job Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
B.2.1. Job Class for CSSS . . . . . . . . . . . . . . . . . . . . . . . . 88
B.2.2. Job Class for DSSS . . . . . . . . . . . . . . . . . . . . . . . . 91
B.3. PolicyType Class . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
B.3.1. PolicyType Class for DSSS . . . . . . . . . . . . . . . . . . . . 94
B.4. ProblemSetting Class . . . . . . . . . . . . . . . . . . . . . . . . . 94
B.5. ProblemType Class . . . . . . . . . . . . . . . . . . . . . . . . . . 95
B.6. Simulator Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
B.6.1. Scheduling Process Algorithms . . . . . . . . . . . . . . . . . . 101
B.6.2. Dispatch Process Algorithms . . . . . . . . . . . . . . . . . . . 103
B.6.3. Simulator Class for CSSS . . . . . . . . . . . . . . . . . . . . . 103
B.6.3.1. Scheduling Process Algorithms in CSSS . . . . . . . . 104
B.6.3.2. Dispatch Process Algorithms in CSSS . . . . . . . . . 106
B.6.4. Simulator Class for DSSS . . . . . . . . . . . . . . . . . . . . . 110
B.6.4.1. Scheduling Process Algorithms in DSSS . . . . . . . . 111
B.6.4.2. Dispatch Process Algorithms in DSSS . . . . . . . . . 114
B.7. Operation of the Simulator . . . . . . . . . . . . . . . . . . . . . . 116
REFERENCES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119

# LIST OF FIGURES

Figure 3.1. Overall Architecture of the Centralized Scheduling System. . . . . 17

Figure 3.2. Dispatch$(t, \pi)$ Algorithm. . . . . . . . . . . . . . . . . . . . . . . . . . 21

Figure 3.3. Sample Run Using the Instantaneous Response Mode. . . . . . . . 24

Figure 3.4. Sample Run Using the Busy Response Mode. . . . . . . . . . . . . 25

Figure 3.5. Sample Run Using the Available Response Mode. . . . . . . . . . 26

Figure 3.6. Overall Architecture of the Distributed Scheduling System. . . . . 27

Figure 3.7. Dispatch$(t, \alpha, \beta)$ Algorithm. . . . . . . . . . . . . . . . . . . . . . . 33

Figure 3.8. Sample Run of Distributed Schedulers. . . . . . . . . . . . . . . . . 35

Figure 5.1. Objective Values and Machine State Percentages for $\{I, 0, Opt\}$. . 44

Figure 5.2. Objective Values and Machine State Percentages for$\{B, \frac{1}{16}, Opt\}$. . 45

Figure 5.3. Objective Values and Machine State Percentages for$\{B, \frac{1}{8}, Opt\}$. . 46

Figure 5.4. Objective Values and Machine State Percentages for$\{B, \frac{1}{4}, Opt\}$. . 46

Figure 5.5. Objective Values and Machine State Percentages for$\{A, \frac{1}{8}, Opt\}$. . 47

Figure 5.6. Objective Values and Machine State Percentages for$\{A, \frac{1}{4}, Opt\}$. . 47

Figure 5.7. Objective Values and Machine State Percentages for$\{A, \frac{1}{2}, Opt\}$. . 48

Figure 5.8. Objective Values and Machine State Percentages for$\{A, 1, Opt\}$. . 48

Figure 5.9. Objective Values and Machine State Percentages for$\{A, 2, Opt\}$. . 49

Figure 5.10. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.5, \tau = 0$. . . . . . . . . . . . . . . . . . . . . . . 51

Figure 5.11. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.5, \tau = 2, \phi = 0.3$. . . . . . . . . . . . . . . . . . . 51

Figure 5.12. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.5, \tau = 2, \phi = 0$. . . . . . . . . . . . . . . . . . . . 52

Figure 5.13. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.5, \tau = 4, \phi = 0.3$. . . . . . . . . . . . . . . . . . . 52

Figure 5.14. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.5, \tau = 4, \phi = 0$. . . . . . . . . . . . . . . . . . . . 53

Figure 5.15. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.7, \tau = 0$. . . . . . . . . . . . . . . . . . . . . . . 53

Figure 5.16. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.7, \tau = 2, \phi = 0.3$. . . . . . . . . . . . . . . . . . . 54

Figure 5.17. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.7, \tau = 2, \phi = 0$. . . . . . . . . . . . . . . . . . . . 54

Figure 5.18. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.7, \tau = 4, \phi = 0.3$. . . . . . . . . . . . . . . . . . . . 55

Figure 5.19. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.7, \tau = 4, \phi = 0$. . . . . . . . . . . . . . . . . . . . . 55

Figure 5.20. Objective Values and Dispatch Decision Percentages of All Distributed Policies for $\rho = 0.5$. . . . . . . . . . . . . . . . . . . . . . . 61

Figure 5.21. Objective Values and Dispatch Decision Percentages of All Distributed Policies for $\rho = 0.7$. . . . . . . . . . . . . . . . . . . . . . . 62

Figure 5.22. Objective Values and Wait Decision Percentages of All Distributed Policies for $\rho = 0.5$. . . . . . . . . . . . . . . . . . . . . . . . . 63

Figure 5.23. Objective Values and Wait Decision Percentages of All Distributed Policies for $\rho = 0.7$. . . . . . . . . . . . . . . . . . . . . . . . . 64

Figure 5.24. Objective Values and Dispatch Decision Percentages of Distributed Policies with $\omega = 0$ under $\gamma = 0$. . . . . . . . . . . . . . . . . . . 65

Figure 5.25. Objective Values and Wait Decision Percentages of Distributed Policies with $\omega = 0$ under $\gamma = 0$. . . . . . . . . . . . . . . . . . . 66

Figure 5.26. Objective Values of Selected Centralized and Distributed Policies for $\rho = 0.5$. . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

Figure 5.27. Objective Values of Selected Centralized and Distributed Policies for $\rho = 0.7$. . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

Figure A.1. The Data Flow Diagram of the Simulation. . . . . . . . . . . . . . 73

Figure B.1. Handle Algorithm of Base Event Class. . . . . . . . . . . . . . . . 81

Figure B.2. Overloaded Handle Algorithm of NEWJOB Event in CSSS Implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

Figure B.3. Overloaded Handle Algorithm of NEWJOB Event in DSSS Implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83

Figure B.4. Overloaded Handle Algorithm of OPERATIONCOMPLETED Event in CSSS Implementation. . . . . . . . . . . . . . . . . . . . . . . 84

Figure B.5. Overloaded Handle Algorithm of OPERATIONCOMPLETED Event in DSSS Implementation. . . . . . . . . . . . . . . . . . . . . . . 85

Figure B.6. Overloaded Handle Algorithm of SCHEDULINGCOMPLETED Event. 86

Figure B.7. Overloaded Handle Algorithm of WAKEUPTODISPATCH Event. 87

Figure B.8. SetDefaultValuesofTheJob Algorithm of Job Class in CSSS Implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89

Figure B.9. InitializeTheJob Algorithm of Job class in CSSS Implementation. . 90

Figure B.10. SetDefaultValuesofTheJob Algorithm of Job Class in DSSS Implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92

Figure B.11. InitializeTheJob Algorithm of Job Class in DSSS Implementation. 93

Figure B.12. InitializeOnce Algorithm of Simulator Class. . . . . . . . . . . . 97

Figure B.13. CreateNewJobEvent Algorithm of Simulator Class. . . . . . . . . . 98

Figure B.14. CreateOperationCompletedEvent Algorithm of Simulator Class. . 99

Figure B.15. CreateSchedulingCompletedEvent Algorithm of Simulator Class. . 100

Figure B.16. CreateWakeupToDispatchEvent Algorithm of Simulator Class. . . 100

Figure B.17. Schedule Algorithm of Simulator Class. . . . . . . . . . . . . . . . 101

Figure B.18. RandomAlgorithm of Simulator Class. . . . . . . . . . . . . . . . . 102

Figure B.19. Dispatch Algorithm of Simulator Class. . . . . . . . . . . . . . . . 103

Figure B.20. StartScheduling Algorithm of Simulator Class in CSSS Implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105

Figure B.21. ReleaseLatestSchedule Algorithm of Simulator Class in CSSS Implementation. . . . . . . . . . . . . . . . . . . . . . . . . . 105

Figure B.22. DispatchBeforePivot Algorithm of Simulator Class in CSSS Implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . 106

Figure B.23. DispatchAfterPivot Algorithm of Simulator Class in CSSS Implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . 107

Figure B.24. PStarts Algorithm of Simulator Class in CSSS Implementation. . . 108

Figure B.25. FIStarts Algorithm of Simulator Class in CSSS Implementation. . 109

Figure B.26. DIStarts Algorithm of Simulator Class in CSSS Implementation. . 109

Figure B.27. QIStarts Algorithm of Simulator Class in CSSS Implementation. . 110

Figure B.28. StartScheduling Algorithm of Simulator Class in DSSS Implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

Figure B.29. AssignTempTimes Algorithm of Simulator Class in DSSS Implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

Figure B.30. ReleaseLatestSchedule Algorithm of Simulator Class in DSSS Implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . 113

Figure B.31. DispatchBeforePivot Algorithm of Simulator Class in DSSS Implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115

Figure B.32. Simulate Algorithm of System Manager Class. . . . . . . . . . . . 117

Figure B.33. SimulateOneProblemSetting Algorithm of Simulator Class. . . . . 117

Figure B.34. SimulateOneReplication Algorithm of Simulator Class. . . . . . . 118

# LIST OF TABLES

Table 4.1. Values tested for the scheduling environment parameters. . . . . . 39

Table 4.2. Values tested for the centralized control policy parameters. . . . . 40

Table 4.3. Values tested for the distributed control policy parameters. . . . . 42

# LIST OF SYMBOLS

| | |
|---|---|
| $a_j$ | Arrival time of job $j$ |
| $c_{[j]}$ | Completion time of the job assigned to the $j$th position |
| $c_j^R$ | Realized completion time of job $j$ on the machine |
| $c_j(\pi)$ | Estimated completion time of $j$ on schedule $\pi$ |
| $d_j$ | Due date of job $j$ |
| $E_{[j]}$ | Earliness value of the job assigned to the $j$th position |
| $E_j^R$ | Realized earliness of job $j$ |
| j | A job |
| $[j]$ | The job assigned to the $j$th position in the sequence |
| $j^{\text{type}}$ | The type of job $j$, $j^{\text{type}} \in \{A, B\}$ |
| $j(t)$ | The job which is being processed on the machine at time $t$ |
| $\bar{j}$ | The job selected to be started on the machine |
| $\bar{j}_\alpha$ | The job selected to be processed at time $t$ according to the schedule $\alpha$ |
| $\bar{j}_\beta$ | The job selected to be processed at time $t$ according to the schedule $\beta$ |
| $J_t$ | Set of jobs that has arrived by time $t$ |
| $J_t^A$ | Schedulable jobs of the scheduler A at time $t$ |
| $J_t^B$ | Schedulable jobs of the scheduler B at time $t$ |
| $J_t^S$ | The set of jobs that are in the system at time $t$ |
| $J(\pi)$ | Set of jobs that are belong to schedule $\pi$ |
| $\bar{J}(t, \pi)$ | The set of jobs that can be processed at time $t$ that exists in schedule $\pi$. |
| L | The expected queue length for an M/M/1 system |
| N | The number of jobs over which we collect statistics |
| $p_j^R$ | Realized processing time of job $j$ |
| $\hat{p}_j$ | Estimated processing time of job j |
| $s_j^R$ | Realized start time of job $j$ on the machine |
| $s_j(\pi)$ | Start time of job $j$ on schedule $\pi$ |

| | |
|---|---|
| t | Time |
| $t^{\text{wakeup}}$ | Requested wakeup time |
| $t^{\text{synch}}$ | Time that the schedulers are synchronized |
| $t^x$ | The time that the recent synchronization process starts |
| $t^S(\pi)$ | Time that the scheduling process to generate $\pi$ has started |
| $t^G(\pi)$ | Time that schedule $\pi$ is generated |
| $T_{[j]}$ | Tardiness value of the job assigned to the $j$th position |
| $T_j^R$ | Realized tardiness of job $j$ |
| $x$ | The response mode of the scheduler |
| $x_{i[j]}$ | Binary variable; it is equal to 1, if job $i$ is assigned to the $j$th position and 0 otherwise |
| $y$ | The length of the decision time as a multiple of $\lambda$ |
| $z$ | The scheduling algorithm used by the scheduler |
| | |
| $\alpha$ | Schedule of scheduler A |
| $\beta$ | Schedule of scheduler B |
| $\delta$ | Decision time |
| $\gamma$ | Processing time variability |
| $\Gamma$ | Scheduling period that statistics during the simulation are collected |
| $\kappa$ | A uniform random number |
| $\lambda$ | Job arrival rate |
| $\omega$ | Time it takes to synchronize |
| $\Omega$ | Synchronization period length |
| $\rho$ | Machine utilization |
| $\phi$ | Urgent job ratio |
| $\pi$ | A Schedule |
| $\pi_t^L$ | Latest schedule released by time $t$ |
| $\sigma$ | A random sequence of jobs in the system ($J_t^S$) |
| $\sigma(i)$ | The $i$th job in sequence $\sigma$ |
| $\tau$ | Due date tightness |

# LIST OF ACRONYMS/ABBREVIATIONS

| | |
|---|---|
| A | Available Mode |
| B | Busy Mode |
| CSSS | Centralized Scheduling System Simulator |
| CST | Current Simulation Time |
| DI | Decision Idleness State |
| DSSS | Distributed Scheduling System Simulator |
| ET | Event Time |
| FI | Forced Idleness State |
| H | A heuristic algorithm |
| I | Instantaneous Mode |
| *Opt* | An optimum algorithm |
| P | Processing State |
| QI | Queue Idleness State |
| RN | Random Number |

# 1. INTRODUCTION

A stream of research, and most scheduling implementations in actual manufacturing companies, handle the dynamic and stochastic nature of the scheduling environment through a rolling scheduling process. At *scheduling process trigger* time points the following steps are executed.

(i) Load the scheduling data, such as open orders, current shop floor status, product specifications (processing times, setup requirements, routings, etc.), available production capacities, and product and material inventories.
(ii) Based on the current captured state of the system in Step 1, run a static deterministic scheduling algorithm to obtain an *optimized* schedule.
(iii) Evaluate the resulting schedule and perform manual overrides.
(iv) Dispatch the generated schedule to the shop floor.

In the literature, different forms of *scheduling process trigger* mechanisms, such as periodic and event based trigger are analyzed [1]. Steps 1 and 4 are accomplished through an integration with an enterprise data source, such as an ERP system. Steps 2 and 3 are optional. In Step 1, the data loaded from the data source is expected to represent the state of the system at a time instance, and it is deterministic. Hence, the scheduling problem on hand in Step 2 is static and deterministic in nature. The stochastic nature of the system is handled by loading the data regarding the current state of the system at each execution of the process.

Studies in the dynamic scheduling literature commonly assume that the whole scheduling process is instantaneous, i.e. executing Steps 1 through 4 does not take time. On the contrary, in real life, all steps take time. Actually, depending on the size and extent of the underlying data source, the scheduling problem and the scheduling method deployed, the time it takes to perform the scheduling process may be very significant. We have accounts of scheduling systems deployed in manufacturing companies where the scheduling process time is in the magnitude of several hours.

Issues related to the integration process (Steps 1 and 4) within a dynamic environment (see [2]) is out of the scope of this paper. Hence, for simplification, we will assume that Steps 1 and 4 do not take time, and the loaded data in Step 1 accurately represent the state of the system at that instance. Since both Steps 2 and 3 deal with the actual scheduling, and there is no added value by considering them separately, we will assume that, within the scheduling process, only Step 2 takes time, and all other processes are instantaneous. We will call the time spent to perform the scheduling process as the *decision time*.

Even when we assume that the decision time is zero, i.e. scheduling process takes no time, using optimum static deterministic scheduling algorithms within the scheduling process to manage dynamic stochastic systems, under some specific operating conditions, may be inferior to using simple heuristics. [3,4] analyze the deterioration of the performance of optimized static schedules under processing time uncertainty and variability in machine availability. [5,6] focus on the robustness of optimum seeking off-line scheduling versus on-line scheduling under uncertainty and processing time variability.

Further studies on different issues regarding scheduling dynamic systems can be found in survey papers [7–9]. Especially [10,11] investigate the dynamic single machine scheduling problem where the performance measure is related to earliness and tardiness of jobs.

Note that, the scheduling process captures the current state of the system at the beginning of the process (Step 1), and the scheduling algorithm runs on that static data. As we discussed previously, running the scheduling algorithm takes some time. Since our system is dynamic and stochastic, the state of the system changes within that time period. Hence, the state of the system when the schedule is released at the end of the process (Step 4) may be significantly different from the data considered within the scheduling algorithm. Hence, the solution obtained may not be a good (or even feasible) solution for the current shop conditions

In a reasonable scheduling setting, this process inherently exposes a basic trade-off: If we spend more time for the scheduling process, by including more detailed information and by employing more sophisticated scheduling algorithms, we will be able to generate better solutions at each instance of the algorithm run. However, consequently, this may increase the discrepancy between the state of the system known to the scheduling algorithm and the actual state of the system at the time we implement the resulting schedule. Hence, we conjecture that, under some specific operating conditions, the dynamic production system will run more efficiently when we use fast simple heuristics instead of relatively slow optimization algorithms. We conjecture also that, again under some specific operating conditions, the dynamic production system will run more efficiently when we use timely accessible local information instead of more detailed global information.

In this study, we investigate the effect of the time it takes to generate a schedule on the performance of a dynamic stochastic production system. Our study is composed of two phases. In the first phase, we test the trade off between spending more time for the scheduling process by employing more sophisticated scheduling algorithms and using simple fast heuristic algorithm. In the second phase, we test the trade off between spending more time by including detailed global information to achieve global optimality under a centralized control structure and using timely accessible local information under distributed control.

In the first phase, we devise a centralized scheduling system. We explicitly model the decision time and analyze its impact on the system performance. To the best of our knowledge, there is very small number of studies in the literature where decision time is explicitly modeled. [12] works on the meeting scheduling problem, where they include the time needed to schedule meetings in their model without analyzing the effects on the system performance. [13] investigates the effect of factors such as agent population, bandwidth, message sizes and decision times in individual agents on the performance of communication system and auctioning process. [14] studies the effect of scheduling time and dispatching time for a specific queuing system. [15, 16] are the only studies which are in the same line of research as our work. The basic difference between those studies and this work is that we consider a scheduling problem where pre-emption is not allowed.

Centralized control structure is in concert with optimization but it requires to manage the data globally, hence it is impractical to timely respond unexpected events in dynamic stochastic systems (i) due to the complexity of scheduling problems, (ii) if the nature of the organization is distributed (i.e. when agents are locally isolated and administratively independent [17]) and (iii) if information sharing is restrictive [11,18].

To handle the dynamic nature of the problem in a responsive manner, one of the proposed solutions is to decompose the problem and distribute to various scheduling servers. This approach is called distributed scheduling. In distributed scheduling, a number of schedulers observe a specific part of the decomposed system, and generate solutions.

Distributing the scheduling problem brings in additional issues such as: (i) Choice of the decomposition methodology [18–20] and the control structure [21–27]; (ii) Handling mechanism for coupling constraints (shared resource) between schedulers [6, 11, 27–33]; (iii) Accuracy and timing of representation of local reality in each scheduler.

We devise a distributed scheduling system for the second phase, then the system performance under the control of fast distributed schedulers is compared with the system performance under a relatively slow centralized scheduler. The centralized scheduler is assumed to has access to all global information at the start of scheduling process but it necessitates more time for running the scheduling algorithm. As mentioned before, the centralized scheduler can provide a global optimum solution according to the state of the system when scheduling process is triggered. However the state of the system when the schedule is released at the end of the scheduling process may be significantly different since our system is dynamic and stochastic. On the other hand, distributing the scheduling problem among various schedulers may improve reactivity by providing smaller local problems. However lack of global information may reduce the quality of the solutions.

Our distributed scheduling system composes of two schedulers. Distributed schedulers handle the dynamic and stochastic nature of the scheduling environment through again a rolling scheduling process. Distributed schedulers observe different parts of scheduling problem, hence in Step 1 distributed schedulers will load the local scheduling data. For simplification we continue to assume that loading local data do not take time and loaded data is accurate. We continue to consider Steps 2 and 3 together. Remember that, in the first phase, a central scheduler running a scheduling algorithm (Step 2) takes time. Under the expectation that distributing the scheduling problem among various schedulers may improve reactivity by providing smaller local problems, we assume that scheduling process does not take time in distributed scheduling system. In Step 4, each scheduler instantaneously dispatch its own generated scheduler that ensures feasibility for its own jobs. We will test the effect of synchronization among schedulers on the distributed scheduling system performance. Besides, we want to test the trade-off between using fast-distributed asynchronous schedulers and the slow-centralized scheduler.

The rest of the report is organized as follows: In Chapter 2, related literature is presented. In Chapter 3, we formally define the dynamic scheduling environment that we investigate. Chapter 4 explains the simulation setup and experimental settings. Experimentation results are discussed in Chapter 5. We conclude the paper in Chapter 6.

# 2. LITERATURE REVIEW

## 2.1. Static Deterministic Scheduling with Earliness/Tardiness Penalties

In this section we review the single machine earliness/tardiness scheduling problem in literature. The earliness/tardiness scheduling problem can be divided into two categories: the problems with distinct due dates and the problems with common due dates. Another distinction on the problem can be made by allowing or not allowing machine idle times. Final categorization of the problems with equal release dates or distinct release dates. In this section, we only review the papers considering the problem with distinct due dates.

### 2.1.1. Scheduling without Machine Idle Times

Most of the earliness/tardiness studies in the literature avoid the issue of inserted idle time by assuming a common due date for all jobs [34]. The earliness/tardiness problem with distinct due dates, equal release dates and no idle time is studied by [35]. [35] decomposes the problem into earliness and tardiness subproblems and shows that the lower bound of the problem is the sum of the lower bounds of the two problems. Lagrangean relaxation method is used to obtain a lower bound of each subproblem. [36, 37] use the same decomposition technique and presents exact approaches for the same problem but with distinct release dates.

### 2.1.2. Scheduling with Inserted Machine Idle Times

Schedules with inserted idle times may be beneficial when when the objective function is not regular [11, 34]. In this section we present exact and approximate studies for the single machine earliness/tardiness scheduling problem in the literature that allow machine idle times.

[38] considers the objective of minimizing the mean tardiness and earliness when

due dates of jobs are not common. [38] proves several properties of an optimal solution. A branch and bound algorithm and heuristic algorithms are developed by using these properties. [39] improves the lower bounds of [38]. [39] presents a nonlinear program for the completion of a partial sequence that is solved by using a timetabling algorithm [40] proposes an approach based on a preemptive relaxation of the problem. [41] proposes the combination of a Lagrangean relaxation of resource constraints and new dominance rules that can solve instances with up to 50 jobs. [42] suggests combining dynamic programming and branch and bound techniques with the application of a transportation problem based lower bound procedure.

For the distinct due date problem, it is shown that insertion of idle time could improve the objective function and this result gave rise a two step procedure to evolve in heuristic procedures [11]: in the first step, a good job processing sequence is determined; in the second step, idle time is inserted optimally [43, 44]. The insertion of idle time problem can easily be solved either by solving a linear program or by using a specialized algorithm [42]. Beside these two step procedures, as a very recent heuristic approach, [45] has developed iterated local search algorithms based on fast neighborhoods. They showed that very good solutions for instances with significantly more than 100 jobs can be derived in a few seconds.

## 2.2. Deterioration of Static Schedules in Dynamic Stochastic Systems

Theoretical scheduling problems, which are concerned with searching for optimal schedules subject to a limited number of constraints, have a combinatorial explosion of possible solutions and are generally NP hard. Although the existence of this fact, most of the literature dealing with production scheduling has been focused primarily on finding optimal, or near-optimal, predictive schedules with respect to various criteria. These approaches have used the implicit assumption of a static deterministic environment where complete knowledge of the problem was available without consideration of any kind of failures. This is rarely the case in the real world.

[8] summarizes the most common unexpected events in the shop floor as; machine

failure, urgent job arrival, job cancellation, due date change, delay in the arrival or shortage of materials, change in job priority, over- or underestimation of process time, and operator absenteeism. These real-time events not only interrupt system operation but also upset the predictive schedule that was previously established.

[3] empirically shows that performance of "optimized" static schedules may deteriorate rapidly with processing time uncertainty and that simple dynamic dispatching heuristics may provide a far superior performance. [4] reports similar observations when considering uncertainties in job processing times and machine availability.

[5] works on a centralized solution methodology for flexible manufacturing systems and shows that online scheduling is more robust to uncertainty and variations in processing times than the optimum seeking offline scheduling. [6] studies methods to improve scheduling robustness under processing time variation for classical job shop problems. Their approach is more robust than static optimization schemes while outperforming best known dynamic heuristics in both performance and robustness.

Further studies on different issues regarding scheduling dynamic systems can be found in survey papers [7–9]. [8] presents definitions appropriate for most applications of rescheduling manufacturing systems and describe a framework for understanding rescheduling strategies, policies and methods. They also discuss studies that show how rescheduling affects the performance of a manufacturing system. [7] considers the problem of executing production schedules in the presence of unforeseen disruptions on the shop floor. A number of issues related to problem formulation is discussed. A taxonomy of the different types of uncertainty is provided. [9] is one of the recent survey of dynamic scheduling outlines the limitations of the static approaches to scheduling in dynamic environments.

## 2.3. Dynamic Scheduling with Earliness/Tardiness Penalties

The dynamic scheduling problem with earliness/tardiness penalties is studied by few researchers [11, 29, 30].

[10, 11] study the dynamic single machine scheduling problem by minimizing earliness/tardiness penalties. They present procedures perform like dispatching procedure to decide the next job to be processed while permitting schedules with idle time between jobs. [10] constructs a schedule by following a two step process consisting of generating a sequence and inserting idle time. [11] proposes an auction model by using Lagrangian relaxation to decompose the problem into job agents and the machine agent and by using Lagrangian multipliers to construct and evaluate bids. [11] defines a parameter to make the machine idle for an amount of time determined by that parameter to involve a new arriving urgent job into the auction. In these studies the performance is evaluated against other well known dispatching procedures.

[30] develops a multi agent method in flexible job shop environment for earliness/tardiness objectives. Multi agent scheduling method is compared with two contract net approaches which are used to minimize average tardiness in heterarchical scheduling of flexible manufacturing systems.

[29] develops a heuristic method based on a multi agent architecture for a parallel identical machine scheduling model and earliness/tardiness objectives. The heuristics of [10] and [46] and EDD are adopted to the online parallel machine case as non agent based heuristics. The performance of multi agent approach is compared with non agent based heuristics to show the effectiveness of such an approach.

[46] investigates the conditions where inserting machine idle times are necessary by considering mean tardiness single machine problem. [46] proposes a decision theory based heuristic by using a simple look ahead procedure to produce tactically delayed schedules. [46] suggests that tactically delayed schedules are important when due dates are arbitrary and utilization low.

## 2.4. Distributed Scheduling

Scheduling problem of manufacturing systems corresponds to a distributed problem from both the physical and the logical point of view. Physically, the system involves several resources and from the logical point of view it is also a distribution problem because several tasks can be carried out at the same time. The studies in literature vary corresponding to decomposition techniques. [19] makes a comprehensive survey on the decomposition of the scheduling problem where the issues such as how the problem is decomposed, how or to whom it is assigned, and the organization for solving the subproblems, strategies used for resolving conflicts, the manner in which communication is conducted, etc. are used as parameters to compare and contrast the approaches reported in the literature

When the problem is decomposed, the subproblems are distributed to various agents. Distributed agents are organized in three different architectures: heterarchical, hierarchical, and a hybrid-based. In hierarchical architectures, the distributed agents are organized in a centralized control structure by using a supervisor agent to act as the controller. In heterarchical architectures, the distributed agents run on a completely decentralized control structure, without the presence of supervisory agent. Hybrid-based systems are organized like hierarchical systems but enabling the self-organization capability of the lower level agents.

This section is to present a survey on several decomposition methodology and the control structure for manufacturing problems in literature, and the handling mechanisms to resolve conflicts among distributed agents.

### 2.4.1. Choice of the Decomposition Methodology

A decomposed problem is not necessarily organized in a decentralized control structure. For example, a single machine problem can be decomposed by using relaxation method, then the information is distributed in multiple sub-production systems (distributed agents). If the entire global information is required to solve the problem, then a supervisory agent is required to manage global data in a centralized control structure.

[18] represents a centralized control structure by using Lagrangian relaxation with

a sub gradient method as a decomposition technique. [18] compares it to a Lagrangian relaxation technique modified to require less global information by considering static single machine problems for total weighted completion times objective. Later method is shown to yield close-to-optimal solutions, so it is beneficial to be applied to situations with more restrictive information sharing.

[20] compares their distributed approach with a centralized system using shifting bottleneck algorithm. The performance measure is weighted make-span for flow shop and job shop configurations. [47] considers a modified shifting bottleneck heuristic for complex job shops as a decomposition technique.

The scheduling system can be decomposed physically into task and resource agents. In this kind of decomposition, the problem is to assign tasks to the resources. The indecision problem may arise either when multiple tasks negotiate with the same resources or multiple resources compete to process the same task. The agreement about which task has to be processed by which resource is made through a message passing among agents. There is a bidding protocol among the agents to achieve mutual agreement. The proposed bids include price and schedule information. This negotiation scheme called the contract net and is the common approach when the system is decomposed into task and resource agents.

The studies in literature vary with the choice of prising mechanism, the selection criterion of the most favorable bids, the communication mechanism among agents, the information required to propose a bid, etc.. The bids are proposed according to local information or global information. The evaluation of bids may be done according to local criteria or a global criterion. There may be direct communication among the group of agents eligible to propose bids in a decentralized negotiation structure, or there is a need for a supervisory agent to manage the negotiation. The selections made on these issues determines the control structure of this kind of distributed systems. [21–25] decompose the system physically, and propose various control mechanisms.

### 2.4.2. Choice of the Control Structure

[21] defines an adaptive production control system by introducing supervisory agents to provide hierarchical control as the production load on resources increases. They compare their conceptual model with a non-hierarchical production control system and they achieve improvements in throughput, resource utilization and work-in-process inventories over non-hierarchical production control system.

As a similar approach, [25] invents a hybrid- based structure by allowing to increase the autonomy of distributed agents when an unexpected disturbance is detected. The proposed idea is that distributed agents should follow the schedule advises proposed by the supervisory agent, and in case of disturbance a faster rescheduling solution is obtained by forcing agents to evolve a heterarchical structure. Superiority of the proposed approach is presented by making comparison with both heterarchical, hierarchical structures.

[22] simulates the behaviors of two heterarchical control structures, characterized by a different degree of decision-making delegation assigned to physical and information units in the production system. The performances of the two architectures are measured both under stable and predictable conditions, and under unexpected events, such as the release of urgent manufacturing orders and the occurrence of failures in production resources. [23] compares a heterarchical control architecture with their proposed unconstrained hierarchical architecture. [24] evaluates the performance of heterarchical and hybrid-based agent control structures.

[26] being a recent review study on distributed scheduling concepts, states that most of distributed scheduling literature focus on designing factors without considering how these factors affect computational time and solution quality. The the study of [27] is the exception.

[27] investigates the computational benefits of distributed decision making in a centralized control structure. They compare a network of four processors with a

single processor by monitoring communication delays, CPU utilization, system utilization and average number of bytes/s transmitted over the network. Moving from a single processor to multiple processors would speed up the computing time for each subproblem but would involve additional communication delays. The result is that if communication delays are less than time taken to solve all subproblems serially on a single processor then distributed implementation provides computing benefits despite network delays. They investigate also the performance of auction-based method over popular dispatching rules when the objective is to minimize weighted squared deviation and presents improvement over the best dispatch rule.

### 2.4.3. Handling Mechanism for the Shared Resource Between Schedulers

[28] studies the performance and design issues in multi-agents information systems for dynamic scheduling in manufacturing. The design and performance issues considered in this research are coordination between agents, number of agents, and frequency of learning. The results indicate that coordination between agents, and learning frequency play a significant role in the performance of multi-agent intelligent systems.

As mentioned above, the contract net negotiation scheme is common in literature and various negotiation mechanisms may be designed by specifying the prising mechanism, the selection criterion of the most favorable bids, the communication mechanism among agents, the information required to propose a bid, etc.. In this section we do not investigate the designing factors of negotiation mechanisms. We rather present several solution approaches to the situations arise in distributed systems.

Distributed agents negotiate with contract net protocol may either present procedure like a dispatching procedure to decide the next job to be processed [11, 29, 30] or may produce a schedule to be used for a length of time horizon [27].

[31] proposes least commitment scheduling and dispatching for dynamic order processing. The distributed decision mechanism is based on the following principles: 1-Any task may be assigned to multiple resources. 2-Resources may have alternative

assignments. The negotiation protocol is as follows: management agent announces iteratively the first unprocessed task of a job, machine agents (with matching resources) prepare schedules and submit bids. Each machine agent may send no, one or several bids to the management agent. Whenever a task has been selected by a machine agent, management agent prompts the other machines to remove all the assignments of this task from their schedules and it announces the successor task of this job.

[6] handles the case of asymmetric information for the independent distributed agents, which may not share private information about their state. A schedule selection game is designed where all participating agents state their preferences via a valuation scheme, and the mechanism selects a final schedule based on the collective input.

The scheduling problems discussed thus far are all single criteria scheduling problems, i.e. either a central scheduler or distributed schedulers try to achieve a global goal. There are also multi criteria scheduling problems. Different agents may have different objectives. [32, 33] investigate the complexity of some scheduling problems with a single machine in which agents, each owing a set of nonpreemtive jobs have to negotiate the usage of the common resource and only the jobs belonging to an agent contribute to that agent's criterion. [32] considers maximum of regular functions, number of late jobs and total weighted completion time. [33] provides extensions for two agent scheduling problem for single machine and considers also identical machines in parallel.

## 2.5. Scheduling Time Considerations

To the best of our knowledge, there is very small number of studies in the literature where decision time is explicitly modeled. [12] works on the meeting scheduling problem, where they include the time needed to schedule meetings in their model without analyzing the effects on the system performance. [13] investigates the effect of factors such as agent population, bandwidth, message sizes and decision times in individual agents on the performance of communication system and auctioning process. [14] studies the effect of scheduling time and dispatching time for a specific queuing system. [15,16] are the only studies which are in the same line of research as our work. The basic difference between those studies and this work is that we consider a scheduling problem where pre-emption is not allowed, and a different performance measure $L_{max}$ is considered in these studies.

# 3. SCHEDULING ENVIRONMENT

In this study, we investigate a single machine system where jobs arrive randomly. Let $a_j$ indicate the arrival time of job $j$, and $J_t$ indicate the set of jobs that has arrived by time $t$. We assume that jobs are indexed in nondecreasing order of their arrival times. There is an infinite queue in front of the machine. Upon arrival, a due date $d_j$ and an estimated processing time $\hat{p}_j$ are assigned to job $j$. Let $\pi$ represent a schedule. The start time of job $j$ in schedule $\pi$ is denoted as $s_j(\pi)$. Machine may process one operation at a time. A job which is started on the machine cannot be interrupted or canceled. Hence, $c_j(\pi) = s_j(\pi) + \hat{p}_j$ is the estimated completion time of $j$ on schedule $\pi$.

Let $s_j^R$ indicate the realized start time of job $j$ on the machine. Since processing times are not deterministic, the machine may complete the processing of job $j$ at time $c_j^R$, where $p_j^R = c_j^R - s_j^R$, and $p_j^R \neq \hat{p}_j$. Originally, $s_j^R, c_j^R$ and $p_j^R$ values of jobs are set to zero. Let $J_t^S = \{j : j \in J_t, \text{ and } (c_j^R = 0 \text{ or } c_j^R > t)\}$ indicate the set of jobs that are in the system at time $t$. We will denote the job which is being processed on the machine at time $t$ as $j(t)$. If $j(t) = 0$ then the machine is idle at time $t$; otherwise, $j(t) \in J_t^S$ and $0 < s_{j(t)}^R \leq t$.

Let realized earliness and tardiness of job $j$ be defined as $E_j^R = \max(0, d_j - c_j^R)$ and $T_j^R = \max(0, c_j^R - d_j)$, respectively. The performance of the system is measured by

$$\lim_{n \to \infty} \frac{\sum_{j=1}^n E_j^R + \sum_{j=1}^n T_j^R}{n}.$$

For the first phase of the study, in order to model the issues originating from decision time, we decouple and simplify the scheduling process described in Chapter 1 into two separate processes: the scheduling process, performed by the *scheduler*, and the dispatch process, performed by the *dispatcher*. The scheduling process combines

Steps 1, 2 and 3. while the dispatch process covers Step 4. The operation of the centralized scheduler and the dispatcher is explained in Section 3.1.

For the second phase of the study, we continue with the same simplification as in the first phase. However, distributed schedulers observe different parts of the global problem, hence they have access to load different scheduling data in Step 1 of the scheduling process. Moreover, since each scheduler performs its own manual overrides in Step 3, there may be more than one generated schedule to dispatch in Step 4. The operation of the distributed scheduler and the dispatcher in distributed scheduling system is explained in Section 3.2.

## 3.1. The Centralized Scheduling System

The overall system works as depicted in Figure 3.1.

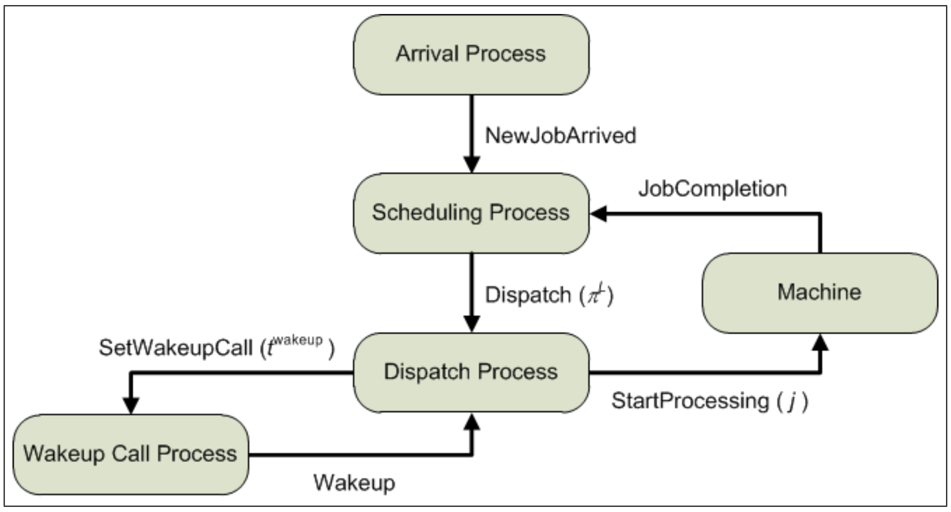


Figure 3.1. Overall Architecture of the Centralized Scheduling System.

Stochastic *Arrival Process* generates *NewJobArrived* events at each time a new job has arrived to the shop, which immediately triggers the *Scheduling Process.* Completion of the *Scheduling Process* releases a new schedule ($\pi^L$) and generates a *Dispatch* command to trigger a *Dispatch Process.* The *Dispatch Process* either sends a *StartProcessing* command to the machine to start the processing of a job ($j$), or it sends a *SetWakeupCall* command to the *Wakeup Call Process*, which may trigger the *Dispatch Process* again at the requested wakeup time ($t^{\text{wakeup}}$) by generating a *Wakeup* event.

When the machine completes processing a job, it updates the realization info of the job that is being processed and generates a *JobCompletion* event, which again triggers the *Scheduling Process*.

In the next sections, we explain the dynamics of the system by providing detailed workings of the scheduler and the dispatcher.

### 3.1.1. The Centralized Scheduler

The centralized scheduler starts the execution of a scheduling process upon receiving a scheduling trigger. As shown in Figure 3.1, arrival of a new job (*NewJobArrived* event) and completion of the processing of a job on the machine (*JobCompletion* event) generate scheduling triggers.

Assume that we are at time $t$, and the scheduler receives a scheduling trigger. Let decision take $\delta$ time units, then the scheduler releases the new schedule at time $t+\delta$. We say that the scheduler is *busy* at time $t$, if the scheduler had already started running the scheduling algorithm in response to a previous scheduling trigger at time $t_1$, where $t_1 < t$ and $t_1 + \delta > t$.

If the scheduler is not busy at time $t$ but $J_t^S = \emptyset$, then there is no job left in the system to schedule, and the scheduler ignores the trigger. However, if $J_t^S \neq \emptyset$, then it constructs the following single machine static deterministic scheduling problem, solves it, and releases the resulting schedule.

**Problem** $\mathcal{P}^{\mathcal{SP}}(t)$ : Given $\hat{p}_j$ and $d_j$, generate a schedule $\pi = \{s_j, c_j : j \in J_t^S\}$ to minimize $\sum_{j\in J_t^S} E_j + \sum_{j\in J_t^S} T_j$, where $E_j = \max(0, d_j - c_j)$, and $T_j = \max(0, c_j - d_j)$, subject to $s_{j(t)} = s_{j(t)}^R$ and $c_{j(t)} = \max(t, s_{j(t)}^R + \hat{p}_{j(t)})$ if $j(t) > 0$.

Problem $\mathcal{P}^{\mathcal{SP}}(t)$ is the well-known non-preemptive $1||\sum(E_j + T_j)$ problem, which is known to be strongly NP-hard [41]. In the literature, there are a number of studies proposing lower bounds and dominance rules to be used within a branch and bound schema to optimally solve the problem. [40] proposes an approach based on a preemptive relaxation of the problem. [41] combines Lagrangean relaxation of resource constraints and new dominance rules. [42] develops a technique which combines dynamic programming and branch and bound by applying a transportation problem based lower bound procedure.

Since the objective of our problem is non-regular, inserting idle time between operations may improve the objective function. Based on this observation a two step heuristic procedure can be defined [11]: in the first step, a good job sequence is obtained; in the second step, idle time is inserted optimally [44, 48]. Insertion of idle time problem can easily be solved either by solving a linear program or by using a specialized algorithm [42]. Beside these two step procedures, [45] reports an iterated local search algorithm based on fast neighborhoods.

