Development of a Run by Run Control Benchmarking and Simulation System

by

Eric J. Stuckey

B.S. in Computer and Electrical Engineering, Purdue University (1996)

Submitted to the Department of Electrical Engineering
and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

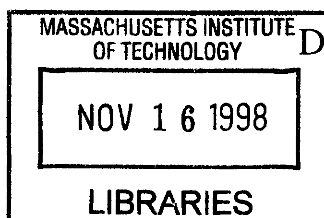
September 1998

© Massachusetts Institute of Technology, 1998. All Rights Reserved.

Author ....
Department of Electrical Engineering and Computer Science
August 24, 1998

Certified by ....
Duane S. Boning, Associate Professor
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith, Chairman
Department Committee on Graduate Theses


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
NOV 16 1998
LIBRARIES

Development of a Run by Run Control Benchmarking and Simulation System

by

Eric J. Stuckey

Submitted to the Department of Electrical Engineering and Computer Science on August 10, 1998, in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science

Abstract

As the semiconductor industry begins to move toward the introduction of fault detection and classification as well as run by run (RbR) process control methodologies, the identification of application scenarios and the means to compare and benchmark available solutions is an essential step. That step has been taken for FDC, and its importance for RbR has been recognized. The work presented here examines the feasibility of such an activity for run by run control including the determination of appropriate scenarios for run by run control, and determining how meaningful comparisons or benchmarking between controllers can be accomplished.

There have been a number of benefits as a result of this project. First, we have shown that the benchmarking of run by run controllers is indeed feasible, and through the development of a run by run control simulation and benchmarking framework, and the determination of a set of process scenarios, SEMATECH is well-prepared for future efforts that might undertake benchmarking and/or demonstration of available commercial and experimental run by run controllers in specific realistic scenarios. Second, we now have a better understanding of the demands on and capabilities of run by run control, as well as a better understanding of the requirements of a successful benchmarking system. This information has been obtained through a literature survey and a questionnaire distributed to SEMATECH member companies to solicit feedback on the requirements and opportunities for run by run control and run by run control benchmarking. Third, we have defined and implemented several control scenarios that can serve as benchmarking cases in a future benchmarking effort. And finally, experience has been gained in using messaging protocols to connect RbR controllers to process simulators (or actual process tools) that can be beneficial in defining a standard set of communications for RbR controllers and other process peripherals..

Thesis Supervisor: Duane S. Boning
Title: Associate Professor, EECS

Acknowledgments

As my masters work comes to a completion here at MIT, I would like to take a moment to thanks a few of the many people who have helped me along the way. First I would like to thank my professor, Duane Boning, for giving me the opportunity and the funding to perform my work amidst such an exciting group and within such an exciting field. I would also like to thank Sematech who provided the actual project (and part of the funding).

I would also like to thank my labmates for all of their help and support. Thank you to Aaron, Han, Sandeep, Taber, Dave, Angie, and Brian. I would finally like to thank all my friends here at MIT for making it such a rewarding experience.

Table of Contents

1 Introduction 12
1.1 Background 12
1.2 Project Tasks 12
1.3 Organization of this Document 13
1.4 Guidance to Audience 13
2 Background 16
2.1 Process Control 16
2.2 Literature Search and Web Page 18
3 RbR Control and Benchmarking Industry Survey 20
3.1 Compilation of Survey Responses 20
3.2 Summary of Information 23
4 RbR Benchmarking Control Scenarios 26
4.1 Description of Control Scenarios 26
4.2 Addressing the Feedback from Industry 32
5 RbR Benchmarking Framework 34
5.1 RbR Benchmarking System Topology 34
5.2 RbR Benchmarking Message Protocol 37
5.3 RbR Benchmarking Server 38
5.4 RbR Benchmarking Clients 39
5.5 RbR Benchmarking Simulators 40
6 Implementing Simulators in the RbR Benchmarking System 42
6.1 Process Simulator Development 42
6.2 Creating a New Simulator 46
6.3 Existing Process Simulators 47
7 Integrating Clients into the RbR Benchmarking System 50
7.1 Client Integration Utilities 50
7.2 Example of an Integrated Controller 55
8 Benchmarking Framework Verification 58
8.1 Connection to the Benchmarking Framework 58
8.2 Results of the Sample Benchmarking 60
9 Conclusions and Future Work 64
Bibliography 68
Appendix A Run by Run Process Control Benchmarking Surveys 70
Appendix B Run by Run Process Control Messaging System 80
Appendix C Example Simulator: Deposition.java 84
Appendix D Example Client Controller: Client.java 88

List of Figures

Figure 2.1: Drifts and Shifts in a Semiconductor Process 16
Figure 2.2: The EWMA Controller...18
Figure 2.3: Sample of the RbR Control Literature Survey Web Page............................19
Figure 5.1: RbR Benchmarking Framework Topology 36
Figure 6.1: ConnectProcess "if else" Model...46
Figure 8.1: Printout of Results from a Sample Control Run 62
Figure 8.2: Control Run on CMP_ext1 by C-based Controller 62
Figure 8.3: Control Run on CMP_ext1 by No-update Java-based Controller................63
Figure 8.4: Control Run on Deposition by C-based Controller 63
Figure 8.5: Control Run on Deposition by Full-update Java-based Controller 63

List of Tables

Table 3.1: Question 3 Results 22
Table 4.1: Control Scenarios 27
Table 4.2: Time-Based Control Scenario Extensions 28
Table 4.3: CMP Scenario Extensions 30
Table 4.4: Etch Scenario Extensions 31
Table 6.1: BenchProcess Fields 44
Table 6.2: Existing Process Simulators 48
Table 7.1: Use of Client Integration Utilities 52
Table 7.2: AbstractClient and ProcessInfo Fields 54
Table 7.3: Bench_help.h Structures 55

Chapter 1

Introduction

1.1 Background

In previous work, the development and use of standardized data sets for the benchmarking and comparison of semiconductor process diagnostic fault detection and classification (FDC) methods and algorithms has been investigated [1]. Such effort was in response to the large number of algorithms available and the relatively difficult task of 1) understanding the various situations in which different methods and algorithms can be applied, and 2) making reasonable comparisons between different FDC products.

The semiconductor industry perspective on run by run (RbR) (or run to run) control algorithms holds a similar standing to that held by fault detection and classification algorithms before the benchmarking effort. Currently, several algorithms exist for RbR control, but minimal efforts at characterizing their performance in a standardized or uniform fashion has been attempted until now. Equally importantly, there is only limited understanding in the industry of the needs, opportunities, and conditions for value-added use of RbR control.

1.2 Project Tasks

The objective of this thesis has been to study and demonstrate the feasibility of RbR control benchmarking. This encompasses two key goals: first, to survey the current state of RbR control and available products, as well as to understand general and specific control scenarios in semiconductor manufacturing; and second, to develop a simulation/benchmark framework for testing and comparing available commercial and experimental RbR controllers in specific realistic scenarios. The results of these efforts and the resulting

benchmarking framework that was developed is presented in this document. The work performed encompasses the following:

1. A literature survey and summary of capabilities and results from the use of RbR control in the semiconductor and other industries
2. A survey for solicitation of feedback from the semiconductor industry of the requirements and opportunities for RbR control
3. Definition of generic and actual RbR control scenarios
4. Development of an RbR control simulation and benchmarking framework for the evaluation and comparison of controllers
5. Verification of the feasibility of benchmarking and simulation by performing a benchmark comparison of two controllers using the benchmarking framework

1.3 Organization of this Document

This document first describes the literature survey performed, the industry questionnaire that was distributed, and the development of the RbR control scenarios from this information in Chapters 2 through 4. The benchmarking framework and the integration of simulators and controllers into this framework is described in Chapters 5 through 7. Chapter 8 presents the results of benchmarking performed on two sample controllers that were integrated into the benchmarking system. Finally, Chapter 9 presents conclusions and suggestions for future work.

1.4 Guidance to Audience

Readers who are interested primarily in the control aspects of this document, including the development of the RbR control scenarios and the performance of the benchmarking system, may start with Chapter 1 and then focus on Chapters 2, 3, 4 and 8 and finish up with

Chapter 9. Those readers more interested in the software implementation issues and use of the benchmarking system including development and integration of new process simulators and connection of client RbR controllers may start with Chapter 1 and then focus on Chapters 5 through 8 as well as Appendices B through D, and finish up with Chapter 9.

Chapter 2

Background

2.1 Process Control

The need for process control arises from systematic variation in the rate or quality of a manufacturing process. Many processes are not stable and well behaved but rather suffer from shifts or drifts. For example, a metal sputter deposition process can exhibit a steady drift in the deposition rate as wafers are processed. This drift can be quite significant. When process kits are changed during maintenance periods, the deposition rate shifts. The drifts and shifts in the process is illustrated in Figure 2.1 [2]. Other semiconductor processes exhibit similar drifts and shifts. In chemical mechanical planarization, as the polishing pad wears away, the removal rate is degraded [3]. In plasma etch, as polymer builds up on the chamber walls the boundary conditions are altered; the plasma changes and so does the etch rate and quality [4].

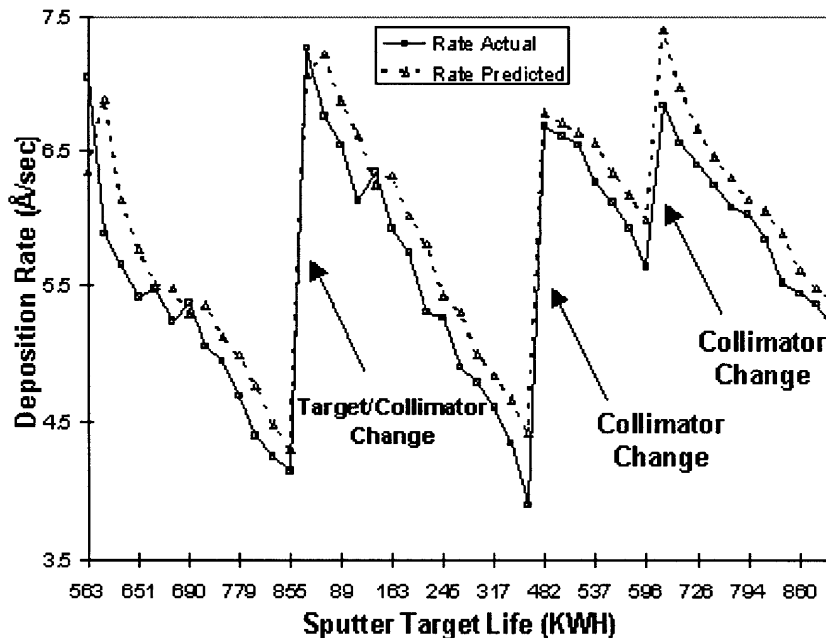


Figure 2.1: Drifts and Shifts in a Semiconductor Process

In order to monitor and correct for these changes to a process (with the goal of maintaining the output quality) open-loop statistical process control (SPC) methods, such as the Shewart control chart, the CUSUM chart, and the moving average, were developed and applied to semiconductor processes [5]. These methods provide a means of monitoring the quality of a process to determine when the process characteristics have changed (i.e. when the process output is no longer acceptable.) When it is determined that the process has changed, or gone out of bounds (i.e. when the metal sputter deposition rate has drifted so much that the resulting output is no longer within an acceptable range) the system is shut down and a corrective action can be taken (i.e. changing the metal sputter deposition's input recipe).

While SPC methods were effective for keeping a process within in a set of bounds, the strict tolerances required in the semiconductor industry called for the development of a methodology that would keep an output more closely centered around a desired value, thus leading to active feedback control. These methodologies are referred to as run by run (RbR) control.

RbR control uses post-process measurements (and possibly *in situ* data) to modify models of the process, and recommend new equipment settings for the next run based on these adaptive models. RbR control differs from real-time control in that it only adjusts process models between runs. RbR control has been shown to be effective in a number of scenarios in which processes characteristics are subject to shifts and drifts [2-4].

One example of a closed-loop feedback RbR control method is the exponentially weighted moving average (EWMA) controller [6]. The EWMA controller uses an affine

model of the process and modifies the offset term to adapt the model to changes in the process. Figure 2.2 presents a flow diagram of the operation of an EWMA controller.