Let schedule $\pi$ be a solution to Problem $\mathcal{P}^{\mathcal{SP}}(t)$ generated and released by the scheduler. We will indicate the latest schedule released by time $t$ as $\pi_t^L$, hence, $\pi_{t'}^L = \pi$ for all $t' \geq t + \delta$ until the scheduler releases a new schedule. We let $J(\pi) = J_t^S$ indicate the set of jobs considered while generating $\pi$, $t^S(\pi) = t$ indicate the time the scheduling process has started and $t^G(\pi) = t + \delta$ indicate the time that $\pi$ is generated.

In case the scheduler is busy at the time that a new scheduling trigger is received, the response of the scheduler to this trigger depends on the response mode installed as a control policy. In this study, we define three response modes: Instantaneous, busy and available response modes.

#### 3.1.1.1. Instantaneous Mode of the Scheduler.

The instantaneous mode is related to the case when executing the scheduling process does not take time, i.e., $\delta = 0$.

*Instantaneous Mode*: Every scheduling trigger starts a new scheduling algorithm run, and the resulting schedule is released instantaneously, which immediately generates a dispatch trigger.

3.1.1.2. Busy Mode of the Scheduler. In the case when executing the scheduling process takes time, say $\delta > 0$ time units, during the execution of the scheduling process, the scheduler is said to be busy and cannot run a second process simultaneously. Busy mode defines one of the two alternative behaviors for the scheduler when a scheduling trigger is received while it is busy.

Assume that we are at time $t$, and the scheduler started running the scheduling algorithm at time $t_1 < t$. Note that $t^G(\pi_t^L) \leq t_1$.

*Busy Mode*: The scheduler ignores the new scheduling trigger and continues with the scheduling process execution.

In the busy mode, a new schedule is not released at time $t$, hence, as a consequence, a dispatch trigger is not generated. In effect, the system waits at least until $t_1 + \delta$, when the scheduler completes the execution of the scheduling process. Hence, in this mode, $\delta$ is a determinant of the length of time that the system stays idle waiting for the scheduler to make a decision.

3.1.1.3. Available Mode of the Scheduler. The available mode is related to the case when executing the scheduling process takes time, i.e., $\delta > 0$. Available mode defines one of the two alternative behaviors during the execution of the scheduling process.

Assume again that we are at time $t$, and the scheduler started running the scheduling algorithm at time $t_1 < t$.

*Available Mode*: The scheduler releases $\pi_t^L$ again at time $t$ and simultaneously, continues with the scheduling process execution.

In the available mode, although the scheduler continues executing the scheduling process to generate a new schedule, at time $t$, it also releases the latest schedule it generated ($\pi_t^L$) one more time. Although, $\pi_t^L$ is generated based on the state of the system as of time $t^S(\pi_t^L) \leq t_1 - \delta < t$, and hence, it contains relatively old data, as a direct result of releasing a schedule, a dispatch trigger is generated, and possibly, a job is dispatched to the machine. In this mode, if $\delta$ gets larger, the difference between the state of the system captured in $\pi_t^L$, which corresponds to time $t^S(\pi_t^L)$, and the current system state increases.

### 3.1.2. The Dispatcher in the Centralized Scheduling System

The dispatcher starts a dispatch process upon receiving a dispatch trigger. As shown in Figure 3.1, completion of a scheduling process execution (*Dispatch* command) and a wakeup call previously set by a previous dispatch process (*Wakeup* event) generate dispatch triggers.

Assume that we are at time $t$, and the dispatcher receives a dispatch trigger. As a response, the dispatcher runs the following Dispatch$(t, \pi)$ algorithm in Figure 3.2, given a previously generated schedule $\pi$ to dispatch. Note that, in case decision takes time, $t^S(\pi) < t$, hence it is possible that there are jobs such that $j \in J(\pi)$ and $0 < c_j^R \leq t$. Let us define $\bar{J}(t, \pi) = \{j : j \in J(\pi) \text{ and } (c_j^R = 0 \text{ or } c_j^R > t)\}$ as the set of jobs that can be processed at time $t$ that exists in schedule $\pi$.

```
**Require** $t, \pi$ are defined.
Step 1. **If** $j(t) > 0$ **then** the machine is busy, EXIT.
Step 2. **If** $\bar{J}(t, \pi) = \emptyset$ **then** there are no jobs left in schedule $\pi$ that can be processed, EXIT.
Step 3. **If** there exists a job $j \in \bar{J}(t, \pi)$ such that $s_j(\pi) \leq t$ **then** start processing $j$ on the machine, i.e., set $s_j^R = t$ and EXIT.
Step 4. **If** for all $j \in \bar{J}(t, \pi), s_j(\pi) > t$ **then** put the dispatcher to sleep, and set a wakeup call at time $t^{\text{wakeup}} = \min_{j \in \bar{J}(t,\pi)}\{s_j(\pi)\}$.
```

Figure 3.2. Dispatch$(t, \pi)$ Algorithm.

Note that in Step 1, we stop the dispatch process because preemption is not allowed. Step 2 checks if there are jobs to be dispatched in the queue that are included in the current schedule. We physically start processing a job in Step 3. We assume that executing the dispatch process does not take time and selected the job can be started immediately at time $t$. In Step 4, although there are jobs in the queue, we leave the machine idle. This may be the optimum behavior since we have a non-regular performance measure.

If a dispatch trigger is received while there is an active wakeup call, the wakeup call is removed.

### 3.1.3. Idle Times on the Machine

Based on the definition of Dispatch$(t, \pi)$ algorithm in Figure 3.2 in Section 3.1.2 and the response modes of the scheduler explained in Section 3.1.1, idle time on the machine may be identified as one of the three types.

3.1.3.1. Queue Idleness (QI). The machine is kept idle until a new dispatch trigger arrives if, in Step 2 of Dispatch$(t, \pi)$ algorithm in Figure 3.2, we realize that the queue is empty ($J_t^S = \emptyset$). Hence, we call this case the queue idleness.

3.1.3.2. Forced Idleness (FI). In Step 4 of Dispatch$(t, \pi)$ algorithm in Figure 3.2, although there are jobs in the queue and the machine is idle, based on schedule $\pi$, we choose to keep the machine idle with the expectation that the performance of the system will be improved. Hence, we call the inserted idle time between $t$ and $\min_{j \in \bar{J}(t,\pi)}\{s_j(\pi)\}$ as the forced idleness.

3.1.3.3. Decision Idleness (DI). Decision idleness is related to the operation modes of the scheduler in case decision time is nonzero. Hence, in a sense, idleness is caused by the scheduling process, and that is why we call this type of idleness the decision idleness. For the busy and available response modes, decision idleness appears as follows: For the *busy mode*, any scheduling trigger is denied by the scheduler if it is already busy.

Hence, no job is dispatched until the scheduling process is completed, and the machine is kept idle.

For the *available mode*, appearance of the idle time is a bit more obscure. In this mode, although the scheduler keeps releasing schedule $\pi^L$ while it is busy with scheduling, at some time $t$, $\bar{J}(t, \pi^L)$ may become empty, and the machine may be kept idle in Step 2 of Dispatch$(t, \pi)$ algorithm in Figure 3.2. Note that since $t(\pi^L) < t$, there may be jobs that has arrived between $t(\pi^L)$ and $t$. This means that actually, there are jobs in the queue, but since the dispatcher is unaware of this situation, the machine is kept idle.

### 3.1.4. Example Run of the Centralized Scheduling System

In this section, we provide example runs of the system for the three distinct modes of the scheduler explained in Section 3.1.1. We use Figure 3.3, 3.4 and 3.5 to demonstrate the generated triggers and commands within the system and activation of processes on a time line (horizontal axis).

In all three figures, $t_0$ indicates the initial state of the system that is captured. The first line indicates job arrivals. The second line represents the scheduler, where a box indicates that the scheduler is busy (executing the scheduling process) in that time period. The third line shows the schedules that are released by the scheduler. Cross symbols are placed on this line at the times that a new schedule is released. The job sequence of the corresponding schedule is also shown close to the cross symbols. The fourth line shows the messages received and sent by the dispatcher. Fifth line shows the wakeup calls set by the dispatcher, where shaded boxes are the times that the dispatcher is sleeping. The sixth line shows the actual processing of jobs and the types of the idle times on the machine.

Figure 3.3 shows a sample run of the system when the scheduler operates under the instantaneous mode. As it can be seen from Figure 3.3, $\pi^L_{t_0} = \{0, 1\}, j(t_0) = 0$ and $c^R_0 = t_1$. At time $t_1$, the machine generates a *JobCompletion* event, which triggers the

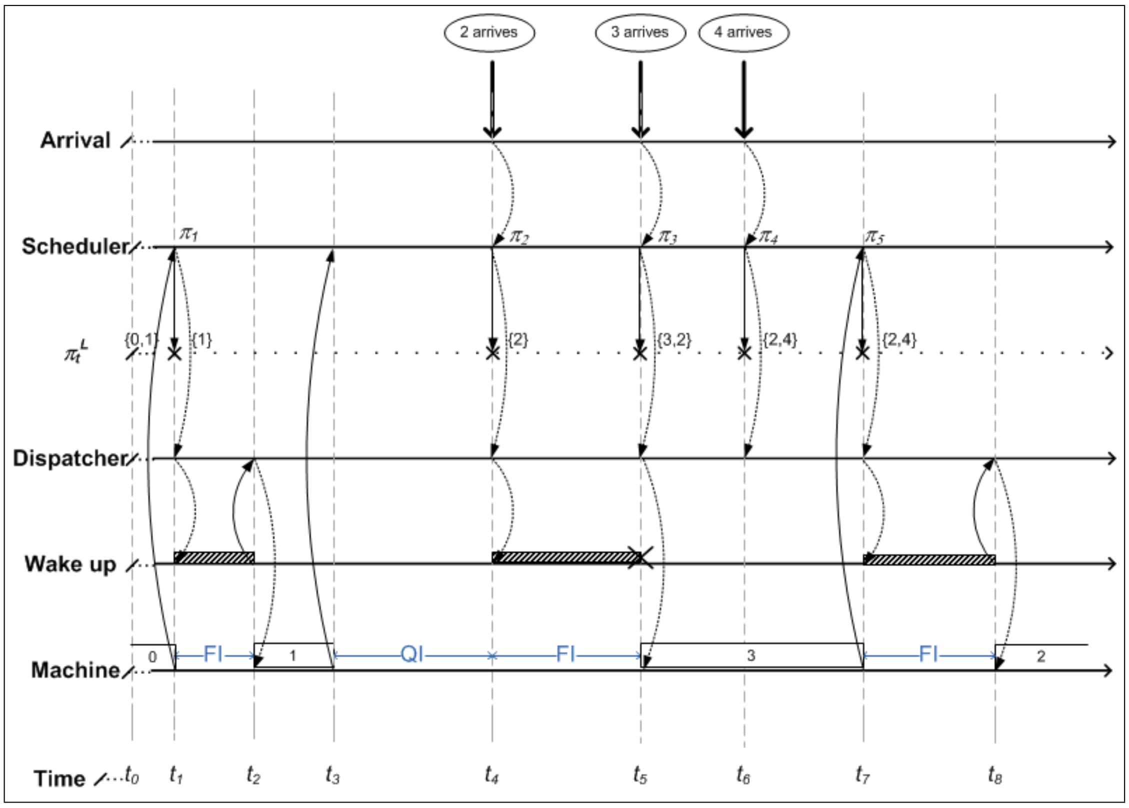


Figure 3.3. Sample Run Using the Instantaneous Response Mode.

scheduler. The scheduler generates schedule $\pi_1$, where the job sequence is $\{1\}$, and releases. From Figure 3.3 we deduce that $s_2(\pi_1) = t_2 > t_1$. Hence,the dispatcher sets a wakeup call, which eventually triggers the dispatcher to command the machine to start processing job 1 at time $t_2$. Note that, since the machine idle time between $t_1$ and $t_2$ is imposed by the inserted idle time in schedule $\pi_1$, it is identified as FI. However, the reason for the idle time between $t_3$ and $t_4$ is that there are no jobs left in the system, hence it is a QI.

At time $t_4$ Job 2 arrives, and the scheduler releases schedule $\pi_2$. Apparently the due date of job 2 is very large and the dispatcher sets a wakeup call. Hence an FI period starts for the machine. This lasts until time $t_5$ when job 3, which is an urgent job, arrives. At this instance, the scheduler releases schedule $\pi_3$, which includes the sequence $\{3, 2\}$. This immediately cancels the current wakeup call, and the dispatcher commands the machine to start job 3. Note that, although the arrival of job 4 triggers the scheduler to release schedule $\pi_4$, the dispatcher does not dispatch the new schedule since the machine is already busy and the current processing cannot be interrupted.

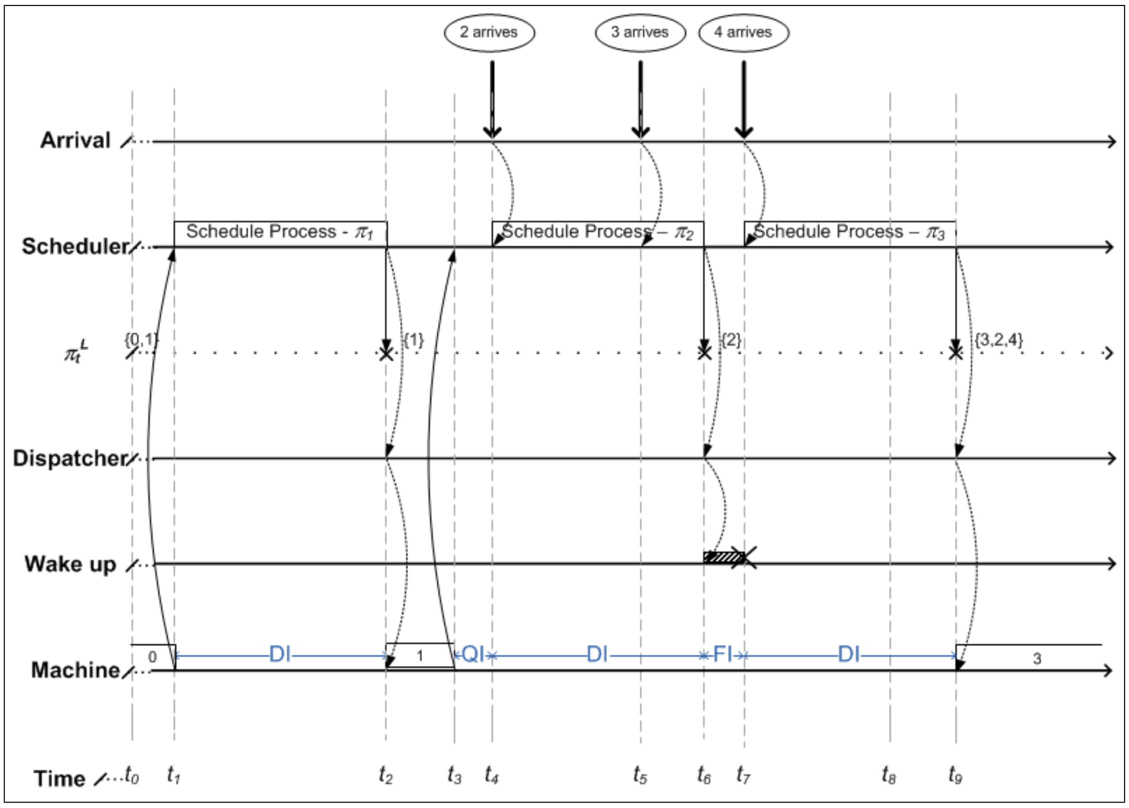

2 arrives
3 arrives
4 arrives
Arrival
Scheduler
Schedule Process - $\pi_1$
Schedule Process – $\pi_2$
Schedule Process – $\pi_3$
$\pi_t^L$
{0,1}
{1}
{2}
{3,2,4}
Dispatcher
Wake up
Machine
0
DI
1
QI
DI
FI
DI
3
Time
$t_0$ $t_1$ $t_2$ $t_3$ $t_4$ $t_5$ $t_6$ $t_7$ $t_8$ $t_9$

Figure 3.4. Sample Run Using the Busy Response Mode.

Note again that, there are no boxes in the scheduler line and there are no idle times of type DI on the machine, since, in this mode, the scheduling process is instantaneous.

Figure 3.4 depicts the case when the scheduler operates under the busy mode. Note that in this case the scheduling process takes time, and if a scheduling trigger is received while the scheduler is busy, such as at time $t_5$, the trigger is ignored. At time $t_3$ job 1 is completed, but since there are no jobs left in the system, the machine enters a QI. At time $t_4$ job 2 arrives. Hence, there is a job that can be processed now. However, since the scheduling process takes time and the job cannot be started on the machine without being dispatched based on a schedule, the system waits until $t_6$. Hence, time period between $t_4$ and $t_6$ is a DI. One interesting issue that originates due to the blindness of the scheduler during the scheduling process is that job 3, which arrives at time $t_5$, can only be included into a released schedule at time $t_9$, i.e. although job 3 is in the system between $t_5$ and $t_9$, it is invisible to the scheduling system.

The difference between the busy mode and the available mode is apparent in Fig-

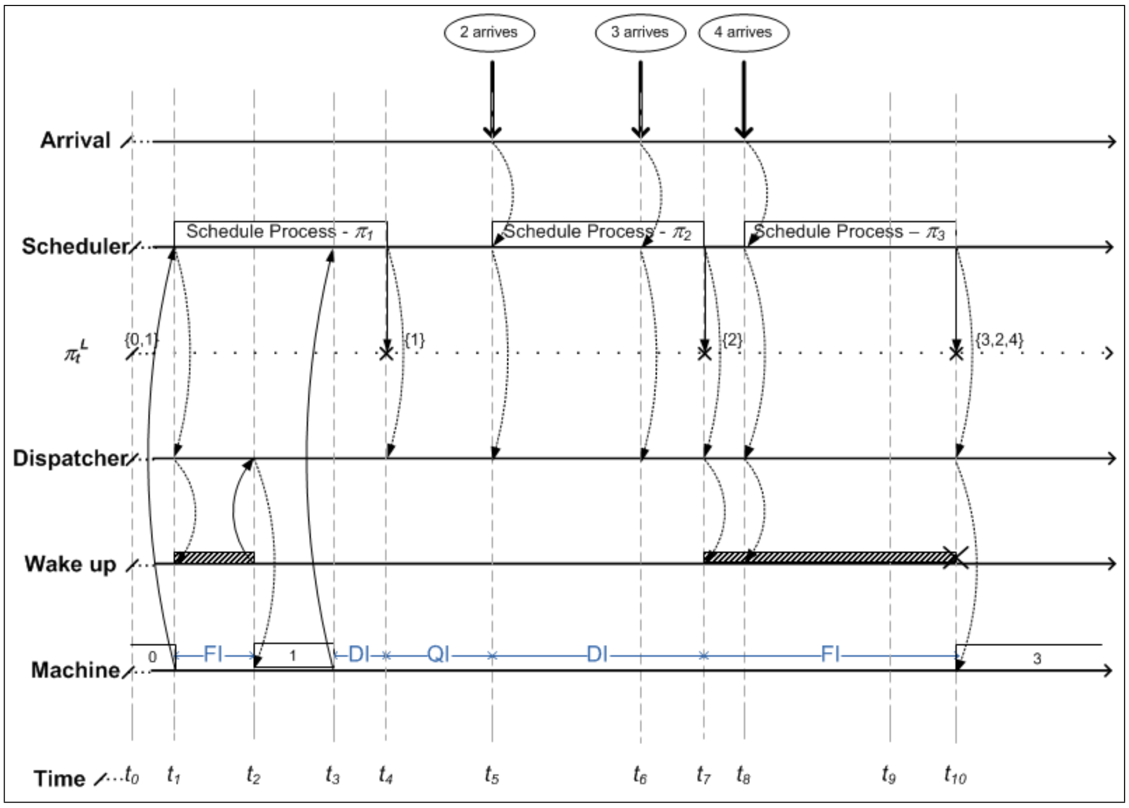


Figure 3.5. Sample Run Using the Available Response Mode.

ure 3.5. At time $t_1$ job 0 completes processing, which triggers execution of a scheduling process in the scheduler. In the busy mode, the system was put into a stand still until the scheduling is over. However, in the available mode, simultaneously, the scheduler releases the last schedule $\pi_0$ it had previously generated. In $\pi_0$, the job sequence is $\{0, 1\}$. Since, job 0 is already completed, and from Figure 3.5 we can deduce that $s_1(\pi_0) = t_2$, the dispatcher sets a wakeup call. Hence, eventually, the dispatcher can command the machine to start processing job 1 at time $t_2$.

When we compare the three realizations in Figure 3.3, 3.4 and 3.5, instantaneous mode is able to process jobs 1 and 3 as efficiently as possible. Busy mode significantly delays both job 1 and job 3, which can be attributes to the inserted DI periods. Although, available mode handles job 1 better, it still significantly delays the start of job 3 due to the blindness of the scheduling system.

## 3.2. The Distributed Scheduling System

The overall system works as depicted in Figure 3.6.

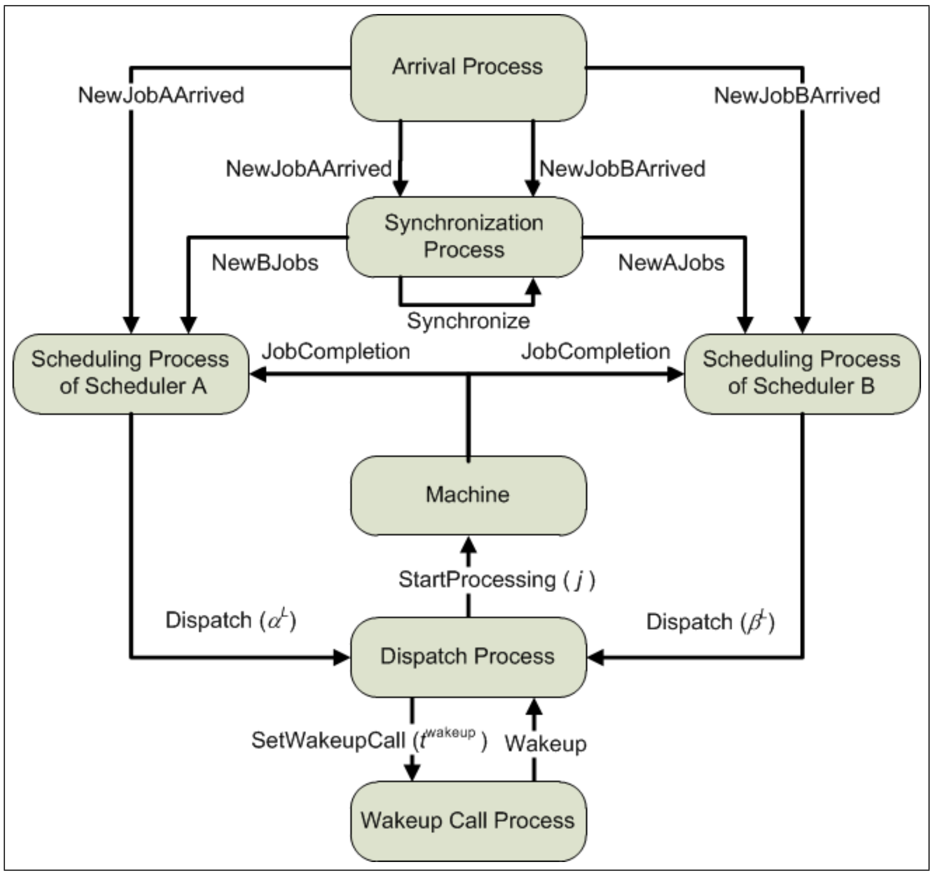


Figure 3.6. Overall Architecture of the Distributed Scheduling System.

There are two types of jobs arriving to the system: type A and type B. Jobs are considered separately by two different schedulers A and B that share a single machine. Each scheduler always knows its own type of arrivals and is informed periodically about other type of arrivals. Schedulers provide solutions for their own local problems, and the dispatcher aims to resolve conflicts among these schedules to dispatch the machine.

Stochastic *Arrival Process* generates *NewJobAArrived* and *NewJobBArrived* events according to the type of arrived jobs at the time a new job has arrived to the shop. *NewJobAArrived* event is received by *Synchronization Process* and *Scheduling Process of Scheduler A*. *Synchronization Process* stores the information of new arrivals and periodically sends to its self *Sychronize* event to update the list of jobs of each scheduler by sending *NewAJobs* and *NewBJobs* events.

*NewJobAArrived* immediately triggers the *Scheduling Process of Scheduler A*. Completion of the *Scheduling Process of Scheduler A* releases a new schedule $\alpha^L$ and generates a *Dispatch* command to trigger a *Dispatch Process.* The operation is similar for *NewJobBArrived* which results the generation of a new schedule $\beta^L$ and a *Dispatch* command to trigger a *Dispatch Process.*

From now on, a schedule generated by the scheduler A is indicated by $\alpha$ and a schedule generated by the scheduler B is indicated by $\beta$. The *Dispatch Process* considers the schedules $\alpha^L$ and $\beta^L$ on hand ,then it either sends a *StartProcessing* command to the machine to start the processing of a job $(j)$, or it sends a *SetWakeupCall* command to the *Wakeup Call Process*, which may trigger the *Dispatch Process* again at the requested wakeup time $(t^{\text{wakeup}})$ by generating a *Wakeup* event. When the machine completes processing a job, it updates the realization info of the job that is being processed and generates a *JobCompletion* event, which again triggers the *Scheduling Process of Scheduler A* and the *Scheduling Process of Scheduler B.*

### 3.2.1. The Distributed Schedulers

The distributed schedulers start the execution of a scheduling process upon receiving a scheduling trigger. As shown in Figure 3.6, arrival of a new job triggers the corresponding scheduler according to the the type of arrived job and completion of the processing of a job on the machine generate scheduling triggers for both of the schedulers.

Under the expectation that distributing the scheduling problem among various schedulers improves reactivity, we assume that distributed schedulers run a scheduling algorithm instantaneously, i.e. upon receiving a scheduling trigger both schedulers are capable of releasing their schedules based on the information that they have at the time of the trigger (for other possible operating modes of a scheduler please see Section 3.1.1).

Suppose that we are at time $t$. We will denote the most recent synchronization time that both schedulers are synchronized by $t^{\text{synch}}$. The scheduler A(B) considers both types of schedulable jobs that are in the system at time $t^{\text{synch}}$ and type A(B) jobs that are arrived during $[t^{\text{synch}}, t]$.

Let $j^{\text{type}}$ denote the type of the job $j$. The set $J_t^A = \{j : (j \in J^S_{t^{\text{synch}}})$ or $(j \in J_t^S$ and $j^{\text{type}} = A\}$ indicate the schedulable jobs of scheduler A when it generates the schedule $\alpha$. The set $J_t^B = \{j : (j \in J^S_{t^{\text{synch}}})$ or $(j \in J_t^S$ and $j^{\text{type}} = B\}$ indicate the schedulable jobs of scheduler B when it generates the schedule $\beta$.

Let $t^x$ denote the time that the recent synchronization process starts. Let $\Omega$ represent synchronization period length, and $\omega$ represent the time it takes to synchronize. If $\Omega = 0$ and $\omega = 0$, then $t^{\text{synch}} = t$. Otherwise there is a synchronization period and it takes some time to synchronize (i.e. $\Omega > 0$ and $\omega \geq 0$). In this case $t^{\text{synch}}$ is determined by following relations: if $(t^x + \omega) \leq t$ then the recent synchronization process is completed before $t$ and the schedulers can access the information of the system at time $t^x$, hence $t^{\text{synch}} = t^x$. Else if $t^x \leq t < t^x + \omega$, then the recent synchronization is not completed yet, so the schedulers can use the previous information at time $t^{\text{synch}} = (t^x - \Omega)$.

Suppose that the scheduler A is received a scheduling trigger at time $t$. If $J_t^A = \emptyset$, then there is no job left in its local system to schedule, and the scheduler A ignores the trigger. If $J_t^A \neq \emptyset$, then the local scheduler A constructs the following single machine static deterministic scheduling problem, solves it, and releases the resulting schedule $\alpha$.

**Problem** $\mathcal{P}^{\mathcal{SPA}}(t)$ : Given $\hat{p}_j$, $d_j$ and $j(t)$, generate a schedule $\alpha = \{s_j, c_j : j \in J_t^A\}$ to minimize $\sum_{j \in J_t^A} E_j + \sum_{j \in J_t^A} T_j$, where $E_j = \max(0, d_j - c_j)$, and $T_j = \max(0, c_j - d_j)$, subject to $s_{j(t)} = s^R_{j(t)}$ and $c_{j(t)} = \max(t, s^R_{j(t)} + \hat{p}_{j(t)})$ if $j(t) > 0$.

Remember that if $j(t) = 0$ then the machine is idle at time $t$; else $0 < s^R_{j(t)} \leq t$.

Similarly, suppose that the scheduler B is received a scheduling trigger at time $t$, and that $J_t^B \neq \emptyset$, then the local scheduler B constructs the following single machine static deterministic scheduling problem, solves it, and releases the resulting schedule $\beta$.

**Problem** $\mathcal{P}^{\mathcal{SPB}}(t)$ : Given $\hat{p}_j$, $d_j$ and $j(t)$, generate a schedule $\alpha = \{s_j, c_j : j \in J_t^B\}$ to minimize $\sum_{j \in J_t^B} E_j + \sum_{j \in J_t^B} T_j$, where $E_j = \max(0, d_j - c_j)$, and $T_j = \max(0, c_j - d_j)$, subject to $s_{j(t)} = s_{j(t)}^R$ and $c_{j(t)} = \max(t, s_{j(t)}^R + \hat{p}_{j(t)})$ if $j(t) > 0$.

### 3.2.2. Issues in Distributed Systems

*Perfect Synchronization*: If $J_t^A = J_t^B = J_t^S$ $\forall t$ then schedulers A and B are perfectly synchronized.

Perfect synchronization among distributed schedulers is possible only when $\Omega = 0$ and $\omega = 0$.

*Conflicting Schedules*: Schedules $\alpha$ and $\beta$ are conflicting schedules, if at least one of the following is true: 1. $\exists j \in J(\alpha) \cap J(\beta) \ni s_j(\alpha) \neq s_j(\beta)$; 2. $\exists i \in J(\alpha)$ and $j \in J(\beta)$ and $t \ni s_i(\alpha) < t < c_i(\alpha)$ and $s_j(\beta) < t < c_j(\beta)$.

*Problem 1*: When $\Omega > O$, $\exists t \ni J_t^A \neq J_t^S$ or $J_t^B \neq J_t^S$, hence $\alpha_t^L$ and $\beta_t^L$ may be conflicting schedules.

Although the scheduling algorithms are same for both schedulers, they produce different schedules regarding to their local informations even they are triggered at the same time.

*Problem 2*: $\exists t$ such that $t^S(\alpha) = t' < t$ and $t^S(\beta) = t$. When $J_{t'}^A \neq J_t^B$, $\alpha$ and $\beta$ are conflicting schedules. When $J_{t'}^A = J_t^B$ and $t^S(\alpha) \neq t^S(\beta)$ , $\alpha$ and $\beta$ may be conflicting schedules.

If the timing of scheduling triggers are different for each scheduler, schedulers may release conflicting schedules even under the assumption that perfect synchronization is possible (i.e. when $\Omega = 0$ and $\omega = 0$). Consider the case when $\Omega = 0$, $\omega = 0$ $t^S(\alpha) = t' < t$ and $t^S(\beta) = t$, then $J^A_{t'} = J^S_{t'}$ and $J^B_t = J^S_t$. If $J^S_{t'} \neq J^S_t$, then $\alpha$ and $\beta$ are conflicting schedules.

As it can be seen from Figure 3.6, type A arrivals trigger scheduler A and not B. Similarly type B arrivals trigger only scheduler B. Hence, Problem 2 is realized in this system.

Moreover, different scheduling start times may produce different schedules for the same problem. For example, lets consider Problems $\mathcal{P}^{\mathcal{SPA}}(0)$ and $\mathcal{P}^{\mathcal{SPB}}(3)$ when $J^A_0 = J^B_3$. Suppose that $J^A_0 = \{1, 2\}$ with $\hat{p}_1 = 5$, $\hat{p}_2 = 1$, $d_1 = 6$ and $d_2 = 5$, hence $\alpha^L_0 = \{s_1 = 1, c_1 = 6, s_2 = 6, c_2 = 7\}$. For $J^B_3 = \{1, 2\}$, $\beta^L_3 = \{s_1 = 4, c_1 = 9, s_2 = 3, c_2 = 4\}$. $\alpha^L_0$ and $\beta^L_3$ are conflicting schedules.

### 3.2.3. The Dispatcher in the Distributed Scheduling System

The dispatcher starts a dispatch process upon receiving a dispatch trigger. In the distributed system presented in Figure 3.6, completion of a scheduling process execution (*Dispatch* command) and a wakeup call previously set by a previous dispatch process (*Wakeup* event) generate dispatch triggers.

In this distributed system the dispatcher may have two different schedules on hand from the two schedulers when it receives a dispatch trigger. In case these are conflicting schedules, the dispatcher resolves conflicts by prioritizing one of them to dispatch the machine.

Suppose that $\alpha$ and $\beta$ are the schedules on hand that we are at time $t$. Let us define $\bar{j}_\alpha = \{j : (c^R_j = 0 \text{ or } c^R_j > t) \text{ and } s_j(\alpha) = \min_{i \in J(\alpha)} s_i(\alpha)\}$ as the job selected to be processed at time $t$ according to the schedule $\alpha$. Similarly, define $\bar{j}_\beta = \{j : (c^R_j = 0 \text{ or } c^R_j > t) \text{ and } s_j(\beta) = \min_{i \in J(\beta)} s_i(\beta)\}$ as the job selected to be processed at time

$t$ according to $\beta$. When there are two schedules on hand, possible decisions that the dispatcher takes are defined as follows:

*Dispatch(-/-)*: If $\bar{j}_\alpha = 0$ and $\bar{j}_\beta = 0$, then there are no jobs selected to be processed.

*Wait(-/w)*: If $\bar{j}_\alpha = 0$ and $\bar{j}_\beta > 0$ with $s_{\bar{j}_\beta}(\beta) > t$, then the dispatcher needs to wait until $s_{\bar{j}_\beta}(\beta)$.

*Wait(w/-)*: If $\bar{j}_\alpha > 0$ with $s_{\bar{j}_\alpha}(\alpha) > t$ and $\bar{j}_\beta = 0$, then the dispatcher needs to wait until $s_{\bar{j}_\alpha}(\alpha)$.

*Dispatch(-/s)*: If $\bar{j}_\alpha = 0$ and $\bar{j}_\beta > 0$ with $s_{\bar{j}_\beta}(\beta) \leq t$, then the dispatcher needs to start $\bar{j}_\beta$.

*Dispatch(s/-)*: If $\bar{j}_\alpha > 0$ with $s_{\bar{j}_\alpha}(\alpha) \leq t$ and $\bar{j}_\beta = 0$, then the dispatcher needs to start $\bar{j}_\alpha$.

*Wait(w/w)*: If $\bar{j}_\alpha > 0$ with $s_{\bar{j}_\alpha}(\alpha) > t$ and $\bar{j}_\beta > 0$ with $s_{\bar{j}_\beta}(\beta) > t$ and $\bar{j}_\alpha \neq \bar{j}_\beta$, then the dispatcher waits until $min\{s_{\bar{j}_\alpha}(\alpha), s_{\bar{j}_\beta}(\beta)\}$.

*Dispatch(w/s)*: If $\bar{j}_\alpha > 0$ with $s_{\bar{j}_\alpha}(\alpha) > t$ and $\bar{j}_\beta > 0$ with $s_{\bar{j}_\beta}(\beta) \leq t$ then the dispatcher starts $\bar{j}_\beta$. (Note that if $\bar{j}_\alpha = \bar{j}_\beta$ then $\alpha$ and $\beta$ are conflicting schedules.)

*Dispatch(s/w)*: If $\bar{j}_\alpha > 0$ with $s_{\bar{j}_\alpha}(\alpha) \leq t$ and $\bar{j}_\beta > 0$ with $s_{\bar{j}_\beta}(\beta) > t$ then the dispatcher starts $\bar{j}_\alpha$. (Note that if $\bar{j}_\alpha = \bar{j}_\beta$ then $\alpha$ and $\beta$ are conflicting schedules.)

*Dispatch(s/s)*: If $\bar{j}_\alpha > 0$ with $s_{\bar{j}_\alpha}(\alpha) \leq t$ and $\bar{j}_\beta > 0$ with $s_{\bar{j}_\beta}(\beta) \leq t$ and $\bar{j}_\alpha \neq \bar{j}_\beta$ then the dispatcher selects the job to start arbitrarily from $\{\bar{j}_\alpha \cup \bar{j}_\beta\}$.

*Wait(w)*: If $\bar{j}_\alpha > 0$ with $s_{\bar{j}_\alpha}(\alpha) > t$ and $\bar{j}_\beta > 0$ with $s_{\bar{j}_\beta}(\beta) > t$ and $\bar{j}_\alpha = \bar{j}_\beta$ then the dispatcher waits until $min\{s_{\bar{j}_\alpha}(\alpha), s_{\bar{j}_\alpha}(\beta)\}$.(Note that if $s_{\bar{j}_\alpha}(\alpha) \neq s_{\bar{j}_\alpha}(\beta)$ then

$\alpha$ and $\beta$ are conflicting schedules.)

*Dispatch(s)*: If $\bar{j}_\alpha > 0$ with $s_{\bar{j}_\alpha}(\alpha) \leq t$ and $\bar{j}_\beta > 0$ with $s_{\bar{j}_\beta}(\beta) \leq t$ and $\bar{j}_\alpha = \bar{j}_\beta$ then the dispatcher selects the job $\bar{j}_\alpha$ to start.(Note that if $s_{\bar{j}_\alpha}(\alpha) \neq s_{\bar{j}_\alpha}(\beta)$ then $\alpha$ and $\beta$ are conflicting schedules.)

Let $\bar{j}$ denote the job selected to be started on the machine. Upon receiving a dispatch trigger at time $t$, the dispatcher runs the following algorithm Dispatch$(t, \alpha, \beta)$ in Figure 3.7 given schedules $\alpha$ and $\beta$ to dispatch.

```
Require $t, \alpha, \beta$ are defined.
Step 1. If $j(t) > 0$ then the machine is busy, EXIT.
Step 2.1. If Dispatch(-/-) then there are no jobs left in schedules $\alpha$ and $\beta$ that can be processed, EXIT.
Step 2.2. If Wait(-/w) then set $t^{\text{wakeup}} = s_{\bar{j}_\beta}(\beta)$ and go to Step 3.
Step 2.3. If Wait(w/-) then set $t^{\text{wakeup}} = s_{\bar{j}_\alpha}(\alpha)$ and go to Step 3.
Step 2.4. If Dispatch(-/s) then set $\bar{j} = \bar{j}_\beta$ and go to Step 4.
Step 2.5. If Dispatch(s/-) then set $\bar{j} = \bar{j}_\alpha$ and go to Step 4.
Step 2.6. If Wait(w/w) then set $t^{\text{wakeup}} = min\{s_{\bar{j}_\alpha}(\alpha), s_{\bar{j}_\beta}(\beta)\}$ and go to Step 3.
Step 2.7. If Dispatch(w/s) then set $\bar{j} = \bar{j}_\beta$ and go to Step 4.
Step 2.8. If Dispatch(s/w) then set $\bar{j} = \bar{j}_\alpha$ and go to Step 4.
Step 2.9. If Dispatch(s/s) then set $\bar{j}$ arbitrarily from $\{\bar{j}_\alpha \cup \bar{j}_\beta\}$ and go to Step 4.
Step 2.10. If Wait(w) then set $t^{\text{wakeup}} = min\{s_{\bar{j}_\alpha}(\alpha), s_{\bar{j}_\beta}(\beta)\}$ and go to Step 3.
Step 2.11. If Dispatch(s) then then $\bar{j} = \bar{j}_\alpha = \bar{j}_\beta$ and go to Step 4.
Step 3. Set a wakeup call at time $t^{\text{wakeup}}$ and EXIT.
Step 4. Start processing $\bar{j}$ on the machine, i.e., set $s^R_{\bar{j}} = t$.
```

Figure 3.7. Dispatch$(t, \alpha, \beta)$ Algorithm.

Note that in Step 1, we stop the dispatch process because preemption is not allowed. Step 2.1 checks if there are jobs to be dispatched in the queue that are included in the current schedules. In Steps 2.2 - 2.5, empty schedules are ignored.

Steps 2.6 - 2.9 handle with the situations when two different jobs are selected. If both jobs have slacks to start, then the machine is left idle until the minimum scheduled start time. If one of the jobs is needed to be started, then it is started immediately. If both of the jobs are needed to be started then selection is done arbitrarily. In Steps 2.10 and 2.11, the two schedules offers the same job. In that case smaller scheduled start time is considered if scheduled start times are different. In Step 3, although there are jobs in the queue, we leave the machine idle. This may be the optimum behavior since we have a non-regular performance measure. We physically start processing a selected job in Step 4.

If a dispatch trigger is received while there is an active wakeup call, the wakeup call is removed.

### 3.2.4. Example Run of the Distributed Scheduling System

The Figure 3.8 demonstrates the generated triggers within the system and activation of processes on a time line (horizontal axis). The first line indicates job arrivals. We show the types of arrivals at the top of the figure. Two types of jobs arrive to the system: A and B. On the arrival line, dashed arrows point the synchronization start times and denoted by $t^{\text{synch}}$. The duration between two $t^{\text{synch}}$ is equal to $\Omega$. Black boxes on this line are the times that it takes to complete a synchronization process, their durations are equal to $\omega$. The second line shows the messages received and sent by schedulers A and B. The third and fourth lines are the schedules that are released by the scheduler A and the scheduler B respectively. Cross symbols are placed on these lines at the times that a new schedule is released. The sequence of corresponding schedule is shown close to the cross symbols. The fifth line represents the dispatcher. The wakeup calls set by the dispatcher are shown in the sixth line, where shaded boxes are the times that the dispatcher is sleeping. The machine line shows the actual processing of jobs and the idle times on the machine.