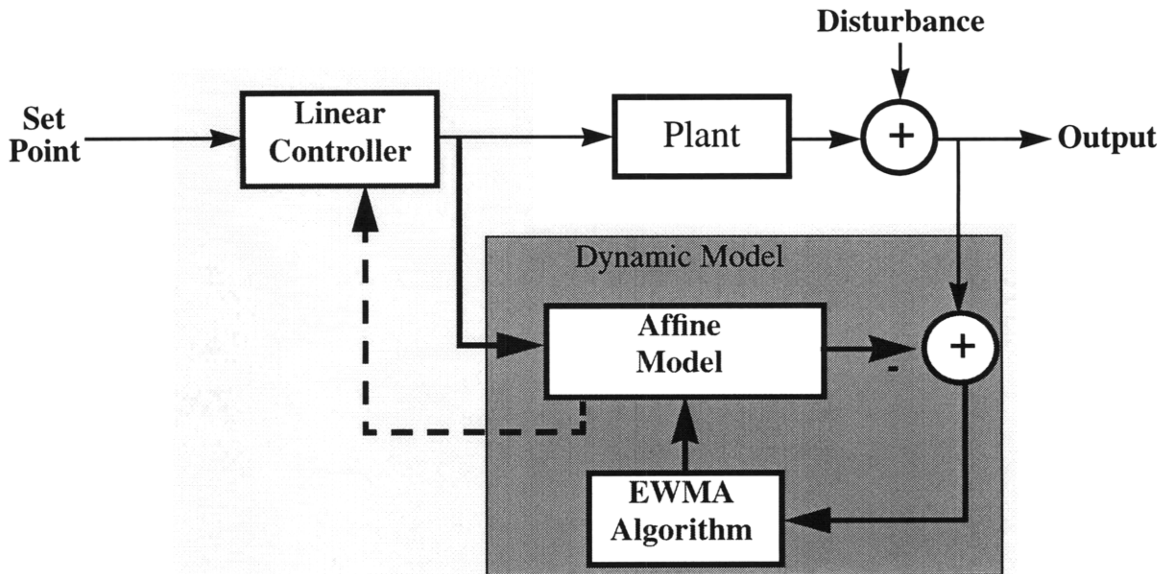


Figure 2.2: The EWMA Controller

A number of other RbR control methods exist, including variations on the EWMA controller [8-10], and other methods [11-15].

2.2 Literature Search and Web Page

The first task performed as part of this thesis was a survey of the literature and a summary of capabilities of RbR control demonstrations and applications. The goal was to identify specific "success stories" of results and benefits gained in the semiconductor and other industries through the use of such control. Performing this task gave us a better understanding of the types of processes that RbR control has been applied to and helped us develop the process scenarios that were determined to best test the ability of RbR controllers. More information about these scenarios, why they were selected, and how they test the full range of RbR controller functionality, is presented in Chapter 4.

In order to organize the results of the literature survey and to make these results (and other information about the RbR benchmarking project) available, an RbR benchmarking web page was created. This web page, located at http://www-mtl.mit.edu/rbrBench, contains much information about the RbR benchmarking effort. See Figure 2.3 for a sample of the entries in the literature survey section of the RbR benchmarking web page. This web page is split into algorithms, applications, and implementation/integration sections in order to organize the articles. While this page is by no means complete, it contains a number of important and relevant publications.

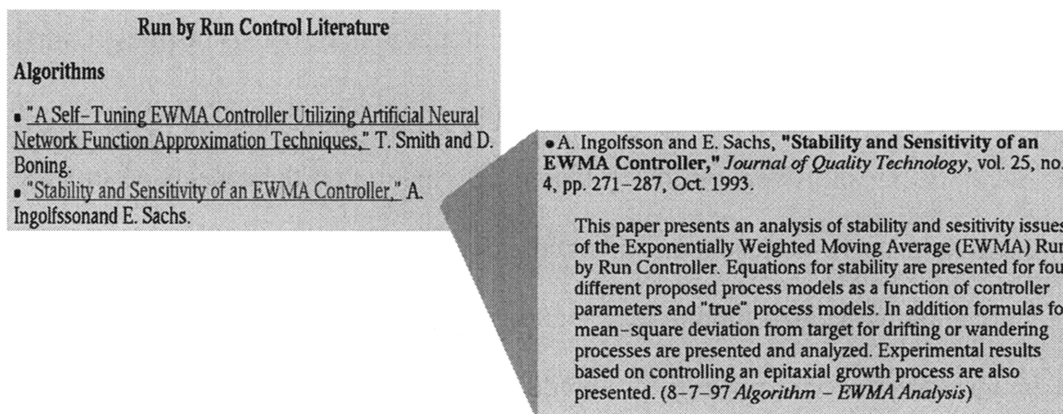


Figure 2.3: Sample of the RbR Control Literature Survey Web Page

Besides the literature survey results, this web page also contains documentation on the benchmarking framework, the industry survey, and various documents pertaining to this project and RbR control. Additionally, the RbR benchmarking web page contains a list of the result files from control runs on the benchmarking system. Each of these components will be described in more detail further into this thesis

Chapter 3

RbR Control and Benchmarking Industry Survey

When developing a benchmarking system it is important to consider the needs of the product's customer. A system that benchmarks the wrong aspects of a product or tests against inappropriate simulations may be of little use to the customer. Therefore it is necessary to institute a means to gather feedback from potential customers during the development of the benchmarking system. For the RbR process control benchmarking project, this was done by developing and distributing to semiconductor industry companies a survey to gather information about processes and RbR issues of interest. Two versions of this survey were developed: a short version that gathered information on the aspects of an RbR controller that industry is interested in, and a more detailed, long version that went much more into depth about processes and implementation and integration issues. Both versions of the survey are presented in Appendix A. In addition, a Java version of the short survey was created and was made available on the RbR benchmarking web page for on-line feedback. The rest of this chapter serves to gather and summarize the information received in the returned surveys. This information was then used to help develop the process scenarios that are described in Chapter 4, and used for input when designing the behavior and performance of the benchmarking framework described in Chapter 6.

3.1 Compilation of Survey Responses

In the following sections, responses to questions from the short version of the survey are presented.

Question 1: What processes and equipment could most benefit from RbR process control and why?

- Track/Stepper links. These systems will affect the wafer more times during its pro-

cess life than any other system or set of systems. These systems could benefit because of the nature of the process with it susceptibility to both sudden and gradual changes.

- Plasma, high temperature processes and photo steps. These processes have the most variation and drive most of the line problems.
- Photolithography: being the bottleneck of the factory throughput, it can benefit from the elimination of pilot runs. Critical dimension (CD) being the major factor in circuit performance, RbR control on CD will have dramatic effects on speed binning, time to yield, and yield. RbR control schemes could also extend the useful life-span of older lithographic technologies.
- All processes could eventually benefit; our current areas of focus are photolithography, CMP and etch.
- Etch, PVD, CVD, etc. Provide disturbance control/compensation for drifts, disturbance, etc. Fault detection/classification.
- Photolithography, CMP, sputter deposition, CVD and epitaxy are viable now. Etch and ion implant will be viable in the future.

Question 2: Are there issues which RbR controllers should address that have not been considered?

- An adequate way of feeding back the information to effect a proper correction on the process.
- How to process the data. Where to store the results, both summary and raw data.
- Yes: in semiconductor manufacturing variability is contributed and accumulated by multiple, sequential steps. This means that in addition to feedback, feed-forward techniques will be effective in reducing variability. Another issue is that of manufacturing flexibility. When a high mixture of products is present, an RbR controller should be able to do well over a wide operating range, allowing manufacturing lines to incorporate new processes quickly.
- Who should own RbR? Equipment vendor or the chip manufacturer?
- Evaluation of costs vs. benefits. Don't confuse process skills with computer integrated manufacturing (CIM) skills, few have both. Consideration of fault detection in addition to/instead of RbR control.

Question 3: Please rate the importance of each of the following criteria for RbR process controllers (1-very important through 5-not important.)

Question	Responses	Mean	Std. Dev.
Simplicity of algorithm	4 1 2 5 4 3	3.17	1.47
Ability to handle different processing conditions	2 2 5 1 1 5	2.67	1.86
Integration with CIM systems	3 1 5 1 1 5	2.67	1.97
Ability to handle complex sensor information	1 4 5 3 5	3.60	1.67
Ability to incorporate complex process models	2 3 5 2 2 3	2.83	1.17
Customer support for RbR packages	1 1 5 1 1 5	2.33	2.07

Table 3.1: Question 3 Results

Question 4: Please provide contact information for any specific controllers you believe should be included in RbR benchmarking.

- In situ particle monitoring (ISPM) (High yield technology (HYT) and particle measuring system [PMS]), residual gas analysis (RGA) (Infocon, Leybold, Ferran,) Radio Frequency (RF) (AE, ENI, FST)
- If this study will extend into non-commercially available packages, and depending on the resources needed and supplied, we will be willing to help benchmark the Berkeley Computer Aided Manufacturing (BCAM) controller (capable of feedback/ feed-forward and supervisory RbR control). Please contact spanos@eecs.berekeley.edu.

Additional Comments

- RbR controllers should be as complex as necessary and as reliable as possible. Nobody demands that photolithography tools be simple.
- Handling complex sensor data is not the RbR controllers job. Relevant information should be extracted before being presented to the controller.
- There are trade-offs for using complex process models. They are more powerful, but they are more expensive to develop and have shorter lifetimes.
- RbR controller companies would be unwilling to figure out the benchmarking framework. It would be better to directly assist them in the integration and then do the testing on-site.
- Benchmarking system should have Java and/or Windows NT support. The RbR controller should be required to run on Windows NT systems.
- What are the costs of the controllers and beyond (i.e. installation, model development, training, downtime.)

3.2 Summary of Information

The information in the returned surveys was collected and used in establishing the framework to be used for the RbR process control benchmarking system. This information was used in several ways. First the scenarios to be used as simulators to test the controllers were determined. By looking at the responses to Question 1, it can be seen that there is significant interest in using RbR control on depositions (both CVD and PVD), CMP, etch and photolithography. Therefore these four processes were chosen as the scenarios to be used in the benchmarking system. Not only are these four process of significant interest to industry, but they also test a wide range of features of RbR controllers, allowing a full evaluation. In addition, extensions to each of these scenarios were created in order to satisfy issues raised in the surveys. For example, using complex models, and providing target dependent plant models to provide wide and varying ranges of process conditions. More information on the scenarios and their extensions are presented in Chapter 4.

Additionally, the surveys were used to help determine the means of employing the benchmarking. There was interest in providing Java support for the benchmarking project. It was decided to implement the benchmarking framework in Java in order to make it distributable. However C support is also provided to make the benchmarking as convenient as possible. Some interest in providing hands-on support to researchers interested in benchmarking their controllers was expressed. This has been considered but a full decision on this must be postponed for a future project.

Finally, the surveys provided much information on how to evaluate the RbR controllers. Obviously process improvement (i.e. attaining target and noise reduction) is critical, but there are many other issues to be considered. Many of these other issues are qualitative and are best evaluated by customers for their specific processes, while others can be tested in the benchmarking system. Some issues brought up include the ability to control over a

wide operating range and the ability to handle different processing conditions. There were mixed responses in the surveys as to whether or not simplicity in the control algorithm is important. But it is an issue worth considering. Other more qualitative issues are customer support for the controller, the CIM/tool integration issues involved with the controller, and the cost of the controller, both in itself and additional costs such as installation and maintenance.

Chapter 4

RbR Benchmarking Control Scenarios

As mentioned earlier, RbR process control is a methodology which utilizes process measurements to modify process settings on a run by run or lot-by-lot basis to improve or maintain performance in the face of a process disturbance state. The process disturbance state includes sources of uncertainty such as process drifts, process shifts due to maintenance or other factors, various noise effects, and model mismatch [6, 8]. A successful RbR controller should be able to compensate for, or minimize the effects of, the various disturbances. In addition, an RbR controller should be capable of dealing with limitations and situations often encountered in a real manufacturing process, such as input bounds, input discretization, non-periodic measurements, variable time delays, process outliers, and the processing of multiple product types or the use of multiple recipes on the same tool [16, 17]. A successful benchmarking system would challenge the controllers with both the fundamental problem and variants that embody these realistic scenarios. The purpose of this chapter is to outline the candidate control scenarios for the RbR benchmarking system, how they challenge controllers, and how they were selected. The four scenarios identified are rate tracking (e.g. deposition), linear multivariate (e.g. CMP), nonlinear multivariate (e.g. plasma etch), and multistep (e.g. photolithography).

4.1 Description of Control Scenarios

The control scenarios presented here are taken from our experience with RbR control, the suggestions and experience of the semiconductor industry gleaned through an RbR process control benchmarking survey described in Chapter 3, and from a review of the literature on RbR control. We determined that we could test the range of issues in RbR control

with four scenarios described by basic underlying control issues: rate tracking with target time adjustment, linear (or nearly linear) multivariate, nonlinear multivariate, and multiple process step. These scenarios are well represented by corresponding canonical fabrication process steps. Respectively, these are deposition, chemical mechanical polishing (CMP), plasma etch, and photolithography. The deposition and CMP problems we define here are somewhat less difficult in terms of the fundamental control action, but provide the opportunity for testing several practical barriers that controllers must face. The etch and photolithography scenarios provide more difficult control problems because the nonlinear and multi-step features of these problems, respectively, make the design of controllers more complex and difficult to implement. We highlight these cases in Table 4.1, and provide further detail in the following sections. In the scope of this thesis, only the first two scenarios (deposition and CMP) have been implemented in order to verify the performance of the benchmarking system.

Scenario	Process Order	Noise	I/O
Deposition	• first order	• white noise • drift	• univariate
CMP	• first order or • second order	• white noise • drift • shifts	• multivariate
Etch	• non-linear	• noise • drift • shifts	• multivariate
Photolithography	• non-linear	• noise	• multistep

Table 4.1: Control Scenarios

4.1.1 Rate Tracking/Time Control (Deposition) Scenario

The simplest scenario that we have is one in which the controller is considerably restricted in the available recipe or control decisions it may make. In several processes, the process

recipe essentially seeks to define a rate (e.g. a material deposition or material removal rate), but the production practice is such that this recipe is considered “fixed.” However, the duration of the process step, or process time, is allowed to be modified in order to achieve a desired target (e.g. thin film thickness deposited or removed).

One example control approach is to simply track the rate of a process and then modify the time parameter to achieve the target output. An example of this type of scenario is control of metal sputter deposition. Metal sputter deposition is characterized by a decrease in the deposition rate, due to sputter target degradation and the build up of material within the collimator, and various amounts of process noise. These systematic sources of variation in the process are well suited to correction or compensation by RbR control [2].

Time-based control is a univariate problem. The rate of a process is taken as the single output and the target output values are achieved via the modification of the process time, which is the single input. In addition, rate tracking and time adjustment control can often be modeled well as a first order process [2]. These aspects make it attractive as a simple test for an RbR controller. However, by extending the simple time control scenario with the many practical scenarios a controller of this type might face in an industrial setting, we create a scenario which will challenge RbR controllers. See Table 4.2 for a list of extensions to the time control scenario.

Extension #	Description
1	Multiple process or product targets
2	Delay in post-process measurements
3	Non-periodic measurements
4	Non-periodic delays
5	Process outliers

Table 4.2: Time-Based Control Scenario Extensions

Extension #1 deals with the fact that some time-based control processes have different rates or process characteristics for different output targets or recipes. For example, the deposition rate of the titanium sputter deposition process is a function of the final film thickness [2], and therefore the process model must take into account this variation. Extension #2 addresses the issue of starting a new run before the post-process measurements from the previous run have been completed. In this case, there is a time delay between the control action and the feedback. Extension #3 considers the fact that measurements are not taken for every wafer in semiconductor processes. There may be varying length delays between successive measurements. Extension #4 is similar to Extension #3 in that it involves non-periodic measurements, but in addition the measurements that do take place are delayed. Finally, Extension #5 addresses the fact that many processes have outliers which complicate the control of processes with even a simple model.

4.1.2 Multivariate Process (CMP) Scenario

The second scenario that we present is one based upon the Chemical-Mechanical Planarization (CMP) process. The CMP process can be characterized as a drifting process, due to pad wear, with noise and occasional large shifts, due to pad changes [3]. The types of systematic disturbances seen in CMP processes make them excellent candidates for RbR control.

CMP is a multivariate process. The two major outputs typically of concern are the removal rate and the wafer-level spatial uniformity. Depending on the CMP tool being used, the inputs can include spindle and table speed, back-pressure, down force and conditioning profile. The CMP process is typically modeled using a first order polynomial response surface, but sometimes second order models are used [7]. CMP, along with the

extensions given in Table 4.3, is a good process with which to test the functionality of a multivariate RbR controller.

Extension #	Description
1	Input bounds and discretization
2	Variations in lot size
3	Infrequent measurements
4	Intermittent control

Table 4.3: CMP Scenario Extensions

Extension #1 deals with the fact that input settings on actual pieces of equipment are not continuous. An RbR controller should provide feasible recipes, in a manner which admits the least error in the controlled output. Extension #2 deals with creating varying amounts of system change between control actions by changing the number of wafers that are processed between output measurements. Extension #3 addresses the need for controllers which can control processes with infrequent or missing measurements. Finally, Extension #4 tests the ability to control a changing process by interpolated control actions, perhaps only on engineer or operator demand, in between infrequent measurements.

4.1.3 Multivariate Nonlinear Process (Plasma Etch) Scenario

The third scenario that we consider is one well represented by the plasma etch process. Etch is characterized by a non-linear response that often requires more complex RbR control. In addition to many input and output variables with complex nonlinear behavior, a number of important extensions can also be identified, as summarized in Table 4.4.

Extension #	Description
1	Spatial uniformity sensor information
2	Time-sampled equipment, process, or wafer state information
3	Need for built-in or add-on diagnostics

Table 4.4: Etch Scenario Extensions

Extension #1 deals with the fact that the etch process often has available a multitude of sensors which provide a large amount of data. A controller's ability to deal with all this information is also important [18, 19]. For example, emerging sensors are capable of providing a wealth of spatial uniformity information which could be used to advantage in the control. Similarly, Extension #2 involves emerging sensors (such as radio frequency monitors, optical emission spectroscopy [OES] sensors) which provide a great deal of finely time-sampled information from the process, which may also provide the opportunity for improved control [4, 20]. These characteristics drive the need for controllers which not only provide control but provide or work with process diagnostic systems as well. Extension #3 could test the ability of the controller to distinguish between events that are controllable and those that require maintenance or operator intervention.

4.1.4 Multiple Step Process (Photolithography) Scenario

The final difficult scenario we examine is one well-represented by photolithography. The previous scenarios focused on individual step or unit process, and assume the use of measurements in a "feedback" sense to improve the performance of the tool or process on wafers that subsequently come to that tool. A multiple step control scenario is one in which information from one step may be used to influence a later step that the *same* wafer or lot will later experience. This "feed-forward" capability offers the opportunity to actively compensate for deviations on the current wafer or lot, and requires substantially better communications infrastructure.

Photolithography is a critical step in the manufacturing of integrated circuits. In many cases, photolithography is the bottleneck in the fab, both in terms of speed and product scrap. More importantly, the process capability requirements for lithography are increasingly stringent and directly impact the performance of the resulting devices and circuits,

motivating attention to this process. The basic characteristic of the process is that it involves both feedback (on unit steps) and feed-forward decision making [21]. Extensions to this scenario are not yet clear, but this scenario is a good candidate for future test and benchmarking development.

4.2 Addressing the Feedback from Industry

The distribution of the RbR benchmarking survey resulted in valuable responses and feedback from industry and the control community which aided this work. The ideas and suggestions received were used to identify the above set of control scenarios.

The types of process disturbances identified in the survey include drift, process recipe memory, time delays, input noise and target variations. Each of these disturbances was identified as causing deviations in the process outputs from what was desired. Every scenario and extension identified earlier contains one or more of these disturbances, and each disturbance is covered.

The processes that were identified as being good candidates for RbR control include photolithography, CMP, CVD, sputter deposition, epi, etch, ion implantation and high-temperature processes. For the control scenarios presented above, we attempt to provide a sampling of the type of control issues that will be encountered in IC production and thus identified the four scenarios of deposition, CMP, plasma etch and photolithography presented above. These scenarios cover the major categories of process disturbances that could be improved through RbR control (simple/univariate rate and time based control, multivariate control, non-linear processes and multiple step sequences) and as such should give a good indication of the controller's performance on other processes. It is believed that these four scenarios cover most of the disturbances and problems that an RbR controller is likely to encounter. These include disturbance states such as complex noise models,

drifts, shifts and noise due to model mismatch, as well as limitations such as input bounds and discretization, asynchronous measurements, time delays, the effect of multiple product or process targets, nonlinear processes, and multiple step process sequences.

Issues relating to the testing of RbR controllers that were brought up in the returned surveys include the need for flexibility in the RbR controllers to allow for a wide range of products produced on a processing line. There were some responses in favor of evaluating RbR controllers based partially on the simplicity of their solution. Also, the respondents were in favor of the ability to incorporate complex process models.

Chapter 5

RbR Benchmarking Framework

After having defined a set of RbR control scenarios and identified the needs and requirements for a benchmarking system, the next step was to develop a benchmarking framework. This software framework is designed to support the simulation, benchmarking and comparison of RbR control products. The goal here is to construct a software framework that allows for the emulation of the above control scenarios, and permits easy connection, testing, and comparison of different RbR control products. Interfaces between the benchmark simulation framework and commercial or experimental controllers are defined using open standards based on socket-based communications and integrated source code stubs for multiple languages (C and Java.) The goal is to support multiple ways for controllers to communicate with the simulation/benchmarking system.

There have been many goals for this thesis, all of which are focused on making the framework as easy to use as possible while still maintaining the feel of a real processing tool as much as possible. These goals have included creating a robust, distributable system that can be run locally or over the internet; simple simulator creation; and multiple easy-to-use client controller connections. Accomplishing these goals has led to the creation of the RbR benchmarking framework that is presented in this chapter. This framework has four major tiers: 1) the benchmarking server that handles the interaction between the controllers and the simulators, 2) the simulators, 3) the client controllers, and 4) the messaging system used to communicate between the client and the benchmarking system.

5.1 RbR Benchmarking System Topology

The system topology for the RbR benchmarking framework is developed to be as flexible

as possible. It consists of three major, independent parts that are connected together via a transmission control protocol/Internet protocol (TCP/IP)-based messaging system. These three parts are client controllers (a client controller is often referred to simply as a client in this thesis), the benchmarking server, and a set of process simulators. Figure 5.1 presents a graphical overview of how these systems interact to form the overall RbR benchmarking framework.

The benchmarking server is a computer program that can run on either a remote or local computer. This server is written in Java to allow it to be easily distributable. The server handles all of the communications between the client controllers and the process simulators. When the server is running, it waits for clients to connect either from the same computer or from any computer connected to the internet. When a client connects, the benchmarking server creates a thread to handle the benchmarking. This is done so that multiple clients can be connected to the benchmarking server at the same time (controlling the same, or different, process simulators). This thread then handles all of the communications with the client controller via a pre-defined set of TCP/IP-based messages, and then passes on this information to the selected simulator which has been loaded into the thread. The messages used in the RbR benchmarking framework allow the client controller to

receive information about the process it is controlling, send recipes to be used in the next run of the simulator, and receive messages about the output of its control.

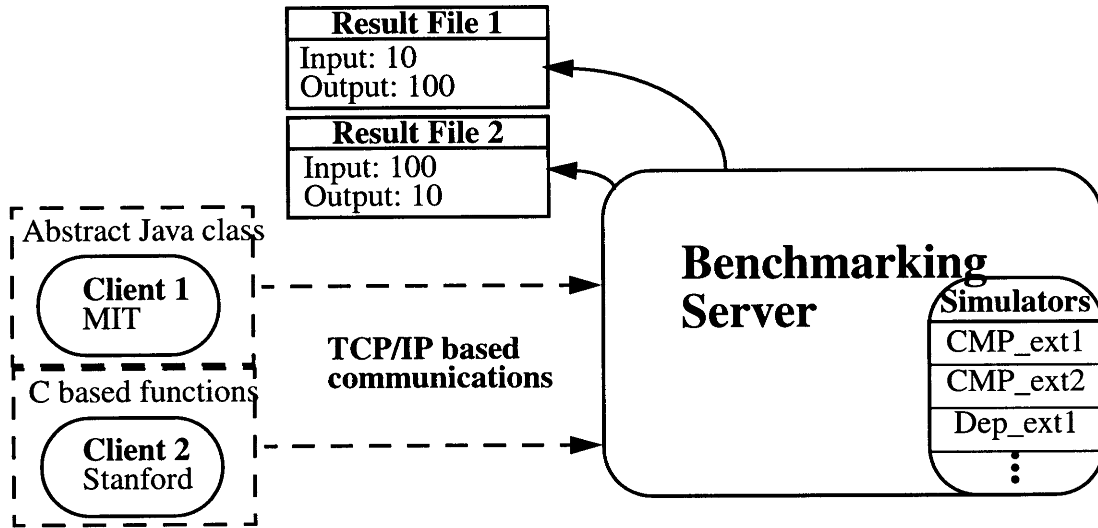


Figure 5.1: RbR Benchmarking Framework Topology

When a client first connects to the benchmarking server, it indicates what process it wishes to control. (The communication does not need to be done directly with TCP/IP communications. A set of classes have been written in Java, and a set of functions written in C to make it easier for client controllers to connect to the benchmarking server. This is indicated by the wrapper around the clients in Figure 5.1, and is described in more detail in Chapter 8.) When the client indicates its simulator selection, the benchmarking server instantiates the proper process simulator class. The benchmarking server then simply acts as a go-between for the client and the simulator. The benchmarking server also creates a log of all the control activities that occur. This log is written to a unique file where it can be viewed as a text file via the world wide web (at http://www-mtl.mit.edu/rbrBench/rbrBenchResults), and therefore downloaded into Matlab or some other mathematical package where further data analysis and plotting can take place. The name of this file is given to the client by the server via the SetSimulator and GetCtrlResults messages.

The next four sections describe each of the tiers of the RbR benchmarking framework in more detail.