As it can be seen from Figure 3.8, $\alpha^L_{t_0} = \{1\}$ and $\beta^L_{t_0} = \{2\}$. Apparently $J^S_{t_0} = \{1, 2\}$, $J^A_{t_0} = \{1\}$ and $J^B_{t_0} = \{2\}$. A synchronization process starts at time $t_1$, and it

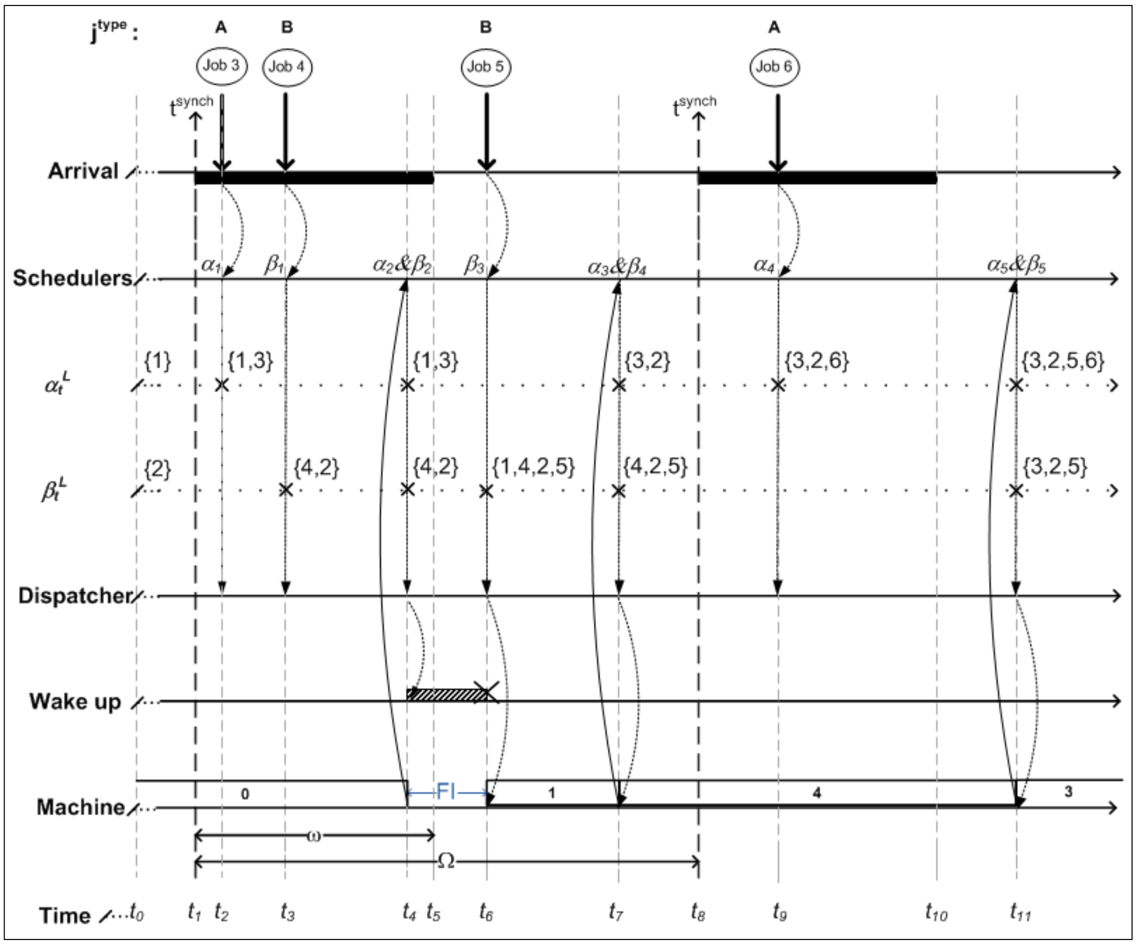


Figure 3.8. Sample Run of Distributed Schedulers.

ends at time $t_5$. At time $t_2$, Job 3 arrives. Since type of Job 3 is A, this arrival triggers the scheduler A, and it releases schedule $\alpha_1$. Note that Job 2 is not included in $\alpha_1$. The scheduler A does not know the information of Job 2 that the scheduler B knows, since the synchronization has not completed yet at the time $t_2$. The dispatcher ignores the dispatch trigger since the machine is already busy at time $t_2$. Similarly,at time $t_3$, type B arrival triggers the scheduler B to release the schedule $\beta_1$ which does not include Job 1. The dispatcher does not dispatch the schedules $\alpha_1$ and $\beta_1$ since the machine is still busy at time $t_3$. At time $t_4$, the machine generates a JobCompletion event, which triggers both of the schedulers. They generate $\alpha_2$ and $\beta_2$ and release them.From Figure 3.8 we deduce that $s_1(\alpha_2) > t_4$ and $s_4(\beta_2) > t_4$. Hence the dispatcher sets a wakeup call, and an FI period starts for the machine. The synchronization process started at time $t_1$ is completed at time $t_5$. At time $t_6$ the arrival of B type Job 5 results the release of the schedule $\beta_3$. Since the synchronization is completed before time $t_6$

all type of jobs that arrived before recent synchronization start time $t_1$ and B type jobs arrived between $t_1$ and the current time $t_6$ are included in $\beta_3$. The completion of schedule generation triggers the dispatcher to dispatch the machine according to $\alpha_2$ and $\beta_3$. Figure 3.8 shows that both schedules offer Job 1 to be selected to dispatch. From the figure we can deduce that $s_1(\beta_3) \leq t_6$ hence the dispatcher stops sleeping and starts Job 1.

When Job 1 is completed at time $t_7$, both schedulers update their released schedules. $J_{t_7}^A$ includes all type of jobs arrived before $t_1$ and type A jobs arrived between $t_1$ and $t_7$ that are not processed yet. Similarly, $J_{t_7}^B$ includes all type of jobs arrived before $t_1$ and type B jobs that arrived between $t_1$ and $t_7$ that are not processed yet. According to $\alpha_3$ and $\beta_4$, the dispatcher starts Job 4 at time $t_7$. At $t_8$ a new synchronization period starts. While synchronizing, the scheduler A receives a trigger due to the arrival of Job 6, but can not access the information of type B arrivals after $t_1$. Thus it produces the schedule $\alpha_4$. At time $t_{10}$ synchronization process is completed, thus at $t_{11}$ both of the schedulers can access the information of the system at time $t_8$.

# 4. EXPERIMENTATION

We control the experimental environment by using two sets of parameters: *scheduling environment* parameters control the design and characteristics of the production system; *control policy* parameters define alternative styles of management.

We simulated the system under various conditions. In total, there are 20 distinct scheduling environment settings, 10 centralized control policy settings and 11 distributed control settings. Hence, for the first phase of the study, we have 200 experiment settings to examine the impact of decision time on the system performance. Similarly, for the second phase of the study, we have 220 experiment settings to examine the impact of synchronization among distributed schedulers.

We obtained results from 10 replications on each experimental setting, and reported the average values. Each replication is run until 1600 jobs are completed. The first 100 completed jobs are excluded from the statistics. Let $N = 1500$ denote the number of jobs over which we collected statistics. Details of parameters tested will be explained in the rest of the chapter.

## 4.1. Scheduling Environment Settings

Scheduling environments with different characteristics are generated using the following parameters.

- Job arrival rate parameter $\lambda$,
- Machine utilization $\rho$,
- Processing time variability parameter $\gamma$
- Urgent job ratio parameter $\phi$, and
- Due date tightness parameter denoted by $\tau$.

Without loss of generality, we set $\lambda$ to a fixed value and report the value for all other time related variables as a multiple of $\lambda$. We generate job arrival times such that $(a_{j+1} - a_j) \sim \mathit{Exponential}(\lambda)$. Let $\mu = \frac{\lambda}{\rho}$. We generate estimated and realized processing times, and due date of job $j$ as follows:

$$\hat{p}_j \sim \mathit{Exponential}(\mu)$$

$$p_j^R \sim \hat{p}_j U(1-\gamma, 1+\gamma)$$

$$d_j = a_j + \hat{p}_j + \kappa \frac{1}{\mu},$$

where

$$L = \frac{\rho}{(1-\rho)}, \quad \text{and}$$

$$\kappa \sim U(\tau \frac{L}{2}, \tau \frac{3L}{2}).$$

Note that, $L$ is the expected queue length for an M/M/1 system, and $\kappa$ is uniformly distributed with mean $\tau L$. and range $\tau \frac{L}{2}$. Hence, a larger value for $\tau$ indicates a looser due date.

To analyze the impact of jobs occasionally arriving to the shop with very tight due dates, we designate some of the arriving jobs as *urgent jobs* and set their $\tau$ values to zero. We control the ratio of urgent jobs using parameter $\phi$, which indicates the probability that an arriving job is indicated as urgent.

Table 4.1 presents the tested values for the scheduling environment parameters in the experiments. Note that, when $\tau = 0$, all jobs are urgent, and parameter $\phi$ has

Table 4.1. Values tested for the scheduling environment parameters.

| | | | |
|---|---|---|---|
| $\boldsymbol{\rho}$ | 0.5 | 0.7 | |
| $\boldsymbol{\gamma}$ | 0 | 1 | |
| $\boldsymbol{\phi}$ | 0 | 0.3 | |
| $\boldsymbol{\tau}$ | 0 | 2 | 4 |

no significance. Hence, there are five valid combinations for $\tau$ and $\phi$ pairs.

## 4.2. Control Policy Settings

### 4.2.1. Centralized Control Policy Settings

We denote a centralized control policy as a triplet $\{x, y, z\}$. Parameter $x$ indicates the response mode of the scheduler that we discussed in Section 3.2.1, and may assume values $I$, $B$ and $A$ to indicate instantaneous, busy and available modes. Parameter $y$ is the length of the decision time as a multiple of $\lambda$. Parameter $z$ represents the scheduling algorithm used by the scheduler. As we introduced in Chapter 1, we want to test the trade-off between using fast - heuristic algorithms and slow - optimal algorithms. Hence, in our tests we used two algorithms to solve Problem $\mathcal{P}^{\mathcal{SP}}(t)$: an optimum algorithm and a heuristic algorithm, denoted by *Opt* and $H$, respectively.

As an example for the notation, triplet $\{B, \frac{1}{4}, Opt\}$ indicates the centralized control policy, where the scheduler operates in the busy mode and running the optimal scheduling algorithm takes $\delta = \frac{1}{4}\lambda$ time units. $\{I, 0, Opt\}$ indicates the centralized control policy, where the scheduler runs the optimal scheduling algorithm instantaneously, so $\{I, 0, Opt\}$ represents the studies in the dynamic scheduling literature which assumes that whole scheduling process is instantaneous. Since the heuristic algorithm is expected to be fast, we assume that running the heuristic algorithm does not take time, and consequently, used only during the instantaneous response mode. Hence, the only centralized control policy using algorithm $H$ is $\{I, 0, H\}$.

Values that we tested for parameter $y$ are $\{\frac{1}{16}, \frac{1}{8}, \frac{1}{4}, \frac{1}{2}\}$. Table **??** summarizes the tested values for the centralized control policy parameters in the experiments.

Table 4.2. Values tested for the centralized control policy parameters.

| $\boldsymbol{x}$ | $I$ | $B$ | $A$ | | | |
|---|---|---|---|---|---|---|
| $\boldsymbol{y}$ | 1/16 | 1/8 | 1/4 | 1/2 | 1 | 2 |
| $\boldsymbol{z}$ | $Opt$ | $H$ | | | | |

In our simulation runs, the optimum solution to Problem $\mathcal{P}^{\mathcal{SP}}(t)$ explained in Section 3.1.1 is obtained by solving a mixed integer linear program by CPLEX solver. [49, 50] study the computational performance of different mixed integer programming formulations for single machine scheduling problem. Based on these studies and our experience, we use the fallowing mixed integer linear program. Let $[j]$ indicate the job assigned to the $j$th position in the sequence.

**Program** $\mathcal{P}^*(t)$ :

$$\min \quad \left(\textstyle\sum_{j\in J_t^S} E_{[j]} + \sum_{j\in J_t^S} T_{[j]}\right) \tag{4.1}$$

s. t.

$$\textstyle\sum_{i\in J_t^S} x_{i[j]} = 1 \qquad j = 1, \ldots, |J_t^S| \tag{4.2}$$

$$\textstyle\sum_{j=1}^{|J_t^S|} x_{i[j]} = 1 \qquad \forall i \in J_t^S \tag{4.3}$$

$$c_{[1]} \geq \textstyle\sum_{i\in J_t^S} x_{i[1]} p_i \tag{4.4}$$

$$c_{[j]} \geq c_{[j-1]} + \textstyle\sum_{i\in J_t^S} x_{i[j]} p_i \qquad j = 2, \ldots, |J_t^S| \tag{4.5}$$

$$c_{[j]} + E_{[j]} - T_{[j]} = \textstyle\sum_{i\in J_t^S} x_{i[j]} d_i \quad j = 1, \ldots, |J_t^S| \tag{4.6}$$

$$E_{[j]}, T_{[j]}, c_{[j]} \geq 0 \qquad j = 1, \ldots, |J_t^S|$$

$$x_{i[j]} \in \{0, 1\} \qquad \forall i \in J_t^S, j = 1, \ldots, |J_t^S|,$$

where $x_{i[j]} = 1$, if job $i$ is assigned to the $j$th position, and $E_{[j]}$ and $T_{[j]}$ are earliness and tardiness values of the job assigned to the $j$th position, respectively.

The heuristic algorithm represents the worst case in terms of performance to isolate the effect of execution speed on the system performance. Therefore, we devised a random algorithm as a heuristic. Let us assume that we run the heuristic algorithm at time $t$. We first generate a random sequence, say $\sigma$, of jobs in the system ($J_t^S$), and use the following linear program to optimally insert idle times [40] to generate schedule $\pi$. Let $\sigma(i)$ indicate the $i$th job in sequence $\sigma$.

**Program $\mathcal{P}^{\mathcal{I}}(\sigma)$ :**

$$\min \quad \textstyle\sum_{j\in J_t^S} E_j + \sum_{j\in J_t^S} T_j \tag{4.7}$$

s. t.

$$s_{\sigma(i)}(\pi) + p_{\sigma(i)} \leq s_{\sigma(i+1)}(\pi) \qquad i = 1, \ldots, |J_t^S| - 1 \tag{4.8}$$

$$s_{\sigma(i)}(\pi) + p_{\sigma(i)} + E_{\sigma(i)} - T_{\sigma(i)} = d_{\sigma(i)} \qquad i = 1, \ldots, |J_t^S| \tag{4.9}$$

$$E_j, T_j, s_j(\pi) \geq 0 \qquad \forall j \in J_t^S \tag{4.10}$$

### 4.2.2. Distributed Control Policy Settings

We denote a centralized control policy as a triplet $\{x, y, z\}$ where parameter $x$ indicates the response mode of the scheduler, parameter $y$ is the length of the decision time as a multiple of $\lambda$, and parameter $z$ represents the scheduling algorithm used by the scheduler. For the current phase, additional to these parameters, we have two more parameters: the synchronization period length $\Omega$ and the duration that a synchronization process takes $\omega$ which are also defined as a multiple of $\lambda$. Hence, we may denote a distributed control policy of the distributed schedulers as a quintet $\{x, y, z, \Omega, \omega\}$.

We want to test the effect of synchronization on the distributed scheduling system performance by testing various $\Omega$, and $\omega$ combinations. Besides, we want to test the trade-off between using fast-distributed asynchronous schedulers and the slow-centralized scheduler. Under the expectation that distributing the scheduling problem among various schedulers improves reactivity, we assume that distributed schedulers run the optimal scheduling algorithm instantaneously. So $x, y$, and $z$ parameters are fixed and equal to $I, 0$, and $Opt$, respectively for the control policies of the distributed schedulers. Hence, instead of using a quintet to denote a control policy of the distributed schedulers, we may use a pair $\{\Omega, \omega\}$ for the convenience of reporting the results.

As an example for the notation, pair $\{10, \frac{1}{2}\}$ indicates the distributed control policy, where the schedulers run the optimal scheduling algorithm instantaneously, the synchronization period length is $\Omega = 10\lambda$ and the synchronization process takes $\omega = \frac{1}{2}\lambda$.

The distributed scheduler A solves Problem $\mathcal{P}^{\mathcal{SPA}}(t)$ and the distributed scheduler B solves Problem $\mathcal{P}^{\mathcal{SPB}}(t)$ (explained in Section 3.2.1) optimally by solving the mixed integer linear program $\mathcal{P}^{*}(t)$ defined in previous section.

Table 4.3 summarizes the tested values for the control policy parameters in the experiments.

Table 4.3. Values tested for the distributed control policy parameters.

| $\boldsymbol{\Omega}$ | 0 | 1 | 10 | 100 | 1000 |
|---|---|---|---|---|---|
| $\boldsymbol{\omega}$ | 0 | 1/2 | 1 | | |

When $\Omega = 0$, parameter $\omega$ has no significance. Distributed control policy $\{0, 0\}$ is tested to make a comparison with the centralized policy $\{I, 0, Opt\}$, and close values obtained provides a validation of simulation. When $\Omega = 1000$, the synchronization period is so long that the distributed scheduler operates almost always asynchronously, thus the effect of parameter $\omega$ is not noticeable, and only $\{1000, 0\}$ is tested. Hence, there are eleven valid combinations for $\Omega$ and $\omega$ pairs.

# 5. RESULTS

The results are discussed based on Figure 5.1 - 5.27. Figure 5.1 - 5.19 are drawn by using the results of the first phase, and examined in the next Section 5.1. Figure 5.20 - 5.27 are drawn by using the results of the second phase, and examined in Section 5.2. In all the figures, the objective values are given as multiples of the job arrival rate ($\lambda$) averaged over all jobs ($N$). Hence, as an example, reported objective value 1.5 means that the actual objective value over $N$ completed jobs is $1.5\lambda \times N$.

We will refer the time period between the completion time of the 100th job and the completion time of the 1600th job (the period that we collect statistics during the simulation) as the *scheduling period* and denote by $\Gamma$.

## 5.1. Effect of Decision Time on Scheduling System Performance

As discussed in Section 3.1.3, during the scheduling period, the machine may be in one of the following states: Processing (P), Forced Idle (FI), Queue Idle (QI) and Decision Idle (DI). In this section, in all Figure 5.1 - 5.19, the values reported in regards to a state of the machine is the ratio of the time that the machine is in the respective state within the scheduling period. So, the reported ratios for the four states always add up to 1.

Figure 5.1 - 5.9 depict the behavior of the objective value under scheduling environment parameter setting for each of the centralized control policy respectively. Figure 5.1 shows the performance of the policy $\{I, 0, Opt\}$, Figure 5.2 - 5.4 display B policies, and Figure 5.7 - 5.9 display A policies.

Figure 5.10 and 5.19 show the objective values and the machine state probabilities of all policies for each scheduling environment, respectively.

### 5.1.1. General Observations

In Figure 5.1, based on the performance of the policy $\{I, 0, Opt\}$, we have the following observations.

- The objective value increases as utilization ($\rho$) increases.
- The objective value increases as processing variability ($\gamma$) increases.
- For a given $\phi$, the objective value increases as due dates become tighter (lower $\tau$ values).
- For a given $\tau$, the objective value increases as urgent job ratio increases (higher $\phi$ values).

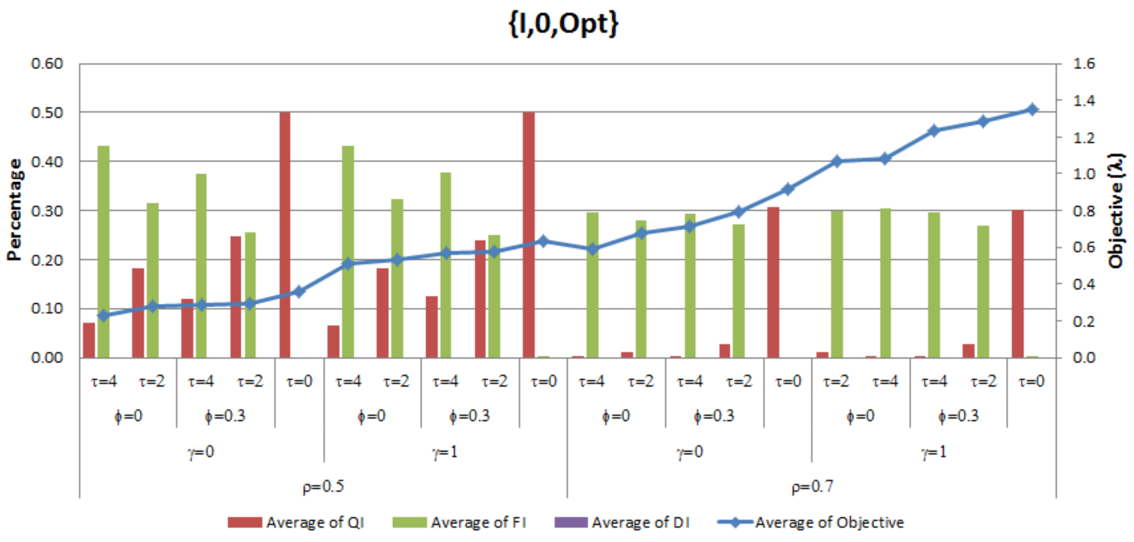


Figure 5.1. Objective Values and Machine State Percentages for $\{I, 0, Opt\}$.

### 5.1.2. Machine States

Figure 5.1 - 5.9 show the ratio of time the machine spends in specific states under various scheduling environment parameters.

First let us consider the base policy, in Figure 5.1. DI does not happen since this policy is in instantaneous mode. When $\tau = 0$, jobs arrive to the system without due date slack. Hence, when $\tau = 0$, FI does not happen, and the machine is idle only due to QI. FI gets larger and QI gets smaller as due dates get looser. Hence, for a given $\phi$, FI increases as $\tau$ increases, similarly for a given $\tau$, FI increases as $\phi$ decreases. But there exist a limit that these ratios may each. As we mentioned in Section 5.1, the ratio for the four P, FI, QI and DI states always add up to 1. We can detect from Figure 5.1 that P ratio is 0.5 and 0.7 when $\rho = 0.5$, and when $\rho = 0.7$ respectively. As due dates get looser, FI may reach up to $(1 - \rho)$.

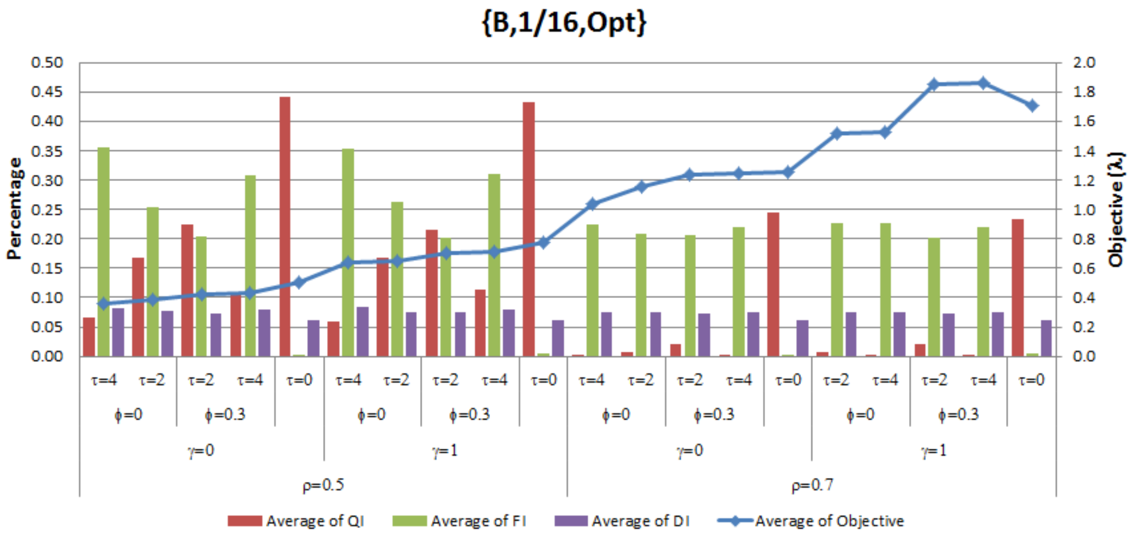


Figure 5.2. Objective Values and Machine State Percentages for$\{B, \frac{1}{16}, Opt\}$.

In Figure 5.2, we observe the behavior of policy $\{B, \frac{1}{16}, Opt\}$. In this figure, DI is almost constant under all scheduling environment settings, and its value is very close to the $\delta$ value. Hence, when $\tau = 0$,the machine is idle due to QI and DI. FI gets larger and QI gets smaller as due dates get looser. P ratio is again close to the $\rho$ value.

We have similar observations for other B policies in Figure 5.3 - 5.4. DI is almost constant under all scheduling environment settings, and DI value is determined by decision time parameter under B policies. As due dates get looser, FI gets larger and QI gets smaller. Since DI is almost constant, FI may reach maximum $((1 - DI) - \rho)$.

Figure 5.5 - 5.9 show the behavior of A policies respectively. For A policies, DI

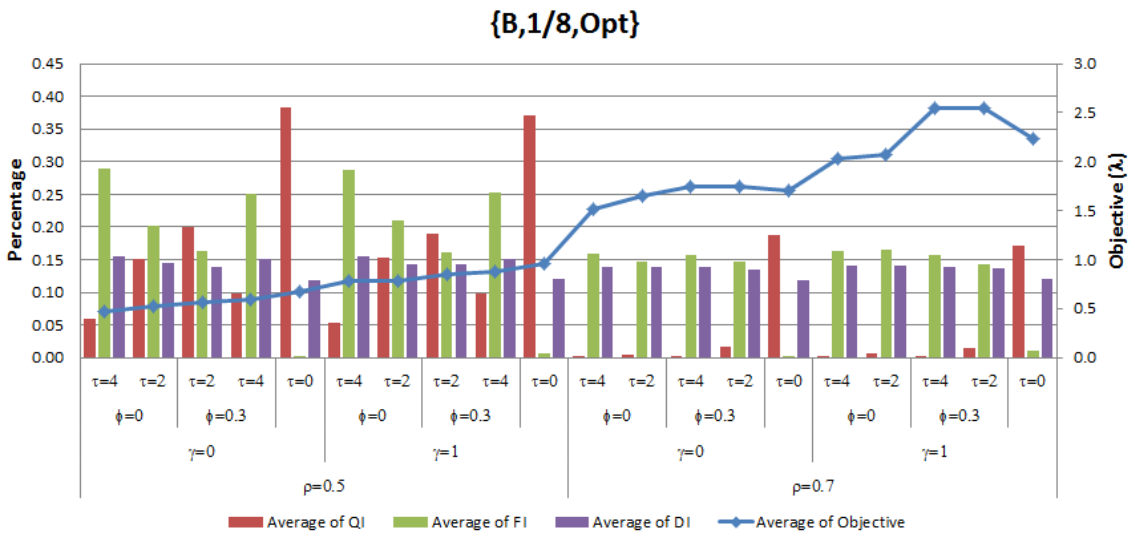


Figure 5.3. Objective Values and Machine State Percentages for$\{B, \frac{1}{8}, Opt\}$.

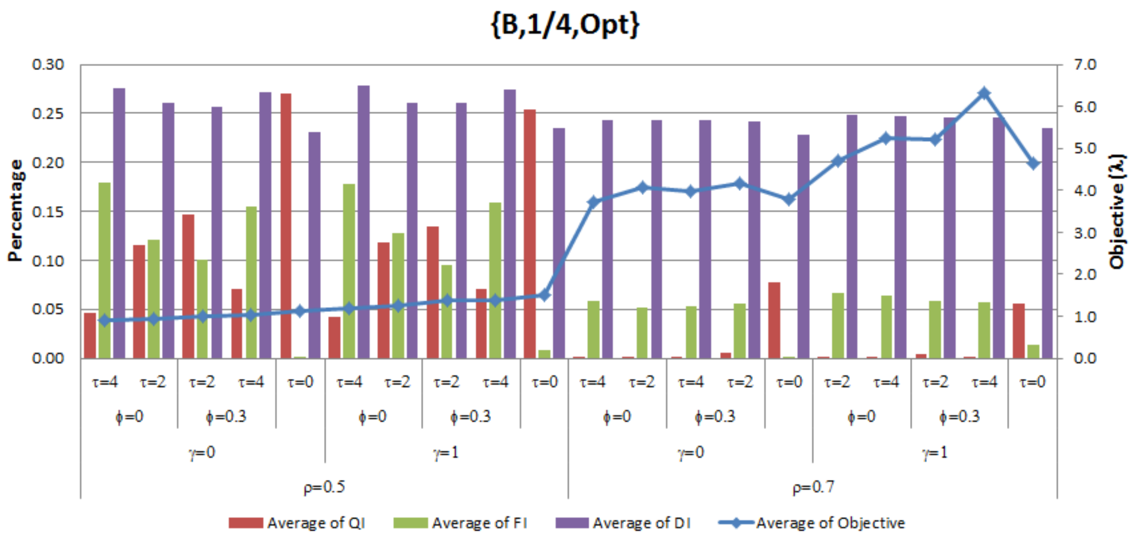


Figure 5.4. Objective Values and Machine State Percentages for$\{B, \frac{1}{4}, Opt\}$.

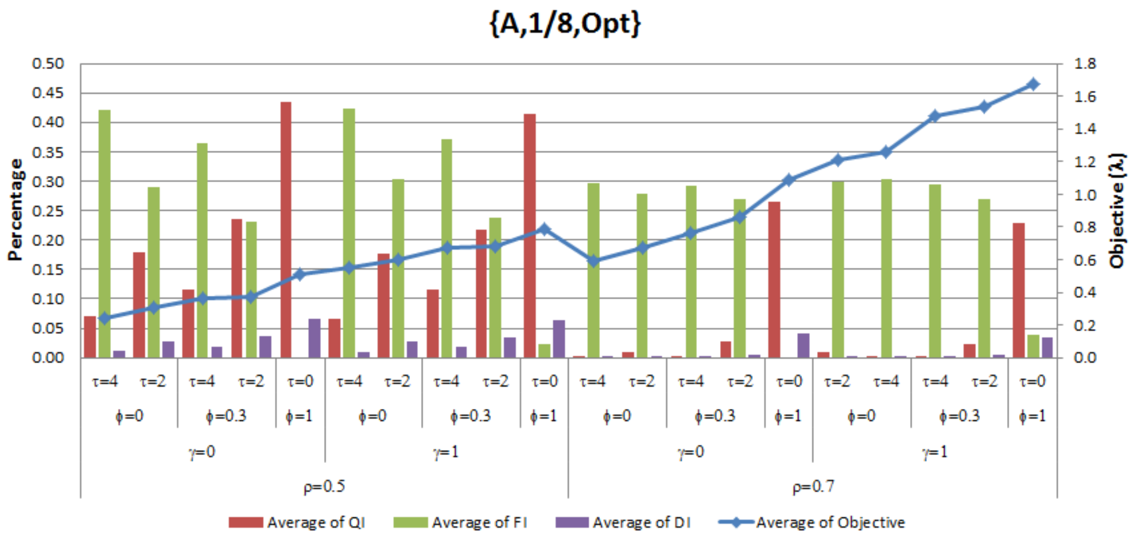


Figure 5.5. Objective Values and Machine State Percentages for$\{A, \frac{1}{8}, Opt\}$.

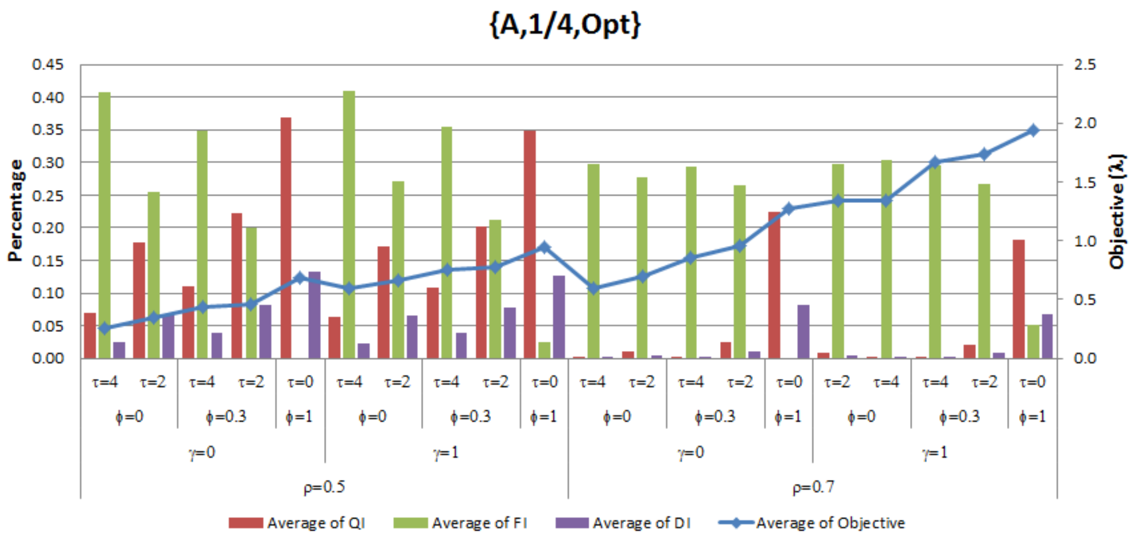


Figure 5.6. Objective Values and Machine State Percentages for$\{A, \frac{1}{4}, Opt\}$.

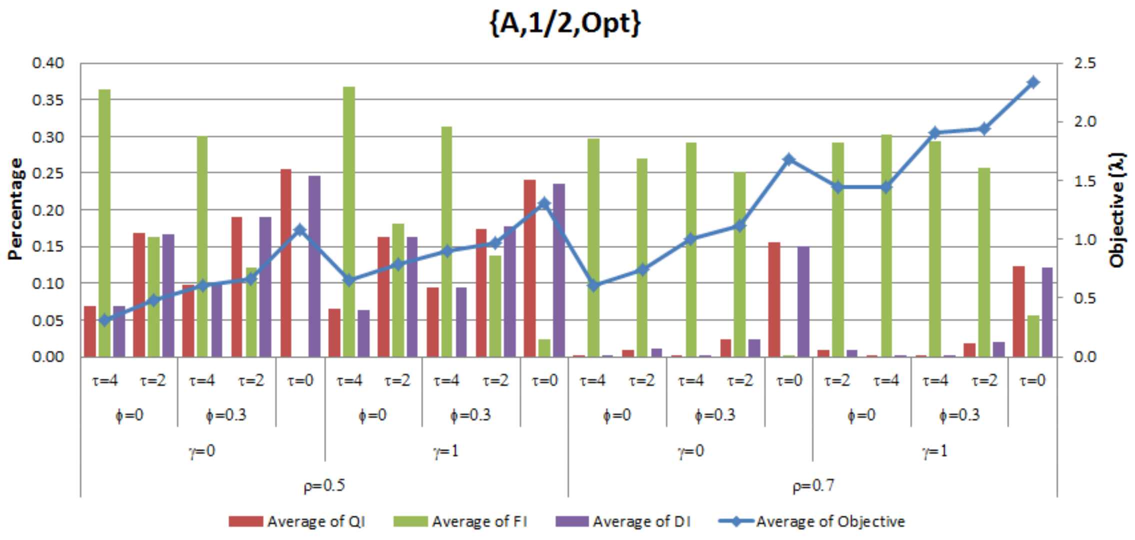


Figure 5.7. Objective Values and Machine State Percentages for$\{A, \frac{1}{2}, Opt\}$.

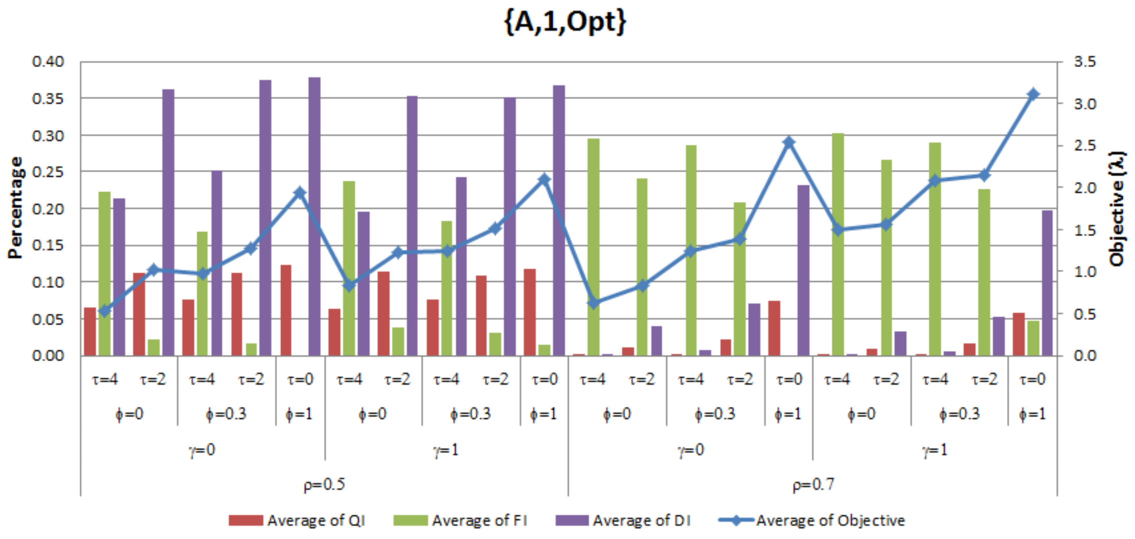


Figure 5.8. Objective Values and Machine State Percentages for$\{A, 1, Opt\}$.

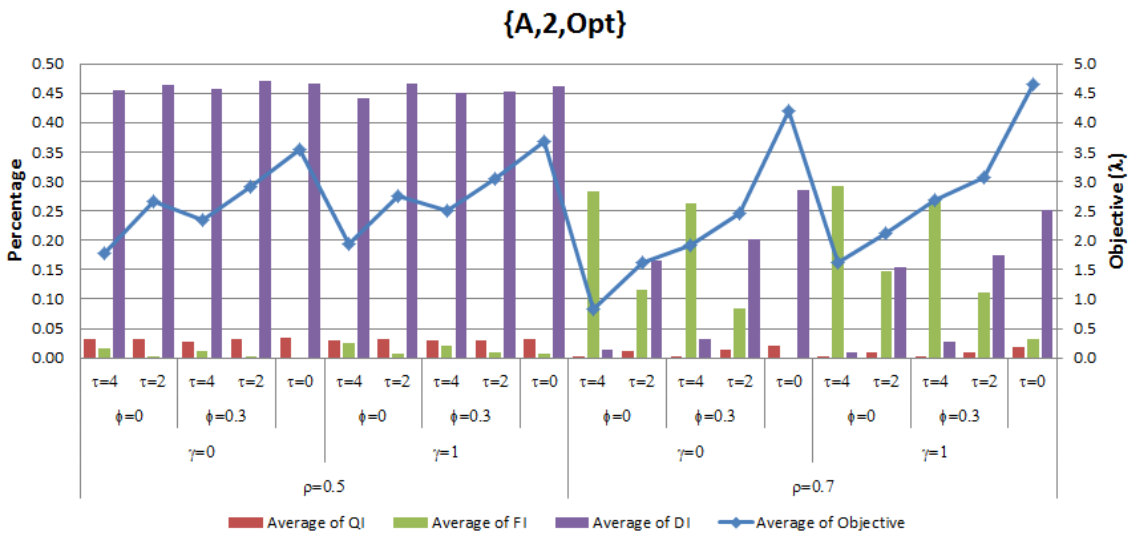


Figure 5.9. Objective Values and Machine State Percentages for$\{A, 2, Opt\}$.

happens when there are jobs in the system but, they cannot be dispatched because of the decision process. DI happens when the the queue size are small. Note that, for a constant $\phi$, the queue size gets larger with larger $\tau$ values (looser due dates). Similarly, for a constant $\tau$, due date gets looser and the queue size gets larger as $\phi$ gets smaller. The queue size gets larger as $\rho$ increase. DI increases as utilization decrease and DI decrease with looser due dates.

We summarize the results of this section as follows.

5.1.2.1. State P. The ratio of the processing state is mainly determined by the value of the utilization ($\rho$) parameter. For a given value of $\rho$, processing state ratio is almost equal to $\rho$, regardless the specific values of other parameters.

5.1.2.2. FI and QI States. Since our objective function includes minimization of earliness, jobs are forced to be delayed until their due dates, which is the reason for FI to occur. Hence, for a given $\rho$, as due dates get looser (larger $\tau$ values), FI gets larger. Since jobs are kept in the system longer, number of jobs waiting in the queue increases and the probability that the machine finds an empty queue and stays idle (QI) decreases. Specifically, for $\tau = 0$, FI is zero, and as $\tau$ increases FI gets larger and QI gets very close to zero.

As $\rho$ gets larger values, the same trend can be observed for FI and QI, while the absolute values get smalller (note that FI + QI $\leq 1 - \rho$).

5.1.2.3. DI State. In the *available mode*, the relation of DI to varying $\tau$ values is similar to the behavior of QI. DI happens only when there are jobs in the system that are ready to be processed, but due to the decision process, the scheduler can not dispatch them to the machine and the machine is kept idle. Hence, as the queue size gets larger with larger $\tau$ values, probability of keeping the machine idle during the scheduling process (DI) gets smaller.

In the *busy mode*, DI is not a function of the state of the queue. The machine is always kept idle during the decision process even if there are jobs waiting in the queue. Hence Figure 5.2 - 5.4 show that, depending on the $\delta$ value, almost a constant percentage of the machine time is spent in the DI state.

As a general result we can state that, for a given $\delta$ value, DI percentage under policy B is always higher then under policy A.

### 5.1.3. Impact of Decision Time

Figure 5.10 - 5.19 show the objective values and the machine state probabilities of all policies for each scheduling environment, respectively. In the horizontal axis, the policies are sorted in the non-decreasing order of their average objective values. In this section results are discussed based on Figure 5.10 - 5.19.

Based on Figure 5.10 - 5.19, we can drive the following general observations.

- $\{I, 0, Opt\}$ always performs better than A and B.