5.2 RbR Benchmarking Message Protocol

The definition of the client to server communication is a critical component of the RbR benchmarking system. Much of the robustness and expandability of a system such as this is dependent upon the messaging system. A system that defines an insufficient or overly limited set of messages quickly outlives it usefulness and must be compensated for, often with deleterious effects. Therefore the messages chosen for the RbR benchmarking system are designed to be as minimal as possible, while still providing the ability to expand to accommodate any simulator or controller. A full listing and specification of the messages is on the web at http://www-mtl.mit.edu/rbrBench/Messages.html, and a copy is contained in Appendix B.

The messaging system has three major types of messages: 1) messages that are specific to the RbR benchmarking framework and are used to configure the system; 2) those that are used to get information, such as process details and outputs, from the simulator to the client; and 3) messages used to give settings and instructions to the simulator from the client. More about these messages and the way they are used will be discussed in Chapter 7, which describes the client integration utilities used to help connect RbR controllers to the benchmarking framework.

Much of the flexibility in the messaging system comes from the ability to fully define the number of inputs, outputs, and parameters for the simulator in the GetProcessInfo message. With these three categories, all of the information that a simulator may have can be given to the controllers. There is much similarity in the way that outputs and parameters are provided to the client, but there is an important difference between the two. The

outputs are used to provide directly controllable process output values, while the parameters can be anything (i.e. uncontrolled outputs, process information, maintenance reports, etc). Using the combination of these two fields, all information can be provided to the controller.

The messaging system is designed to have the look and feel of an actual process tool. Since no industry standard has been defined for what information should be passed (or how it should be passed) between an RbR controller and an actual process tool, we have had to determine our own set of messages. This has been a good exercise and has garnered some good information about what is necessary to allow a controller to control a process tool. We believe that a robust and expandable messaging system has been formulated.

5.3 RbR Benchmarking Server

The RbR benchmarking server is the framework that allows the controller and simulator to come together. The server accepts connections from clients, sets up the appropriate simulator, handles the communication between the client and the simulator, and records all of the control decisions and results.

The RbR benchmarking server is run by executing the command java Benchmark.Bench from the command line. When it is run, the server listens for client controllers to connect via TCP/IP on socket 4444. Clients can connect from either a remote system or the local system. The server is set up to handle multiple connected clients at the same time (by spawning a separate thread for each attached client), so only one server needs to be running for any given application.

One of the first things that a newly connected client does is send a message indicating which process to control via a SetSimulator message. Assuming that the indicated process is valid, the server instantiates an object of that process' simulator class to be used for

the rest of the connection. Once the simulator is connected, the server simply serves to forward messages back and forth between the client controller and the simulator. The server does however also make a record (between each run and after each Reset message) of the control actions taking place. It writes these values, and some additional summary and process information, to a unique file in the directory http://www-mtl.mit.edu/rbrBench/rbrBenchResults.html which is then accessible via the web. The name of this file is given to the client in the return messages to both the SetSimulator and GetCtrlResults messages. The owner of that client can then get on the web and view the results of the control. When either the client or server disconnects, the thread spawned to handle that connection ends and the server returns to the state it started in.

5.4 RbR Benchmarking Clients

The clients to the RbR benchmarking system are RbR controllers that have had the ability to communicate with the RbR benchmarking server incorporated into them. Currently there are three ways to incorporate this ability into an RbR controller. The first way is to directly implement the messaging scheme, which is done via the John Carney Messaging system [22]. A much simpler way is to use one of the two sets of client integration utilities that have been written that handle all of the communications with the server. These two integration utilities are an extendable Java abstract class called BenchClient, and a set of C functions and structures in the file Bench_help.c. Both of these utilities provide functions for each message in the messaging system that perform the same operation as the corresponding message, but frees the client from having to deal with actually sending and receiving the messages. These functions can be used just like the messages would be used, except much more easily and readily because the communication implementation is hidden. More details on connecting a controller using these client integration utilities is

described in Chapter 7, and an example of connecting controllers to the benchmarking system is presented in Chapter 8.

5.5 RbR Benchmarking Simulators

The RbR benchmarking simulators are process simulators that are written to simulate a given semiconductor process. Because of the flexibility of the RbR benchmarking framework and messaging system, there is a wide range of processes that can be simulated. It is straightforward to create a new process simulator to be used in the RbR benchmarking framework. There are two classes provided in the BenchSimulations package that are used to create a new simulator, BenchProcess and ConnectProcess. BenchProcess is an abstract class that is extended in order to create a new process simulator. In this extended class, the process detail fields should be set in the constructor, and the methods Run() and Reset() should be overridden to perform a run for the process, and to reset the simulator's state, respectively. Then, an entry should be added to the ConnectProcess class to let the RbR benchmarking server know that this new simulator exists. More details on creating and integrating a new simulator are presented in Chapter 6.

An important aspect of the RbR benchmarking system is the ability of a client to specify a random number generator seed in the SetSimulator message, which is then passed to the simulator by the benchmarking server. This seed can then be used to provide a seed value for the disturbance state used in the simulator. This allows for repeatable control runs by allowing the exact same disturbance state to be used multiple times, and thus allowing for easier comparisons between control runs. By convention, a seed value of zero indicates that a random seed should be given to the disturbance state.

Chapter 6

Implementing Simulators in the RbR Benchmarking System

This chapter describes the creation and implementation of new process simulators to be used in the RbR benchmarking framework. For this benchmarking framework to be usable, the integration of client controllers and the creation of new simulators must be straight-forward and as simple as possible, and so a set of classes that were created to implement simulators is described. This chapter also contains descriptions of the currently existing process simulators that are based on the benchmarking process scenarios that were described in Chapter 4.

6.1 Process Simulator Development

In the BenchSimulations package within the RbR benchmarking system there are two Java classes that are used to create a process simulator: BenchProcess, and ConnectProcess. The abstract class BenchProcess should be extended and used as the base of all process simulators. The class ConnectProcess, which is used by the Benchmark package to connect to the process indicated by a client controller, should be modified each time a new process simulator is created by adding an entry that lets the Benchmark package know that the new simulator exists. In the rest of this section, the abstract class BenchProcess, the ConnectProcess class, and the way to use these classes to create a new process simulator are described. On-line documentation on these classes are available at http://www-mtl.mit.edu/rbrBench/BenchDoc/Package-BenchSimulations.html.

BenchProcess

The BenchProcess class contains three things: fields, methods that can be overridden, and

final methods (methods that cannot be overridden). The fields defined in BenchProcess contain all of the details about the process. There are two main types of fields defined: those whose values remain constant and should be set only once, upon initialization, and those whose values change with each run. See Table 6.1 for a listing and a description of the fields in each category. The constant fields are variables which should not change over the course of a simulation, and so should be initialized in the constructor of the simulator. The dynamic fields are variables that can change with each run and are typically set in the Run() method of a simulator. Memory is not pre-allocated for these fields except for in_setting which is set in the SetRecipe() method, and so this should be taken into consideration. Param_value is a dynamic field, but it can also be used to provide some preliminary parameter values to the client. Therefore, its values should be set in the constructor of the simulator since the client will receive these values immediately after connection. After the initial setting, param_value can be treated just like a dynamic field and changed after each run. However, num_params and param_name are constant fields and cannot be changed outside of the constructor. The num_wafers_in_run and out_target fields are also expected to have initial values set in the constructor before the Run() method is executed for the first time, and so should therefore be initialized in the simulator's constructor. There is one additional field that is not mentioned in Table 6.1, the field Active which is used to indicate if this simulator can be used. If for some reason a simulator should not be used (i.e. it is still under development), the Active field should be overridden and set to false, in which case no client can connect to it.

Most of the numerical manipulation done within the BenchSimulations package, as well as the rest of the RbR benchmarking system, is done using the MatrixCafe Java mathematical libraries (more information about these libraries can be found at http://www-mtl.mit.edu/~taber/MatrixCafe/MatrixCafe.html). Each of the fields in Table 6.1 are

defined as DoubleMatrix (DoubleMatrix is the class defined in MatrixCafe for use in Matrix manipulations). Note however, that in_name, out_name, and param_name are arrays of strings.

Constant Fields	Dynamic Fields	Description
num_inputs		The number of inputs
num_outputs		The number of outputs
num_params		The number of parameters
in_name		The inputs' names
out_name		The outputs' names
param_name		The parameters' names
in_hi_range		The inputs' upper bounds
in_lo_range		The inputs' lower bounds
in_resolution		The inputs' resolution
in_weight		The input weights
out_weight		The output weights
	in_setting	The input recipe
	out_target	The output targets
	out_value	The output values
	param_value	The parameter values
	num_wafers_in_run	The number of wafers in next run

Table 6.1: BenchProcess Fields

In the BenchProcess class there are a number of final methods that cannot be overridden. These methods provide information to be used by the RbR benchmarking server and do not need to be considered when writing new simulators.

There are two abstract methods which must be overridden, Run() and Reset(), and one method that can be overridden, SetRecipe(). SetRecipe() is called by the benchmarking

server to give the simulator the next recipe from the client controller (and so therefore, is not directly used by the simulator). There is a default implementation of SetRecipe() which takes the input recipe values, checks them for correct bounding and discretization, and stores them in the in_setting field. However, this method can be overridden if additional functionality is desired (i.e. to add input noise). The Run() method is the method that is called by the benchmarking server to actually perform a run. It should use the value in in_setting to compute values for the out_value and param_value fields. The Run() method should also determine new settings for the num_wafers_in_run and out_target fields as needed. The Reset() method should be used to reset the process (i.e. set the drift component back to zero). Implementations of Run() and Reset() must be provided for any new process simulator.

ConnectProcess

The purpose of the ConnectProcess class is to provide the RbR benchmarking server with a means to connect to the available process simulators. This class consists of a static method named Connect() that takes the name of a simulator and returns an instantiated object of that simulator. The outcome of the connect attempt is recorded in the ErrorValue field. The ErrorValue field is set to zero for a successful connection and set to nonzero for a non-existent or inactive process. The Connect() process operates via a set of "if else" statements. So, when a new simulator is written, an "if else" statement needs to be added to ConnectProcess. See Figure 6.1 for the structure of the "if else" statement to be inserted. This comment block is in the ConnectProcess.java file, near the end. Simply copy this group of code directly above itself, remove the comments, and replace each instance of the word *NewProcess* with the name of the new simulator. This will allow the RbR benchmarking sys-

tem, and thus all client controllers, to connect to the new simulator.

```
/***** This is an example of a how to add a new process *****
 else if (processName.equals("NewProcess"))
   {
    if (INFO) System.out.println("NewProcess Process chosen");
    if (NewProcess.ACTIVE)
      {
       ErrorValue = 0;
       return (new NewProcess(seed));
      }
    else
      {
       if (DEBUG) System.out.println("NewProcess Process not active.");
       ErrorValue = 3;
       return null;
      }
   }
*****************************************************************/
```

Figure 6.1: ConnectProcess “if else” Model

6.2 Creating a New Simulator

The classes required to make a new process simulator for the RbR benchmarking system have been described in the previous two sections. There are two tasks to create a new simulator: 1) create a new class by extending the abstract BenchProcess class, and 2) add an entry for this newly developed simulator class to the ConnectProcess class. In this section, the creation of an actual simulator will be described. Appendix C contains a listing of the code for the Deposition simulator, and can be consulted to trace through the following discussion.

The first step is to make an extension of the abstract BenchProcess class and call it Deposition. This new class needs to be in the BenchSimulations package in order to be able to access the fields in BenchProcess. Also, since we will be using matrixCafe, and Java's Random class, we want to import those classes. Next, all the private fields and state variables that will be used to compute the output within the simulator should be declared.

The next step is to write a constructor, Deposition (long seed), that sets values for all of the constant fields and allocates space for the dynamic fields. In addition the constructor

should take care of any tasks specific to that controller, such as how to handle the seed provided by the client. In this case, if a non-zero seed is provided, it is used to seed the noise used for the disturbance state in the simulator, otherwise a random seed is used. As mentioned before, most of the fields are DoubleMatrixs, but a few are arrays of strings. In this simulator, since there are no parameters, the param_name and param_value fields are set to null. Notice that space is allocated for out_value, while the rest of the fields are given specific initial values. (One note should be made here about the format of strings used in the RbR benchmarking framework. Strings must be a continuous set of letters, numbers, or underscores and must start with a letter (i.e. no spaces). This includes the input, output, and parameter names as well as the name of the process itself. This is the reason for the dash in the input name.)

The next step is to override the int Run() method and use in_setting, and the rest of the fields and local variables, to generate out_value and update the dynamic variables as appropriate. Next, the int Reset() method should be overwritten and made to re-initialize the state variables of the process when it is called. For example, in this simulator, the two state variables, num_runs and elapsed_time, are set back to zero. Here there is no reason to override SetRecipe() since no additional functionality is desired in setting the input values, beyond checking for proper discretization and bounding.

Finally, after the new simulator, Deposition, is completed, an entry must be made for it in ConnectProcess by inserting the code within the comments in Figure 6.1 (with the "NewProcess" entries replaced with "Deposition") into the "if else" structure in ConnectProcess.Connect().

6.3 Existing Process Simulators

So far, ten process simulators have been implemented. These simulators are different

extensions of two different processes: a four input, two output CMP process, and a single input, single output metal sputter deposition process. The choice of these processes and the scenarios for these simulators was made with the help of feedback from the semiconductor industry as described in Chapter 4. (More information about each of these simulators can be found at http://www-mtl.mit.edu/rbrBench/BenchDoc/PackageBenchSimulations.html.) Table 6.2 contains the names and a description of each of these process simulators.

- CMP - a 4 input 2 output linear process model
  - Scenario 1: Discrete steps and bounds on recipes (CMP_ext1.java)
  - Scenario 2: Same as Scenario 1 with random lot sizes (CMP_ext2.java)
  - Scenario 3: Same as Scenario 2 with infrequent measurements (CMP_ex3.java)
  - Scenario 4: Same as Scenario 3 with missing measurements (CMP_ext4.java)
- Sputter Deposition - single input single output process model
  - Scenario 1: Discrete steps and bounds on the input (Deposition.java)
  - Scenario 2: Same as Scenario 1 with various thickness goals (Dep_ext1.java)
  - Scenario 3: Same as Scenario 2 with a constant measurement delay (Dep_ext2.java)
  - Scenario 4: Same as Scenario 3 with non-periodic measurements (Dep_ext3.java)
  - Scenario 5: Same as Scenario 4 with non-periodic delay (Dep_ext4.java)
  - Scenario 6: Same as Scenario 1 with outliers (Dep_ext5.java)

Table 6.2: Existing Process Simulators

Chapter 7

Integrating Clients into the RbR Benchmarking System

The ease of the connection of client controllers to the benchmarking system is an important barrier to overcome in developing a benchmarking system. Developers want to spend as little time as possible getting their controllers connected to the system so they can spend more time testing. Because of the expandable and distributed nature of the RbR benchmarking system, connection will have some complexities. However, much of the problem of having to communicate over the network via TCP/IP sockets has been alleviated by two sets of client integration utilities that have been written. The structure and use of these two integrations utilities, BenchClient and Bench_help, is described in this chapter. Actual uses of these integration utilities to connect two controllers to the benchmarking framework, and subsequent testing of these controllers against the RbR benchmarking simulators are presented in Chapter 8.

7.1 Client Integration Utilities

BenchClient is a client integration utility for controllers that are written in Java. Bench_help is one for controllers written in C. Both of these integration utilities perform the same task for their respective languages; they simplify the communications with the RbR benchmarking server by providing functions for each message in the RbR benchmarking messaging protocol. BenchClient is an abstract Java class (that provides a method and fields for each message) which can be extended to make a client. Bench_help is a set of C functions that implement the messages and a set of data structures for information storage that can be integrated into a client controller. For example, to send a Reset message to the benchmarking system, the controller would call the Reset() function. (For a complete listing of

the messages in the RbR benchmarking system see Appendix B.)

These two integration utilities are both used in a similar way. Each has the same functions and variables available for use. The rest of this section will describe the manner in which these functions and variables should be used, and the following two sub-sections will describe the individual integration utilities.

See Table 7.1 for a listing of the typical order in which the functions in the client integration utilities should be used and the order in which the variables should be accessed. Steps 1 through 4 in Table 7.1 are steps used to set up the connection with the RbR benchmarking server and the appropriate simulator, and so only need to be performed once for each connection. Steps 5 and 6 are used to reset the state of the simulator and to set the mode of the controller. The two types of modes are EXPERIMENTAL and CONTROL. Experimental mode should be used for design of experiments (DOEs) to determine a model for the process. Control mode should be used for actual control. Statistics used for evaluating the performance of controllers are only computed for runs made in control mode. Steps 7 through 11 are the steps needed to actually perform a run, and should be repeated for each desired run. Step 12 is used to get information on the runs performed during the entire connection, and Step 13 is a function that should be overwritten to locally record this information. Step 14 is used to disconnect from the RbR benchmarking server.

#	Function	Vars set before	Vars viewed after
1	ConnectToHost()	HOST, SOCK	
2	Init()	name	
3	SetSimulator()	process, seed	results_file

#	Function	Vars set before	Vars viewed after
4	GetProcessInfo()		num_inputs, num_outputs, num_params, in_name, out_name, param_name, in_hi_range, in_lo_range, in_resolution, in_weight, out_weight, param_value
5	Reset()		
6	SetMode()	mode	
7	GetRunInfo()		num_wafers_in_run, out_target
8	GenerateRecipe()		
9	SetRecipe()	num_runs, in_setting	
10	Run()		num_runs, out_value, param_value
11	RecordRunResults()		
12	GetCtrlResults()		num_exp_runs, num_ctrl_runs, results_file
13	RecordCtrlResults()		
14	Disconnect()		

Table 7.1: Use of Client Integration Utilities

BenchClient

BenchClient is the java version of the client integration utility described above. It consists of two classes BenchClient and ProcessInfo. (On-line documentation for these classes is available at http://www-mtl.mit.edu/rbrBench/BenchDoc/Package-BenchClient.html.) BenchClient is an abstract class that provides all of the (RbR benchmarking message implementing) methods necessary to connect a client controller. It contains one abstract method, GenerateRecipe, that should be overridden with the actual RbR control algorithm, and two unimplemented methods, RecordRunResults, and RecordCtrlResults that can be overridden to keep track of controller performance locally. BenchClient provides all of the methods described in Table 7.1. Table 7.2 describes where each of the fields used in integrating a

client controller are located. Note that most of the fields are contained in an object of the ProcessInfo class. This class stores most of the process information and provides some methods used by BenchClient.

ProcessInfo Fields	AbstractClient Fields	Description
num_inputs		The number of inputs
num_outputs		The number of outputs
num_params		The number of parameters
in_name		The inputs' names
out_name		The outputs' names
param_name		The parameters' names
in_hi_range		The inputs' upper bounds
in_lo_range		The inputs' lower bounds
in_resolution		The inputs' resolution
in_weight		The input weights
out_weight		The output weights
recipe		The input recipe
out_target		The output targets
out_value		The output values
param_value		The parameter values
num_wafers_in_run		The number of wafers in next run
num_runs		The number of runs in next Run message
num_exp_runs		Total number of experimental mode runs performed
num_ctrl_runs		Total number of control mode runs performed
	mode	Mode of the controller
	name	Name of the controller
	process	Name of the process to control

ProcessInfo Fields	AbstractClient Fields	Description
	results_file	Name of file results are written to
	seed	The seed value to be used

Table 7.2: AbstractClient and ProcessInfo Fields

Bench_help

Bench_help is the C version of the client integration utility. It is designed to look and perform just like the java version of the integration utility. There are three files, Bench_help.c, Bench_help.h and ProcessInfo.c. Bench_help.c and ProcessInfo.c contain the same functions as the corresponding classes in the java version. Bench_help.c contains the functions defined in Table 7.1, and ProcessInfo.c contains functions used by Bench_help.c. However the key difference is the way in which the variables described in Table 7.1 are stored. In the C version of the integration utility, Bench_help.h defines a set of structures that contain entries for each variable. To make a client, a program should be written which creates these structures, sets the appropriate variables and calls the functions in Bench_help.c, as described in Table 7.1. See Table 7.3 for a description of the variables that are contained in these structures.

struct proc_details Variables	struct run_info Variables	struct ctrl_results Variables	struct ctrl_details Variables
num_inputs			
num_outputs			
num_params			
in_name			
out_name			
param_name			
in_hi_range			
in_lo_range			

struct proc_details Variables	struct run_info Variables	struct ctrl_results Variables	struct ctrl_details Variables
in_resolution			
in_weight			
out_weight			
	recipe		
	out_target		
	out_value		
	param_value		
	num_wafers_in_run		
	num_runs		
	prev_num_runs		
		num_exp_runs	
		num_ctrl_runs	
			mode
			name
			process
			results_file
			seed

Table 7.3: Bench_help.h Structures

7.2 Example of an Integrated Controller

As a specific example of how to integrate an RbR controller into the RbR benchmarking framework, the listing of the code for the client controller Client.java in Appendix D is provided and can be consulted in parallel with the following discussion. Since this is a Java controller, it uses the java version (the AbstractClient and ProcessInfo classes) of the client integration utilities.

The first step is to make an extension of the abstract AbstractClient class and call it Client. Since we will be using matrixCafe, and the BenchClient class, we want to import those classes. Next, all the private fields and state variables that will be used to compute the next recipe within the controller should be declared.

The next step is to write a constructor, Client (), that sets values for the name, process, and seed fields. In this case, the Simple_Java_EWMA controller is controlling the Deposition process with a seed of 200. (Again note that the format of strings used in the messages must be alphanumeric with no spaces.)

The next step is to override the GenerateRecipe() method to cause it to compute the next recipe to use. This new recipe, and the number of runs to be performed via the next run message should be entered into the recipe and num_runs fields of the ProcessInfo instance. Note that the num_runs field is provided because the benchmarking system allows for the entry of multiple recipes (i.e. allows for multiple runs to be specified) in one Run message. This is done by entering an array of recipes into the recipe field, and the corresponding results from the simulator will be an array of outputs. Note however, in Client.java num_runs is always set to one.

Next, the RecordRunResults() and RecordCtrlResults() methods could be overwritten if desired, but in this case they are left unimplemented.

Since Client.java is a free-standing program, it needs a main() method which is located at the end of the listing. In this client's main() method, it instantiates an instance of itself, and then calls the Operate() method. The Operate() method is where the communications with the RbR benchmarking system takes place. (Using the Operate() method to do the communications was chosen out of convenience, but is not the only way to do it.)

The Operate() method uses the methods in the sequence described in Table 7.1. Note that a value is given to the mode field before the SetMode() method is called. In this imple-

mentation Steps 7 through 11 in Table 7.1 are repeated 100 times, to perform the actual control runs.

Chapter 8

Benchmarking Framework Verification

The final task in the development of the RbR benchmarking system was to perform a sample benchmarking of two RbR controllers. This chapter reports on the integration of these two controllers into the benchmarking framework and presents the results of a set of benchmark control runs performed. One of the controllers connected to the system was a previously existing exponentially weighted moving average (EWMA) controller written in C. This EWMA controller was then ported over to Java, and then this new Java version of the EWMA controller was connected to the benchmarking system.

8.1 Connection to the Benchmarking Framework

In the benchmarking framework, two client integration utilities are provided that help to connect an RbR controller to the RbR benchmarking server so that simulations can be performed and a benchmarking performed as described in Chapter 7. BenchClient is the client integration utility for controllers that are written in Java, and Bench_help is the one for controllers written in C. Both of these integration utilities perform the same task for their respective languages; they simplify the communications with the RbR benchmarking server. These integration utilities are described in more detail in Chapter 7 and on-line documentation can be found at http://www-mtl.mit.edu/rbrBench/BenchDoc/Package-BenchClient.html.

These integration utilities were used to connect the two RbR controllers to the RbR benchmarking server. The first controller is an EWMA controller written in C. Bench_help was used to help integrate this controller. The second controller is a Java-based version of

the EWMA controller that uses BenchClient to integrate into the RbR benchmarking system.

Connection of C RbR Controller

The C RbR controller that was connected to the RbR benchmarking system is an EWMA controller, written in previous work [23], that performs bounding and discretization and allows for input and output weightings. This controller was integrated into the benchmarking system with the help of the Bench_help integration utility as described in Chapter 7. The integrated controller prompts the user for the type of rounding to use, the system model for the process being simulated, and the EWMA weights to use. The rest of the information needed for control comes from the RbR benchmarking system and the specific simulator being accessed through the messaging system.

Connection of Java RbR Controller

In order to test the ability to connect a Java-based RbR controller to the benchmarking system, it was necessary to create one. Therefore, the C-based EWMA RbR controller described above was implemented in Java and then integrated into the benchmarking system with the help of the BenchClient integration utility. This controller was integrated almost exactly the same as the integrated controller shown in Appendix D, except the GenerateRecipe() method was replaced with one that simply makes a call to the new Java-based EWMA controller.

In the sample benchmarkings described in Section 8.2, the Java-based EWMA controller was specially set up to allow easier (qualitative) comparisons to be made. There were two modes it was used in. In the first mode, the EWMA weights were set to 0, which causes no change in the input to occur, thus providing the same recipe for each run. This is referred to as the no-update controller. In the second mode, the EWMA weights were set

to 1. In this mode, the next recipe is changed completely after each run based only on the previous output, thus not providing any smoothing (or averaging). This is referred to as the full-update controller.