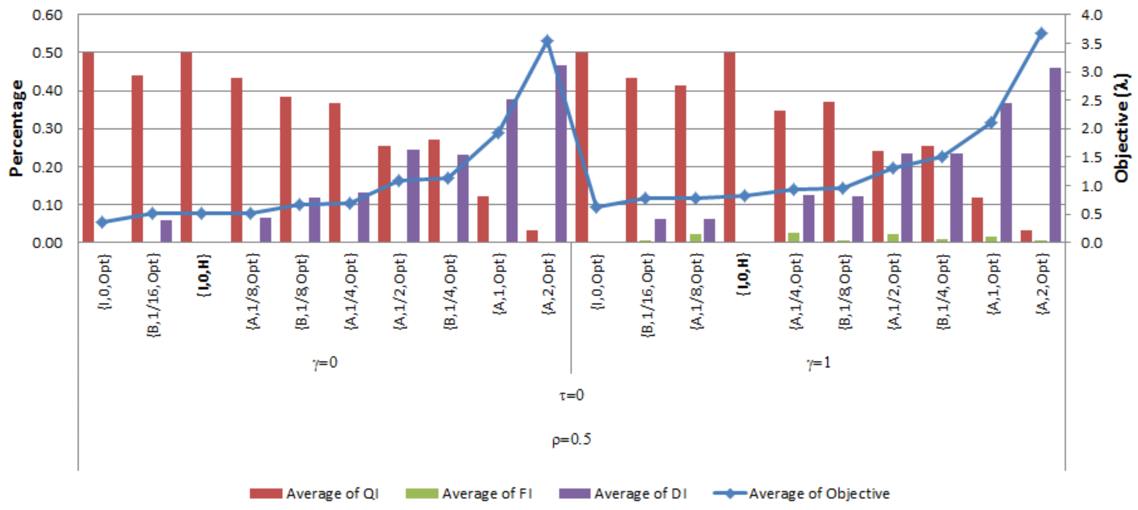


Figure 5.10. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.5, \tau = 0$.

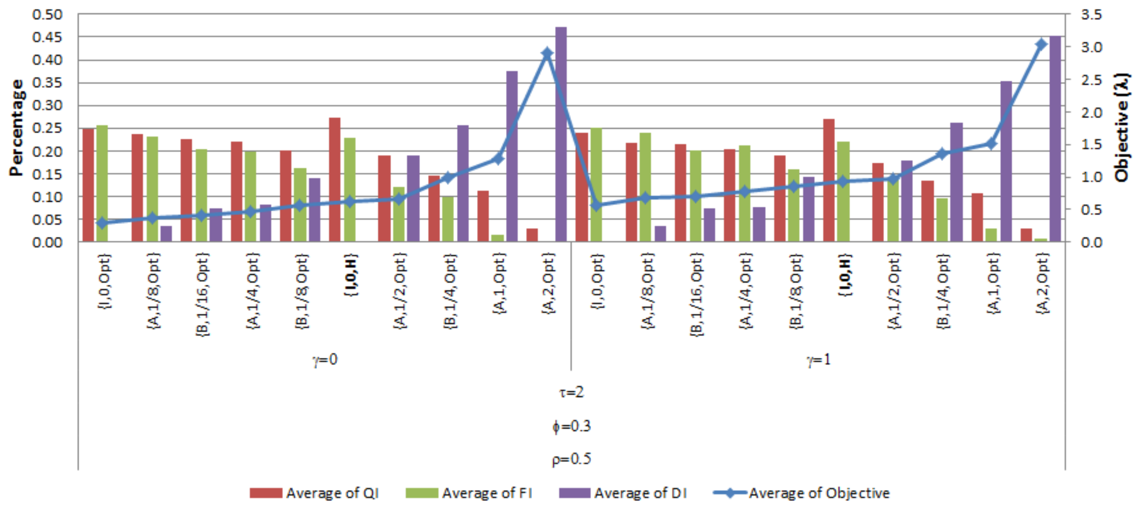


Figure 5.11. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.5, \tau = 2, \phi = 0.3$.

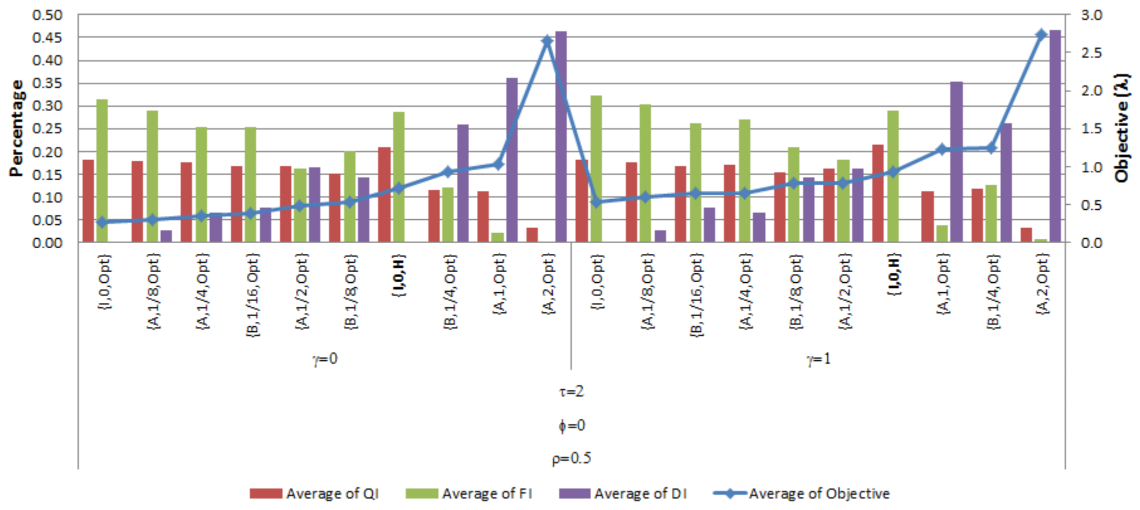


Figure 5.12. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.5, \tau = 2, \phi = 0$.

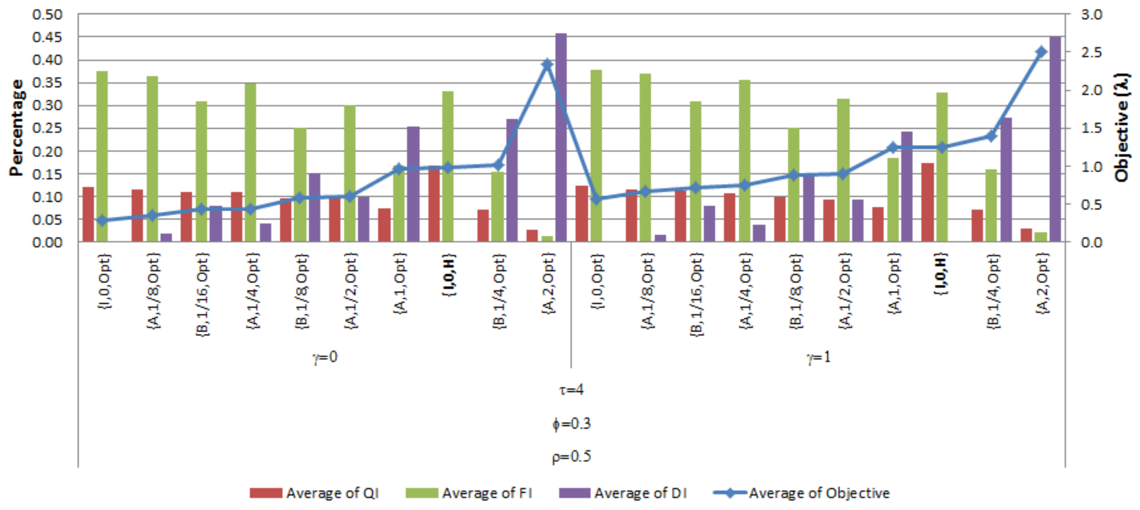


Figure 5.13. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.5, \tau = 4, \phi = 0.3$.

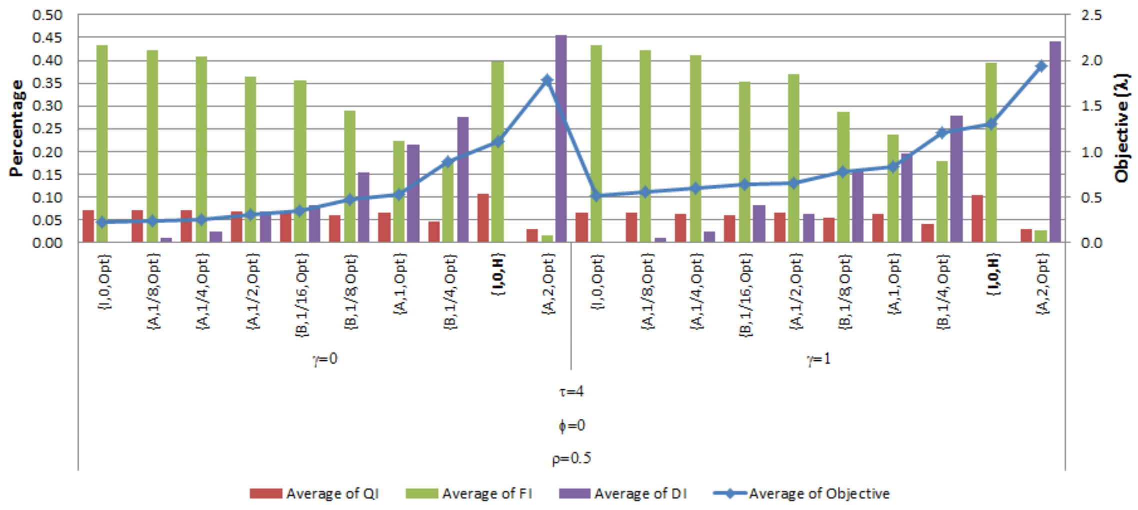


Figure 5.14. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.5, \tau = 4, \phi = 0$.

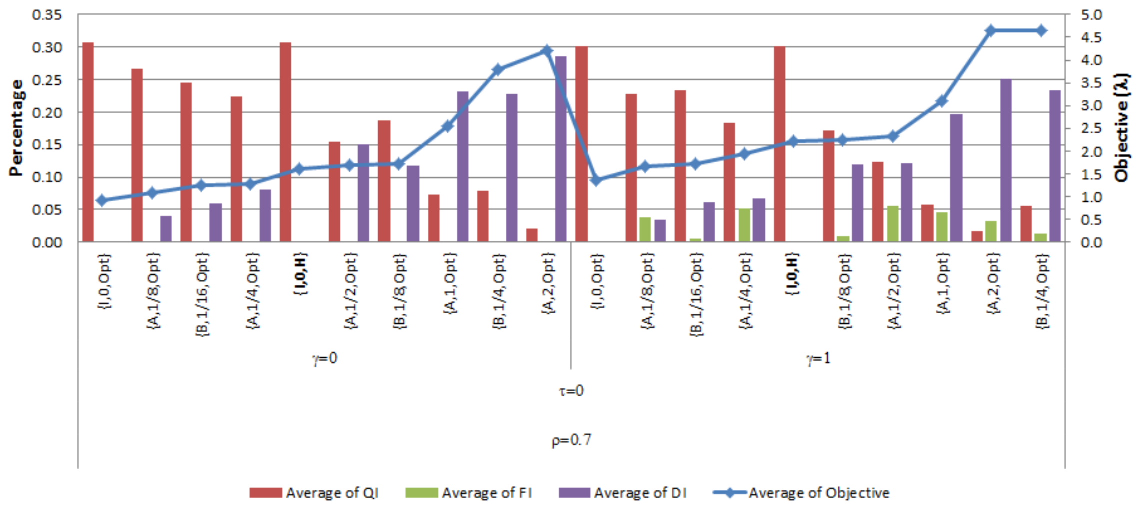


Figure 5.15. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.7, \tau = 0$.

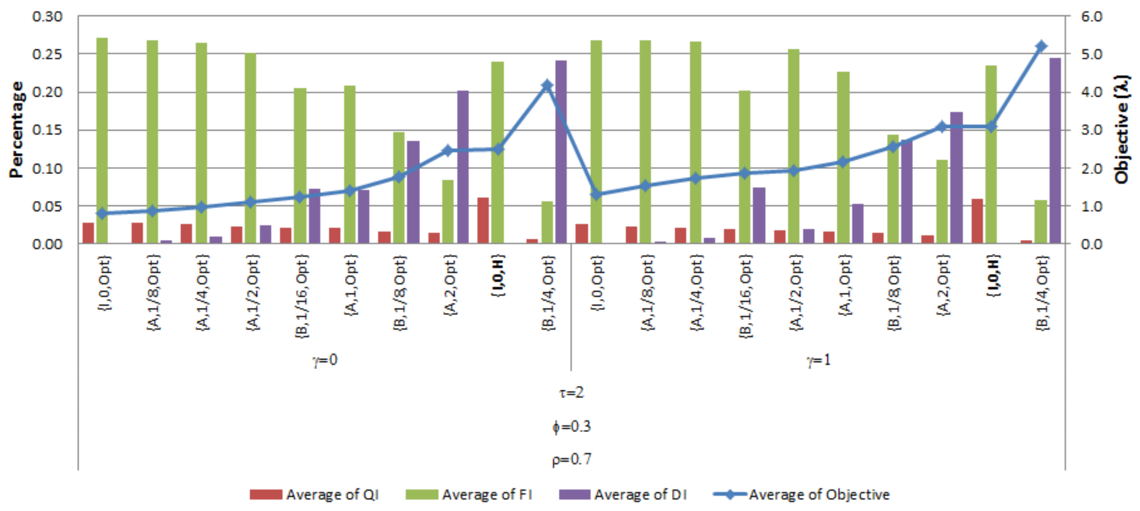


Figure 5.16. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.7, \tau = 2, \phi = 0.3$.

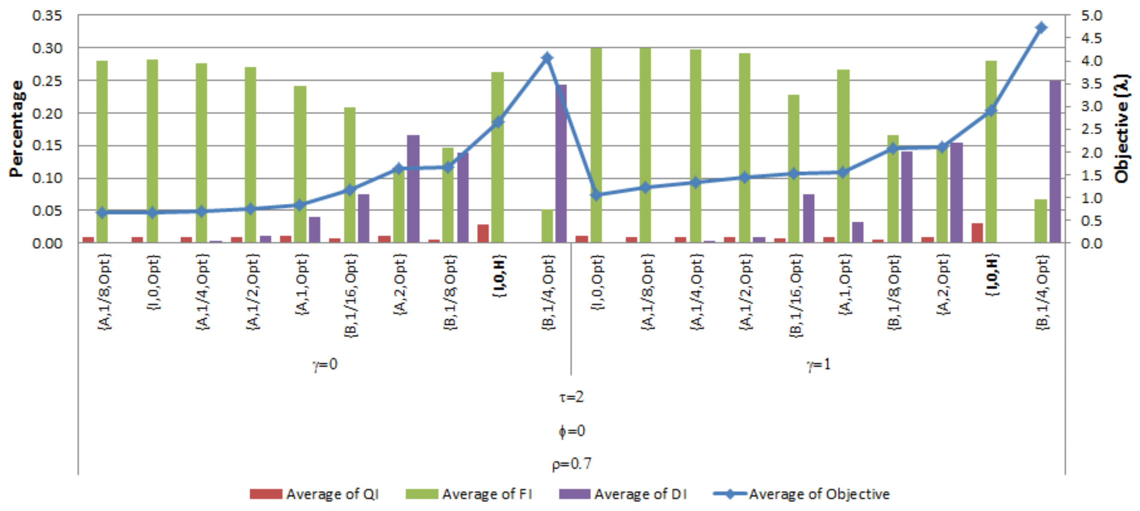


Figure 5.17. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.7, \tau = 2, \phi = 0$.

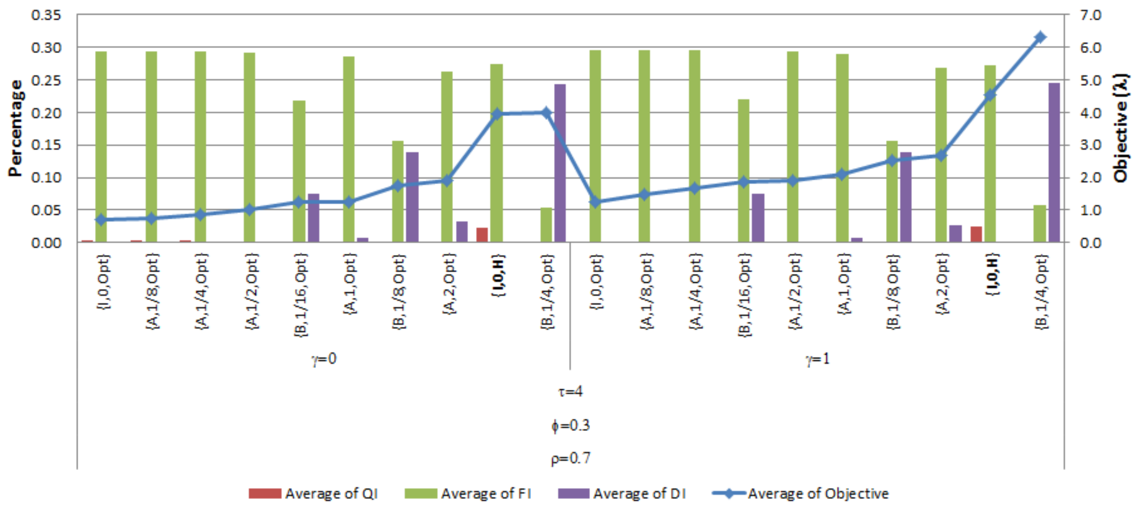


Figure 5.18. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.7, \tau = 4, \phi = 0.3$.

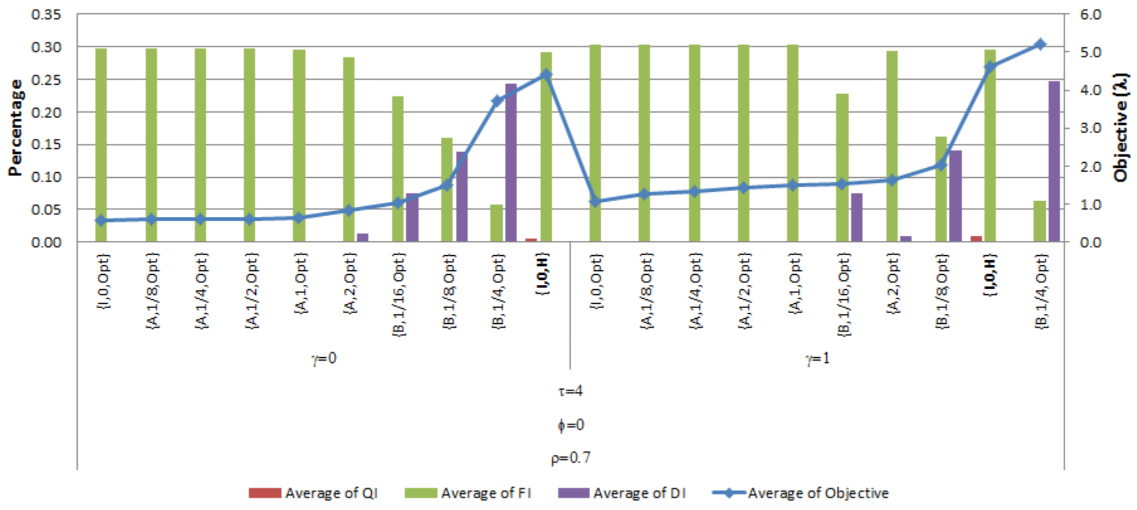


Figure 5.19. Objective Values and Machine State Percentages of All Centralized Policies for $\rho = 0.7, \tau = 4, \phi = 0$.

- For a given $\delta$, A always performs better than B.
- For a given operating mode and given $\gamma, \phi, \tau$ and $\rho$ values, as $\delta$ gets higher, performance of the system degrades.
- Performance of the system under $\{I, 0, H\}$, in comparison to A and B, depends on specific system parameters, and will be discussed in detail in Section 5.1.4.

As a general observation, we state that as $\delta$ gets larger, the system performance deteriorates. In this section, we will analyze the specific reason for this performance loss. Decision time affects the performance of the system in two different ways.

- *Inserted idle time.* During the time period that the scheduler is making a decision, the machine may be kept idle even though there are jobs that may be processed (which appears as DI). These inserted idle times contribute to increased completion times and a higher objective value.
- *Dispatching a wrong job to the machine.* The scheduler generates a schedule based on the state of the system at the start time of the decision process. Since decision takes time, by the time the generated schedule is released, the state of the system probably changes (e.g. new jobs with higher priority might have arrived) and the schedule does not reflect the current state. Hence, it is possible that a wrong job is dispatched, which eventually results with a higher objective value.

In Figure 5.10 - 5.19, we can observe by considering A policies alone that DI is increased with increased objective value as decision times are increased. The same situation is also true for B policies. As a result, increased DI (and hence, increased inserted idle time) results with increased objective value.

To show the impact of dispatching the wrong job, consider Figure 5.10 - 5.19 by comparing A policies with B policies. We can observe that in Figure 5.13, 5.16 and 5.18, there are B policies with higher DI values but they do not perform worse than some A policies with longer decision times. More specifically in Figure 5.13, $\{B, \frac{1}{8}, Opt\}$ has larger DI than $\{A, \frac{1}{2}, Opt\}$ but the performance of $\{B, \frac{1}{8}, Opt\}$ is no worse than $\{A, \frac{1}{2}, Opt\}$. Similarly, in Figure 5.16 by comparing $\{B, \frac{1}{16}, Opt\}$ with $\{A, 1, Opt\}$ and in Figure 5.18 by comparing $\{B, \frac{1}{8}, Opt\}$ with $\{A, 2, Opt\}$ we can obtain the same result. We can explain the difference as the impact of dispatching the wrong job to the machine as a result of using older information.

### 5.1.4. Responsiveness Versus Optimality

The trade off that we would like to emphasize in this section is between the responsiveness of the decision process and the quality of the decisions. To show this trade off, we compare the results for the base policy $\{I, 0, H\}$ with A and B policies with a non-zero decision time. Note that $\{I, 0, H\}$ reflects the scenario where the scheduler is highly responsive, i.e., it does not spend any time to make decisions, but the quality of decisions are very low. Contrary to this, under policies A and B, the scheduler spends time for decision making and this improves the quality of the decisions. Hence, under policies A and B, the scheduler solves the static scheduling problem at each decision epoch optimally.

When we examine the Figure 5.10 - 5.19, we can see that, when $\delta$ is small, the quality of the solutions obtained compensates the performance lost during the decision process. However, for large $\delta$ values, time lost during the decision process is so high that being responsive (no matter how inferior the decision process is) results in a better system performance. For example, in Figure 5.13, under, $\rho = 0.5$, $\phi = 0.3$ and $\tau = 2$, $\{A, \frac{1}{8}, Opt\}$ outperforms $\{I, 0, H\}$, but if we increase the $\delta$ value from $\frac{1}{8}$ to 1 or higher values, the impact of additional time lost becomes significant, and optimal scheduling under A policies yield inferior system performance compared to the random dispatching of jobs to the machine.

Even though control policy B seems to be quite restrictive and unintuitive, in some scheduling environments and for some $\delta$ values, performance of the system is better under B compared to {I, 0, H} and A. As an example, in Figure 5.13, $\{B, \frac{1}{8}, Opt\}$ performs better than all A policies with $\delta$ larger than $\frac{1}{2}$.

Another interesting observation based on Figure 5.10 - 5.19 is that the relative rank of policy $\{I, 0, H\}$ among optimization based policies increases as we increase the value of $\rho$ or decrease the value of $\phi$ for a given $\tau$ or increase the value of $\tau$ for a given $\phi$, i.e., optimization based policies perform relatively better for larger utilization and looser due date values. This can be explained by analyzing the *value of scheduling.* For example, Figure 5.10 and 5.15 depict the case under $\tau = 0$ where the due dates are so tight that all the jobs are almost tardy when they arrive to the system. Especially in low utilization environments (Figure 5.10 represents small $\rho$), queue lengths are very small, hence there is no value that can be obtained by better scheduling, and responsive dispatching performs favorably. Higher utilization and moderately loose due dates generates local problems with significant queues, and the probability of dispatching a wrong job to the machine increases considerably. Hence, the impact of better scheduling can be observed.

A similar analysis applies for the comparison of A and B policies. As $\rho$ and $\tau$ get larger, policy A becomes more effective with larger $\delta$ values compared to B policies with smaller $\delta$ values. Under A policies, ability to dispatch old schedules during the decision process does not add too much value if the utilization is low and queue lengths are small.

Note that, in our problem increasing the $\tau$ value does not necessarily generate a more relaxed environment in terms of the scheduling problem, since we have a non-regular performance measure. To minimize earliness, we keep the job in the queue until its due date becomes tight (which increases FI), and this increases the average queue lengths.

Based on these observations we can conclude that, for each dynamic system, there exists an upper bound on the time that you can spend on the optimization process for the optimization effort to yield better system performance compared to making random decisions in a responsive manner.

## 5.2. Effect of Synchronization on Distributed Scheduling System Performance

Remember that, we refer the time period between the completion time of the 100th job and the completion time of the 1600th job (the period that we collect statistics during the simulation) as the scheduling period. During the scheduling period, the dispatcher decides to dispatch the machine 1500 times. As discussed in Section 3.2.3, the type of the dispatch decision may be as one of the followings: (-/s), (s/-), (w/s), (s/w), (s,s) and (s). While collecting statistics, we count the number of dispatch decisions according to dispatch type then we divide these numbers by 1500 to obtain dispatch decision percentages. Figure 5.20 and 5.21 show the objective value and the dispatch decision percentages of all distributed policies for $\rho = 0.5$ and $\rho = 0.7$, respectively.

Since we have a non-regular performance measure, the dispatcher decides sometimes to keep the machine idle and wait a duration of time. In Section 3.2.3 we define the types of wait decisions as (w), (-/w), (w/-) and (w/w). While collecting statistics, we count the number of all types of wait decisions then we calculate the wait decision percentages. Figure 5.22 and 5.23 show the objective value and the wait decision percentages of all distributed policies for $\rho = 0.5$ and $\rho = 0.7$, respectively.

The percentage of (s) is an indicator of synchronization because both schedulers suggest to start the same job. (s/-) and (-/s) reflect the times that one scheduler suggest to start a job while the other scheduler is not aware of that job and any other job. Hence (s/-) and (-/s) decisions in the distributed scheduling system reflect the situations that if it were a centralized scheduling system, it also suggests to dispatch the same job, and synchronization is inoperative. When the percentage of (s/s) and (s/w) and (w/s) decisions increases the probability to dispatch the wrong job to the machine increases and system performance may deteriorate.

With a similar discussion, the percentage of (w) is an indicator of synchronization since both schedulers suggest to wait for the same job. (w/-) and (-/w) reflect the times that distributed scheduling system behaves like a centralized scheduling system. When the percentage of (w/w) decisions increases, the system performance may deteriorate because of keeping machine idle instead of dispatching a job.

### 5.2.1. Dispatch Decision Percentages

In Figure 5.20-5.23 the policies are sorted primarily in increasing order of $\Omega$ values then secondarily in increasing order of $\omega$ values in the horizontal axis. Note that the policy {0,0} reflects synchronous operation of the distributed schedulers.

Based on Figure 5.20- 5.23, we can drive the following observations.

- The percentage of (s) decisions decreases and the percentage of (s/-), (-/s), (s/w), (w/s) and (s/s) decisions increases as $\Omega$ increases.
- The percentage of (w) decisions decreases and the percentage of (w/-), (-/w) and (w/w) decisions increases as $\Omega$ increases.
- The objective value increases at a value and then it stays constant as synchronization period length ($\Omega$) increases. Hence, for a given $\rho$, $\tau$, $\phi$ and $\gamma$, there is a $\Omega$ value that separates the local problems of two schedulers and induce asynchronous operation.
- The operation of the synchronization time ($\omega$) is similar to the operation of the synchronization period length ($\Omega$). When both of these parameters take comparable values, the effect of $\omega$ may be observable. The objective value slightly increases as $\omega$ increases when $\Omega = 1$. The effect of $\omega$ is not significant when $\Omega$ is high.
- For a given a distributed policy and given $\rho$, $\tau$, $\phi$ values, the objective value increases as processing variability ($\gamma$) increases while dispatch decision percentages are similar in Figure 5.20 - 5.21 and wait percentages are similar in Figure 5.22 - 5.23. Hence, deterioration in objective value is due to the uncertainty in processing times but not due to synchronization.

We deduce from Figure 5.20-5.23 that the effect of $\omega$ is not significant, so we can

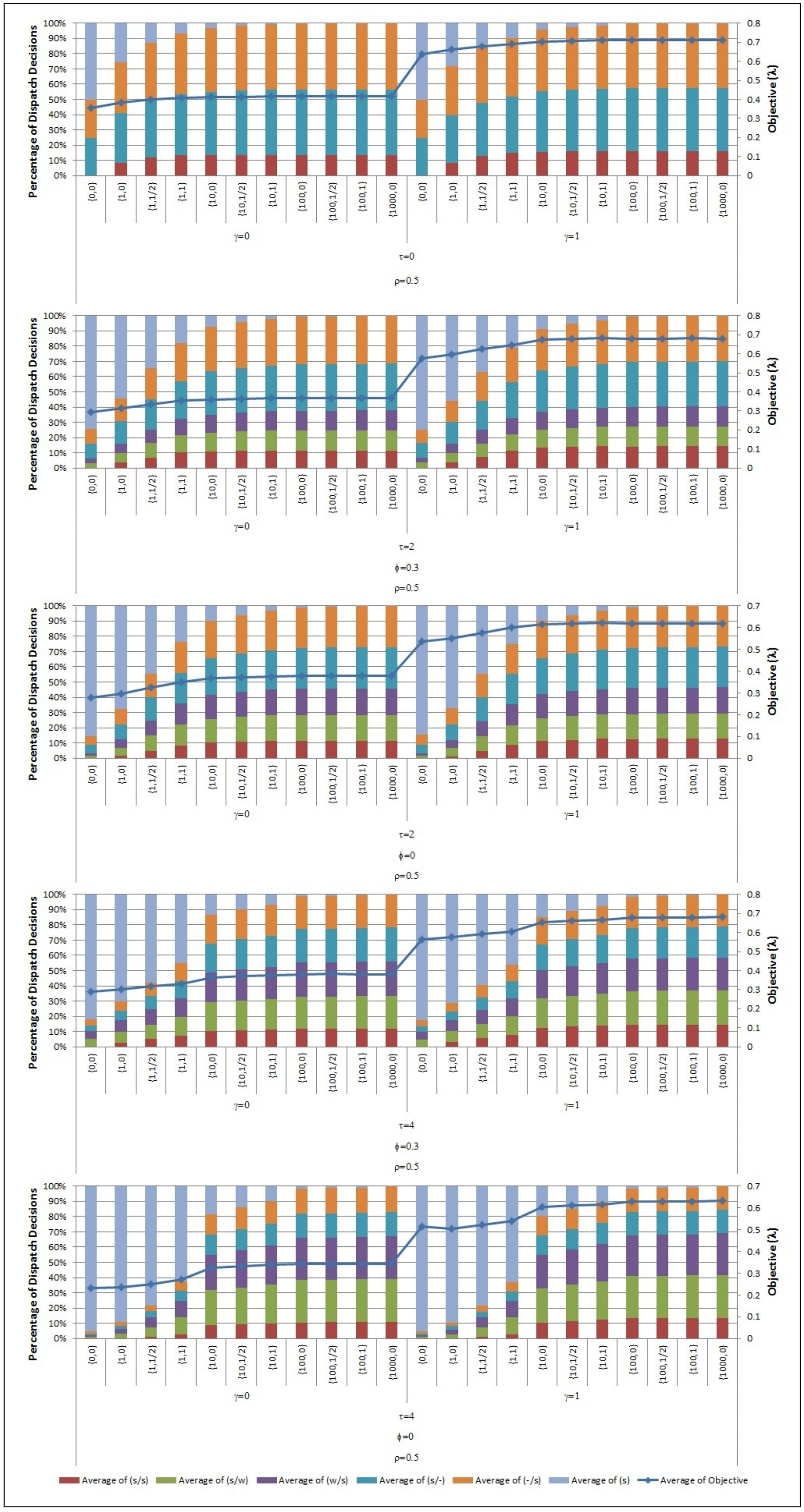


Figure 5.20. Objective Values and Dispatch Decision Percentages of All Distributed Policies for $\rho = 0.5$.

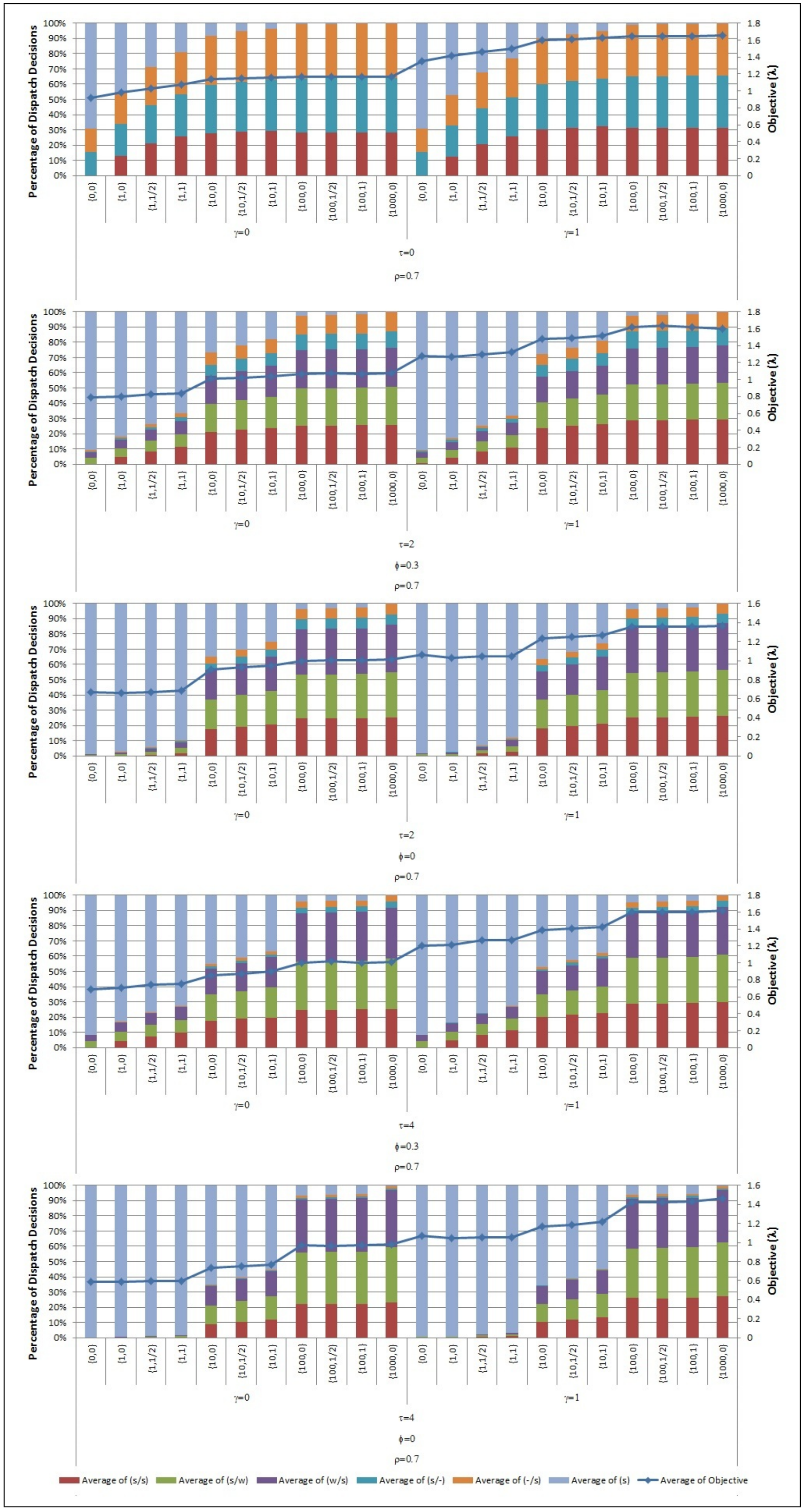


Figure 5.21. Objective Values and Dispatch Decision Percentages of All Distributed Policies for $\rho = 0.7$.

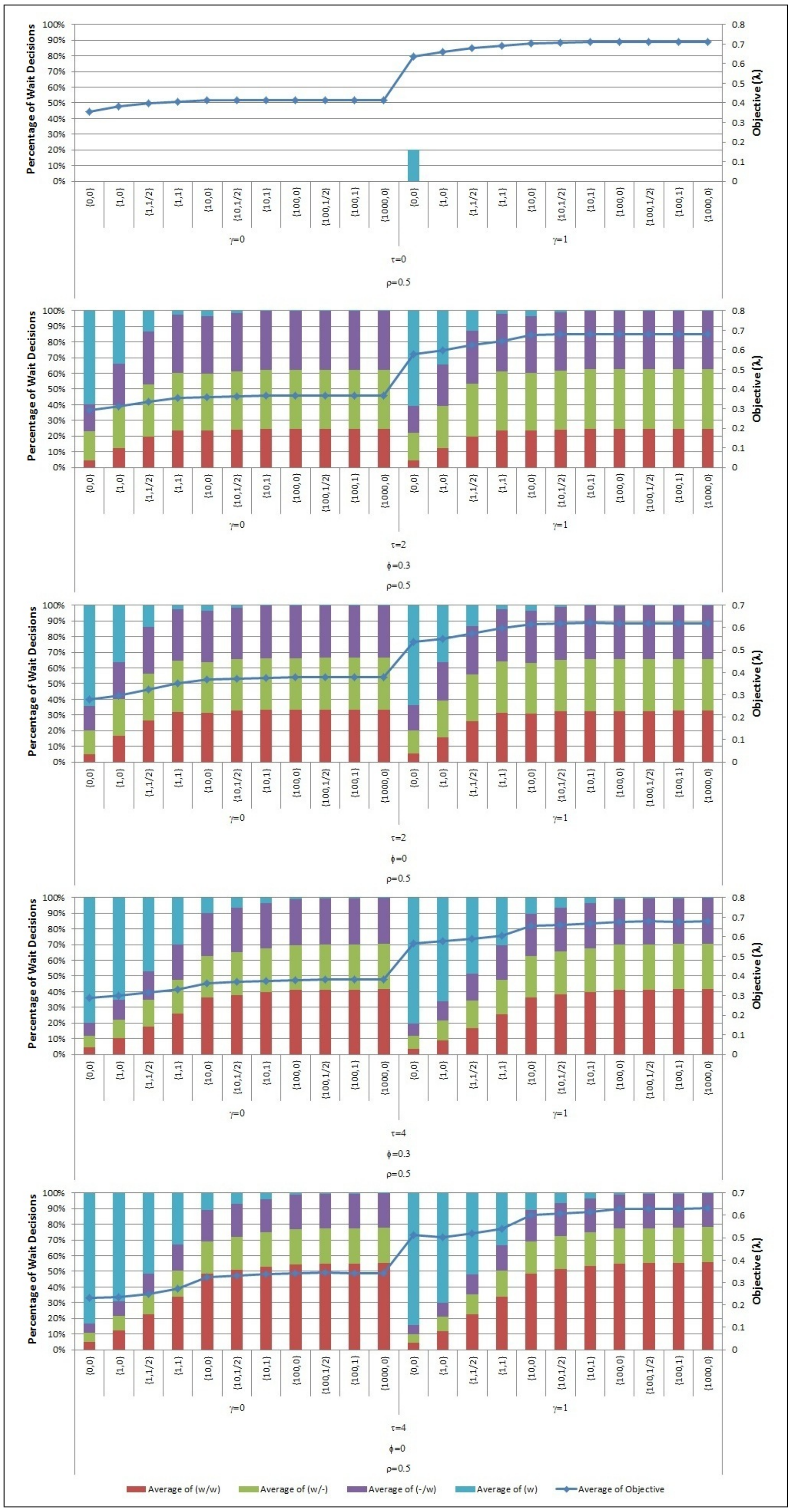


Figure 5.22. Objective Values and Wait Decision Percentages of All Distributed Policies for $\rho = 0.5$.

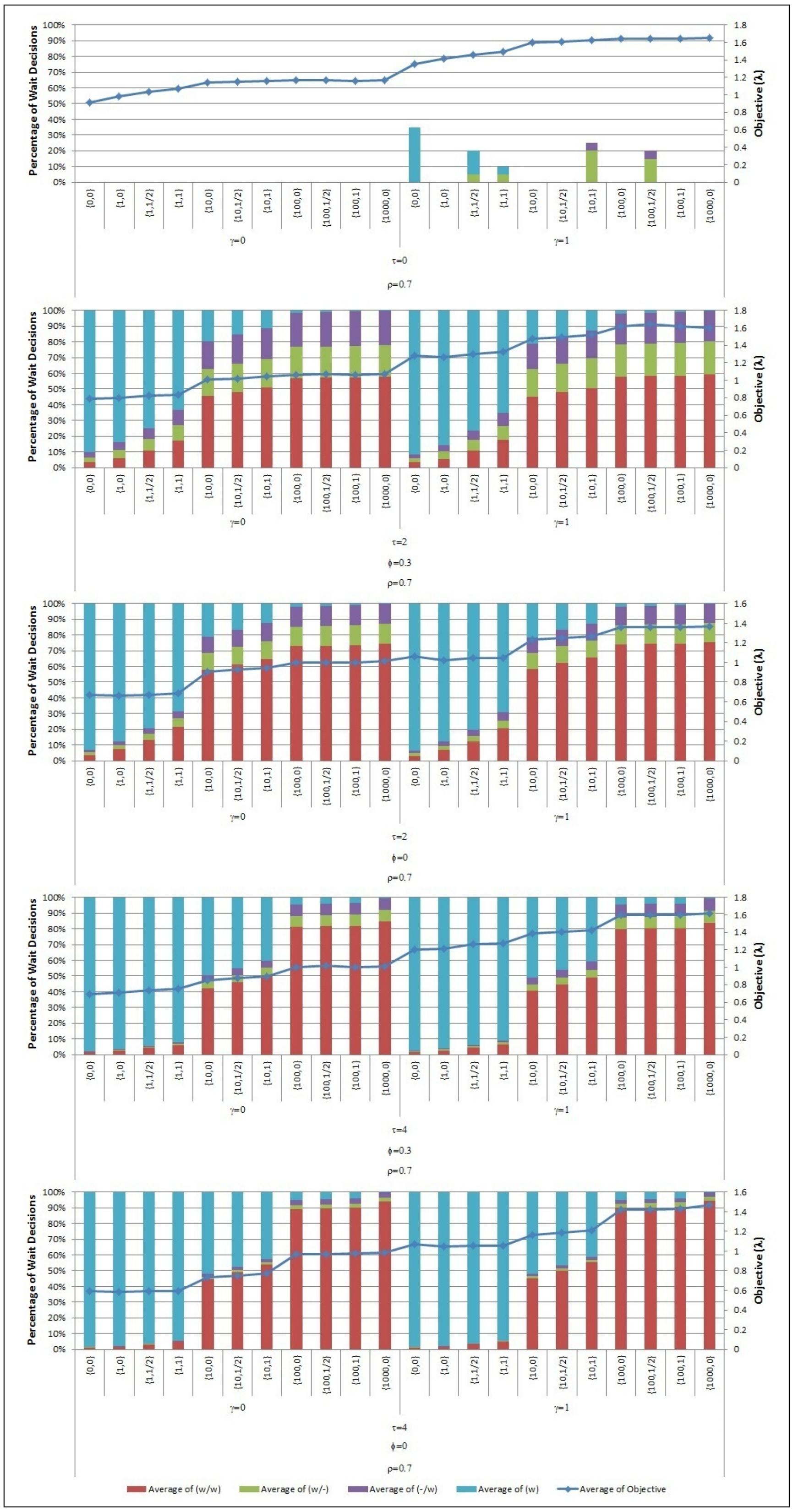


Figure 5.23. Objective Values and Wait Decision Percentages of All Distributed Policies for $\rho = 0.7$.

consider only distributed policies with $\omega = 0$. Processing variability $\gamma$ does not effect the dispatch decision percentages and wait decision percentages, so scheduling environments with processing variability are ignored when investigating decisions. Figure 5.24 provides a simplified version of Figure 5.20 and 5.21 where similar dispatch decisions are presented together. Figure 5.25 provides a simplified version of Figure 5.22 and 5.23 where similar wait decisions are presented together. Based on Figure 5.24 and 5.25 we can drive the following observations.

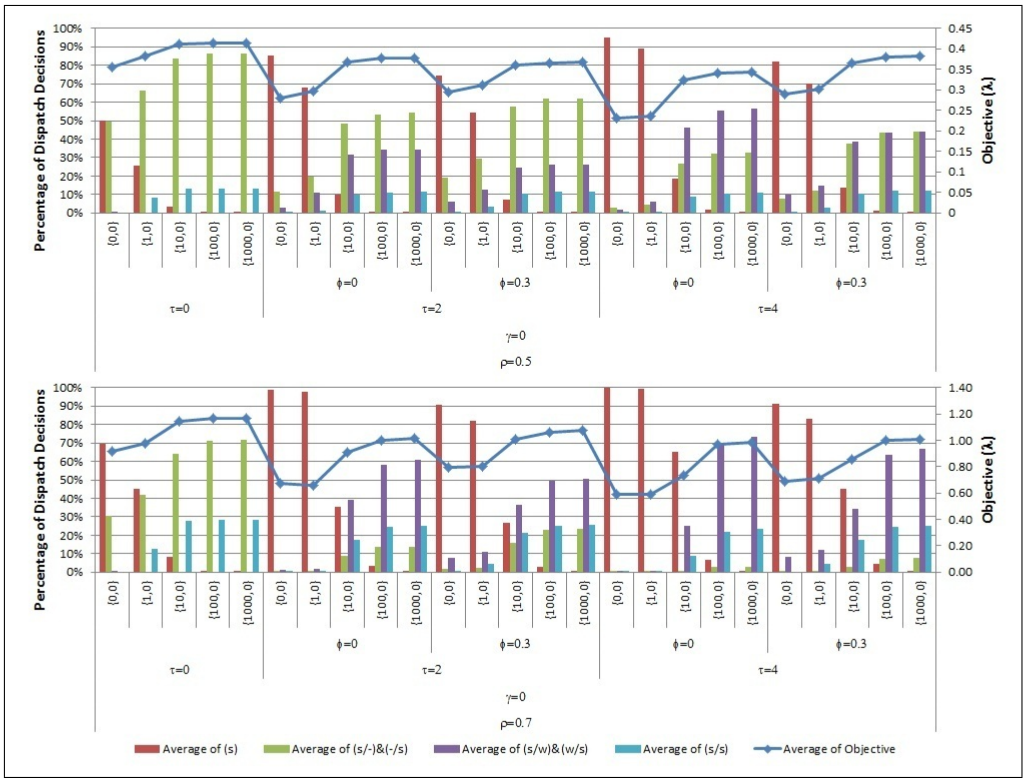


Figure 5.24. Objective Values and Dispatch Decision Percentages of Distributed Policies with $\omega = 0$ under $\gamma = 0$.

- For a given $\phi$, (s/-) and (-/s) percentages increases as due dates become tighter (lower $\tau$ values).
- For a given $\tau$, (s/-) and (-/s) percentages increases as urgent job ratio increases (higher $\phi$ values).
- For a given $\phi$, (w/-) and (-/w) percentages increases as due dates become tighter (lower $\tau$ values). But when $\tau = 0$, the system never decide to wait.
- For a given $\tau$, (w/-) and (-/w) percentages increases as urgent job ratio increases (higher $\phi$ values).

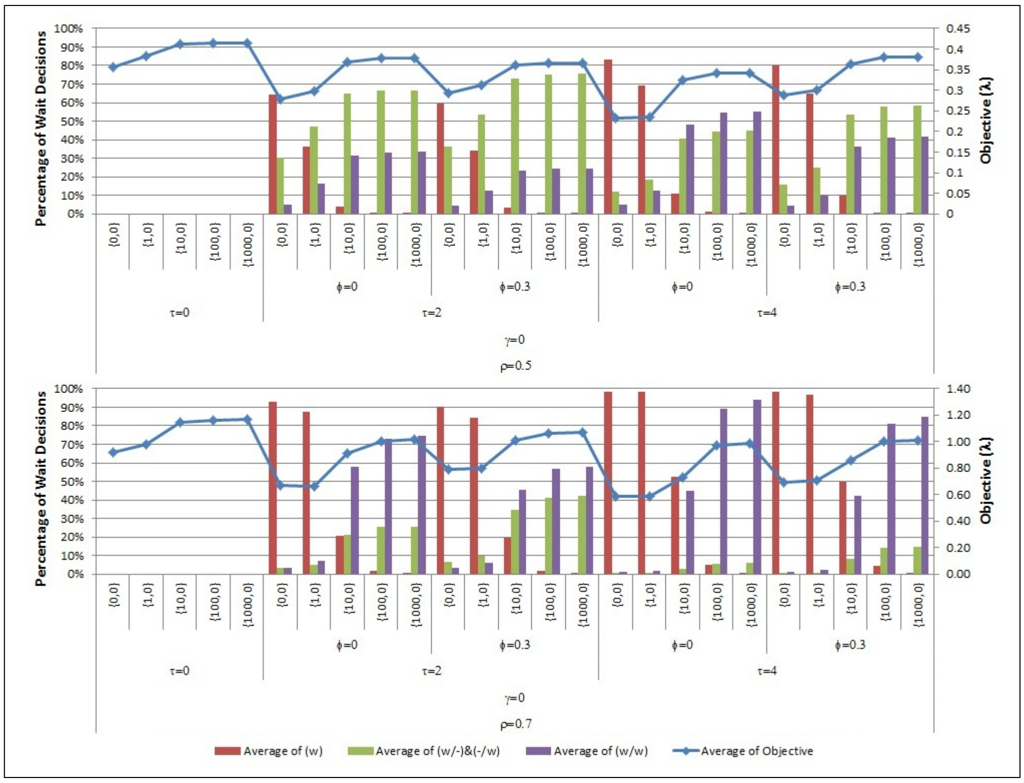


Figure 5.25. Objective Values and Wait Decision Percentages of Distributed Policies with $\omega = 0$ under $\gamma = 0$.

Under very tight due dates even when the distributed schedulers operate asynchronously, the distributed scheduling system decide same as a centralized scheduling system most of the time. In Figure 5.24, for $\tau = 0$, (s/-) and (-/s) percentages reach up to 86% under $\rho = 0.5$ and 70% under $\rho = 0.7$.

- Higher utilization and moderately loose due dates generates local problems with significant queues. In such environments, the probability of dispatching a wrong job to the machine increases as (s/w) and (w/s) and (s/s) decisions increases and the probability of giving a wrong wait decision increases as (w/w) increases. Hence, the impact of synchronization can be observed.

In low utilization environments (small $\rho$), queue lengths are very small, hence there is a little value that can be obtained by synchronization. For example, consider $\tau = 4$ with $\phi = 0$ in Figure 5.24, under $\rho = 0.5$ the performance of {0,0} is around $0.23\lambda$ and the performance of {1000,0} is around $0.34\lambda$, so the deterioration is around

0.11 $\lambda$. Whereas under $\rho = 0.7$, the performance of {0,0} is around $0.59\lambda$ and the performance of {1000,0} is around $0.98\lambda$, the deterioration is around 0.39 $\lambda$.

In Figure 5.24 for $\rho = 0.7$, as $\Omega$ gets higher, the deterioration in system performance under $\tau = 0$ is small whereas it is higher under $\tau = 4$ and $\phi = 0$.

For $\tau = 0$, we can confirm from Figure 5.25 that the schedulers never decide to wait, so there is no chance for a wrong wait decision (w,w). Similarly we can check from Figure 5.24 that (s/w) and (w/s) percentages are zero. Thus deterioration in system performance is induced by some of (s/s) decisions.

### 5.2.2. Centralized Versus Distributed Scheduling Systems

The centralized scheduler is able to generate better solutions at each instance of the algorithm run by including more detailed information but it necessitates more time for the scheduling process. This may increase the discrepancy between the state of the system known to the scheduling algorithm and the actual state of the system at the time the resulting schedule is implemented. Distributing the scheduling problem among various schedulers may improve reactivity by providing smaller local problems. However, lack of global information may reduce the quality of the solutions. In this section, we want to compare the performance of the fast distributed schedulers with the slow centralized scheduler.

In Section 3.1.1, we define three response mode of a scheduler: instantaneous, busy and available response modes. As mentioned in Section 3.2.1, we assume that distributed schedulers run the optimal scheduling algorithm in instantaneous response mode. For the centralized scheduler, we show in Section 5.1.3 that for a given response mode and given $\gamma, \phi, \tau$ and $\rho$ values, as $\delta$ gets higher, performance of the system degrades. Hence, for this section we select a representative control policy for each operating modes A and B instead of presenting the performance of all centralized control policies.

Figure 5.26 and 5.27 show the objective values of selected centralized and distributed policies for $\rho = 0.5$ and $\rho = 0.7$ respectively. In the horizontal axis, the policies are sorted in the non-decreasing order of their average objective values.

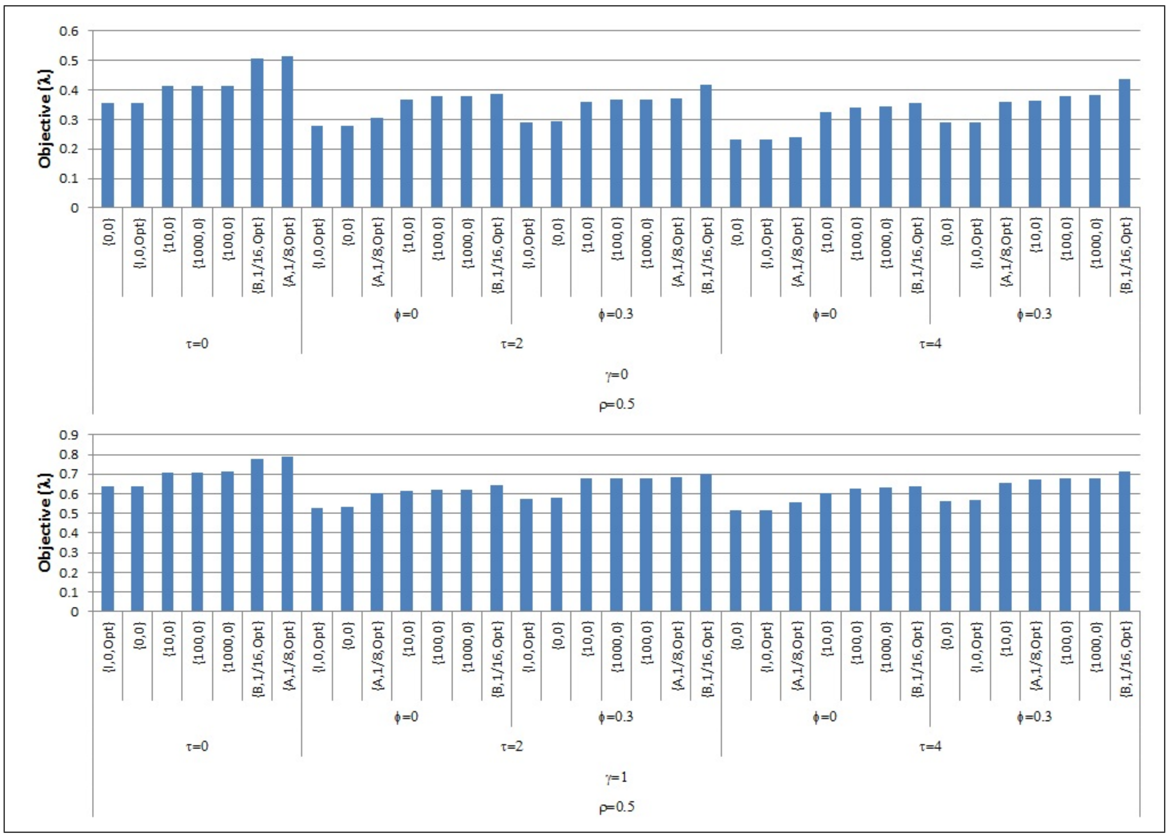


Figure 5.26. Objective Values of Selected Centralized and Distributed Policies for $\rho = 0.5$.

Based on Figure 5.26 and 5.27, we can drive the fallowing observations.

- Centralized policy $\{I, 0, Opt\}$ and distributed policy $\{0, 0\}$ performs very close to each other. The little difference between objective values of these policies is caused by scheduling trigger mechanism in distributed system mentioned as *Problem 2* in Section 3.2.2.
- All distributed policies outperform the centralized policy $\{B, \frac{1}{16}, Opt\}$.
- For $\gamma = 0$, and for a given $\tau$, the performance of centralized A policy degrades as urgent job ratio increases (higher $\phi$ values) and approximates to the performance of asynchronous distributed policy $\{1000, 0\}$.
- For $\gamma = 0$, and for $\phi = 0.3$, the relative rank of A policy increases as due dates

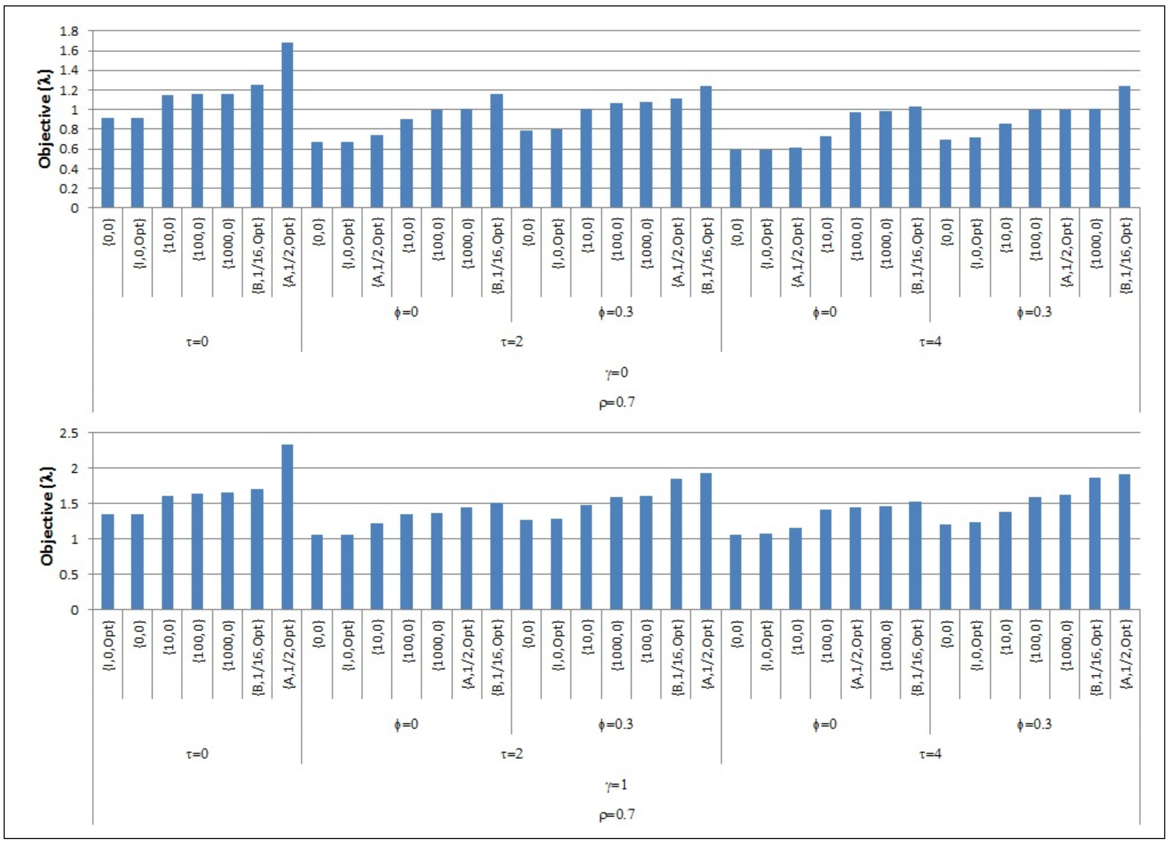


Figure 5.27. Objective Values of Selected Centralized and Distributed Policies for $\rho = 0.7$.

become tighter (lower $\tau$ values).

- The ranking of centralized A and B policies increases as processing variability increase for $\rho = 0.7$ in Figure 5.27. The effect of processing variability on the ranking of policies is not that significant for $\rho = 0.5$ in Figure 5.26.

We conclude that, all distributed policies outperform the selected centralized B policy. We also conclude that selected centralized A policies perform relatively better for looser due date values. This can be explained by analyzing the queue length. For a given $\tau$, queue length increases as $\phi$ decreases. For a given $\phi$, queue length increases as $\tau$ increases. Hence, the impact of better scheduling can be observed in problems with significant queues.

In summary, we conjecture that, under some specific operating conditions, the dynamic production system will run more efficiently when we use fast distributed asynchronous schedulers instead of relatively slow centralized scheduler.

# 6. CONCLUSION

In this study, we analyze a single machine dynamic scheduling environment with both a centralized and a distributed scheduling system. The objective is to minimize the summation of earliness and tardiness. In the scheduling literature involving similar problems, a common assumption is that, at an instance that the scheduler needs to dispatch a job to the machine, regardless of how complex the scheduling algorithm is, the schedule can be obtained instantaneously. Our experience in real life scheduling environments is that, depending on the complexity of the algorithm deployed and depending on the details of information included, the scheduling process may take a significant amount of time.

In this study we explicitly model the scheduling (decision) time and analyze its impact on the system performance. We define two basic modes of operation for the scheduler during the decision process. In the available mode, the scheduler continues dispatching the machine using the last schedule it generated, although the schedule may not reflect the current state of the system correctly. In the busy mode, the scheduler ignores all dispatch requests and keeps the machine idle until a new schedule is obtained.

Our study is composed of two phases. In the first phase, we test the trade off between spending more time for the scheduling process by employing more sophisticated scheduling algorithms and using simple fast heuristic algorithm. In the second phase, we test the trade off between spending more time by including detailed global information to achieve global optimality under a centralized control structure and using timely accessible local information under distributed control.

For the first phase, we define various centralized control polices by using different operation modes and decision times. One of these centralized control policy represents a fast simple heuristic. Another one represents the studies in the dynamic scheduling literature which assumes that whole scheduling process is instantaneous. The rest of the centralized control policies denotes slow optimization algorithms.

We simulated the system under various scheduling environments controlled by due date tightness, urgent job ratios, operation time variability and utilization using different centralized control polices. The results of the first phase indicate that decision time is a very significant determinant of system performance. If the time it takes to make a decision is relatively high, even a responsive system using a random dispatching mechanism may outperform a system guided by an optimal scheduling process. In other words, in order for an optimization algorithm to be effective in a scheduling system, the time it takes to obtain the optimal result must be relatively short.

For the second phase of the study, we devise a distributed scheduling system composes of two schedulers, each owing different sets of jobs. The schedulers are allowed to synchronize their local information periodically. We define various distributed control polices by using different synchronization period lengths. We simulated the system again under various scheduling environments controlled by due date tightness, urgent job ratios, operation time variability and utilization using different distributed control polices. We analyze how lack of global information affects solution quality 1- by examining the effect of synchronization period on the performance of distributed schedulers, 2- by making comparison between the selected centralized and distributed control policies.

The results indicate that synchronization period length affect the distributed system performance and lack of global information results the deterioration in system performance. If the synchronization period length is relatively high, the schedulers operates almost always asynchronously, and over this synchronization period length value, deterioration stops and the performance stays constant.

The comparison between the performance of a fast distributed scheduling system with slow centralized scheduling system reveals that if the time it takes to make a decision is relatively high, even an asynchronous distributed system may outperform a centralized system guided.

# APPENDIX A: DISCRETE EVENT SIMULATION

In this study, dynamic single machine scheduling system is simulated by discrete event simulation. Simulation events and processes are explained by referring the data flow diagram of simulation shown in Figure A.1. In the data flow diagram, processes are shown in circles, data stores are shown in contours defined by parallel lines, and simulation events are shown in diamonds.

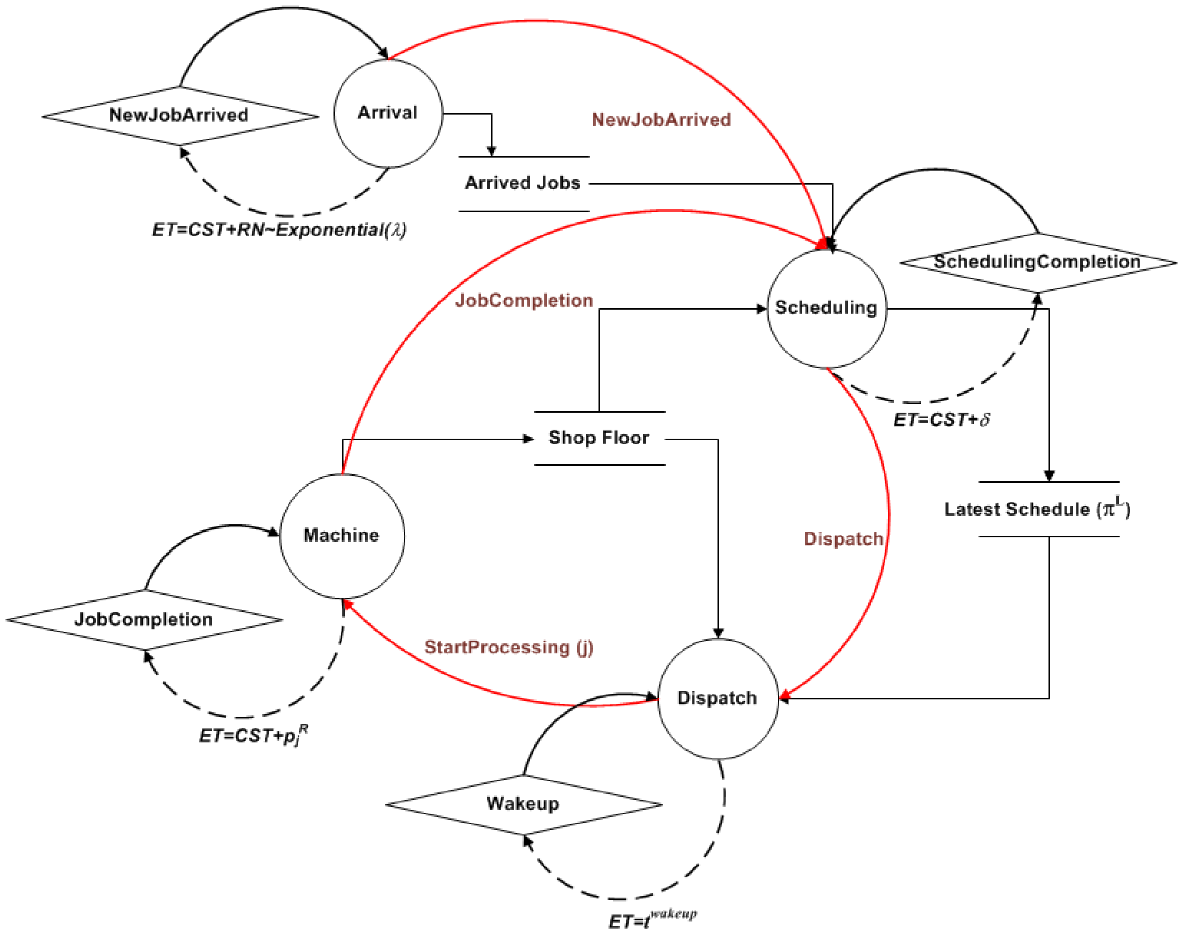


Figure A.1. The Data Flow Diagram of the Simulation.

A black dashed arrow points to an event represents creation of a new simulation event. A skewed arrow points to a process is a process trigger. There are two types of triggers: (i) black skewed arrows represent a simulation event triggers a process; (ii) red skewed arrows represent a process sends a command to trigger another process. There may be more than one trigger points to a process. Each process trigger calls a specific function or functions defined in that process. Processes may result modifications of data: updating /writing data is represented by straight black arrows coming into data stores; synchronizing /reading data is represented by straight black arrows going out from data stores. A process may end by creating a new simulation event and/or a new process trigger.

The simulator has a simulation clock to control simulation time, and has an events list to operate simulation. Current Simulation Time ($CST$) is initialized in the beginning of simulation, and initial event(s) is created by defining a specific event time ($ET$), and put into events list. In discrete-event simulation, the operation is represented as a chronological sequence of events. If all events in events list have greater $ET$ than $CST$, then $CST$ is raised up to the smallest $ET$ present in events list. Later the event is handled and removed from the list when the $CST$ is equal to the $ET$. At any time in simulation, $CST$ is equal to the $ET$ of currently activated event. Each event is designed to create the consecutive event by defining next event time as $CST$ plus a positive number/time.

Each simulation event starts one of the process of simulation. As shown in Figure A.1, there are four process of simulation: Arrival, Scheduling, Dispatching and Production. In Figure A.1, name of events are consistent with the context of Section 3.1 and 3.2. However in implementation, the event names are different.

## A.1. Arrival Process

The *Arrival Process* triggered by a *NewJobArrived* event, controls new job arrivals to the system. Assume that job $j$ is the new job arrived to the scheduling system. The arrival time of the new job is assigned to event time, $a_j = ET$. The other attributes $(\hat{p}_j, p_j^R, d_j)$ of the job $j$ are initialized according to the guidelines defined in Section 4.1. *Arrived Jobs* data store is updated by writing the information of recently arrived job. Remember that job arrival times are generated such that $(a_{j+1} - a_j) \sim Exponential(\lambda)$. Note that $CST = ET$. Hence the next *NewJobArrived* event is created by $ET =$

$CST + RN \sim Exponential(\lambda)$.

A new job arrival may or may not trigger a scheduling process. For periodic scheduling policies, new arrivals are collected and contained in a set until the next scheduling period starts, and then they are released to system as a set of jobs. In this study, we assume that (re)scheduling is triggered with each new job arrival, therefore *NewJobArrived* command is send to the Scheduling process at the end of *Arrival Process*.

## A.2. Scheduling Process

The *Scheduling Process* includes loading recent data of the dynamic system and running a scheduling algorithm, and releasing the generated schedule. As explained in Section 3.1.1, three different operation modes of the scheduler are defined. In instantaneous response mode, the schedule is assumed to be generated immediately, else scheduling generation takes time. In busy and available response modes where scheduling generation takes time,*Scheduling Process* is divided into two sequential processes: 1- *Schedule Generation Process* 2-*Schedule Release Process*.

Referring to Figure A.1, *NewJobArrived* and *JobCompletion* commands may trigger the *Schedule Generation Process. SchedulingCompletion* event triggers the *Schedule Release Process*.

In the *Schedule Generation Process*, the scheduler recognizes released jobs by reading recent data from arrived jobs data store $J_{CST} = \{j : a_j > 0\}$, and it follows up shop floor realizations by synchronizing with shop floor data store $\{c_j^R : j \in J_{CST}\}$. Then it starts schedule generation by considering the set of jobs that are in the system at time $CST$, i.e. $J_{CST}^S = \{j : j \in J_{CST}$, and $(c_j^R = 0$ or $c_j^R > CST)\}$.

*Schedule Generation Process* takes $\delta$ time. During that process, the schedule in generation is incomplete and cannot be used. Once a *Schedule Generation Process* starts, it cannot be interrupted (retriggered) by events occurring during $\delta$. At the end of *Schedule Generation Process* 1- The corresponding *SchedulingCompletion* event is created by defining $ET = CST + \delta$, 2- A *Dispatch* command to start the *Dispatch Process* is send.

*SchedulingCompletion* event determines the end of the *Schedule Generation Process* and the start of the *Schedule Release Process.* In *Schedule Release Process*, last generated schedule $\pi^L$ is written in *Latest Schedule* data store. The *Schedule Release Process* ends by sending a *Dispatch* command to start then *Dispatch Process.*

The details of response modes are provided in following subsections.

### A.2.1. Instantaneous Mode

In instantaneous mode, *Schedule Generation Process* and *Schedule Release Process* are executed together without creating a *SchedulingCompletion* event.

Referring to Figure A.1, suppose that a *Job Release* (or *Job Completion*) trigger at CST is handled. It considers recent realizations $J^S_{CST}$, and starts scheduling process to generate schedule $\pi$, $t^S(\pi) = CST$. Scheduling takes no time, $\pi$ is generated instantaneously $t^G(\pi) = CST$. Latest schedule data store is updated by writing $\pi^L_{CST}$. The *Dispatch* command is send at time $CST$.

### A.2.2. Busy Mode

In busy mode, *Schedule Generation Process* and *Schedule Release Process* are executed separately by creating a *SchedulingCompletion* event.

Referring to Figure A.1, suppose again that a *Job Release* command (or *Job Completion* event) at CST is handled. If the scheduler is not already generating a schedule, then it starts *Schedule Generation Process* to generate schedule $\pi$, $t^S(\pi) = CST$ by considering $J^S_{CST}$. In busy response mode at the beginning of *Schedule Generation Process*, the current schedule $\pi^L$ in *Latest Schedule* data store is removed. At the end of the process *SchedulingCompletion* event is created by defining $ET = CST + \delta$, and it is put into the event list of simulation. If the scheduler is already generating a schedule when it receives a *Scheduling Process* trigger, this trigger is ignored in busy mode.

### A.2.3. Available Mode

In available mode, *Schedule Generation Process* and *Schedule Release Process* are executed separately by creating a *SchedulingCompletion* event. Unlike busy mode, in available response mode at the beginning of *Schedule Generation Process*, the current schedule in *Latest Schedule* data store is not removed so that previously generated old schedule $\pi^L$ with $t^G(\pi^L) < CST$ is available.

Suppose again that a *Scheduling Process* trigger is received at CST. If the scheduler is not already generating a schedule, then it starts *Schedule Generation Process* to generate schedule $\pi$, $t^S(\pi) = CST$ by considering $J^S_{CST}$. At the end of the process 1- *SchedulingCompletion* event is created by defining $ET = CST + \delta$, and put into the event list of simulation. 2- *Dispatch* command is send at time $CST$. If there is an ongoing *Schedule Generation Process*, then any initiator event realized in the mean time is ignored by saying that the scheduler is busy.

## A.3. Dispatch Process

In this study, the assumption is that scheduling is the decision process that takes time, but dispatching is done instantaneously. In Figure A.1, a *Dispatch* command or a *Wakeup* event triggers *Dispatch Process.* The operation of the dispatcher in the centralized scheduling system and in the distributed scheduling system is explained in Section 3.1.2 and Section 3.2.3 respectively in details. At the end of the *Dispatch Process* either the machine is dispatched by sending *StartProcessing (j)* command to the machine or the machine is kept idle by creating a *Wakeup* event with $ET = t^{wakeup}$.

The *Wakeup* event is created to remind the time to dispatch a job. The *Wakeup* event is active if it is included in the events list of the simulator. There may be at most one *Wakeup* event in the events list. If a dispatch trigger is received while there is an active *Wakeup* event, then the *Wakeup* event becomes redundant and it is removed from the events list.

To illustrate, suppose that events list contains an active *Wakeup* event with $ET = t_1$ and a *NewJobArrived* event with $ET = t_2$. Let $CST < t_2 < t_1$. If all events in events list have greater $ET$ than $CST$, then $CST$ is raised up to the smallest $ET$ present in events list. Hence, *NewJobArrived* event is handled first. From Figure A.1, *NewJobArrived* event results triggering of *Scheduling Process*. As explained above, if the operation is in instantaneous mode or in available mode a *Dispatch* command is send to *Dispatch Process*. Note that $CST = t_2$. The dispatcher decides according to conditions at time $CST$. If the schedule in *Latest Schedule* data store is changed, the previous dispatch decision may become useless. Else if *Latest Schedule* data store is still the same, then a *Wakeup* event with $ET = t_1$ is created again and it is put into the events list.

## A.4. Machine Process

*Machine Process* process controls shop floor realizations by updating shop floor data store. It is initiated either by a *StartProcessing (j)* command or a *JobCompletion* event.

When *StartProcessing (j)* command is received, the real start time of the job $s_j^R$ is recorded in *Shop Floor* data store as $s_j^R = CST$, then the corresponding *JobCompletion* event created by defining its $ET$ as $ET = CST + p_j^R$ at the end of the process.

When *JobCompletion* event is handled, real completion time of recently processed job $j(CST)$ is recorded as $c_j^R = ET$. At the end of the process, *JobCompletion* command is sent to *Scheduling Process* in order to start new decision process.

# APPENDIX B: ICRON IMPLEMENTATION

In this study, ICRON Planning and Optimization applications software is chosen to implement the simulator and evaluate the issues discussed. ICRON is a general purpose visual algorithm modeling software. It has a very special visual modeling tool named GSAMS. GSAMS is an object-oriented graphical algorithm modeling system which is used to construct and modify algorithms within graphical environments. GSAMS allows users to drag and drop algorithmic objects on the design window and visually construct algorithms, thereby the algorithms are presented as graphical flow charts and it is easy to follow and understand the logic.

In order to model an algorithm, the user works on nodes. Each node represents a particular operation on the current object. Nodes have input and output link points. Objects flow between the nodes through link points and all of the data manipulations are managed by node and link structures of the algorithm model. GSAMS provides basic nodes to execute elementary programming operations under the classifications like Datasource Manipulation, Execution Flow, Input/Output, List Manipulation, Messaging, Object Model Navigation, Reporting, System Construction and XML. GSAMS offers a rich set of planning objects. The user can benefit from existing system components, and also can define new components or extend existing ones. In addition to these advantages, ICRON uses the latest available versions of CPLEX, COIN-Clp and COIN-Cbc solvers.

In this study, we develop a simulation model named SFSim in ICRON. Both of the Centralized Scheduling System Simulator (CSSS) and the Distributed Scheduling System Simulator (DSSS) are constructed by using SFSim model. In following sections, the SFSim model is explained in details by using Object Orientation terminology. Object Orientation allows programmers to separate program-specific data types through the use of classes. Classes define types of data structures and the functions that operate on those data structures. Instances of these data types are known as objects.

The structured design of SFSim explained in Chapter A is common for both of CSSS and DSSS. However some of the classes are modified according to the type of the simulator. We present the implementation according to the type of the simulator. The common attributes and the algorithms belong to each class will be given in a main section. Algorithms specific to CSSS and DSSS are separately explained under subsections. Hence, for example, in order to examine the attributes of a job object in DSSS implementation, the reader should consider both of the Section B.2 and B.2.2. The type of an attribute is specified in parenthesis after its name is declared.

In Chapter 5, we present the results of experiments where the distributed schedulers operates only in the instantaneous mode. However, by using DSSS implementation, it is possible to simulate the other cases where the distributed scheduler's decision process takes time. In following subsections, while explaining DSSS implementation, algorithms required only for instantaneous mode will be presented.

## B.1. Event Class

There is a base Event class in SFSim model, and all simulation events inherit from this base class. The attributes of Event class are:

- TheSimulator (object)
- Time (time)
- Code (string)

Event class has an algorithm named *Handle* shown in Figure B.1. *Handle* algorithm is overloaded in the classes inherit from Event class. The algorithm shown in Figure B.1 evaluates the event, and finds the base object of the Event type arrived, and invokes the *Handle* algorithm of the base object. For example, if the event type is NEWJOB, then this algorithm invokes the *Handle* method of the NEWJOB event object. This is same for all the other event types inherits from Event class.

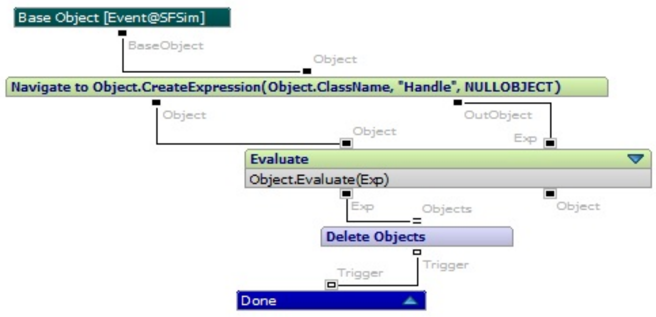


Figure B.1. Handle Algorithm of Base Event Class.

### B.1.1. NEWJOB Event for CSSS

The name of *NewJobArrived* event in Figure A.1, is NEWJOB in implementation. NEWJOB is an Event class. Overloaded *Handle* algorithm in CSSS implementation is shown in Figure B.2. The steps of the algorithm are as follows:

*Handle Algorithm*

Step 1. The Jobs list of the simulator is reached. The first job in the list that is not assigned an ArrivalTime yet is picked.

Step 2. The attributes of the selected job are assigned by using *InitializeTheJob* algorithm of the Job class presented in Figure B.9 in Section B.2.1.

Step 3. If the centralized scheduler is in a scheduling process, (IsScheduling variable returns true), then *Dispatch* algorithm of the simulator is called, else the *Schedule* algorithm of the simulator is called. *Schedule* and *Dispatch* algorithms are presented in Figure B.17 in Section B.6.1 and Figure B.19 in Section B.6.2 respectively.

Step 4. At the end of algorithm the next NEWJOB event is created by calling *CreateNewJobEvent* algorithm in Figure B.13 in Section B.6.

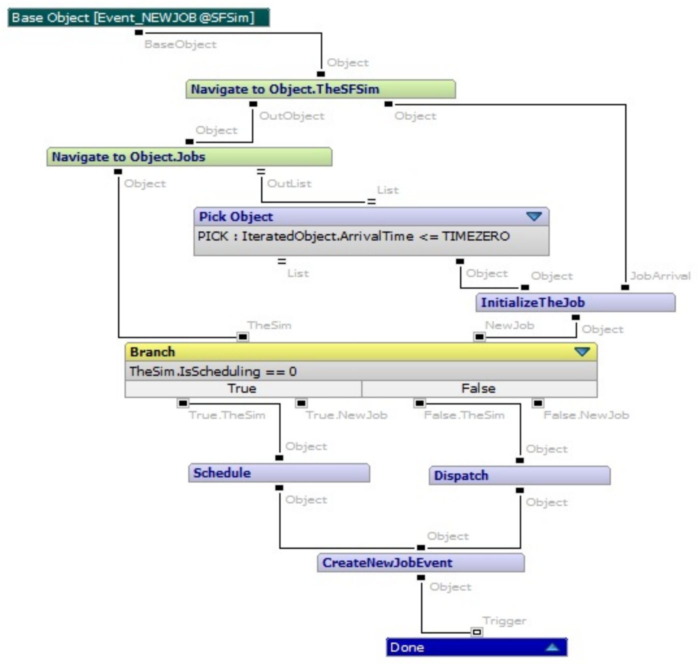


Figure B.2. Overloaded Handle Algorithm of NEWJOB Event in CSSS Implementation.

### B.1.2. NEWJOB Event for DSSS

In DSSS implementation, according to the JobType of arriving job, one of the schedulers is triggered. Overloaded *Handle* algorithm of NEWJOB event in DSSS implementation is shown in Figure B.3. The steps of the algorithm are as follows:

*Handle Algorithm*

Step 1. The Jobs list of the simulator is reached. The first job in the list that is not assigned an ArrivalTime yet is picked.

Step 2. The attributes of the selected job are assigned by using *InitializeTheJob* algorithm of the Job class presented in Figure B.11 in Section B.2.2. Note that, JobType of the job is determined by using *InitializeTheJob* algorithm.

Step 3. TheCurrentSchedulerType variable of DSSS is set to JobType of the job. TheCurrentSchedulerType denotes the type of scheduler relevant to the arrival.

Step 4. If TheCurrentSchedulerType is "A" and the the scheduler A is not busy (IsSchedulingA variable returns false), or If TheCurrentSchedulerType is "B" and the the scheduler B is not busy (IsSchedulingB variable returns false), then *Schedule* algorithm in Figure B.17 in Section B.6.1 is called. If the relevant scheduler is busy, it ignores the trigger and *Dispatch* algorithm in Figure B.19 in Section B.6.2 is called.

Step 5. At the end of algorithm the next NEWJOB event is created by calling *CreateNewJobEvent* algorithm in Figure B.13 in Section B.6.