8.2 Results of the Sample Benchmarking

Once these two controllers were integrated into the RbR benchmarking system, they were run and tested against two of the process scenarios available in the system. Tests were run on the CMP_ext1 and Deposition simulation scenarios. CMP_ext1 is a 4 input, 2 output CMP simulator that has discrete inputs and input bounds. Deposition is a 1 input, 1 output sputter deposition simulator that has discrete inputs, input bounds, and a constant target. More information on these and the other scenarios can be found at http://www-mtl.mit.edu/rbrBench/BenchDoc/Package-BenchSimulations.html. An important component of these simulators is the RbR controller's ability to provide a seed in the SetSimulator message. If a non-zero seed is provided by the controller, then the noise and randomness present in that control run can be recreated exactly in another run or by another controller by providing the same seed (while a seed value of zero provides different noise for each run). This is very useful in a benchmarking effort as it allows different controllers to be more accurately compared.

Figures 8.1 through 8.5 present some of the results from the control runs performed. Figure 8.1 is an example of the results printout from a control run that can be found at http://www-mtl.mit.edu/rbrBench/rbrBenchResults after a run has been performed. This particular printout is from control of the CMP_ext1 simulator by the C EWMA controller. Figures 8.2 and 8.3 are plots of the outputs and inputs from control runs performed on the CMP_ext1 simulator. Figure 8.2 is from a control run performed by the C controller, while Figure 8.3 is from the no-update Java controller. In both of these figures, the exact same

set of noise values are used so that one can clearly see the difference in the ability of the controllers to keep the process on target. The ability to use the same noise is achieved by giving a non-zero seed value to the benchmarking system; in this case the seed was set to 100 for both control benchmark runs. These two figures demonstrate the ability of the benchmarking system to compare the performances of different controllers. In this case, Figure 8.3 (with no model update) also illustrates the types of disturbances implemented by the CMP_ext1 scenario. In particular, one sees long drifts, large shifts (at run 30), and random "noisy" measurements (e.g. at runs 15 and 81). It is obvious that the C controller does a much better job at maintaining the targets than the no-update Java controller. Also notice that the inputs of the Java controller do not change, while those of the C controller provide more and more aggressive control to compensate for the drift.

Similarly, Figures 8.4 and 8.5 compare the performances of the C controller and the full-update Java controller on the Deposition process simulator with a seed value of 200. In Figure 8.5, the Java controller does modify the input to compensate for the change in the process model (unlike in the no-update case in Figure 8.3) as can be seen by comparing the actual output to the baseline output (the output that would have occurred had there been no control actions taken) represented by the dashed lines in Figures 8.4 and 8.5. Notice, however, the larger number of input modifications in Figure 8.5 than in Figure 8.4. The higher EWMA weight in the full-update Java controller causes more reactionary control actions than those of the C controller.

These test scenarios demonstrate the ability of the RbR benchmarking system to compare the performances of RbR controllers, and demonstrates the feasibility of RbR benchmarking and this benchmarking framework.These examples demonstrate the ability of the RbR benchmarking system to provide both a qualitative and quantitative comparison of controller performances.

```
Controller name:          MIT_C_EWMA
Process simulator selected:      CMP_ext1
Seed value used:          0
Number of inputs:         4
Input names:     Speed    Pressure         Force      Profile
Input low range:          11.0     1.0      1.0        -1.8
Input high range:         45.0     7.5      18.0       0.9
Input resolution:         0.5      0.1      0.1        0.05
Input weights:  28.0      10.0     32.0     4.0
Number of outputs:        2
Output names:    RemovalRate       Non-Uniformity
Output weights: 0.8       0.3
Number of parameters:     0
Parameter names:
Initial parameter values:

RUN DATA
numExp   numCtrl  setmodeMsg       mode     runMsg   num_runs            batch_size          inpu
0        0        1        1
0        1        0        1        1        1        1        36.0     4.4      6.4      -0.4
0        2        0        1        1        1        1        38.0     4.4      5.4      -0.4
0        3        0        1        1        1        1        39.0     4.4      5.0      -0.4
0        4        0        1        1        1        1        40.0     4.4      4.6      -0.4
0        5        0        1        1        1        1        40.0     4.4      4.6      -0.4

SUMMARY STATISTICS
Number of Experimental Runs: 0
Number of Control Runs: 5
Output Mean Squared Error:       3526.375731455176         64.33703930093097
Input Mean Squared Change:       1.5      0.0      0.3300000000000002       0.0
```

Figure 8.1: Printout of Results from a Sample Control Run

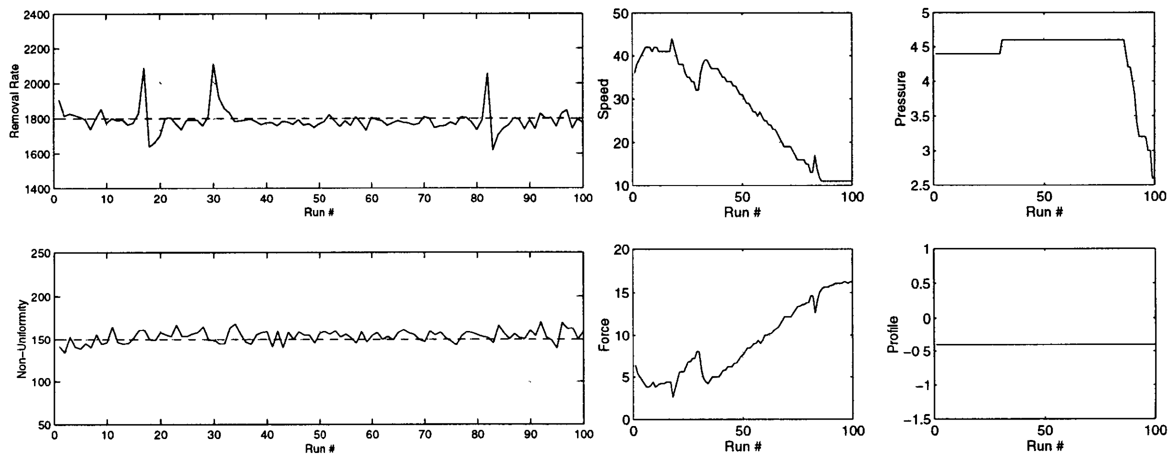


Figure 8.2: Control Run on CMP_ext1 by C-based Controller

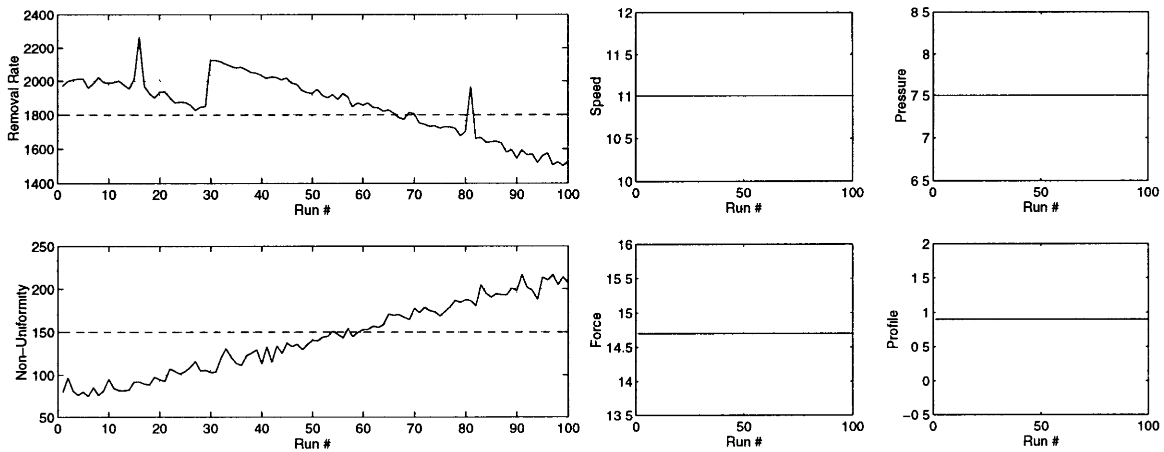


Figure 8.3: Control Run on CMP_ext1 by No-update Java-based Controller

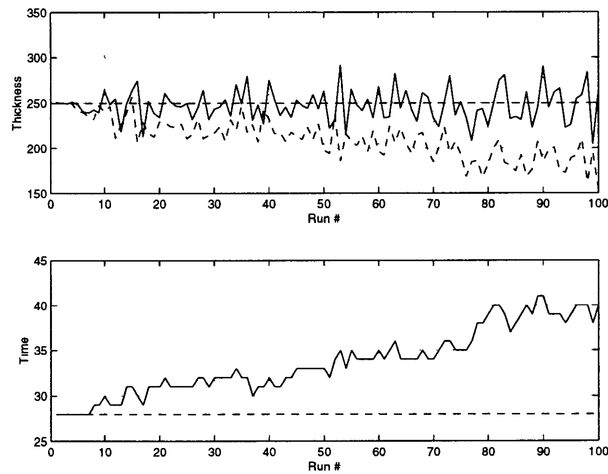


Figure 8.4: Control Run on Deposition by C-based Controller

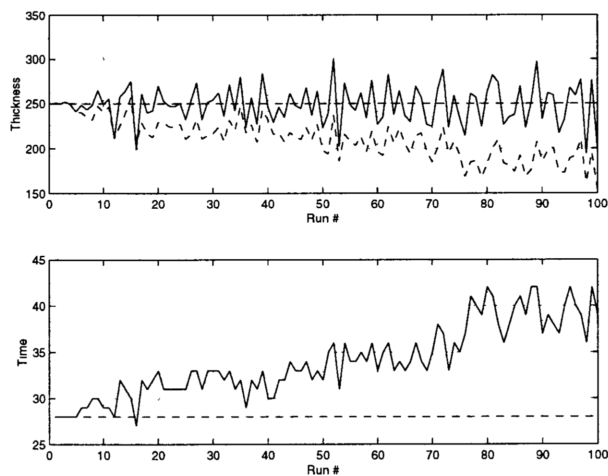


Figure 8.5: Control Run on Deposition by Full-update Java-based Controller

Chapter 9

Conclusions and Future Work

This thesis has presented the definition and development of an RbR control benchmarking and simulation system. We have shown the feasibility of RbR benchmarking and have created a framework that allows simple connection of controllers to a set of process scenario simulators to support the simulation, benchmarking and comparison of RbR control products. This thesis contained two primary phases: first, the determination of the capabilities and performance of RbR controllers, and the requirements and specifications for a benchmarking system and its process scenarios; and second, the creation and verification of a benchmarking system including a server, simulators, and client integration utilities.

The first phase involved a gathering of information through a survey of RbR control literature, and the development of a questionnaire that was distributed to semiconductor industry companies. This phase of the thesis was critical in the development of a benchmarking framework that is indeed feasible, and served to determine the needs and wants of the benchmarking framework user. The goal of the literature search was to identify results and benefits gained in the semiconductor and other industries through the use of RbR control, while the industry questionnaire aimed to identify the key problems, opportunities, and scenarios for RbR control, as well as the needs or requirements for successful product adoption. Once this information was gathered it was used to help determine the makeup of the benchmarking system, including the specific process scenarios that should be used to fully test the ability of an RbR controller. Four control scenarios, with extensions, were selected. These four scenarios (deposition, CMP, plasma etch, and photolithography) cover most of the disturbances and problems that an RbR controller is likely to encounter.

The second phase consisted of developing a software framework to support the benchmarking process. This framework was designed to be a robust, expandable, distributable, platform-independent system that is easy to use and connect to, while still maintaining the look and feel or a real process tool. This framework incorporates four tiers: a messaging system, a server, clients, and simulators.

1. The first tier is a messaging system that allows the client controllers to communicate with the server and the simulators. This messaging system was designed to be robust and flexible enough to handle all types of RbR control situations that may be encountered.
2. The next tier is the benchmarking server that allows the controller and the simulator to come together. It handles all of the communications between the two and also handles the recording of the results of the control run taking place.
3. The third tier is the connection of RbR controllers to the benchmarking system to act as clients to control a simulator through the benchmarking server. There are two client integration utilities that are included in the framework: BenchClient for Java-based controllers, and Bench_help.c for controllers written in C.
4. The fourth tier is the set of process simulators themselves. These simulators are based on the control scenarios and extensions defined in phase one of this thesis and were integrated into the framework with the help of a set of java classes that are used to incorporate new simulators into the framework, BenchProcess and ConnectProcess.

Finally, once these two phases of the thesis were completed (background research, and system development), it was time to test the feasibility of benchmarking RbR controllers. This was done by incorporating two RbR controllers into the benchmarking framework and running control runs against some of the simulators. This proved that the benchmarking framework does indeed perform as designed, and that the ability to connect controllers

and simulators to the system does indeed exist. Therefore the ability of the benchmarking system to support simulation, benchmarking and controller comparison was demonstrated.