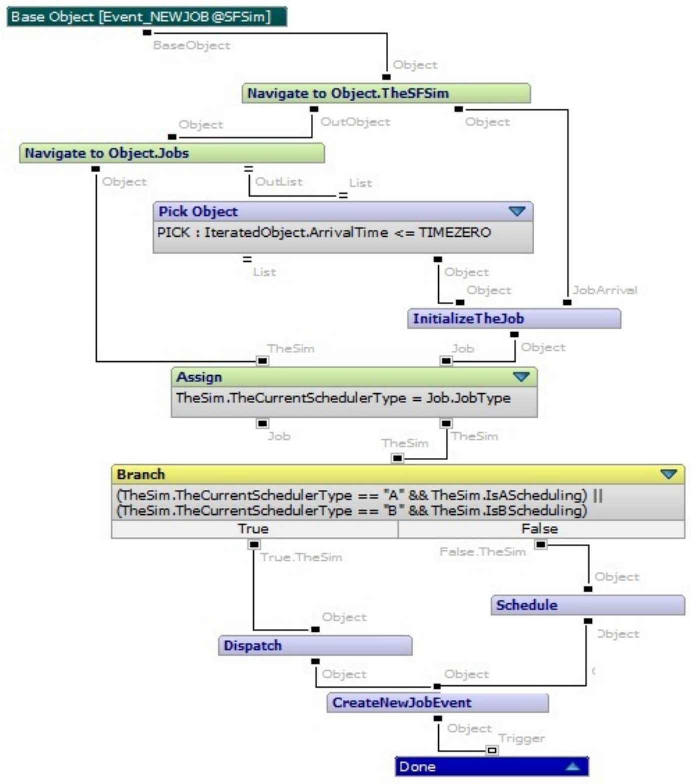


Figure B.3. Overloaded Handle Algorithm of NEWJOB Event in DSSS Implementation.

### B.1.3. OPERATIONCOMPLETED Event for CSSS

The name of *JobCompletion* event in Figure A.1, is OPERATIONCOMPLETED in implementation. OPERATIONCOMPLETED is an Event class. Overloaded *Handle* algorithm in CSSS is shown in Figure B.4.

The steps of the algorithm are as follows:

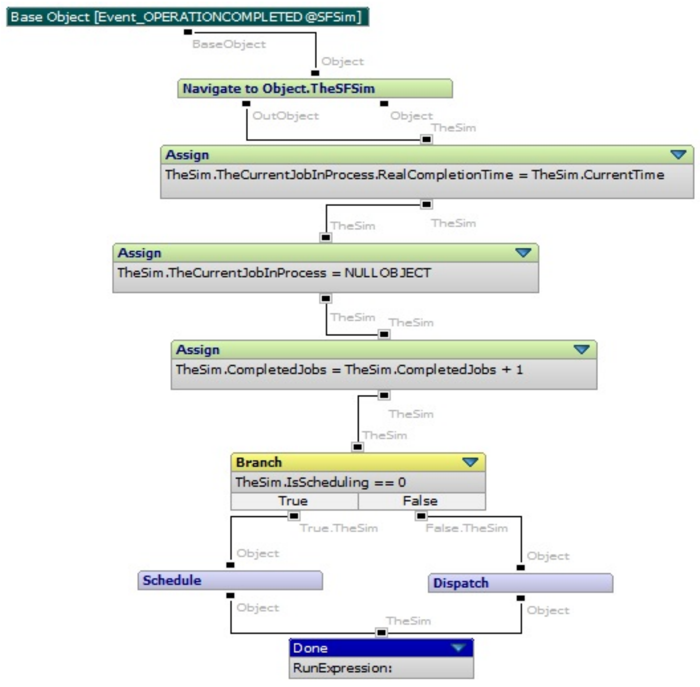

Base Object [Event_OPERATIONCOMPLETED @SFSim]
BaseObject
Object
Navigate to Object.TheSFSim
OutObject
Object
TheSim
Assign
TheSim.TheCurrentJobInProcess.RealCompletionTime = TheSim.CurrentTime
TheSim
TheSim
Assign
TheSim.TheCurrentJobInProcess = NULLOBJECT
TheSim
TheSim
Assign
TheSim.CompletedJobs = TheSim.CompletedJobs + 1
TheSim
TheSim
Branch
TheSim.IsScheduling == 0
True
False
True.TheSim
False.TheSim
Object
Object
Schedule
Dispatch
Object
Object
TheSim
Done
RunExpression:

Figure B.4. Overloaded Handle Algorithm of OPERATIONCOMPLETED Event in CSSS Implementation.

*Handle Algorithm*

Step 1. TheRealCompletionTime field of TheCurrentJobInProcess object of the simulator job is set to CurrentTime.

Step 2. TheCurrentJobInProcess is set to zero.

Step 3. CompletedJobs number is increased by one.

Step 4. If the centralized scheduler is in a scheduling process, (IsScheduling variable returns true), then *Dispatch* algorithm of the simulator is called, else the *Schedule* algorithm of the simulator is called. *Schedule* and *Dispatch* algorithms are presented in Figure B.17 in Section B.6.1 and Figure B.19 in Section B.6.2 respectively.

### B.1.4. OPERATIONCOMPLETED Event for DSSS

In DSSS, OPERATIONCOMPLETED event triggers both of the schedulers. Overloaded *Handle* algorithm of OPERATIONCOMPLETED event in DSSS implementation is shown in Figure B.5. The steps of the algorithm are as follows:

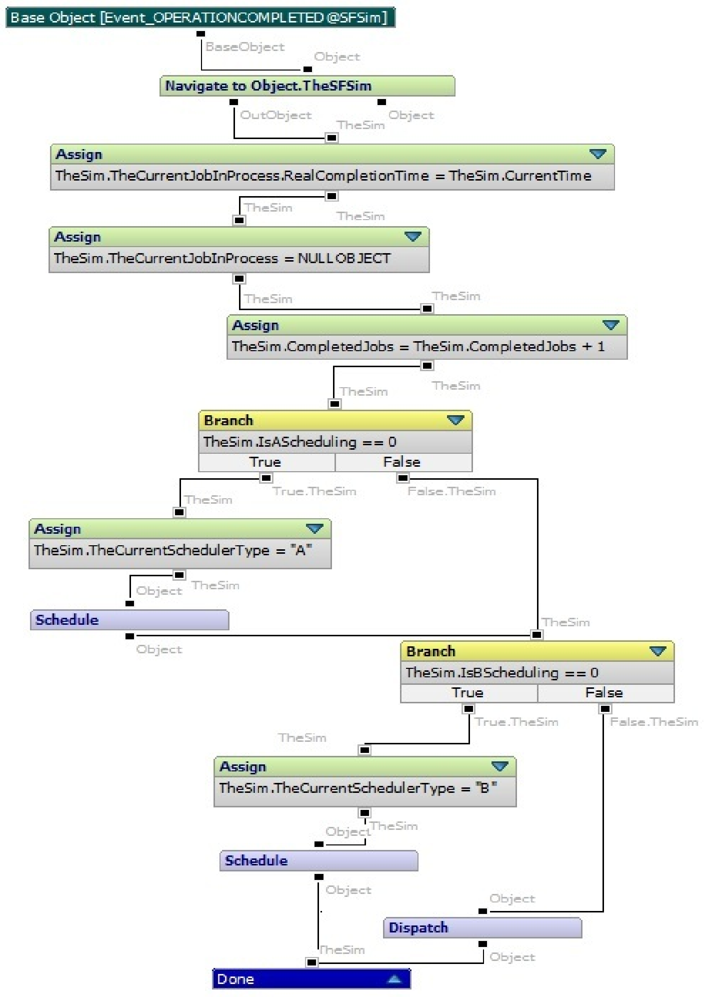

Base Object [Event_OPERATIONCOMPLETED@SFSim]
BaseObject
Object
Navigate to Object.TheSFSim
OutObject
TheSim
Object
Assign
TheSim.TheCurrentJobInProcess.RealCompletionTime = TheSim.CurrentTime
TheSim
TheSim
Assign
TheSim.TheCurrentJobInProcess = NULLOBJECT
TheSim
TheSim
Assign
TheSim.CompletedJobs = TheSim.CompletedJobs + 1
TheSim
TheSim
Branch
TheSim.IsAScheduling == 0
True
False
TheSim
True.TheSim
False.TheSim
Assign
TheSim.TheCurrentSchedulerType = "A"
Object
TheSim
Schedule
Object
TheSim
Branch
TheSim.IsBScheduling == 0
True
False
True.TheSim
False.TheSim
TheSim
Assign
TheSim.TheCurrentSchedulerType = "B"
Object
TheSim
Schedule
Object
Object
Dispatch
TheSim
Object
Done

Figure B.5. Overloaded Handle Algorithm of OPERATIONCOMPLETED Event in DSSS Implementation.

*Handle Algorithm*

Step 1. TheRealCompletionTime field of TheCurrentJobInProcess object of the simulator job is set to CurrentTime.

Step 2. TheCurrentJobInProcess is set to zero.

Step 3. CompletedJobs number is increased by one.

Step 4. In DSSS implementation OPERATIONCOMPLETED event triggers both of the schedulers to start a new scheduling process.

Step 5. If the scheduler A is not busy, (IsSchedulingA variable returns false), then TheCurrentSchedulerType is set to “A” and *Schedule* algorithm is executed. If IsSchedulingB variable returns false, then TheCurrentSchedulerType is set to “B” and *Schedule* algorithm is executed, else *Dispatch* algorithm is called. *Schedule* and *Dispatch* algorithms are presented in Figure B.17 in Section B.6.1 and in Figure B.19 in Section B.6.2 respectively.

### B.1.5. SCHEDULINGCOMPLETED Event

The name of *SchedulingCompletion* event in Figure A.1, is SCHEDULINGCOMPLETED in implementation. SCHEDULINGCOMPLETED is an Event class. Overloaded *Handle* algorithm is shown in Figure B.6.

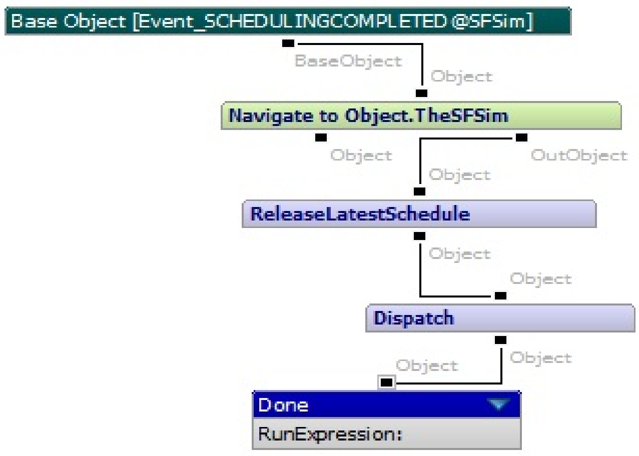


Figure B.6. Overloaded Handle Algorithm of SCHEDULINGCOMPLETED Event.

The steps of the algorithm are as follows:

*Handle Algorithm*

Step 1. *ReleaseLatestSchedule* algorithm is called. In CSSS, the algorithm is developed as shown in Figure B.21 in Section B.6.3.1. In DSSS, the algorithm is developed as shown in Figure B.21 in Section B.6.4.1.

Step 2. After the latest schedule released, *Dispatch* presented in Figure B.19 in Section B.6.2 is called.

### B.1.6. WAKEUPTODISPATCH Event

The name of *Wakeup* event in Figure A.1, is WAKEUPTODISPATCH in implementation. WAKEUPTODISPATCH is an Event class. Overloaded *Handle* algorithm in Figure B.7 calls emphDispatch algorithm presented in Figure B.19 in Section B.6.2

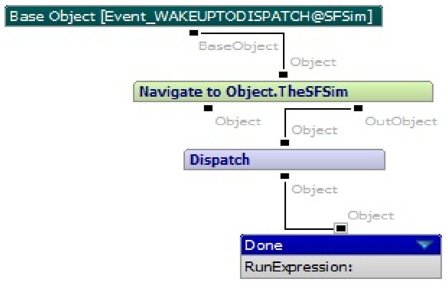


Figure B.7. Overloaded Handle Algorithm of WAKEUPTODISPATCH Event.

## B.2. Job Class

The attributes of Job class are:

- Index (number)
- PredictedProcessingTime (number)
- RealProcessingTime (number)
- ArrivalTime (time)
- DueDate (item)
- RealCompletionTime (time)
- RealStartTime (time)

- TempCompletionTime (time)
- TempStartTime (time)

When a job is created, it is assigned an index. PredictedProcessingTime, RealProcessingTime, ArrivalTime, DueDate, RealCompletionTime, RealStartTime variables are used for $\hat{p}_j, p_j^R, a_j, d_j, c_j^R, s_j^R$ of job $j$ respectively. TempCompletionTime, TempStartTime variables of a job are used in the schedule generation process. In a scheduling algorithm, TempCompletionTime, TempStartTime variables of each job are scheduled completion time and scheduled start time respectively.

### B.2.1. Job Class for CSSS

A job in CSSS implementation has all of the attributes defined in Section B.2 and additional to those it has the following attributes:

- ReleasedCompletionTime (time)
- ReleasedStartTime (time)

As it is explained in Section A.2, in case that scheduling generation takes time, scheduling process is divided into two sequential processes: (i) Schedule Generation Process, (ii) Schedule Release Process. In schedule generation TempCompletionTime, TempStartTime variables of a job are used to determine scheduled times. When the schedule is released ReleasedCompletionTime and ReleasedStartTime variables are assigned to TempCompletionTime, TempStartTime variables respectively. Hence, ReleasedCompletionTime and ReleasedStartTime variables denotes scheduled completion and start times in latest available schedule.

A job object has two functions: *SetDefaultValuesofTheJob* algorithm in Figure B.8 and *InitializeTheJob* algorithm in Figure B.9. Originally, PredictedProcessingTime, RealProcessingTime, ArrivalTime, DueDate, RealCompletionTime, RealStartTime, TempCompletionTime, TempStartTime, ReleasedCompletionTime and Released-

StartTime of a job is set to zero by *SetDefaultValuesofTheJob* algorithm.

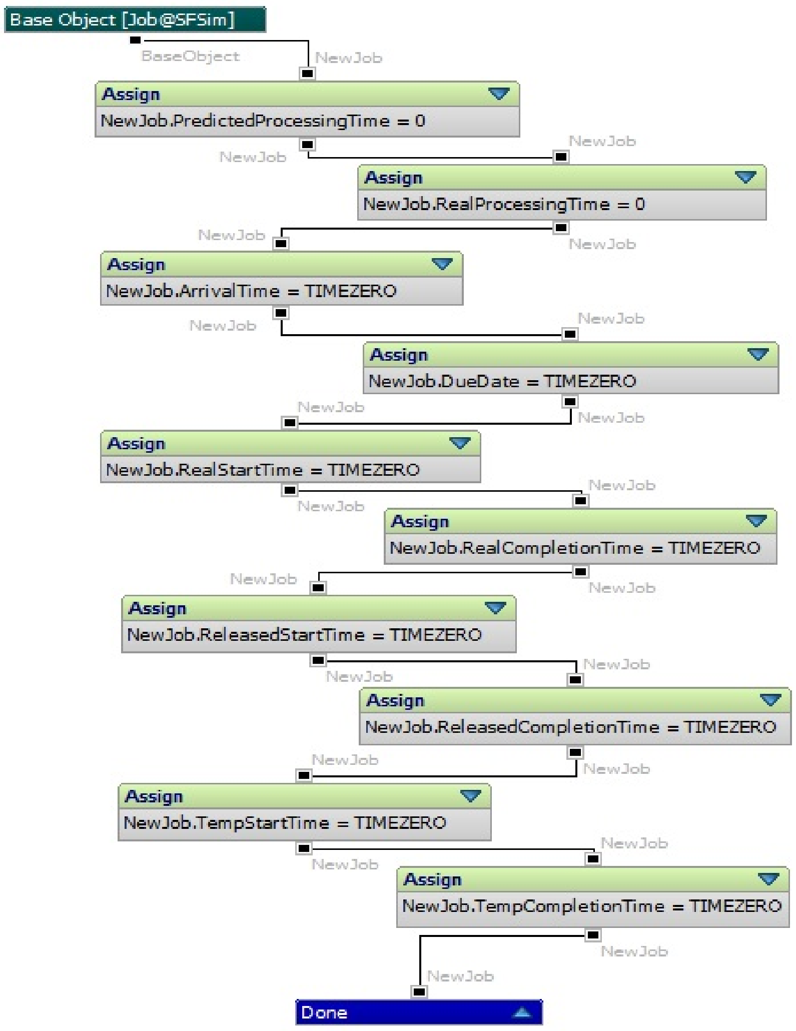
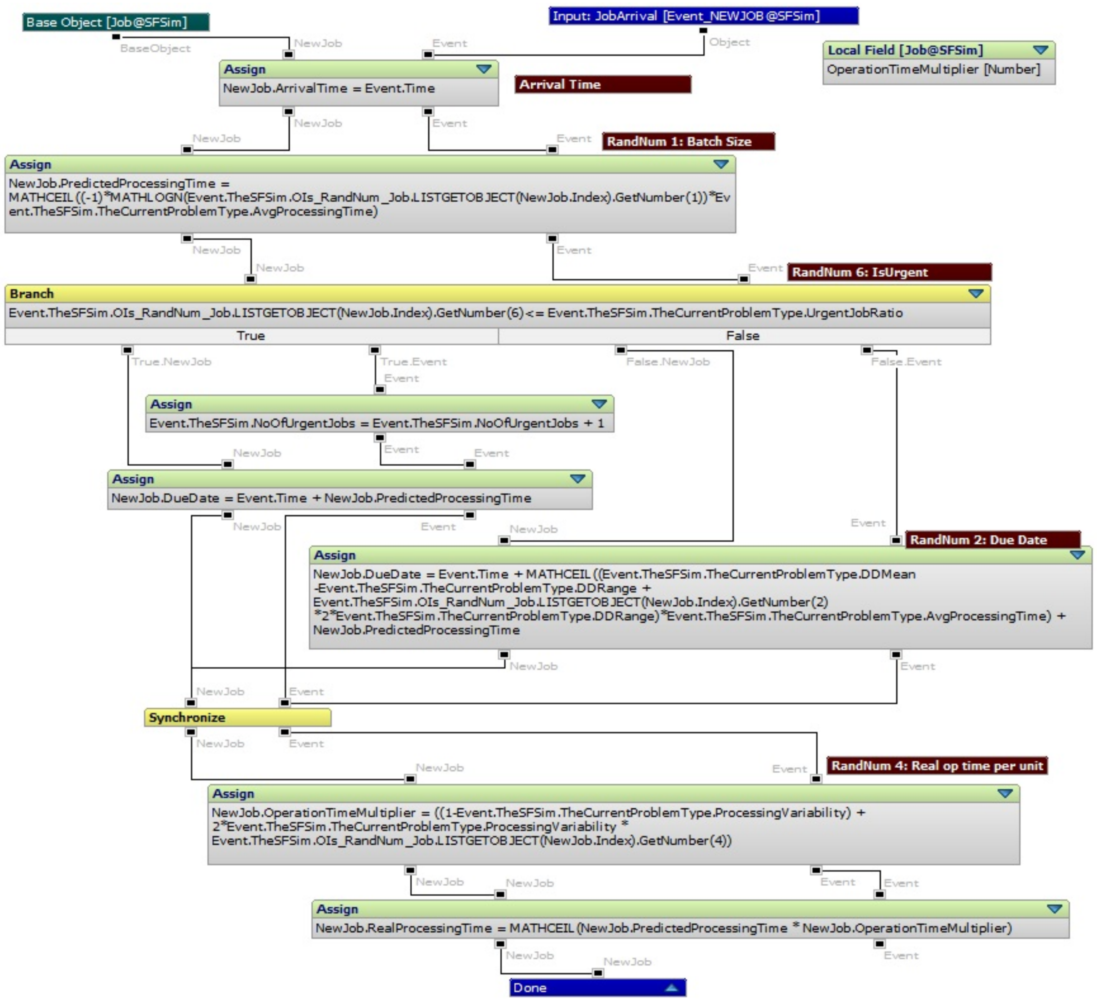

Base Object [Job@SFSim]
BaseObject
NewJob
Assign
NewJob.PredictedProcessingTime = 0
Assign
NewJob.RealProcessingTime = 0
Assign
NewJob.ArrivalTime = TIMEZERO
Assign
NewJob.DueDate = TIMEZERO
Assign
NewJob.RealStartTime = TIMEZERO
Assign
NewJob.RealCompletionTime = TIMEZERO
Assign
NewJob.ReleasedStartTime = TIMEZERO
Assign
NewJob.ReleasedCompletionTime = TIMEZERO
Assign
NewJob.TempStartTime = TIMEZERO
Assign
NewJob.TempCompletionTime = TIMEZERO
Done

Figure B.8. SetDefaultValuesofTheJob Algorithm of Job Class in CSSS Implementation.

The input of *InitializeTheJob* algorithm an Event-NewJobArrived object as shown in Figure B.9. ArrivalTime is set to Time of Event, PredictedProcessingTime, RealProcessingTime, DueDate of the job is calculated as explained in Section 4.1 by using this algorithm.



```
NewJob.ArrivalTime = Event.Time
```

```
NewJob.PredictedProcessingTime =
MATHCEIL ((-1)*MATHLOGN(Event.TheSFSim.OIs_RandNum_Job.LISTGETOBJECT(NewJob.Index).GetNumber(1))*Ev
ent.TheSFSim.TheCurrentProblemType.AvgProcessingTime)
```

```
Event.TheSFSim.OIs_RandNum_Job.LISTGETOBJECT(NewJob.Index).GetNumber(6)<= Event.TheSFSim.TheCurrentProblemType.UrgentJobRatio
```

```
Event.TheSFSim.NoOfUrgentJobs = Event.TheSFSim.NoOfUrgentJobs + 1
```

```
NewJob.DueDate = Event.Time + NewJob.PredictedProcessingTime
```

```
NewJob.DueDate = Event.Time + MATHCEIL ((Event.TheSFSim.TheCurrentProblemType.DDMean
-Event.TheSFSim.TheCurrentProblemType.DDRange +
Event.TheSFSim.OIs_RandNum_Job.LISTGETOBJECT(NewJob.Index).GetNumber(2)
*2*Event.TheSFSim.TheCurrentProblemType.DDRange)*Event.TheSFSim.TheCurrentProblemType.AvgProcessingTime) +
NewJob.PredictedProcessingTime
```

```
NewJob.OperationTimeMultiplier = ((1-Event.TheSFSim.TheCurrentProblemType.ProcessingVariability) +
2*Event.TheSFSim.TheCurrentProblemType.ProcessingVariability *
Event.TheSFSim.OIs_RandNum_Job.LISTGETOBJECT(NewJob.Index).GetNumber(4))
```

```
NewJob.RealProcessingTime = MATHCEIL (NewJob.PredictedProcessingTime * NewJob.OperationTimeMultiplier)
```

Figure B.9. InitializeTheJob Algorithm of Job class in CSSS Implementation.

### B.2.2. Job Class for DSSS

A job in DSSS implementation has the following additional attributes to those defined in in Section B.2:

- ReleasedCompletionTimeA (time)
- ReleasedCompletionTimeB (time)
- ReleasedStartTimeA (time)
- ReleasedStartTimeB (time)
- TempCompletionTimeA (time)
- TempCompletionTimeB (time)
- TempStartTimeA (time)
- TempStartTimeB (time)
- JobType (string)

As explained in Section B.2.1, for the busy and available operation modes, scheduling process is divided into two sequential processes: (i) Schedule Generation Process (ii) Schedule Release Process. In schedule generation, TempCompletionTime, TempStartTime variables of a job are used.

In DSSS there are two schedulers A and B. The scheduled start time and the scheduled completion time determined by the scheduler A are denoted by TempStartTimeA and TempCompletionTimeA respectively. Similarly, TempStartTimeB and TempCompletionTimeB are determined by the scheduler B. Since the scheduling algorithms are common for both of the schedulers, TempCompletionTime, TempStartTime variables are used to determine scheduled times, then TempCompletionTime, TempStartTime variables are assigned to either TempStartTimeA, TempCompletionTimeA variables or TempStartTimeB, TempCompletionTimeB variables according to the type of the scheduler.

When a schedule generated by the scheduler A is released, then ReleasedCompletionTimeA and ReleasedStartTimeA variables are assigned to TempCompletionTimeA, TempStartTimeA variables respectively. Otherwise, ReleasedCompletionTimeB and ReleasedStartTimeB variables are assigned to TempCompletionTimeB, TempStartTimeB variables respectively.

*SetDefaultValuesofTheJob* and *InitializeTheJob* algorithms in Section B.2.1 modified in DSSS implementation are presented in Figure B.10 and B.11. Originally, all of the attributes expect Index of a job is set to zero by *SetDefaultValuesofTheJob* algorithm.

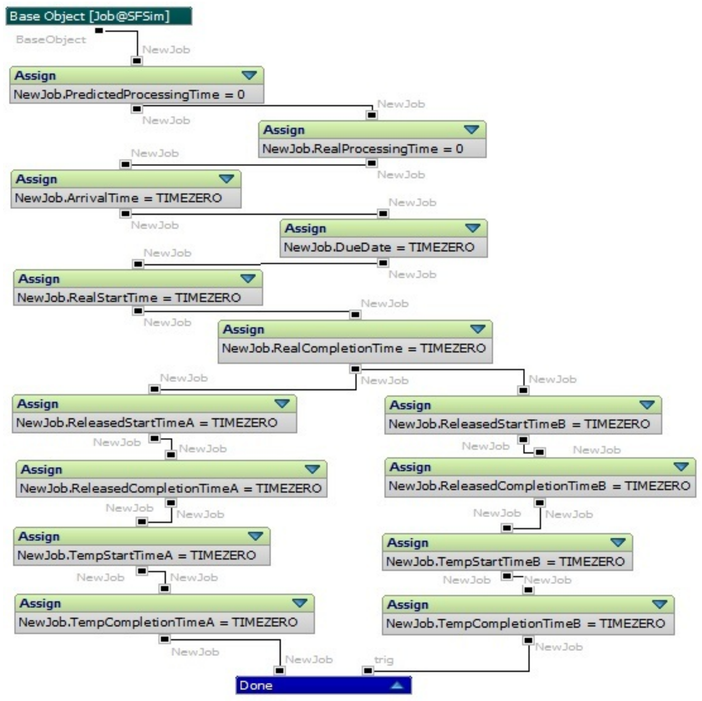
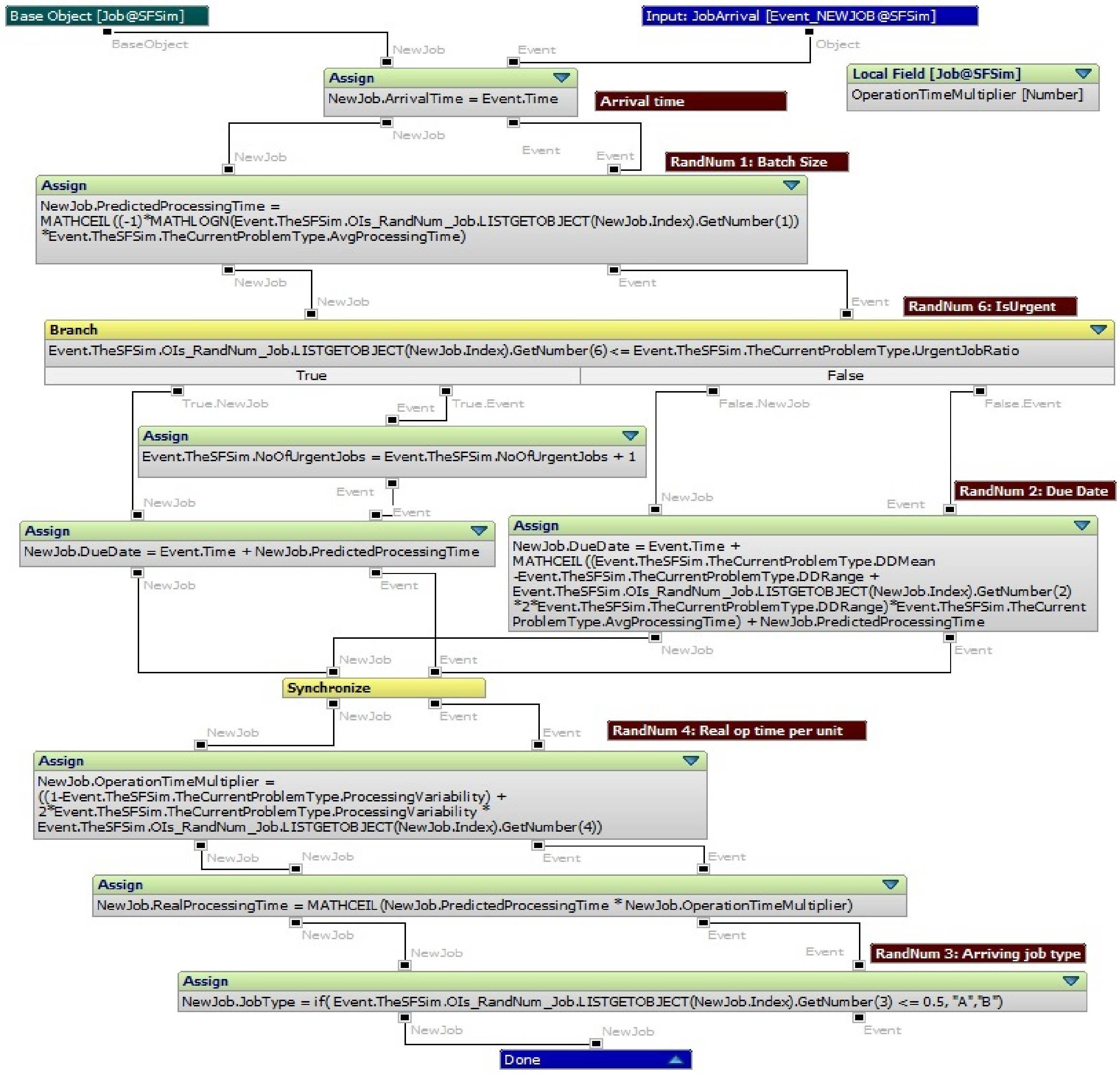

Base Object [Job@SFSim]
BaseObject
NewJob
Assign
NewJob.PredictedProcessingTime = 0
Assign
NewJob.RealProcessingTime = 0
Assign
NewJob.ArrivalTime = TIMEZERO
Assign
NewJob.DueDate = TIMEZERO
Assign
NewJob.RealStartTime = TIMEZERO
Assign
NewJob.RealCompletionTime = TIMEZERO
Assign
NewJob.ReleasedStartTimeA = TIMEZERO
Assign
NewJob.ReleasedStartTimeB = TIMEZERO
Assign
NewJob.ReleasedCompletionTimeA = TIMEZERO
Assign
NewJob.ReleasedCompletionTimeB = TIMEZERO
Assign
NewJob.TempStartTimeA = TIMEZERO
Assign
NewJob.TempStartTimeB = TIMEZERO
Assign
NewJob.TempCompletionTimeA = TIMEZERO
Assign
NewJob.TempCompletionTimeB = TIMEZERO
trig
Done

Figure B.10. SetDefaultValuesofTheJob Algorithm of Job Class in DSSS Implementation.

The input of *InitializeTheJob* algorithm an Event-NewJobArrived object as shown in Figure B.9. ArrivalTime is set to Time of Event, PredictedProcessingTime, RealProcessingTime, DueDate of the job is calculated as explained in Section 4.1. Finally, a RN$\sim U[0,1]$ is generated . if RN$<= 0.5$ then JobType is assigned as “A”, else it is assigned as “B”.



```
Assign
NewJob.ArrivalTime = Event.Time

Assign
NewJob.PredictedProcessingTime =
MATHCEIL((-1)*MATHLOGN(Event.TheSFSim.OIs_RandNum_Job.LISTGETOBJECT(NewJob.Index).GetNumber(1))
*Event.TheSFSim.TheCurrentProblemType.AvgProcessingTime)

Branch
Event.TheSFSim.OIs_RandNum_Job.LISTGETOBJECT(NewJob.Index).GetNumber(6) <= Event.TheSFSim.TheCurrentProblemType.UrgentJobRatio

Assign
Event.TheSFSim.NoOfUrgentJobs = Event.TheSFSim.NoOfUrgentJobs + 1

Assign
NewJob.DueDate = Event.Time + NewJob.PredictedProcessingTime

Assign
NewJob.DueDate = Event.Time +
MATHCEIL((Event.TheSFSim.TheCurrentProblemType.DDMean
-Event.TheSFSim.TheCurrentProblemType.DDRange +
Event.TheSFSim.OIs_RandNum_Job.LISTGETOBJECT(NewJob.Index).GetNumber(2)
*2*Event.TheSFSim.TheCurrentProblemType.DDRange)*Event.TheSFSim.TheCurrent
ProblemType.AvgProcessingTime) + NewJob.PredictedProcessingTime

Assign
NewJob.OperationTimeMultiplier =
((1-Event.TheSFSim.TheCurrentProblemType.ProcessingVariability) +
2*Event.TheSFSim.TheCurrentProblemType.ProcessingVariability *
Event.TheSFSim.OIs_RandNum_Job.LISTGETOBJECT(NewJob.Index).GetNumber(4))

Assign
NewJob.RealProcessingTime = MATHCEIL(NewJob.PredictedProcessingTime * NewJob.OperationTimeMultiplier)

Assign
NewJob.JobType = if( Event.TheSFSim.OIs_RandNum_Job.LISTGETOBJECT(NewJob.Index).GetNumber(3) <= 0.5, "A","B")
```

Figure B.11. InitializeTheJob Algorithm of Job Class in DSSS Implementation.

## B.3. PolicyType Class

PolicyType object is a control policy to define alternative styles of management in experimentation. Control Policy Settings are explained in Section 4.2. Each PolicyType object has the following attributes:

- DecisionTime (number)
- OperatingMode (string)
- SchedulingAlgorithm (string)

As their name imply, OperatingMode, DecisionTime, SchedulingAlgorithm parameters are used for $x, y$, and $z$ respectively.

### B.3.1. PolicyType Class for DSSS

The additional attributes of a PolicyType object in DSSS implementation are:

- SynchronizationDelta (number)
- SynchronizaitionPeriod(number)

SynchronizationDelta is the parameter $\omega$ and SynchronizaitionPeriod is the parameter $\Omega$ in SFSim model.

## B.4. ProblemSetting Class

The attributes of a ProblemSetting object are:

- EarlinessWeight (number)
- MaxNoOfJobsArrived (number)
- MaxNoOfJobsCompleted (number)
- NoOfReplications (number)

- Pivot (number)
- TardinessWeight (number)

EarlinessWeight and TardinessWeight parameters are the penalty weights for realized earliness and tardiness of a job. In SFSim model, it is possible to test weighted earliness/tardiness problem. In this study, we present the experiments where these parameters are set to 1. MaxNoOfJobsArrived parameter determines the number of jobs arrived to the system. We stop the experiment when the number of completed jobs reach the value of MaxNoOfJobsCompleted parameter. NoOfReplications parameter determines the number of replication of each problem instance.We collect statistics over the completed jobs list of the simulator. Pivot determines the number of completed jobs that is excluded from the statistics.

## B.5. ProblemType Class

PolicyType object is a scheduling environment to control the design of the production system. Scheduling environment settings are explained in Section 4.1. The attributes of a PolicyType object are:

- DDMean (number)
- DDRange (number)
- JobInterarrival (number)
- ProcessingVariability (number)
- UrgentJobRatio (number)
- Utilization (number)
- AvgProcessingTime (expression)

As their names imply JobInterarrival, DDMean, ProcessingVariability, UrgentJobRatio and Utilization parameters are $\lambda, \tau, \gamma, \phi$ and $\rho$ respectively. DDRange parameter determines the range of the interval that the due date of a job is generated. Utilization is multiplied with JobInterarrival to obtain AvgProcessingTime.

## B.6. Simulator Class

The attributes of a Simulator object are:

- CurrentReplicationIndex (number)
- CompletedJobs (number)
- NoOfUrgentJobs (number)
- Seed (number)
- CurrentTime (time)
- SchedulingPeriodStartTime (time)
- SchedulingProcessStartTime (time)
- TheCurrentJobInProcess (Job object)
- TheCurrentPolicyType (Policy Type object)
- TheCurrentProblemSetting (Problem Setting object)
- TheCurrentProblemType (Problem Type object)
- Events (a list of event objects)
- Jobs (a list of job objects)
- PolicyTypes (a list of Policy Type objects)
- Problem Types (a list of Policy Type objects)
- ArrivedJobs (expression)
- CompletedJobsList (expression)

NoOfUrgentJobs is used to count the number of urgent jobs during simulation. The ratio of NoOfUrgentJobs to MaxNoOfJobsArrived of TheCurrentProblemSetting object should be close to UrgentJobRatio of TheCurrentProblemType object for the verification of simulation.

A problem instance can be repeated by using the same seeds to generate random numbers. The experiments can be conducted by repeating specific instances with the help of the Seed attribute. CurrentReplicationIndex is used to control the replication number.

CurrentTime is the current simulation time CST. SchedulingProcessStartTime is the time that a scheduling generation starts i.e. $t^S(\pi)$ to generate $\pi$. SchedulingPeriodStartTime determines the beginning of a schedule. ArrivedJobs expression filter the jobs with ArrivalTime $> 0$ in Jobs list. TheCurrentJobInProcess is $j(t)$. CompletedJobsList expression filter the jobs with RealCompletionTime $> 0$ in Jobs list, and this list is used in statistic calculations. CompletedJobs denotes the value of CompletedJobsList list size. A simulation ends when CompletedJobs reaches the value of MaxNoOfJobsCompleted of TheCurrentProblemSetting object of the simulator.

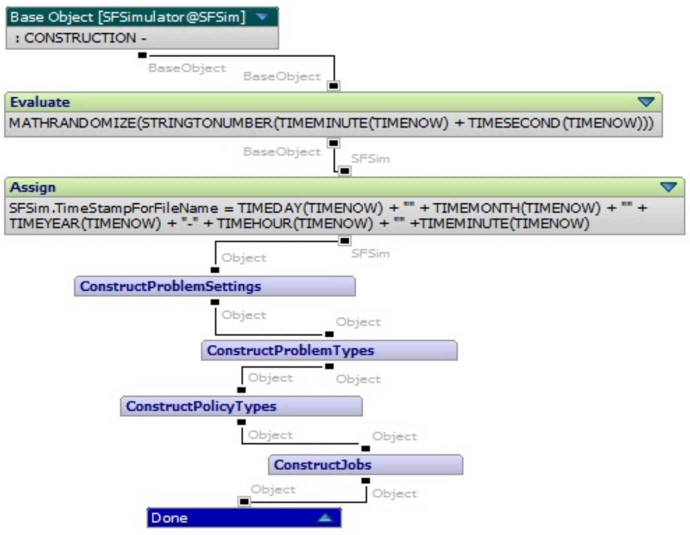

```
MATHRANDOMIZE(STRINGTONUMBER(TIMEMINUTE(TIMENOW) + TIMESECOND(TIMENOW)))
```
```
SFSim.TimeStampForFileName = TIMEDAY(TIMENOW) + "" + TIMEMONTH(TIMENOW) + "" +
TIMEYEAR(TIMENOW) + "-" + TIMEHOUR(TIMENOW) + "" +TIMEMINUTE(TIMENOW)
```

Figure B.12. InitializeOnce Algorithm of Simulator Class.

*InitializeOnce* algorithm presented in Figure B.12 is used to construct the simulator. The data is read from an Access database. ConstructProblemSettings node creates TheCurrentProblemSetting object of the simulator. ConstructProblemTypes and ConstructPolicyTypes nodes create ProblemTypes and PolicyTypes list of the simulator. The list size of Jobs is determined by MaxNumberOfJobsArrived parameter of TheCurrentProblemSetting of the simulator and created by ConstructJobs node.

*CreateNewJobEvent*, *CreateOperationCompletedEvent*, *CreateSchedulngCompleted* and *CreateWakeupToDispatchEvent* algorithms presented in Figure B.13, B.14, B.15, B.16 respectively are used to create consecutive events during simulation. In event creation algorithms, the common part of the algorithms starts from "Create Object info” node, and ends with the “Done” node. As its name implies, “Create Object” info node creates necessary information object. “Create Objects From Object Info” node use this information object and a sample object to create a new object. In order to create an Event Object, Time and Code fields are needed. The created event object is put into the Events list of simulator in Create Objects From Object Info node.

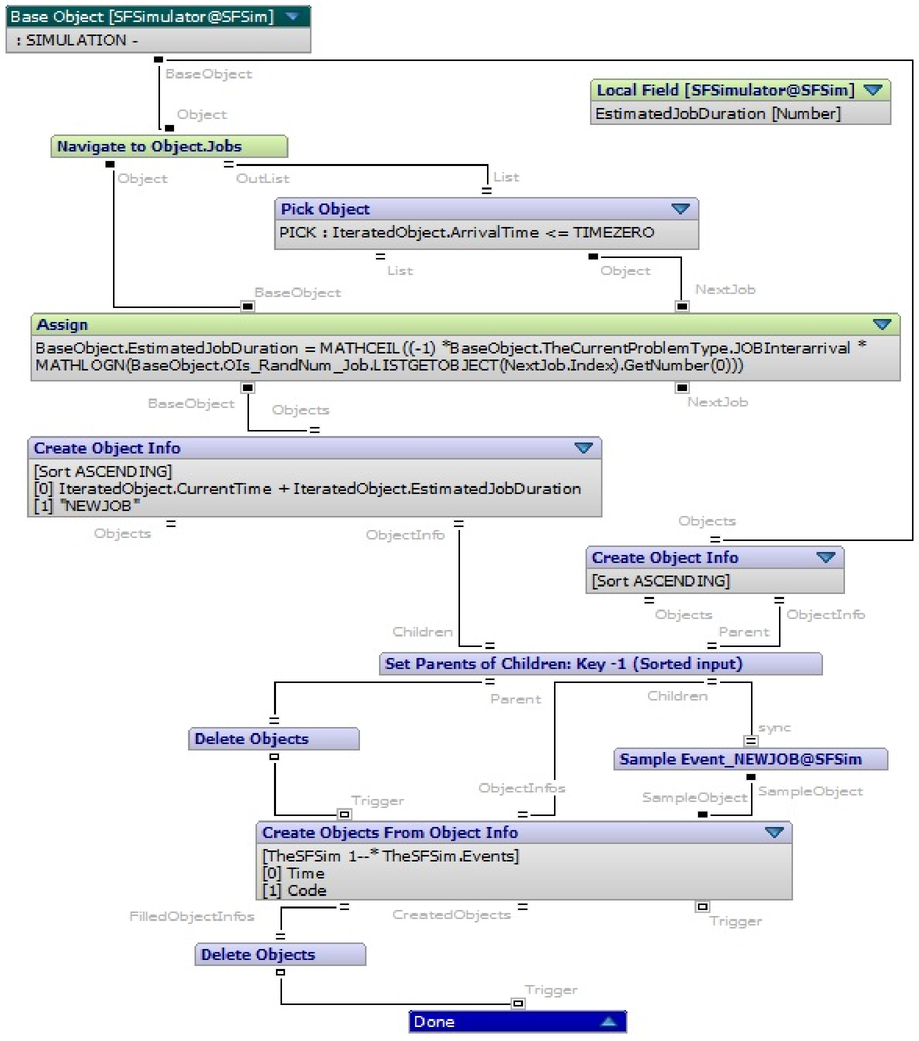

Base Object [SFSimulator@SFSim]
: SIMULATION -
BaseObject
Object
Local Field [SFSimulator@SFSim]
EstimatedJobDuration [Number]
Navigate to Object.Jobs
Object
OutList
List
Pick Object
PICK : IteratedObject.ArrivalTime <= TIMEZERO
List
Object
NextJob
BaseObject
Assign
BaseObject.EstimatedJobDuration = MATHCEIL((-1) *BaseObject.TheCurrentProblemType.JOBInterarrival * MATHLOGN(BaseObject.OIs_RandNum_Job.LISTGETOBJECT(NextJob.Index).GetNumber(0)))
BaseObject
Objects
NextJob
Create Object Info
[Sort ASCENDING]
[0] IteratedObject.CurrentTime + IteratedObject.EstimatedJobDuration
[1] "NEWJOB"
Objects
ObjectInfo
Objects
Create Object Info
[Sort ASCENDING]
Objects
ObjectInfo
Children
Parent
Set Parents of Children: Key -1 (Sorted input)
Parent
Children
sync
Delete Objects
Sample Event_NEWJOB@SFSim
ObjectInfos
Trigger
SampleObject
SampleObject
Create Objects From Object Info
[TheSFSim 1--* TheSFSim.Events]
[0] Time
[1] Code
FilledObjectInfos
CreatedObjects
Trigger
Delete Objects
Trigger
Done

Figure B.13. CreateNewJobEvent Algorithm of Simulator Class.

In Figure B.13, the simulator navigates its jobs list, and pick the first job whose ArrivalTime is zero. Then it calculates next arrival time by using $Exponential(\lambda)$ distribution and specify that time as Time of event object. The Code of the event is set to “NEWJOB”.

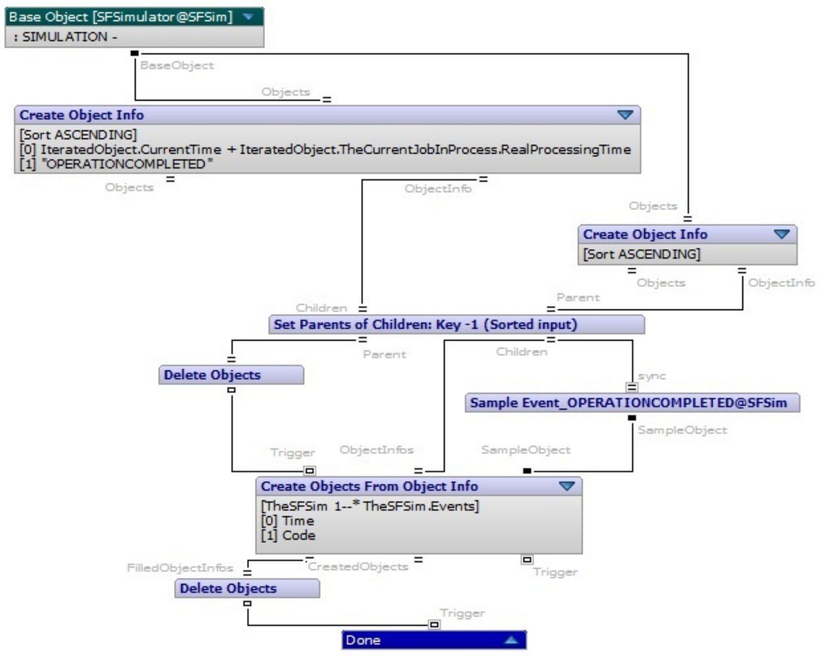


Figure B.14. CreateOperationCompletedEvent Algorithm of Simulator Class.

In Figure B.14, the simulator determines Time attribute of event object by adding RealProcessingTime value of TheCurrentJobInProcess object of the simulator to its CurrentTime field. The Code of the event is set to “OPERATIONCOMPLETED”.

*CreateSchedulngCompleted* algorithm is shown in Figure B.15. The simulator determines Time of event object by adding DecisionTime value of TheCurrentPolicyType object of the simulator to its SchedulingProcessStartTime. The Code of the event is set to “SCHEDULINGCOMPLETED”.

As shown in Figure B.16, *CreateWakeupToDispatchEvent* algorithm requires a time type input. Time of Event-Wakeup object is set to the input value. The Code of the event is assigned as “WAKEUPTODISPATCH”.

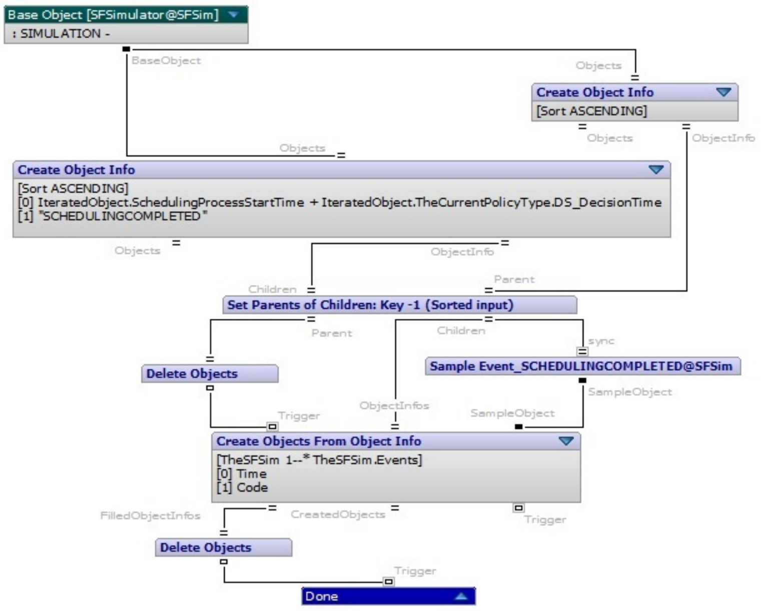


Figure B.15. CreateSchedulingCompletedEvent Algorithm of Simulator Class.

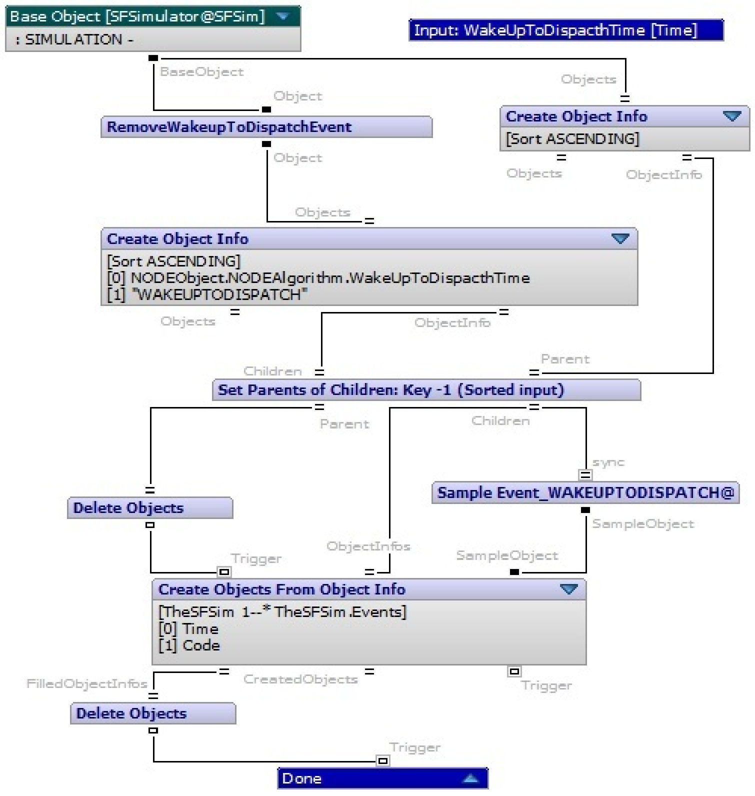


Figure B.16. CreateWakeupToDispatchEvent Algorithm of Simulator Class.

### B.6.1. Scheduling Process Algorithms

*Schedule* algorithm common in both of CSSS and DSSS, is presented in Figure B.17. In the first step, SchedulableJobs list is received. If there is no job to schedule then, the simulator calls *Dispatch* algorithm given in Figure B.19. If SchedulableJobs list is full, then the simulator calls *StartScheduling* algorithm. When *StartScheduling* algorithm is done, it is checked if the current operating mode takes time. If it is so, then *CreateSchedulingCompleted* algorithm in Figure B.15 is run, else generated schedule is released immediately by *ReleaseLatestSchedule* algorithm. At the end of *Schedule* algorithm, *Dispatch* algorithm in Figure B.19 is called. *Schedule* algorithm is done when *Dispatch* algorithm is done.

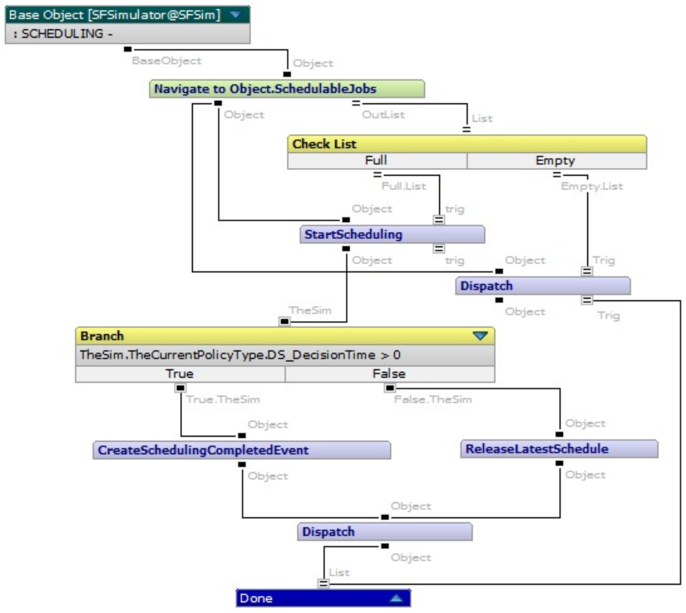


Figure B.17. Schedule Algorithm of Simulator Class.

*StartScheduling* and *ReleaseLatestSchedule* algorithms are different in CSSS and DSSS implementations. *StartScheduling* and *ReleaseLatestSchedule* algorithms in CSSS presented respectively in Figure B.20 and B.21 are explained in Section B.6.3.1. *StartScheduling* and *ReleaseLatestSchedule* algorithms in DSSS are shown respectively in Figure B.28 and B.30 are explained in Section B.6.4.1.

Scheduling algorithms are *RandomAlgorithm*, *Optimization* and *OptimalTiming* algorithms. As explained in Section 4.2, the optimum solution to the scheduling problem we considered is obtained by solving a mixed integer program. We solve the mixed integer program in by *Optimization* algorithm. As we mentioned at the beginning of this chapter, ICRON uses the latest available versions of CPLEX and COIN solvers. We use CPLEX solver in emphOptimization algorithm.

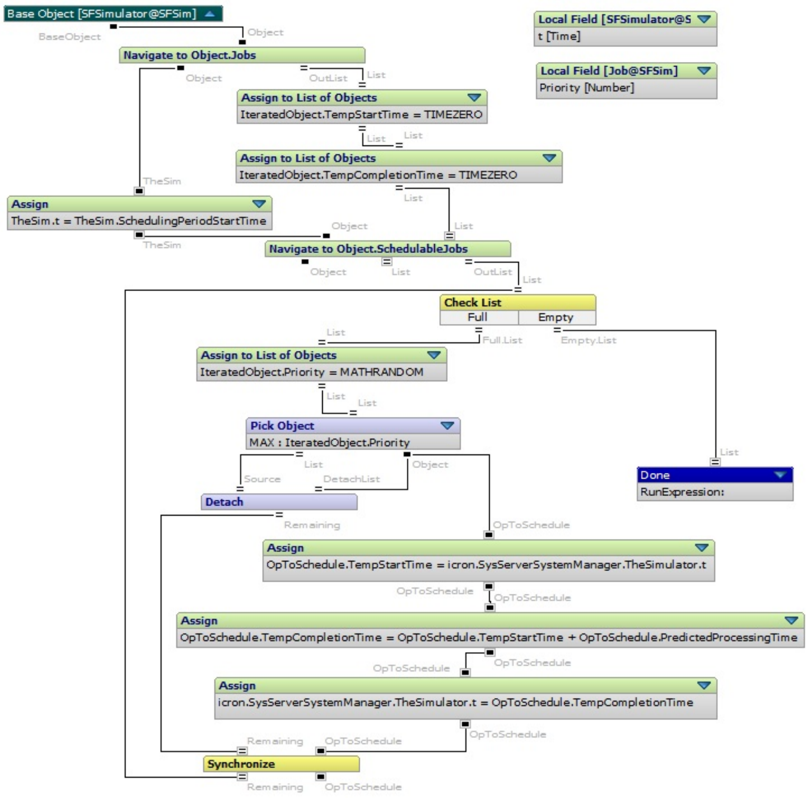

Base Object [SFSimulator@SFSim]
BaseObject
Object
Navigate to Object.Jobs
Object
OutList
List
Assign to List of Objects
IteratedObject.TempStartTime = TIMEZERO
List
List
Assign to List of Objects
IteratedObject.TempCompletionTime = TIMEZERO
List
TheSim
Assign
TheSim.t = TheSim.SchedulingPeriodStartTime
TheSim
Object
List
Navigate to Object.SchedulableJobs
Object
List
OutList
List
Check List
Full
Empty
List
Full.List
Empty.List
Assign to List of Objects
IteratedObject.Priority = MATHRANDOM
List
List
Pick Object
MAX : IteratedObject.Priority
List
Object
Source
DetachList
Detach
Remaining
OpToSchedule
Assign
OpToSchedule.TempStartTime = icron.SysServerSystemManager.TheSimulator.t
OpToSchedule
OpToSchedule
Assign
OpToSchedule.TempCompletionTime = OpToSchedule.TempStartTime + OpToSchedule.PredictedProcessingTime
OpToSchedule
OpToSchedule
Assign
icron.SysServerSystemManager.TheSimulator.t = OpToSchedule.TempCompletionTime
OpToSchedule
Remaining
OpToSchedule
Synchronize
Remaining
OpToSchedule
Local Field [SFSimulator@S
t [Time]
Local Field [Job@SFSim]
Priority [Number]
List
Done
RunExpression:

Figure B.18. RandomAlgorithm of Simulator Class.

We devise *RandomAlgorithm* as a heuristic. In *RandomAlgorithm*, we generate a random sequence of jobs. Given a sequence mixed integer program becomes a linear program. *OptimalTiming* algorithm solves the linear program to optimally insert idle times to generate a schedule.

*Optimization* and *OptimalTiming* algorithms are not presented in figures since their implementations include lots of details specific to ICRON software. *RandomAlgorithm* is presented in Figure B.18.

### B.6.2. Dispatch Process Algorithms

*Dispatch* algorithm common in CSSS and DSSS, is presented in Figure B.19. At the beginning of the algorithm it is check if TheCurrentJobInProcess exists or not. If it exists, then Figure B.19 is done without doing anything, else one of the *DispatchBeforePivot* or *DispatchAfterPivot* algorithms is invoked according to Pivot value of TheCurrentProblemSetting.

We collect statistics over the CompletedJobsList of the simulator. When CompletedJobs attribute to denote CompletedJobsList list size reaches Pivot value, we start to collect statistics. In *DispatchBeforePivot* algorithm statistics are ignored. *DispatchAfterPivot* algorithm is extended version of *DispatchBeforePivot* algorithm since it includes the calculation of some of the statistics.

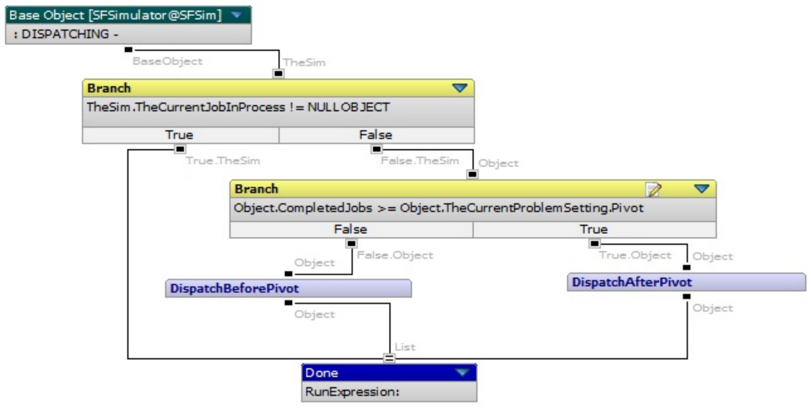


Figure B.19. Dispatch Algorithm of Simulator Class.

*DispatchBeforePivot* and *DispatchAfterPivot* algorithms are different in CSSS and DSSS implementations. They are explained in Section B.6.3.2  B.6.4.2 according to the type of simulator.

### B.6.3. Simulator Class for CSSS

The additional attributes in CSSS implementation are:

- IsScheduling (number)
- DecisionIdleness (number)
- ForcedIdleness (number)
- QueueIdleness (number)
- DecisionIdlenessStartTime (time)
- ForcedIdlenessStartTime (time)
- QueueIdlenessStartTime (time)
- SchedulableJobs (expression)

IsScheduling is used to control if the scheduler is in a schedule generation process or not. SchedulableJobs expression filters the jobs with RealStartTime <= 0 in ArrivedJobs list to obtain $J^S$.

DecisionIdlenessStartTime is assigned to simulation time when the machine is turn in DI state. The difference between the time when DI state ends and DecisionIdlenessStartTime determines DecisionIdleness value. DecisionIdleness is accumulated during each simulation. The ratio of DecisionIdleness to the simulation period $\Gamma$ gives DI ratio. The calculations are similar for ForcedIdleness and QueueIdleness attributes.

<u>B.6.3.1. Scheduling Process Algorithms in CSSS.</u> *StartScheduling* and *ReleaseLatestSchedule* algorithms are shown in Figure B.20 and B.21 respectively.

In schedule generation process, the scheduled times are determined by using TempCompletionTime, TempStartTime variables of each job in *StartScheduling* algorithm. At the beginning of *StartScheduling* algorithm, IsScheduling field of the simulator is set to 1. SchedulingProcessStartTime is set to CurrentTime. If the current operating mode is busy mode, then ReleasedStartTime and ReleasedCompletionTime attributes of each job is set to zero, hence the latest available scheduled times are removed. If TheCurrentJobInProcess exists then SchedulingPeriodStartTime is adjusted. Then according to current control policy, either *RandomAlgorithm* in Figure B.18 and *OptimalTiming* are run consecutively or *Optimization* algorithm is called.

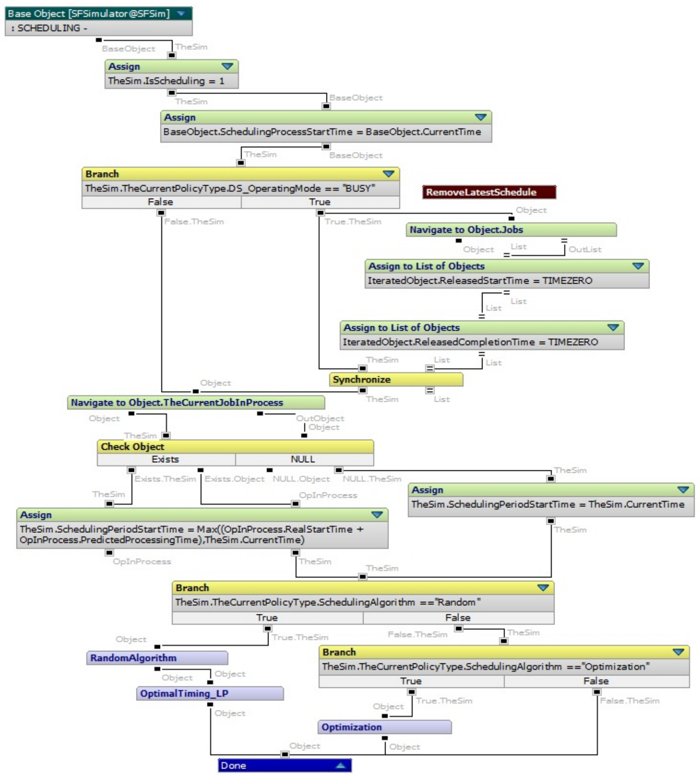

Base Object [SFSimulator@SFSim]
: SCHEDULING -
Assign
TheSim.IsScheduling = 1
Assign
BaseObject.SchedulingProcessStartTime = BaseObject.CurrentTime
Branch
TheSim.TheCurrentPolicyType.DS_OperatingMode == "BUSY"
False
True
RemoveLatestSchedule
Navigate to Object.Jobs
Assign to List of Objects
IteratedObject.ReleasedStartTime = TIMEZERO
Assign to List of Objects
IteratedObject.ReleasedCompletionTime = TIMEZERO
Synchronize
Navigate to Object.TheCurrentJobInProcess
Check Object
Exists
NULL
Assign
TheSim.SchedulingPeriodStartTime = TheSim.CurrentTime
Assign
TheSim.SchedulingPeriodStartTime = Max((OpInProcess.RealStartTime + OpInProcess.PredictedProcessingTime),TheSim.CurrentTime)
Branch
TheSim.TheCurrentPolicyType.SchedulingAlgorithm =="Random"
True
False
RandomAlgorithm
OptimalTiming_LP
Branch
TheSim.TheCurrentPolicyType.SchedulingAlgorithm =="Optimization"
True
False
Optimization
Done

Figure B.20. StartScheduling Algorithm of Simulator Class in CSSS Implementation.

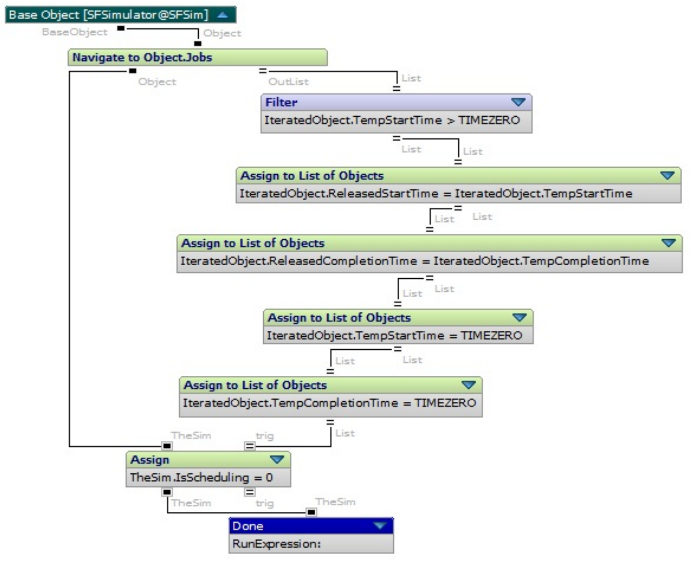

Base Object [SFSimulator@SFSim]
Navigate to Object.Jobs
Filter
IteratedObject.TempStartTime > TIMEZERO
Assign to List of Objects
IteratedObject.ReleasedStartTime = IteratedObject.TempStartTime
Assign to List of Objects
IteratedObject.ReleasedCompletionTime = IteratedObject.TempCompletionTime
Assign to List of Objects
IteratedObject.TempStartTime = TIMEZERO
Assign to List of Objects
IteratedObject.TempCompletionTime = TIMEZERO
Assign
TheSim.IsScheduling = 0
Done
RunExpression:

Figure B.21. ReleaseLatestSchedule Algorithm of Simulator Class in CSSS Implementation.

In schedule release process ReleasedCompletionTime and ReleasedStartTime variables are assigned to TempCompletionTime, TempStartTime variables respectively and IsScheduling field is set to zero by *ReleaseLatestSchedule* algorithm.

<u>B.6.3.2. Dispatch Process Algorithms in CSSS.</u> *DispatchBeforePivot* and *DispatchAfterPivot* algorithms are the implementation of the Dispatch algorithm given in Section 3.1.2. *DispatchAfterPivot* in Figure B.23 algorithm includes the same steps of *DispatchBeforePivot* in Figure B.22 algorithm and additional statistics calculation steps.

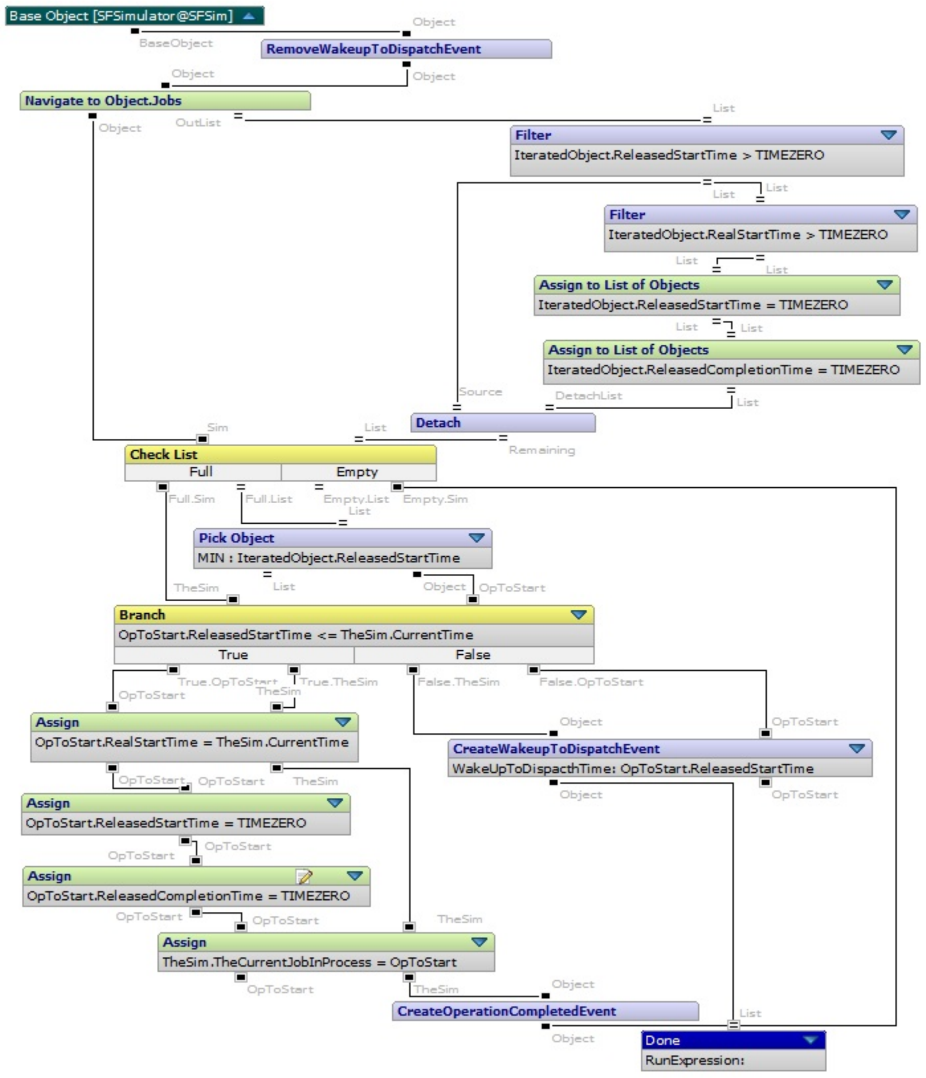
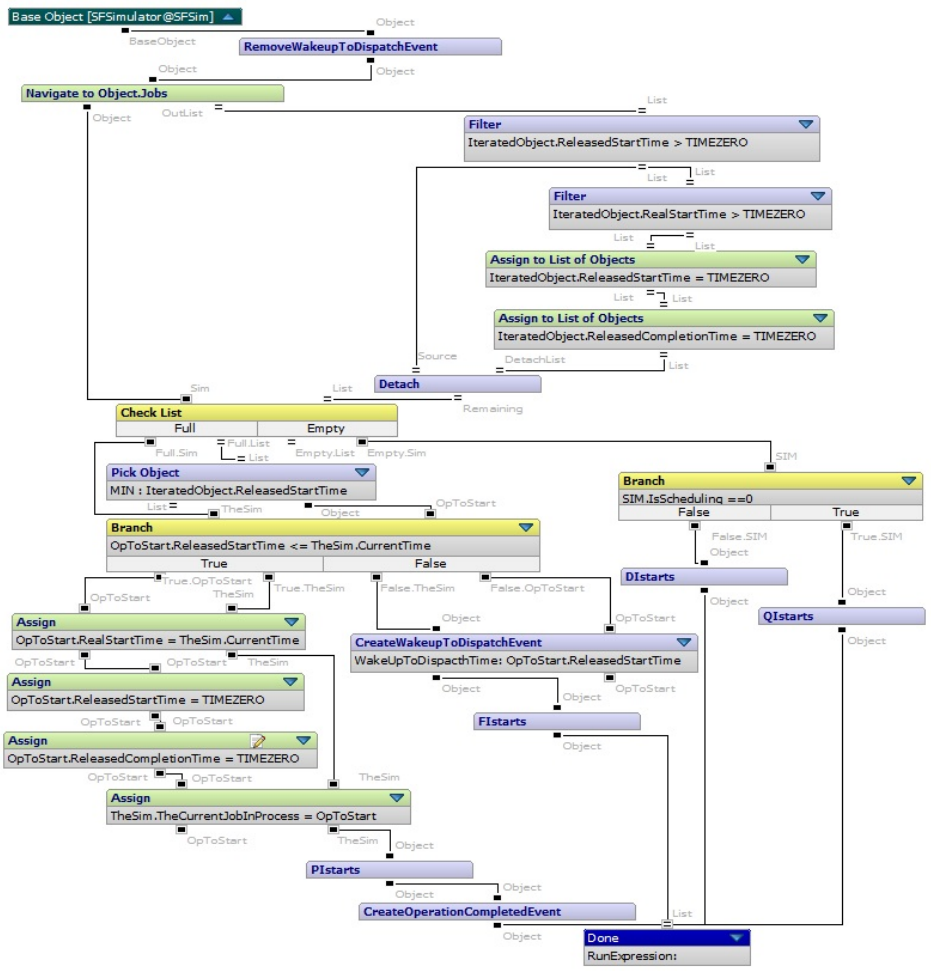

Base Object [SFSimulator@SFSim]
RemoveWakeupToDispatchEvent
Navigate to Object.Jobs
Filter
IteratedObject.ReleasedStartTime > TIMEZERO
Filter
IteratedObject.RealStartTime > TIMEZERO
Assign to List of Objects
IteratedObject.ReleasedStartTime = TIMEZERO
Assign to List of Objects
IteratedObject.ReleasedCompletionTime = TIMEZERO
Detach
Check List
Full
Empty
Pick Object
MIN : IteratedObject.ReleasedStartTime
Branch
OpToStart.ReleasedStartTime <= TheSim.CurrentTime
True
False
Assign
OpToStart.RealStartTime = TheSim.CurrentTime
CreateWakeupToDispatchEvent
WakeUpToDispacthTime: OpToStart.ReleasedStartTime
Assign
OpToStart.ReleasedStartTime = TIMEZERO
Assign
OpToStart.ReleasedCompletionTime = TIMEZERO
Assign
TheSim.TheCurrentJobInProcess = OpToStart
CreateOperationCompletedEvent
Done
RunExpression:

Figure B.22. DispatchBeforePivot Algorithm of Simulator Class in CSSS Implementation.

In the first step of the algorithm displayed in Figure B.23, *RemoveWakeupToDispatchEvent* algorithm is called, i.e. if there is an active WakeupToDispatchEvent in Events list of the simulator, it is deleted. The simulator navigates to Jobs list. It filters the jobs that has positive ReleasedStartTime. Then among those scheduled jobs, jobs with positive RealStartTime are filtered. These are the jobs exist in the current schedule but they are already started, hence they are excluded from the schedule by assigning their ReleasedStartTime and ReleasedCompletionTime fields to zero.



Base Object [SFSimulator@SFSim]
RemoveWakeupToDispatchEvent
Navigate to Object.Jobs
Filter
IteratedObject.ReleasedStartTime > TIMEZERO
Filter
IteratedObject.RealStartTime > TIMEZERO
Assign to List of Objects
IteratedObject.ReleasedStartTime = TIMEZERO
Assign to List of Objects
IteratedObject.ReleasedCompletionTime = TIMEZERO
Detach
Check List
Full
Empty
Pick Object
MIN : IteratedObject.ReleasedStartTime
Branch
OpToStart.ReleasedStartTime <= TheSim.CurrentTime
True
False
Branch
SIM.IsScheduling ==0
False
True
DIstarts
QIstarts
Assign
OpToStart.RealStartTime = TheSim.CurrentTime
Assign
OpToStart.ReleasedStartTime = TIMEZERO
Assign
OpToStart.ReleasedCompletionTime = TIMEZERO
Assign
TheSim.TheCurrentJobInProcess = OpToStart
CreateWakeupToDispatchEvent
WakeUpToDispacthTime: OpToStart.ReleasedStartTime
FIstarts
PIstarts
CreateOperationCompletedEvent
Done
RunExpression:

Figure B.23. DispatchAfterPivot Algorithm of Simulator Class in CSSS Implementation.

If the remaining list is full, the job with minimum ReleasedStartTime is picked as a candidate to dispatch. If ReleasedStartTime of the selected job is less than or equal to CurrentTime, then RealStartTime is set to CurrentTime, ReleasedStartTime and ReleasedCompletionTime are set to zero and TheCurrentJobInProcess is assigned to selected job. *PStarts* in Figure B.24 and *CreateOperationCompletedEvent* in Figure B.14 are called consecutively, then the algorithm is done. If ReleasedStartTime of the selected job is greater than CurrentTıme, *CreateWakeupToDispatchEvent* in Figure B.16 and *FIStarts* in Figure B.25 are executed consecutively, then the algorithm is done.

If the remaining list after the shop floor realizations is empty, and if IsScheduling is true then *DIStarts* algorithm in Figure B.26 is executed, else *QIStarts* algorithm in Figure B.27 is executed.

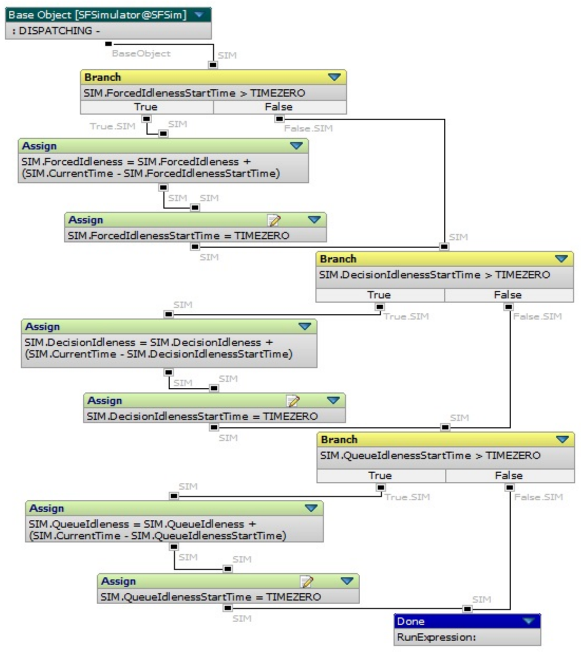

Base Object [SFSimulator@SFSim]
: DISPATCHING -
BaseObject
SIM
Branch
SIM.ForcedIdlenessStartTime > TIMEZERO
True
False
True.SIM
False.SIM
Assign
SIM.ForcedIdleness = SIM.ForcedIdleness + (SIM.CurrentTime - SIM.ForcedIdlenessStartTime)
Assign
SIM.ForcedIdlenessStartTime = TIMEZERO
Branch
SIM.DecisionIdlenessStartTime > TIMEZERO
Assign
SIM.DecisionIdleness = SIM.DecisionIdleness + (SIM.CurrentTime - SIM.DecisionIdlenessStartTime)
Assign
SIM.DecisionIdlenessStartTime = TIMEZERO
Branch
SIM.QueueIdlenessStartTime > TIMEZERO
Assign
SIM.QueueIdleness = SIM.QueueIdleness + (SIM.CurrentTime - SIM.QueueIdlenessStartTime)
Assign
SIM.QueueIdlenessStartTime = TIMEZERO
Done
RunExpression:

Figure B.24. PStarts Algorithm of Simulator Class in CSSS Implementation.

*PStarts*, *FIStarts*, *DIStarts*, *QIStarts* algorithms are used to calculate the duration of time that the machine is in P, FI, DI, and QI states respectively. The machine may be in one of these states at any time. Hence when one of the machine state starts, the previous state ends by these algorithms.

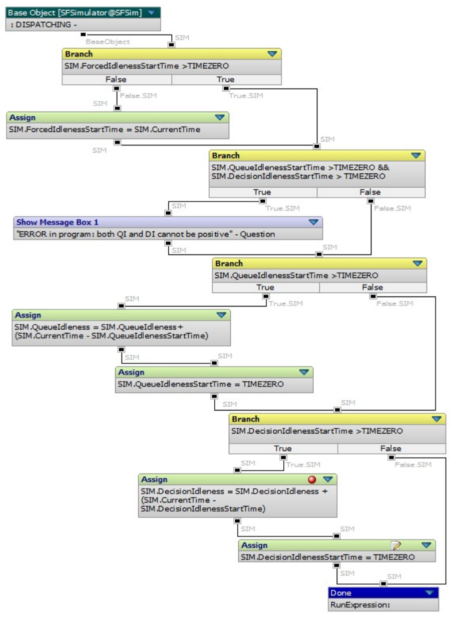

Base Object [SFSimulator@SFSim]
: DISPATCHING -
BaseObject
SIM
Branch
SIM.ForcedIdlenessStartTime >TIMEZERO
False
True
False.SIM
True.SIM
SIM
Assign
SIM.ForcedIdlenessStartTime = SIM.CurrentTime
SIM
SIM
Branch
SIM.QueueIdlenessStartTime >TIMEZERO &&
SIM.DecisionIdlenessStartTime > TIMEZERO
True
False
SIM
True.SIM
False.SIM
Show Message Box 1
"ERROR in program: both QI and DI cannot be positive" - Question
SIM
SIM
Branch
SIM.QueueIdlenessStartTime >TIMEZERO
True
False
SIM
True.SIM
False.SIM
Assign
SIM.QueueIdleness = SIM.QueueIdleness+
(SIM.CurrentTime - SIM.QueueIdlenessStartTime)
SIM
SIM
Assign
SIM.QueueIdlenessStartTime = TIMEZERO
SIM
SIM
Branch
SIM.DecisionIdlenessStartTime >TIMEZERO
True
False
SIM
True.SIM
False.SIM
Assign
SIM.DecisionIdleness = SIM.DecisionIdleness +
(SIM.CurrentTime -
SIM.DecisionIdlenessStartTime)
SIM
SIM
Assign
SIM.DecisionIdlenessStartTime = TIMEZERO
SIM
SIM
Done
RunExpression:

Figure B.25. FIStarts Algorithm of Simulator Class in CSSS Implementation.

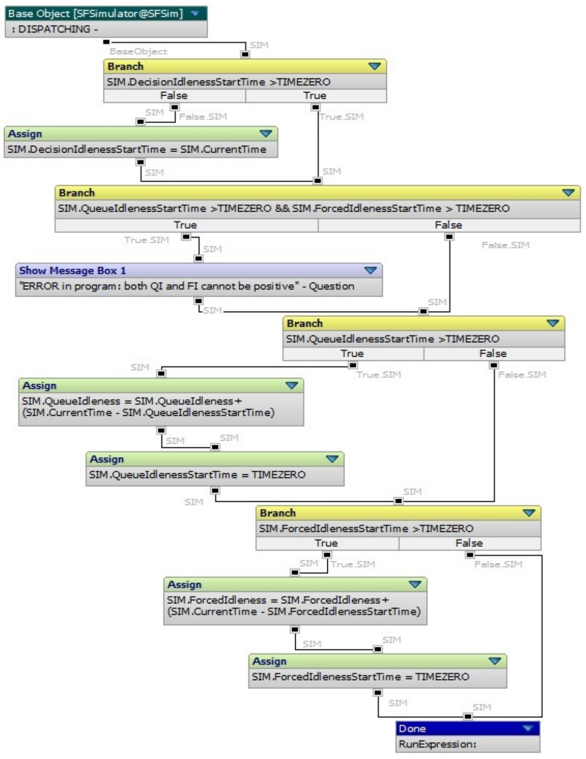

Base Object [SFSimulator@SFSim]
: DISPATCHING -
BaseObject
SIM
Branch
SIM.DecisionIdlenessStartTime >TIMEZERO
False
True
SIM
False.SIM
True.SIM
Assign
SIM.DecisionIdlenessStartTime = SIM.CurrentTime
SIM
SIM
Branch
SIM.QueueIdlenessStartTime >TIMEZERO && SIM.ForcedIdlenessStartTime > TIMEZERO
True
False
True.SIM
SIM
False.SIM
Show Message Box 1
"ERROR in program: both QI and FI cannot be positive" - Question
SIM
SIM
Branch
SIM.QueueIdlenessStartTime >TIMEZERO
True
False
SIM
True.SIM
False.SIM
Assign
SIM.QueueIdleness = SIM.QueueIdleness+
(SIM.CurrentTime - SIM.QueueIdlenessStartTime)
SIM
SIM
Assign
SIM.QueueIdlenessStartTime = TIMEZERO
SIM
SIM
Branch
SIM.ForcedIdlenessStartTime >TIMEZERO
True
False
SIM
True.SIM
False.SIM
Assign
SIM.ForcedIdleness = SIM.ForcedIdleness+
(SIM.CurrentTime - SIM.ForcedIdlenessStartTime)
SIM
SIM
Assign
SIM.ForcedIdlenessStartTime = TIMEZERO
SIM
SIM
Done
RunExpression:

Figure B.26. DIStarts Algorithm of Simulator Class in CSSS Implementation.