There are many exciting directions that this project can take in the future. As with all projects of this nature, the gathering of information (as performed in phase one) should be an on-going process. Close interactions with industry and developers of RbR controllers should be maintained and the benchmarking system should be left open to changes should new information become available. During development of this framework we have discovered that the RbR benchmarking framework promises utility not only as a benchmarking tool but also as a simulation and testing tool because of its design features. Future work could include a general RbR controller benchmarking effort where industry members could evaluate RbR controllers. A smaller scale benchmarking could also be performed on a specific process simulator, by a tool company or a specific fab, to determine the best RbR controller for that process. In either case, the creation of a standard set of scenarios and benchmarks will foster comparisons and discussion within industry. Another avenue of exploration for this framework could be as a tool in developing controller algorithms and/or process models. This could become even more important if the messages used in this framework help lead to an industry standard for the protocol or message content for communicating between RbR controllers, process sensors, and tools. Evaluating the merits of such a standard could be a very interesting continuation of this thesis.

References

[1] V. Bakshi, "Benchmarking of Commercial Software for Fault Detection and Classification (FDC) of Plasma Etchers for Semiconductor Manufacturing Equipment," *Proc. of the Amer. Ctrl. Conf.*, Vol. 3, pp. 1579-82, June 1997.

[2] T. Smith, D. Boning, J. Stefani, and S. Butler, "Run By Run Advanced Process Control of Metal Sputter Deposition," *TI Internal Report*, 1996.

[3] D. Boning, W. Moyne, T. Smith, J. Moyne, R. Telfeyan, A. Hurwitz, S. Shellman, and J. Taylor, "Run By Run Control of Chemical Mechanical Polishing," *IEEE Trans. on Components, Packaging, and Manufacturing Technology*, Vol. 19, No. 4, pp. 307-314, Oct. 1996.

[4] M. Le, "Variation Reduction in Plasma Etching via Run-to-Run Process Control and Endpoint Detection," S.M. Thesis, MIT EECS, July 1997.

[5] D. C. Montgomery, *Introduction to Statistical Quality Control*, John Wiley, New York, 1985.

[6] E. Sachs, A. Hu, and A. Ingolfsson, "Run by Run Process Control: Combining SPC and Feedback Control," *IEEE Trans. on Semi. Manuf*, Feb. 1995.

[7] T. Smith, "Novel Techniques for the Run by Run Process Control of Chemical-Mechanical Polishing," S. M. Thesis, MIT EECS, May 1996.

[8] E. Del Castillo and A. Hurwitz, "Run to Run Process Control: a Review and Some Extensions," J. Qual. Tech., 1997.

[9] S. Butler, and J. Stefani, "Application of Predictor Corrector Control to Polysilicon Gate Etching," *Proc. of the American Control Conference*, San Francisco, June 1993.

[10] T. Smith, and D. Boning, "A Self-Tuning EWMA Controller Utilizing Artificial Neural Network Function Approximation Techniques," *International Electronics Manufacturing Symposium*, IEMT '96, Oct. 1996.

[11] E. Del Castillo, and J. Yeh, "An Adaptive Run-to-Run Optimizing Controller for Linear and Nonlinear Semiconductor Processes," *IEEE Transactions on Semiconductor Manufacturing,* Mar. 1997.

[12] J. Baras, and N. Patel, "Designing Response Surface Model Based Run by Run Controllers: A New Approach," *IEEE/CMPT International Manufacturing Technology Symposium*, pp. 210-217, 1995.

[13] J. Mullins, W. Campbell, and A. Stock, "An Evaluation of Model Predictive Control in Run to Run Processing in Semiconductor Manufacturing."

[14] M. Hankinson, T. Vincent, K. Irani, and P. Khargonekar, "Combined Real-Time and Run-to-Run Control of Etch Depth and Spatial Uniformity in Plasma Etching," *Journal of the Electrochemical Society*, vol. 144, no. 7, pp. 2473-2479. July 1997.

[15] Z. Ning, J. Moyne, T. Smith, D. Boning, E. Del Castillo, J. Y. Yeh, and A. Hurwitz, "A Comparative Analysis of Run-to-Run Control Algorithms in the Semiconductor Manufacturing Industry," *Seventh Annual SEMI/IEEE Advanced Semiconductor Manufacturing Conference and Workshop*, Cambridge, Mass., Nov. 1996.

[16] D. Boning, W. Moyne, T. Smith, J. Moyne, and A. Hurwitz, "Practical Issues in Run by Run Control," *Sixth Annual SEMI/IEEE ASMC*, Boston, Nov. 1995.

[17] D. Boning, "Run by Run Control Benchmarking," A White Paper, Oct. 1996.

[18] G. Smith and I. Highberg, "Real Time Statistical Process Control for Improved Etch Tool Performance," *J. Vac. Sci. Tech.*, Jan. 1996.

[19] S. Butler and J. Stefani, "Supervisory Run-to-Run Control of Polysilicon Gate Etch Using *In Situ* Ellipsometry," *IEEE Trans. on Semi. Manuf.*, May 1994.

[20] A. Gower, "An Architecture for Flexible Distributed Experimentation and Control with an AME 5000 Plasma Etcher," S.M. Thesis, MIT EECS, August 1996.

[21] S. Leang, S.-Y. Ma, J. Thomson, B. J. Bombay, and C. J. Spanos, "A Control System for Photolithographic Sequences," *IEEE Trans. on Semi. Manuf.*, Vol. 9, No. 2, pp. 191-207, May 1996.

[22] J. Carney, "Message Passing Tools for Software Integration," S. M. Thesis, MIT EECS, June 1995.

[23] W. Moyne, "Run by Run Control: Interfaces, Implementation, and Integration," S. M. Thesis, MIT EECS, May 1995.

Appendix A

Run by Run Process Control Benchmarking Surveys

SEMATECH Run By Run Process Control Benchmarking

SURVEY - Also available at http://www-mtl.mit.edu/rbrBench/Survey.html

Name:
Address:

Phone:
Fax:
Email:

Please return survey to:
Prof. Duane S. Boning
MIT Room 39-567
Cambridge, MA 02139
Email: boning@mit.edu
Ph: 617-253-0931
Fax: 617-253-9606

Run by run (RbR) process control is a methodology which utilizes process measurements to modify process settings on a run by run or lot by lot basis to improve or maintain process performance in the face of process drift or other disturbances. We are currently soliciting input from SEMATECH member companies regarding the benchmarking of process controllers.

2. What processes and equipment could most benefit from RbR process control, and why?

3. Are there issues which RbR controllers should address that have not been considered?

4. Please rate the importance of each of the following criteria for RbR process controllers (1 - very important through 5 - not important.)

____ Simplicity of algorithm

____ Ability to handle different processing conditions

____ Integration with CIM systems

____ Ability to handle complex sensor information

____ Ability to incorporate complex process models

____ Customer support for RbR packages

5. Please provide contact information for any specific controllers you believe should be included in RbR benchmarking.

Closing Remarks - The usefulness and appropriateness of the benchmarking project depends on your response to this survey. Please return the survey and any additional comments or questions to authors at the address above. Thank you. - T. Smith, A. Gower, E. Stuckey, and D. Boning

SEMATECH Run By Run Process Control Benchmarking

Appendix - Extended Survey

Name:	Please return survey to:
Address:	Prof. Duane S. Boning
	MIT Room 39-567
	Cambridge, MA 02139
Phone:	Email: boning@mit.edu
Fax:	Ph: 617-253-0931
Email:	Fax: 617-253-9606

Run by run (RbR) process control is a methodology which utilizes process measurements to modify process settings on a run by run or lot by lot basis to improve or maintain process performance in the face of process drift or other disturbances. We are currently soliciting input from SEMATECH member companies regarding the benchmarking of process controllers.

General Overview

The SEMATECH run by run (RbR) process control benchmarking project is aimed at providing information about the technical capability and available implementations (commercial or otherwise) to the general semiconductor manufacturing community.

We are currently soliciting input on the issues the community feels are important to the success of the RbR process control benchmarking project. In addition to the "short form" (one page) survey, we have put together this more extensive set of questions. The questions are divided into the following three main sections: general questions about processes and the benchmarking project, questions regarding the criteria by which the technical capability of the RbR process control methodologies will be judged, and questions regarding the basis for which implementations of these control algorithms will be compared and contrasted.

General Questions

These are questions probing which process technologies have difficult problems which run by run controllers may be able to address, what the major categories of interest are in comparing RbR control methods, and finally what tasks are needed to accomplish the project goals.

1. What are current process or equipment barriers and/or control problems (e.g. significant uniformity problems, machine wear and target drift, tuning complexity, etc.) in the following:
   a) Epitaxial Film Growth

   b) Chemical Vapor Deposition

   c) Thermal Oxidation

d) Diffusion

e) Ion Implantation

f) Sputter Deposition

g) Lithography

h) Plasma Etching

i) Chemical-Mechanical Polishing

j) Rapid Thermal Processing

2. Where and when do you see RbR process control being used?

3. Are there particular advanced processes which you believe may require RbR process control in order to achieve future requirements?

4. What current processes might most benefit from RbR process control, and why?

5. What criteria do you see as being most important to the benchmarking of RbR process control solutions (e.g. applicability to specific processing scenarios, experimental verification, level of implementation or commercialization)?

6. What can the benchmarking team do to make the benchmarking system widely available?

7. What are effective ways to get the community involved in the project?

8. What else can the benchmarking team do to make the RbR process control benchmarking project a success?

Technical Capability Issues

This section of the survey is designed to assess the technical capability of the various RbR control algorithms in the face of several different semiconductor process control scenarios. The focus here is on technical quality of algorithms, rather than focusing on whether a commercialized version with factory integration exists.

We have initially chosen to segment the technical capability section into realistic scenarios a controller might face in the industry. This allows the controller the freedom to run design of experiments, simulated test runs, and simulated control runs. The following questions are aimed at determining what scenarios are important and challenging.

1. Should the RbR technical capability assessment be segmented into scenarios? If not, how should the methodologies be segmented and tested?

2. What processes are least and most formidable to control on a run to run or lot to lot basis?

3. What types of sensor information should a RbR process controller be able to handle?

4. What practical issues are important for RbR controllers to address (e.g. bounds checking, recipe rounding, dead-bands, or embedded SPC)?

5. Should RbR process controllers be assessed with regard to the complexity (order) of processes which they are able to control?

6. What types of process disturbances are common in semiconductor manufacturing?

7. What other technical criteria should RbR process controllers be compared and contrasted on (e.g. questions 4-6)?

Commercial Availability and Implementation Issues

This section of the survey is aimed at assessing the implementation quality of RbR process control solutions. Although a RbR process control methodology may be technically capable, its ability to perform reliably in an integrated fashion with factory computer integrated manufacturing (CIM) systems and equipment automation systems is often an essential criteria in the assessment of the benefit of a particular control solution.

The questions that follow attempt to determine the critical integration issues for RbR control software solutions.

1. What are the key integration issues a controller must address in order to be viable for use in a fab?

2. What communication tools should a controller be able to support
   (e.g. SECS, CORBA, CIM Compliant, stand-alone GUI)?

3. Should controllers have time restrictions on recipe generation (i.e. a limitation on how long it takes to generate or suggest a new recipe)? If so, should these be process dependent?

4. What platforms should controller systems run on?
   What platforms should controller systems be *required* to run on?

5. Should control systems support plug and play modular control methodologies
   (Plug and play control modules within a cell controller)?

6. Should control solutions be client/server?

7. Should control solutions have embedded database technologies?

8. Should our benchmarking project report which controllers are available for which processes?

9. Should the project report pricing for commercial products?

Closing Remarks

The usefulness and appropriateness of the benchmarking project depends on your response to this survey. Please return the survey and any additional comments or questions to authors at the address above. Thank you!

Appendix A

Run by Run Process Control Messaging System

This appendix contains a description of the messages used to communicate with the RbR benchmarking server. These messages use the John Carney Messaging system [22]. The S_HDR entry identifies the initial message, and the R_HDR entry identifies the resulting return message. The entries that follow these are the corresponding initial and reply variables. The columns of the tables correspond to the variable name, the variable type, and the value to be entered in that variable. For example, in the GetProcessInfo message below, the client sends a message with no variables, and the server replies by sending variables describing the details of the process to be simulated.

GetProcessInfo Message

S_HDR:	OB_ASCII	"S_GETPROCESSINFO"
R_HDR:	OB_ASCII	"R_GETPROCESSINFO"
NUM_INPUTS:	OB_LONG	n (Number of controllable inputs to the system)
IN_NAME:	OB_BYTE[]	(Names of the inputs embedded in a sub-message with title IN_NAMES with an ASCII entry (labeled Name1, Name2...) for each input.)
IN_HI_RANGE:	OB_DOUBLE[n]	(Upper limit of range for input i)
IN_LO_RANGE:	OB_DOUBLE[n]	(Lower limit of range for input i)
IN_RESOLUTION:	OB_DOUBLE[n]	(Spacing between possible settings of input i)
IN_WEIGHT:	OB_DOUBLE[n]	(Preferences for changing input i)
OUT_WEIGHT:	OB_DOUBLE[k]	(Preferences for changing output i)
NUM_OUTPUTS:	OB_LONG	k (Number of controlled outputs of the system)
OUT_NAME:	OB_BYTE[]	(Names of the outputs embedded in a sub-message with title OUT_NAMES with an ASCII entry (labeled Name1, Name2...) for each output.)
NUM_PARAMS:	OB_LONG	r (Number of uncontrollable parameters in system)
PARAM_NAME:	OB_BYTE[]	(Names of the parameters embedded in a sub-message with title PARAM_NAMES with an ASCII entry (labeled Name1, Name2...) for each parameter.)
PARAM_VALUE:	OB_DOUBLE[r]	(Setting of parameter i)
RETCODE:	OB_SHORT	(return code) • 0 = Success • 1 = Error parsing message • 99= Error, no process specified yet via Connect msg

SetMode Message

S_HDR:	OB_ASCII	"S_SETMODE"
SETTING:	OB_SHORT	(Set to Experimental or Control mode) • 0 = Set to Experimental mode (Default) • 1 = Set to Control mode
R_HDR:	OB_ASCII	"R_SETMODE"
RETCODE:	OB_SHORT	(return code) • 0 = Success • 1 = Error parsing message • 2 = Error, invalid mode

SetRecipe Message

S_HDR:	OB_ASCII	"S_SETRECIPE"
NUM_RUNS:	OB_LONG	l (Num runs this message specifies inputs for)
IN_SETTING:	OB_DOUBLE[l*n]	(Setting for input i. n inputs in a row for each of the l runs)
R_HDR:	OB_ASCII	"R_SETRECIPE"
RETCODE:	OB_SHORT	(return code) • 0 = Success • 1 = Error parsing message • 99 = Error, no process specified yet via Connect msg

GetRunInfo Message

S_HDR:	OB_ASCII	"S_GETRUNINFO"
R_HDR:	OB_ASCII	"R_GETRUNINFO"
NUM_WAFERS:	OB_LONG	(Number of wafers to be processed in next run)
TARGET:	OB_DOUBLE[k]	(Target for output i)
RETCODE:	OB_SHORT	(return code) • 0 = Success • 1 = Error parsing message • 99 = Error, no process specified yet via Connect msg

Run Message

S_HDR:	OB_ASCII	"S_RUN"
S_HDR:	OB_ASCII	"R_RUN"
RETCODE:	OB_SHORT	(return code) ● 0 = Success ● 1 = Error parsing message ● 99 = Error, no process specified yet via Connect msg

GetCtrlResults Message

S_HDR:	OB_ASCII	"S_GETCTRLRESULTS"
R_HDR:	OB_ASCII	"R_GETCTRLRESULTS"
NUM_EXP:	OB_LONG	(Number of experimental runs performed)
NUM_CTRL:	OB_LONG	(Number of control runs performed)
RESULTS_FILE:	OB_ASCII	(File where results from this run are stored)
RETCODE:	OB_SHORT	(return code) ● 0 = Success ● 1 = Error parsing message ● 99 = Error, no process specified yet via Connect msg

Disconnect Message

S_HDR:	OB_ASCII	"S_DISCONNECT"

Reset Message

S_HDR:	OB_ASCII	"S_RESET"
R_HDR:	OB_ASCII	"R_RESET"
RETCODE:	OB_SHORT	(return code) ● 0 = Success ● 1 = Error parsing message ● 99 = Error, no process specified yet via Connect msg

Server -> Controller Messages

RunResults Message

(Sent after a Run message is sent by controller)

S_HDR:	OB_ASCII	"S_RUNRESULTS"
SUCCESS:	OB_SHORT	(Flag to indicate successful run) • 0 = Success • 1 = Failure
NUM_RUNS:	OB_LONG	l (Num runs this message gives results for)
OUT_MEAS:	OB_DOUBLE[k]	(Measured output after a run. k outputs in a row for each of the l runs)
OUT_PARAMS:	OB_DOUBLE[r]	(Parameters from the previous run. r params in a row for each of the l runs)
R_HDR:	OB_ASCII	"R_RUNRESULTS"
RETCODE:	OB_SHORT	(return code) • 0 = Success • 1 = Error parsing message

Disconnect Message

S_HDR:	OB_ASCII	"S_DISCONNECT"

Appendix B

Example Simulator: Deposition.java

```
package BenchSimulations;

import matrixCafe.*;
import java.util.Random;

/**
 * The <code>Deposition</code> class is a single input, single output
 * metal sputter deposition process simulator.
 * The input is Time-seconds.
 * The output is Thickness
 * This extension uses a constant target thickness.
 *
 * @author  Eric Stuckey
 * @version 11/27/97
 */
public class Deposition extends BenchProcess
  {
   private static final boolean DEBUG = false;
   private static final boolean ERRMSG = true;

    /**
     * The number of runs so far.
     */
   private DoubleMatrix num_runs = new DoubleMatrix(new double[]{0});

    /**
     * This is the baseline rate for the process.
     */
   private DoubleMatrix rate = new DoubleMatrix(new double[] {8.735});

    /**
     * Parameters for the Deposition disturbance state model.
     */
   private DoubleMatrix drift_ratio=new DoubleMatrix(new double[] {.3});
   private DoubleMatrix drift_time=new DoubleMatrix(new double[] {10});
   private DoubleMatrix kWH_per_sec=
        new DoubleMatrix(new double[] {.0027777778});
   private DoubleMatrix noise_std=new DoubleMatrix(new double[] {.5});
   private DoubleMatrix elapsed_time=new DoubleMatrix(new double[] {0});

    /**
     * Whether or not the process is active.
     */
   public static final boolean ACTIVE = true;

    /**
     * The Random noise class to use.
     */
```

```
private Random thick_noise;


 /**
  * Constructor for <code>Deposition</code>.
  *
  * @param     seed    The seed (for noise) provided by the client.
  *                    A value of zero indicates random noise.
  */
public Deposition(long seed)
  {
   super(seed);
   if (DEBUG) System.out.println("In Deposition's Constructor");

   if (seed != 0)
     {
      thick_noise = new Random(seed);
     }
   else
     {
      thick_noise = new Random();
     }

   num_inputs = new DoubleMatrix(new double[] {1});
   in_name = new String[] {"Time-seconds"};
   in_hi_range = new DoubleMatrix(new double[] {150});
   in_lo_range = new DoubleMatrix(new double[] {0});
   in_resolution = new DoubleMatrix(new double[] {1});
   in_weight = new DoubleMatrix(new double[] {1});
   out_weight = new DoubleMatrix(new double[] {1});

   num_outputs = new DoubleMatrix(new double[] {1});
   out_name = new String[] {"Thickness"};
   out_target = new DoubleMatrix(new double[] {250});
   out_value = new DoubleMatrix(1,1);

   num_params = new DoubleMatrix(new double[] {0});
   param_name = null;
   param_value = null;

   num_wafers_in_run = new DoubleMatrix(new double[] {1});
  }

 /**
  * Performs a single run.
  *
  * @return     An error code:  0 for a successful run,
  *                             nonzero otherwise.
  */
public int Run()
  {
   String src = "Deposition.Run(): ";

   DoubleMatrix noise;
```

```
      DoubleMatrix drift;

      try
        {
          // Keep track of the number of runs.
         num_runs = num_runs.plus(num_wafers_in_run);

/* Calculate the disturbance state. */
          // Set up Noise.
         noise = noise_std.times
              (new DoubleMatrix(1,1,thick_noise.nextGaussian()));

          // Set up Drift.
         drift = DoubleMatrix.ones(1).minus
            ((drift_ratio.times(kWH_per_sec.divide(drift_time)))
            .times(elapsed_time));

          // Calculate the rate for this run.
         rate = (drift.times(rate)).plus(noise);

         /* Calculate the output. */
         out_value = rate.times(in_setting);

          // Keep track of the amount of time machine has been run.
         elapsed_time = elapsed_time.plus(in_setting);
        }
      catch (MatrixDimensionException e)
        {
         if (DEBUG) System.out.println (src+e.getMessage());
         return 1;
        }
      catch (MatrixException e)
        {
         if (DEBUG) System.out.println (src+e.getMessage());
         return 1;
        }

      return 0;
    }

   /**
    * Resets the controller.
    *
    * @return     An error code:  0 for a successful reset,
    *                             nonzero otherwise.
    */
  public int Reset()
    {
     String src = "Deposition.Reset(): ";

     try
       {
        num_runs.setElement(0,0,0);
        elapsed_time.setElement(0,0,0);
```

```
        }
       catch (MatrixDimensionException e)
        {
         if (DEBUG) System.out.println (src+e.getMessage());
         return 1;
        }

       return 0;
      }
}
```

Appendix C

Example Client Controller: Client.java

```
import BenchClient.*;
import matrixCafe.*;

/**
 * The <code>Client</code> class is an example of a Client controller
 * program written in Java.
 *
 * @author  Eric Stuckey
 * @version 08/25/97
 */
public class Client extends AbstractClient
  {
   private static final boolean DEBUG = false;
   private static final boolean ERRMSG = true;

    /**
     * These are the parameters for the CMP_ext1 plant model.
     */
   private DoubleMatrix A = new DoubleMatrix(new double[] {10});
   private DoubleMatrix invA = new DoubleMatrix(new double[] {0.1});
   private DoubleMatrix b = (new DoubleMatrix(new double[] {0}));
   private boolean FirstRun = true;
   private DoubleMatrix num_runs = new DoubleMatrix(new double[] {1});
    // recipe's dimensions are nun_runs by num_inputs.
   private DoubleMatrix recipe = new DoubleMatrix(1,1);

    /**
     * Constructor for <code>Client</code>.
     */
   Client()
     {
      super();
      String src = "Client.Client(): ";

      name = "Simple_Java_EWMA";
      process = "Deposition";
      seed = 200;
     }


    /**
     * This method generates the Recipe for the next run
     * (or sequence of runs.)
     */
   public void GenerateRecipe()
     {
      String src = "Client.GenerateRecipe(): ";
```

```
    try
      {
        // If it's the first run pick a dummy last recipe.
       if (FirstRun)
         {
          recipe.setElement(0,0,30.0);
          FirstRun = false;
         }
        // Otherwise compute the next recipe.
       else
         {
           // Update the EWMA offset. A full update controller.
  b = (processInfo.out_value.transpose()).
              minus(A.mtimes(recipe.transpose()));

   // Compute the next recipe.
          recipe = invA.mtimes((processInfo.out_target).minus(b));

           // Round and discretize the recipe.
          for (int j=0; j<(processInfo.num_inputs).getElement(0); j++)
            {
             double temp = ((double)((int)((recipe.getElement(0,j)
                  / processInfo.in_resolution.getElement(j,0))+0.5)))
                  * processInfo.in_resolution.getElement(j,0);
             if (temp < processInfo.in_lo_range.getElement(j,0))
                temp = processInfo.in_lo_range.getElement(j,0);
             else if (temp > processInfo.in_hi_range.getElement(j,0))
                temp = processInfo.in_hi_range.getElement(j,0);
             recipe.setElement(0,j,temp);
            }
         }

        // Set the recipe and num_runs in the processInfo object.
       processInfo.num_runs = num_runs;
       processInfo.recipe = recipe;
      }
   catch (MatrixDimensionException e)
      {
       if (ERRMSG) System.err.println(src+e.getMessage());
       Exit(1);
      }
  }


  /**
   * This method records the control results after each run.
   * It is not currently implemented.
   */
 public void RecordRunResults()
  {
  }


  /**
```

```
   * This method records the final control results.
   * It is not currently implemented.
   */
 public void RecordCtrlResults()
   {
   }


  /**
   * This is the <code>Operate</code> method.
   * It does all of the control for the given process.
   */
 private void Operate()
   {
    ConnectToHost();

    Init();
    SetSimulator();
    GetProcessInfo();

    Reset();
    mode = EXPERIMENTAL;
    SetMode();

    for (int i=0; i < 100; i++)
      {
       GetRunInfo();
       GenerateRecipe();
       SetRecipe();
       Run();
       RecordRunResults();
      }

    /******
    Reset();
    mode = CONTROL;
    SetMode();

    for (int i=0; i < 100; i++)
      {
       GetRunInfo();
       GenerateRecipe();
       SetRecipe();
       Run();
       RecordRunResults();
      }
    ******/

    GetCtrlResults();
    RecordCtrlResults();

    Disconnect();
   }
```

```
  /**
   * This is the main method for the <code>Client2</code> program.
   * It does all of the control for the given process.
   *
   * @param     args  the command line arguments to the program.
   *                  None currently implemented.
   */
 public static void main(String[] args)
   {
    Client client = new Client();
    client.Operate();
   }
}
```