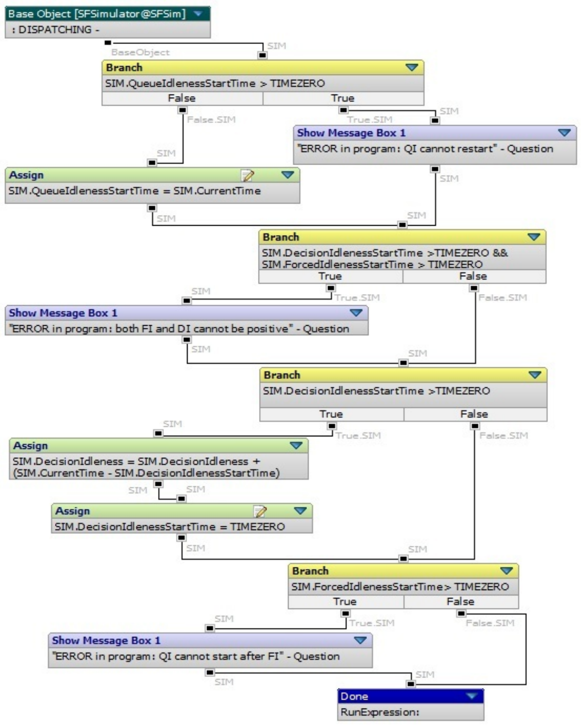

Base Object [SFSimulator@SFSim]
: DISPATCHING -
Branch
SIM.QueueIdlenessStartTime > TIMEZERO
False
True
Show Message Box 1
"ERROR in program: QI cannot restart" - Question
Assign
SIM.QueueIdlenessStartTime = SIM.CurrentTime
Branch
SIM.DecisionIdlenessStartTime >TIMEZERO &&
SIM.ForcedIdlenessStartTime > TIMEZERO
True
False
Show Message Box 1
"ERROR in program: both FI and DI cannot be positive" - Question
Branch
SIM.DecisionIdlenessStartTime >TIMEZERO
True
False
Assign
SIM.DecisionIdleness = SIM.DecisionIdleness +
(SIM.CurrentTime - SIM.DecisionIdlenessStartTime)
Assign
SIM.DecisionIdlenessStartTime = TIMEZERO
Branch
SIM.ForcedIdlenessStartTime> TIMEZERO
True
False
Show Message Box 1
"ERROR in program: QI cannot start after FI" - Question
Done
RunExpression:

Figure B.27. QIStarts Algorithm of Simulator Class in CSSS Implementation.

### B.6.4. Simulator Class for DSSS

The additional attributes in DSSS implementation are:

- IsAScheduling (number)
- IsBScheduling (number)
- ReleaseType (string)
- TheCurrentSchedulerType (string)
- SchedulableJobs (expression)
- SynchronizationTime (expression)
- startAstartB
- startAwaitB
- waitAstartB
- startAnullB
- nullAstartB

- startAB
- waitAwaitB
- waitAnullB
- nullAwaitB
- waitAB

IsAScheduling and IsBScheduling are used to control the schedule generation process of the scheduler A and B respectively. SynchronizationTime expression returns $t^{synch}$. If TheCurrentSchedulerType is "A", SchedulableJobs expression returns schedulable job set of the scheduler A, $J_t^A$. Else TheCurrentSchedulerType is "B" and SchedulableJobs expression returns $J_t^B$. (Please refer Section 3.2.1 for details.)

Dispatch decisions startAstartB, startAwaitB, waitAstartB, startAnullB, nullAstartB and startAB and wait decisions waitA,waitB, waitAnullB, nullAwaitB, waitAB are defined in Section 3.2.1.

#### B.6.4.1. Scheduling Process Algorithms in DSSS.

*StartScheduling*, and *ReleaseLatestSchedule* algorithms are shown in Figure B.28 and B.30 respectively.

In Figure B.28, at the beginning of *StartScheduling* algorithm, IsSchedulingA or IsSchedulingB field of the simulator is set to 1 according to TheCurrentSchedulerType. SchedulingProcessStartTime is set to CurrentTime. If the current operating mode is busy mode, then *RemoveLatestSchedule* algorithm deletes ReleasedStartTimeA and ReleasedCompletionTimeA or ReleasedStartTimeB and ReleasedCompletionTimeB according toTheCurrentSchedulerType. If TheCurrentJobInProcess exists then SchedulingPeriodStartTime is adjusted. Then according to TheCurrentPolicyType, either *RandomAlgorithm* in Figure B.18 then *OptimalTiming* or *Optimization* is called.

In schedule generation process, *RandomAlgorithm* or *Optimization* uses TempCompletionTime, TempStartTime variables of each job to calculate the scheduled

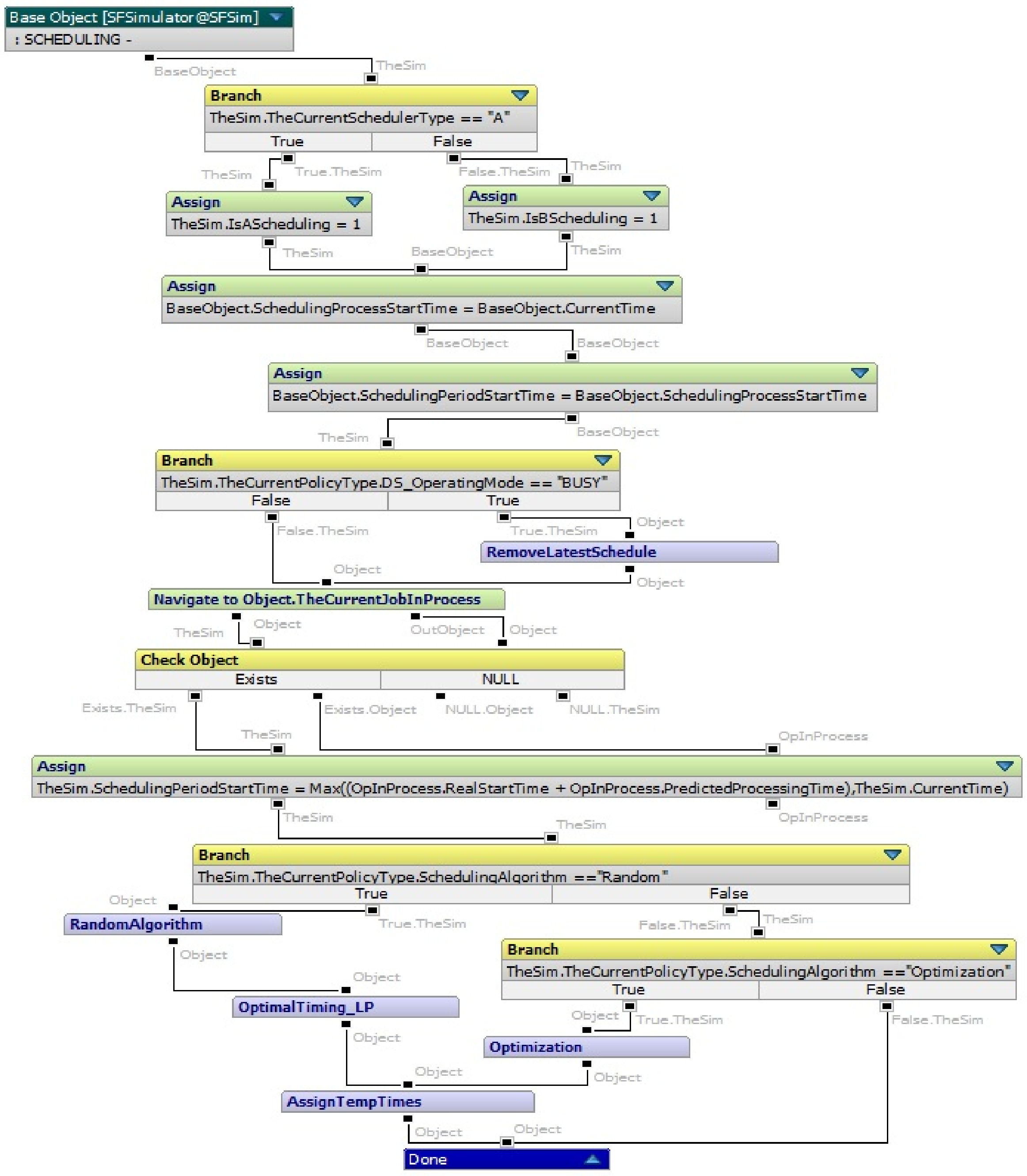


Figure B.28. StartScheduling Algorithm of Simulator Class in DSSS Implementation.

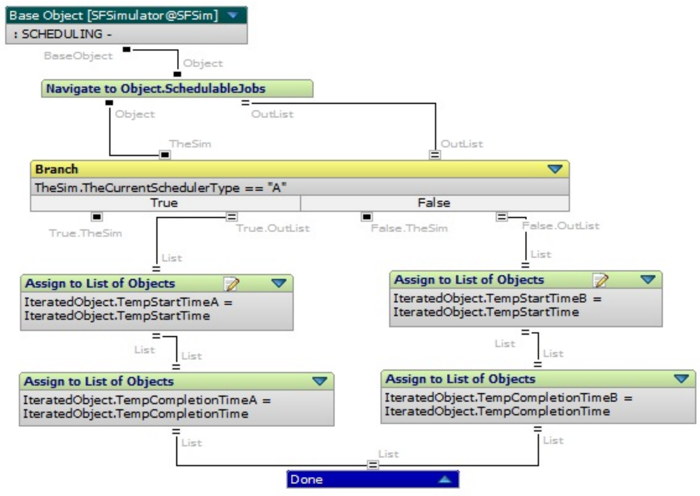

Base Object [SFSimulator@SFSim]
: SCHEDULING -
BaseObject
Object
Navigate to Object.SchedulableJobs
Object
OutList
TheSim
OutList
Branch
TheSim.TheCurrentSchedulerType == "A"
True
False
True.TheSim
True.OutList
False.TheSim
False.OutList
List
List
Assign to List of Objects
IteratedObject.TempStartTimeA = IteratedObject.TempStartTime
Assign to List of Objects
IteratedObject.TempStartTimeB = IteratedObject.TempStartTime
List
List
List
List
Assign to List of Objects
IteratedObject.TempCompletionTimeA = IteratedObject.TempCompletionTime
Assign to List of Objects
IteratedObject.TempCompletionTimeB = IteratedObject.TempCompletionTime
List
List
List
Done

Figure B.29. AssignTempTimes Algorithm of Simulator Class in DSSS Implementation.

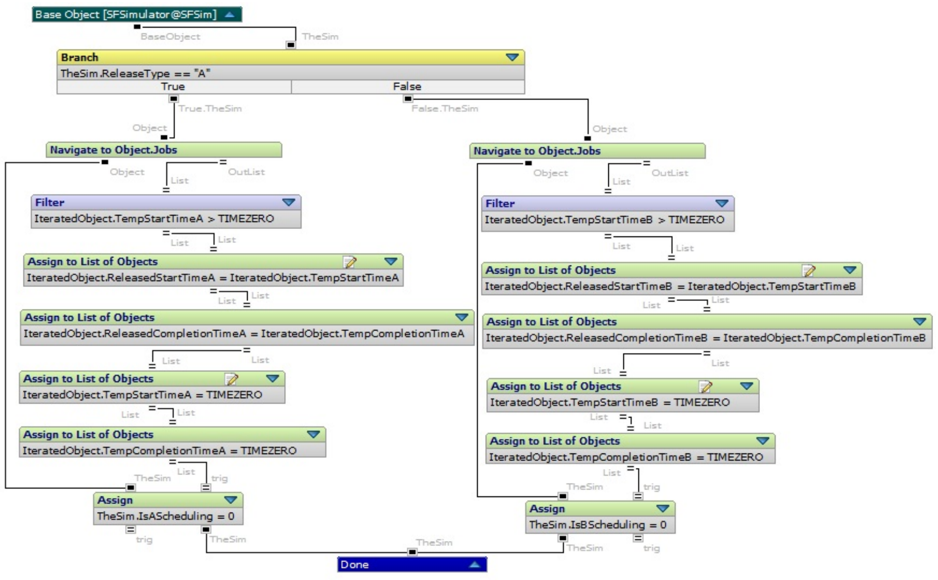

Base Object [SFSimulator@SFSim]
BaseObject
TheSim
Branch
TheSim.ReleaseType == "A"
True
False
True.TheSim
False.TheSim
Object
Object
Navigate to Object.Jobs
Navigate to Object.Jobs
Object
OutList
Object
OutList
List
List
Filter
IteratedObject.TempStartTimeA > TIMEZERO
Filter
IteratedObject.TempStartTimeB > TIMEZERO
List
List
Assign to List of Objects
IteratedObject.ReleasedStartTimeA = IteratedObject.TempStartTimeA
Assign to List of Objects
IteratedObject.ReleasedStartTimeB = IteratedObject.TempStartTimeB
Assign to List of Objects
IteratedObject.ReleasedCompletionTimeA = IteratedObject.TempCompletionTimeA
Assign to List of Objects
IteratedObject.ReleasedCompletionTimeB = IteratedObject.TempCompletionTimeB
Assign to List of Objects
IteratedObject.TempStartTimeA = TIMEZERO
Assign to List of Objects
IteratedObject.TempStartTimeB = TIMEZERO
Assign to List of Objects
IteratedObject.TempCompletionTimeA = TIMEZERO
Assign to List of Objects
IteratedObject.TempCompletionTimeB = TIMEZERO
TheSim
List
trig
Assign
TheSim.IsAScheduling = 0
Assign
TheSim.IsBScheduling = 0
trig
TheSim
TheSim
TheSim
trig
Done

Figure B.30. ReleaseLatestSchedule Algorithm of Simulator Class in DSSS Implementation.

times. TempCompletionTime, TempStartTime variables are assigned to either TempStartTimeA, TempCompletionTimeA variables or TempStartTimeB, TempCompletionTimeB variables according to the type of the scheduler by using *AssignTempTimes*.

In schedule release process, if ReleaseType is “A”, then ReleasedCompletionTimeA and ReleasedStartTimeA variables are assigned to TempCompletionTimeA, TempStartTimeA variables respectively and IsSchedulingA field is set to zero by *ReleaseLatestSchedule* algorithm. Otherwise“B” type attributes are assigned similarly.

B.6.4.2. Dispatch Process Algorithms in DSSS. *DispatchAfterPivot* algorithm includes the same steps of *DispatchBeforePivot* algorithm and additional statistics calculation steps for machine states. Since machine states results obtained in DSSS are not presented, only *DispatchBeforePivot* algorithm is explained in this section.

*DispatchBeforePivot* in DSSS is presented in Figure B.31. If there is an active WakeupToDispatchEvent in Events list of the simulator, it is deleted by *RemoveWakeupToDispatchEvent* algorithm in the first step. The simulator navigates to ArrivedJobs list. Note that, a job is scheduled if it has positive ReleasedStartTimeA and/or ReleasedStartTimeB. Jobs exist in at least one of the schedules are filtered and among them the jobs with positive RealStartTime are eliminated.

If the remaining list after the shop floor realizations is empty, then the algorithm is done. If the remaining list is full, *SelectTheJobToStart* algorithm is called. The figure of *SelectTheJobToStart* algorithm is not presented because it is very crowed and unclear.

In *SelectTheJobToStart* algorithm, in the first step, jobs with ReleasedStartTimeA are filtered, and the job with minimum ReleasedStartTimeA is picked, hence the candidate of $\alpha$ is received. The candidate of $\beta$ is received in a similar way. If both of the schedules offer the same candidate, this job is the output object of the algorithm. Else there are two different candidates. When *SelectTheJobToStart* is executed, one

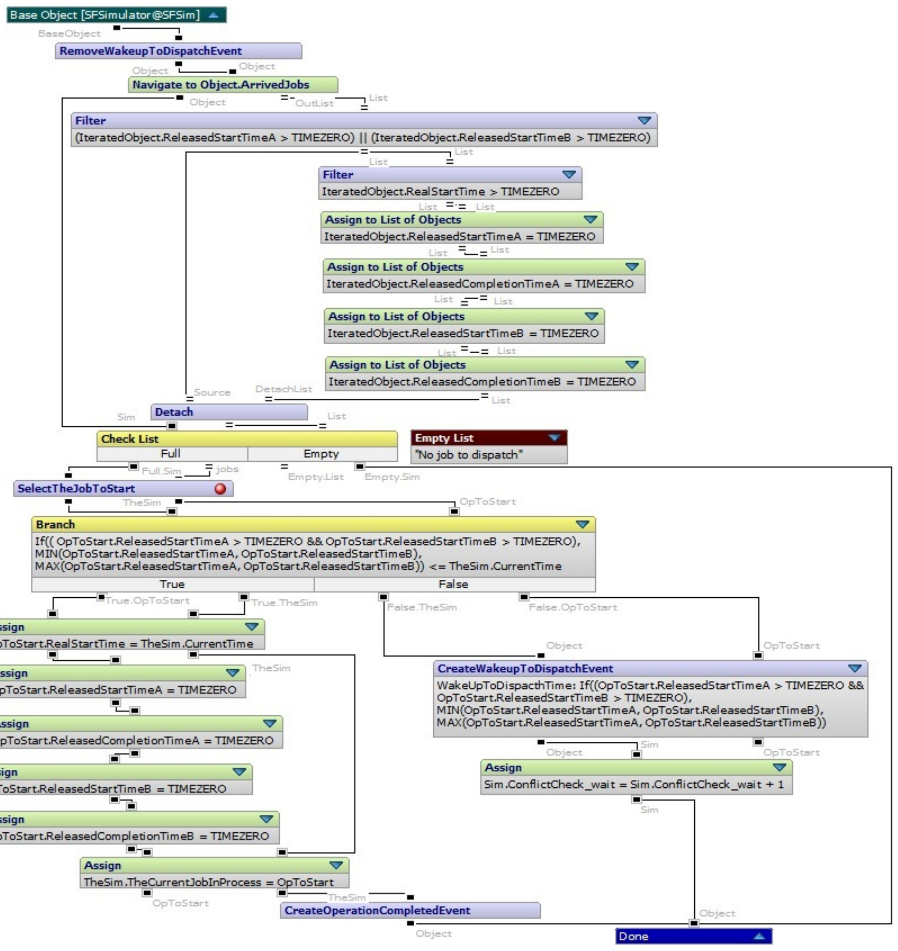

Base Object [SFSimulator@SFSim]
BaseObject
RemoveWakeupToDispatchEvent
Object
Object
Navigate to Object.ArrivedJobs
Object
OutList
List
Filter
(IteratedObject.ReleasedStartTimeA > TIMEZERO) || (IteratedObject.ReleasedStartTimeB > TIMEZERO)
List
List
Filter
IteratedObject.RealStartTime > TIMEZERO
List
List
Assign to List of Objects
IteratedObject.ReleasedStartTimeA = TIMEZERO
List
List
Assign to List of Objects
IteratedObject.ReleasedCompletionTimeA = TIMEZERO
List
List
Assign to List of Objects
IteratedObject.ReleasedStartTimeB = TIMEZERO
List
List
Assign to List of Objects
IteratedObject.ReleasedCompletionTimeB = TIMEZERO
List
Source
DetachList
Sim
Detach
List
Check List
Full
Empty
Empty List
"No job to dispatch"
Full.Sim
jobs
Empty.List
Empty.Sim
SelectTheJobToStart
TheSim
OpToStart
Branch
If(( OpToStart.ReleasedStartTimeA > TIMEZERO && OpToStart.ReleasedStartTimeB > TIMEZERO),
MIN(OpToStart.ReleasedStartTimeA, OpToStart.ReleasedStartTimeB),
MAX(OpToStart.ReleasedStartTimeA, OpToStart.ReleasedStartTimeB)) <= TheSim.CurrentTime
True
False
True.OpToStart
True.TheSim
False.TheSim
False.OpToStart
Assign
OpToStart.RealStartTime = TheSim.CurrentTime
TheSim
Assign
OpToStart.ReleasedStartTimeA = TIMEZERO
Assign
OpToStart.ReleasedCompletionTimeA = TIMEZERO
Assign
OpToStart.ReleasedStartTimeB = TIMEZERO
Assign
OpToStart.ReleasedCompletionTimeB = TIMEZERO
Assign
TheSim.TheCurrentJobInProcess = OpToStart
OpToStart
TheSim
CreateOperationCompletedEvent
Object
Object
OpToStart
CreateWakeupToDispatchEvent
WakeUpToDispacthTime: If((OpToStart.ReleasedStartTimeA > TIMEZERO &&
OpToStart.ReleasedStartTimeB > TIMEZERO),
MIN(OpToStart.ReleasedStartTimeA, OpToStart.ReleasedStartTimeB),
MAX(OpToStart.ReleasedStartTimeA, OpToStart.ReleasedStartTimeB))
Sim
Object
OpToStart
Assign
Sim.ConflictCheck_wait = Sim.ConflictCheck_wait + 1
Sim
Object
Done

Figure B.31. DispatchBeforePivot Algorithm of Simulator Class in DSSS Implementation.

of startAstartB, startAwaitB, waitAstartB, startAnullB, nullAstartB, startAB, waitAwaitB, waitAnullB, nullAwaitB, waitAB variables is increased by one. For example, if there is a single candidate, and if it has greater ReleasedStartTimeA and ReleasedStartTimeB fields than CurrentTime, then waitAB value is increased by one. The output object is determined by executing the steps of the *Dispatch* algorithm given in Section 3.2.3.

If the selected job has a scheduled start time that is less than CurrentTime, then RealStartTime is set to CurrentTime, ReleasedStartTimeA, ReleasedCompletionTimeA, ReleasedStartTimeB, ReleasedCompletionTimeB are set to zero. TheCurrentJobInProcess is assigned to selected job. *CreateOperationCompletedEvent* in Figure B.14 is called. Else *CreateWakeupToDispatchEvent* in Figure B.16 is executed. The algorithm is done.

## B.7. Operation of the Simulator

*Simulate* algorithm displayed in Figure B.32 manages the creation and operation of the simulator. In the first step, a new simulator object is created and InitializeOnce algorithm presented in Figure B.12 is used to construct the simulator. After the construction, the simulator has a full PolicyTypes list and a full ProblemTypes. *Simulate* algorithm simulates each control policy under all scheduling environments. Hence Each PolicyType is simulated iteratively for all ProblemTypes by using *SimulateOneProblemSetting* algorithm given in Figure B.33.

*SimulateOneProblemSetting* algorithm invokes *SimulateOneReplication* algorithm many times until the specified number of replication is reached. The simulator is initialized before each replication. Statistics of each replication is obtained by *CalculateStatistics* algorithm. *SimulateOneReplication* algorithm is demonstrated in Figure B.34.

The simulator has a CurrentTime attribute to control simulation time, and has an Events list to operate simulation. Each time the event object with minimum Time value is selected from Events list. If Time is greater than CurrentTime, CurrentTime is raised

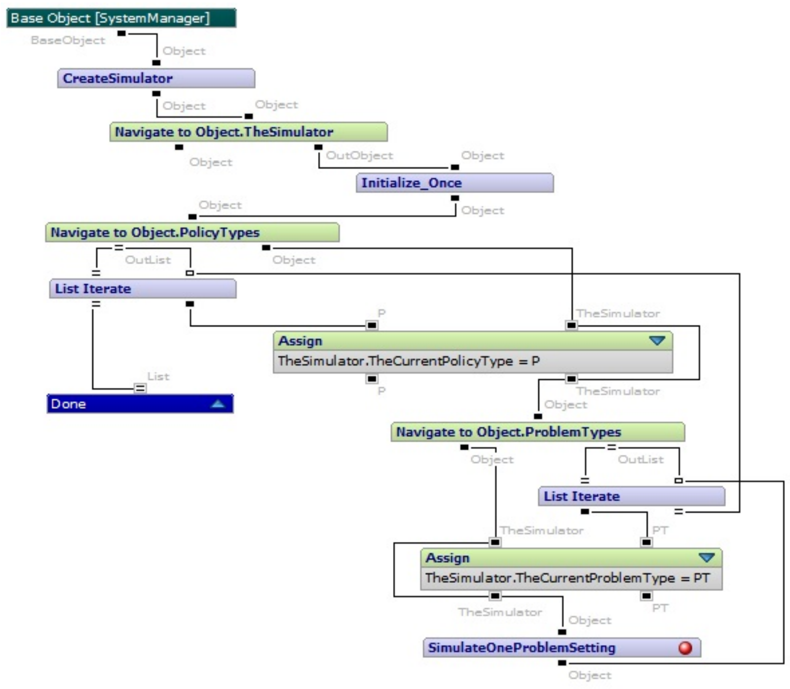


Figure B.32. Simulate Algorithm of System Manager Class.

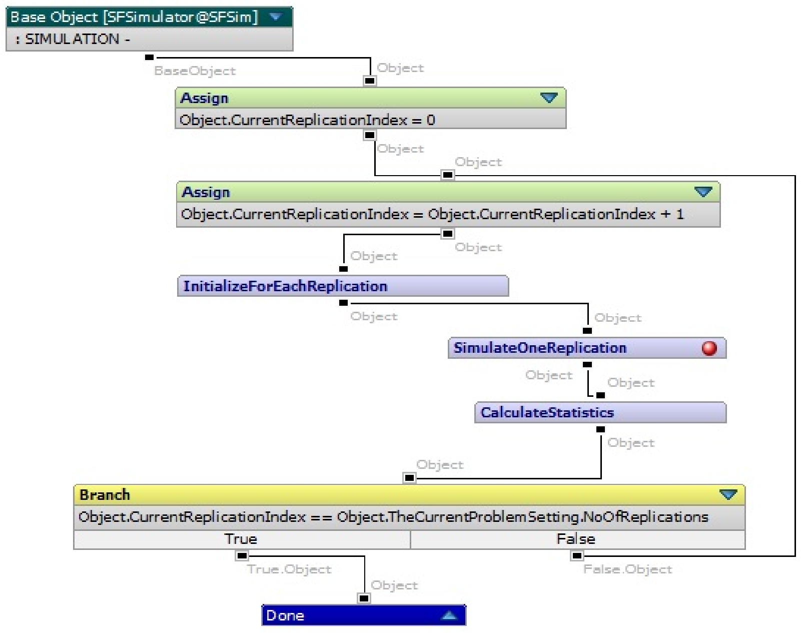

Base Object [SFSimulator@SFSim]
: SIMULATION -
BaseObject
Object
Assign
Object.CurrentReplicationIndex = 0
Object
Object
Assign
Object.CurrentReplicationIndex = Object.CurrentReplicationIndex + 1
Object
Object
InitializeForEachReplication
Object
Object
SimulateOneReplication
Object
Object
CalculateStatistics
Object
Object
Branch
Object.CurrentReplicationIndex == Object.TheCurrentProblemSetting.NoOfReplications
True
False
True.Object
False.Object
Object
Done

Figure B.33. SimulateOneProblemSetting Algorithm of Simulator Class.

up to Time, else selected event is activated. According to the Code of the selected event object one of the overloaded *Handle* algorithm explained in Section B.1 is invoked. The executed event is removed from Events list. After *Handle* algorithm is done it is checked if CompletedJobs reaches NoOfJobsCompleted value of TheCurrentProblemSetting or not. *SimulateOneReplication* is done when the desired number of completed jobs is achieved.

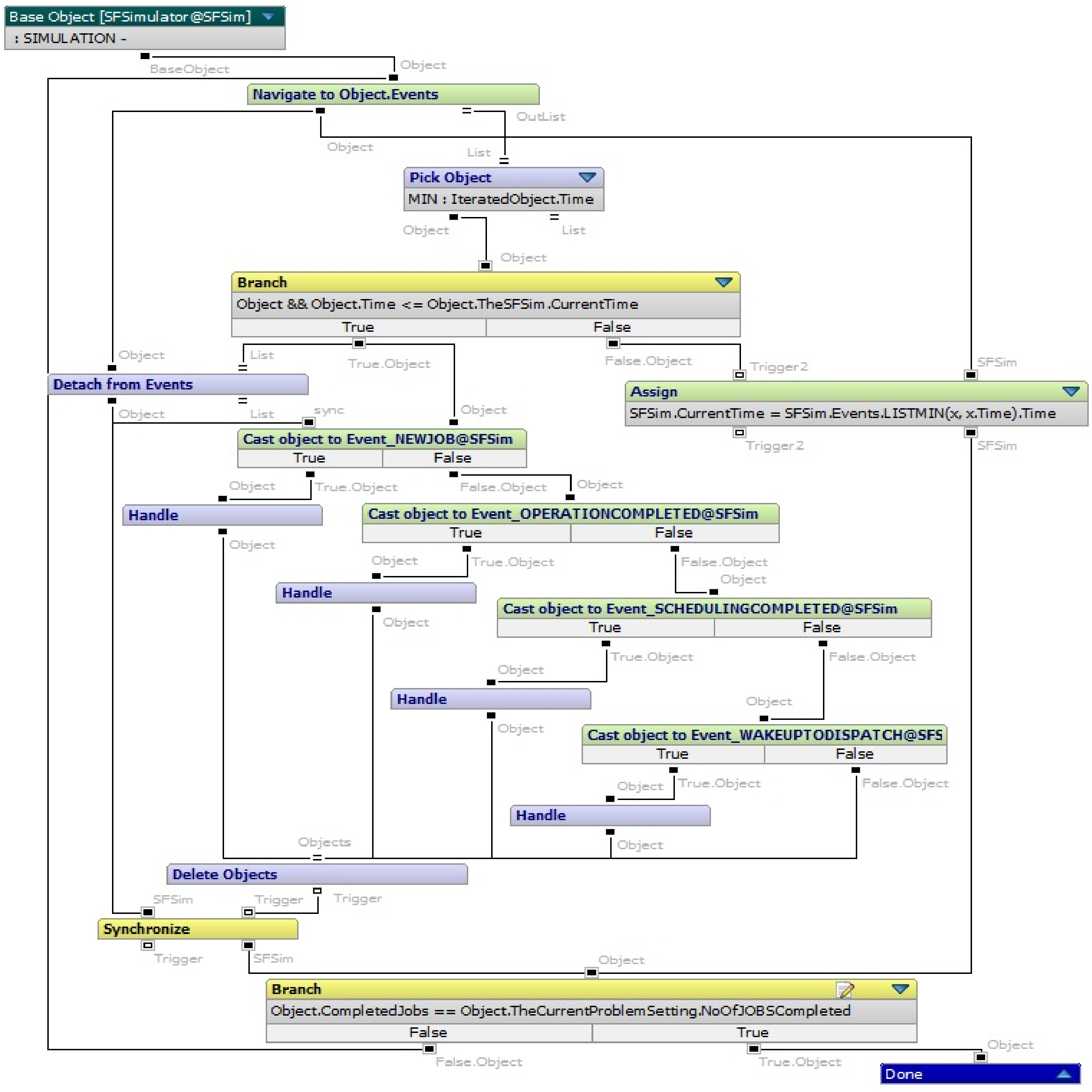


Figure B.34. SimulateOneReplication Algorithm of Simulator Class.

# REFERENCES

1. Church, L. and R. Uzsoy, “Analysis of Periodic and Event-Driven Rescheduling Policies in Dynamic Shops”, *International Journal of Computer Integrated Manufacturing*, Vol. 5, No. 3, pp. 153–163, 1992.

2. Sozer, K., *State Based Modeling of Scheduling Agents in Intelligent Manufacturing*, M.S. Thesis, Bogazici University, 2007.

3. Lawrence, S. and E. Sewell, “Heuristic, Optimal, Static, and Dynamic Schedules when Processing Times are Uncertain”, *Journal of Operations Management*, Vol. 15, No. 1, pp. 71–82, 1997.

4. Kutanoglu, E. and I. Sabuncuoglu, “Experimental Investigation of Iterative Simulation-Based Scheduling in a Dynamic and Stochastic Job Shop”, *Journal of Manufacturing Systems*, Vol. 20, No. 4, pp. 264–279, 2001.

5. Sabuncuoglu, I. and O. Kizilisik, “Reactive Scheduling in a Dynamic and Stochastic FMS Environment”, *International Journal of Production Research*, Vol. 41, No. 17, pp. 4211–4231, 2003.

6. Kutanoglu, E. and S. Wu, “Incentive Compatible, Collaborative Production Scheduling with Simple Communication among Distributed Agents”, *International Journal of Production Research*, Vol. 44, No. 3, pp. 421–446, 2006.

7. Aytug, H., M. Lawley, K. McKay, S. Mohan and R. Uzsoy, “Executing Production Schedules in the Face of Uncertainties: A Review and Some Future Directions”, *European Journal of Operational Research*, Vol. 161, No. 1, pp. 86–110, 2005.

8. Vieira, G., J. Herrmann and E. Lin, “Rescheduling Manufacturing Systems: A Framework of Strategies, Policies, and Methods”, *Journal of Scheduling*, Vol. 6, No. 1, pp. 39–62, 2003.

9. Ouelhadj, D. and S. Petrovic, "A Survey of Dynamic Scheduling in Manufacturing Systems", *Journal of Scheduling*, Vol. 12, No. 4, pp. 417–431, 2009.

10. Nandkeolyar, U., M. Ahmed and P. Sundararaghavan, "Dynamic Single-Machine-Weighted Absolute Deviation Problem: Predictive Heuristics and Evaluation", *International Journal of Production Research*, Vol. 31, No. 6, pp. 1453–1466, 1993.

11. Dewan, P. and S. Joshi, "Dynamic Single-Machine Scheduling under Distributed Decision-Making", *International Journal of Production Research*, Vol. 38, No. 16, pp. 3759–3777, 2000.

12. Sen, S. and E. Durfee, "A Contracting Model for Flexible Distributed Scheduling", *Annals of Operations Research*, Vol. 65, No. 1, pp. 195–222, 1996.

13. Veeramani, D. and K. Wang, "Performance Analysis of Auction-Based Distributed Shop-Floor Control Schemes from the Perspective of the Communication System", *International Journal of Flexible Manufacturing Systems*, Vol. 9, No. 2, pp. 121–143, 1997.

14. Qin, X. and H. Jiang, "A Dynamic and Reliability-Driven Scheduling Algorithm for Parallel Real-Time Jobs Executing on Heterogeneous Clusters", *Journal of Parallel and Distributed Computing*, Vol. 65, No. 8, pp. 885–900, 2005.

15. Kursun, M., *Modeling and Analysis of Agent Based Distributed Scheduling Systems*, Ph.D. Thesis, Bogazici University, 2008.

16. Kursun, M., A. Unal and K. Sozer, "Issues in Distributed Scheduling", *Proceedings of the Eleventh International Workshop on Project Management and Scheduling*, April 2008.

17. Heydenreich, B., R. Müller and M. Uetz, "Mechanism Design for Decentralized Online Machine Scheduling", *Operations Research*, Vol. 58, No. 2, pp. 445–457, 2010.

18. Jeong, I. and V. Leon, “A Single-Machine Distributed Scheduling Methodology Using Cooperative Interaction via Coupling Agents”, *IIE Transactions*, Vol. 37, No. 2, pp. 137–152, 2005.

19. Tharumarajah, A., “Survey of Resource Allocation Methods for Distributed Manufacturing Systems”, *Production Planning & Control*, Vol. 12, No. 1, pp. 58–68, 2001.

20. Greer, K., J. Stewart and B. McCollum, “Comparison of Centralised and Distributed Approach for a Generic Scheduling System”, *Journal of Intelligent Manufacturing*, Vol. 19, No. 1, pp. 119–129, 2008.

21. Ottaway, T. and J. Burns, “An Adaptive Production Control System Utilizing Agent Technology”, *International Journal of Production Research*, Vol. 38, No. 4, pp. 721–737, 2000.

22. Cavalieri, S., M. Garetti, M. Macchi and M. Taisch, “An Experimental Benchmarking of Two Multi-Agent Architectures for Production Scheduling and Control”, *Computers in Industry*, Vol. 43, No. 2, pp. 139–152, 2000.

23. Brennan, R. and D. Norrie, “Evaluating the Performance of Reactive Control Architectures for Manufacturing Production Control”, *Computers in Industry*, Vol. 46, No. 3, pp. 235–245, 2001.

24. Wong, T., C. Leung, K. Mak and R. Fung, “Dynamic Shopfloor Scheduling in Multi-Agent Manufacturing Systems”, *Expert Systems with Applications*, Vol. 31, No. 3, pp. 486–494, 2006.

25. Leitão, P. and F. Restivo, “A Holonic Approach to Dynamic Manufacturing Scheduling”, *Robotics and Computer-Integrated Manufacturing*, Vol. 24, No. 5, pp. 625–634, 2008.

26. Toptal, A. and I. Sabuncuoglu, “Distributed Scheduling: A Review of Concepts

and Applications", *International Journal of Production Research*, Vol. 48, No. 17-18, pp. 5235–5262, 2010.

27. Dewan, P. and S. Joshi, "Implementation of an Auction-Based Distributed Scheduling Model for a Dynamic Job Shop Environment", *International Journal of Computer Integrated Manufacturing*, Vol. 14, No. 5, pp. 446–456, 2001.

28. Pendharkar, P., "A Computational Study on Design and Performance Issues of Multi-Agent Intelligent Systems for Dynamic Scheduling Environments", *Expert Systems with Applications*, Vol. 16, No. 2, pp. 121–133, 1999.

29. Boccalatte, A., A. Gozzi, M. Paolucci, V. Queirolo and M. Tamoglia, "A Multi-Agent System for Dynamic Just-In-Time Manufacturing Production Scheduling", *IEEE International Conference on Systems, Man and Cybernetics, 2004*, Vol. 6, pp. 5548–5553, IEEE, 2004.

30. Wu, Z. and M. Weng, "Multiagent Scheduling Method with Earliness and Tardiness Objectives in Flexible Job Shops", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 35, No. 2, pp. 293–301, 2005.

31. Váncza, J. and A. Márkus, "An Agent Model for Incentive-Based Production Scheduling", *Computers in Industry*, Vol. 43, No. 2, pp. 173–187, 2000.

32. Agnetis, A., D. Pacciarelli and A. Pacifici, "Multi-Agent Single Machine Scheduling", *Annals of Operations Research*, Vol. 150, No. 1, pp. 3–15, 2007.

33. Leung, J., M. Pinedo and G. Wan, "Competitive Two-Agent Scheduling and Its Applications", *Operations Research*, Vol. 58, No. 2, pp. 458–469, 2010.

34. Kanet, J. and V. Sridharan, "Scheduling with Inserted Idle Time: Problem Taxonomy and Literature Review", *Operations Research*, Vol. 48, No. 1, pp. 99–110, 2000.

35. Li, G., "Single Machine Earliness and Tardiness Scheduling", *European Journal of Operational Research*, Vol. 96, No. 3, pp. 546–558, 1997.

36. Valente, J. and R. Alves, "An Exact Approach to Early/Tardy Scheduling with Release Dates", *Computers & Operations Research*, Vol. 32, No. 11, pp. 2905–2917, 2005.

37. Valente, J., "Improved Lower Bounds for the Single Machine Earliness/Tardiness Scheduling Problem with Release Dates", *International Journal of Operations Research*, Vol. 2, No. 2, pp. 9–16, 2005.

38. Kim, Y. and C. Yano, "Minimizing Mean Tardiness and Earliness in Single-Machine Scheduling Problems with Unequal Due Dates", *Naval Research Logistics*, Vol. 41, No. 7, pp. 913–934, 1994.

39. Schaller, J., "A Comparison of Lower Bounds for the Single-Machine Early/Tardy Problem", *Computers & Operations Research*, Vol. 34, No. 8, pp. 2279–2292, 2007.

40. Bulbul, K., P. Kaminsky and C. Yano, "Preemption in Single Machine Earliness/Tardiness Scheduling", *Journal of Scheduling*, Vol. 10, No. 4, pp. 271–292, 2007.

41. Sourd, F. and S. Kedad-Sidhoum, "A Faster Branch-and-Bound Algorithm for the Earliness-Tardiness Scheduling Problem", *Journal of Scheduling*, Vol. 11, No. 1, pp. 49–58, 2008.

42. Yau, H., Y. Pan and L. Shi, "New Solution Approaches to the General Single-Machine Earliness-Tardiness Problem", *IEEE Transactions on Automation Science and Engineering*, Vol. 5, No. 2, p. 349, 2008.

43. Yano, C. and Y. Kim, "Algorithms for a Class of Single-Machine Weighted Tardiness and Earliness Problems", *European Journal of Operational Research*, Vol. 52, No. 2, pp. 167–178, 1991.

44. Mazzini, R. and V. Armentano, “A Heuristic for Single Machine Scheduling with Early and Tardy Costs”, *European Journal of Operational Research*, Vol. 128, No. 1, pp. 129–146, 2001.

45. Kedad-Sidhoum, S. and F. Sourd, “Fast Neighborhood Search for the Single Machine Earliness-Tardiness Scheduling Problem”, *Computers & Operations Research*, Vol. 37, No. 8, pp. 1464–1471, 2010.

46. Sridharan, S. and Z. Zhou, “Dynamic Non-Preemptive Single Machine Scheduling* 1”, *Computers & Operations Research*, Vol. 23, No. 12, pp. 1183–1190, 1996.

47. Mönch, L. and R. Drießel, “A Distributed Shifting Bottleneck Heuristic for Complex Job Shops”, *Computers & Industrial Engineering*, Vol. 49, No. 3, pp. 363–380, 2005.

48. Yeong-Dae, Y. and C. Arai, “Algorithms for a Class of Single-Machine Weighted Tardiness and Earliness Problems”, *European Journal of Operational Research*, Vol. 52, No. 2, pp. 167–178, 1991.

49. Lasserre, J. and M. Queyranne, “Generic Scheduling Polyhedra and a New Mixed-Integer Formulation for Single-Machine Scheduling”, *Proceedings of the 1992 Integer Programming and Combinatorial Optimization Conference*, 1992.

50. Keha, A., K. Khowala and J. Fowler, “Mixed Integer Programming Formulations for Single Machine Scheduling Problems”, *Computers & Industrial Engineering*, Vol. 56, No. 1, pp. 357–367, 2009.