Run by Run Control: Interfaces, Implementation, and Integration

by

William P. Moyne

B.S. in Electrical Engineering, The University of Michigan-Dearborn (1992)

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

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

Author .....
Department of Electrical Engineering and Computer Science
February 3rd, 1995

Certified by.....
Donald E. Troxel
Professor of Electrical Engineering
Thesis Supervisor

Certified by.....
Duane S. Boning
Assistant Professor of Electrical Engineering
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Students

Run by Run Control: Interfaces, Implementation, and Integration

by

William P. Moyne

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

Abstract

Run by run (RbR) control is a form of adaptive model based process control where recipe changes are performed between runs of the process. It is becoming popular in the area of VLSI processing but its acceptance has been hindered by integration issues. Existing systems cannot be replaced due to massive capital investments, so the RbR controller must mesh with these systems without requiring large modifications. Two steps have been taken to ease this integration. First, an abstract data model has been developed for RbR control which can be easily communicated between dissimilar modules. Second, a three tier communication model has been developed to allow multiple levels of interaction with the RbR control module. These layers complement the underlying data model.

An RbR control server has been implemented to demonstrate the robustness of the communication model and data abstraction. This server provides RbR control to a variety of clients via TCP/IP network sockets. One of these clients is a graphical user interface that allows direct operation of the control algorithms. This can be a powerful tool when evaluating new control algorithms or testing equipment models. The interface contains a set of simulation, graphing, and archiving tools to aid in the testing and operation of the server. The controller has been integrated into a local computer integrated manufacturing (CIM) system, as well as a control framework being developed by The University of Michigan and SEMATECH.

In addition to interface issues, the control algorithms themselves have been enhanced to enable a variety of constraints and bias terms to be added to the models. The controller currently contains two control algorithms, but is designed to be expanded to include more algorithms as they are developed. Data needed for these new algorithms can be constructed using the data abstraction without disrupting existing modules. Implementation, interface, and integration barriers to the adoption of run by run control have been reduced by the definitions and demonstrations presented in this thesis.

Thesis Supervisor:	Duane S. Boning
Title:	Assistant Professor of Electrical Engineering
Thesis Supervisor:	Donald E. Troxel
Title:	Professor of Electrical Engineering

Acknowledgments

There are many people who have helped me with my research in the past year. I cannot possibly thank them all, but I will try.

When I first arrived at MIT I had no financial support and no research topic. I would like to thank Professor Donald Troxel and Professor Duane Boning for asking me to join their research group and funding my education under contracts with ARPA and SEMATECH. I would also like to thank Professor Boning for providing a research topic and guiding me to its completion.

I am also fortunate to be part of the SEMATECH J88D run by run control project. This project has been key in pushing me toward my thesis goal, and has complemented my research. Each member of the team has, at one time or another, provided useful if not vital insight into some aspect of my work. Arnon Hurwitz, the team leader, has been both patient and supportive of my research efforts. James Moyne has always made sure that I kept a clear view of my objective and do not let the views of others cloud my judgment. John Taylor has shown that humor is sometimes the best research method of all, and backs it up with a solid grasp of reality. Even with the unpleasant job of understanding my code, Roland Telfeyan has been supportive in my attempts to define and develop a generic run by run controller interface.

I would like to thank Arthur Altman for performing run by run tests at Digital Equipment Corporation and providing both data and interface suggestions along the way.

My work has also been complemented by my fellow research assistants and office mates. John Carney has simplified my TCP/IP work greatly by writing a generic interface library. Thomas Lohman has provided endless support on object oriented programming as well as UNIX tips. Greg Fischer has been key in the run by run - CAFE integration. Myron Freeman has helped with the arduous NeXT Step installation along with various other network tasks. Taber Smith provided insight into the run by run algorithms. Michael McIllrath, Jimmy Kwon, James Kao, Aaron Gower, Ka Shun Wong, David White have provided feedback and ideas that enhanced my work.

My research would not be possible without emotional support from my mother, my father and Kathie.

My work has been made possible by support from the Advanced Research Projects Agency (ARPA) contract #N00174-93-C-0035 and SEMATECH.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Integration is a key issue in determining the eventual success of a component in a manufacturing process. If the component cannot complement the other pieces of the existing system, it will be rejected in order to protect the previous system wide investment. This leads to a demand for technology to be flexible so that it can not only provide new and improved functionality, but can also mesh seamlessly with other systems. The primary goal of this thesis is to design a generic interface for one such technology, run by run control, and demonstrate the flexibility of this interface by examining multiple system integrations. A positive side-effect of work done toward integration has been a refinement of the algorithms used for the controller. This chapter reviews previous run by run control research, and discusses of issues faced when attempting integration. The chapter concludes with a short example followed by a thesis overview.

1.1 Background: Run by Run Control

Run by run (RbR) control is a form of process control. It uses post-process measurements and possibly *in situ* data to modify models of the process, and based on these adaptive models recommends new equipment settings for the next run. Although run by run control may use *in situ* data, it differs from real-time control in that it only adjusts process variables between runs. Run by run control can be effective in keeping a process on target and in statistical control in the presence of process noise and drift detected between runs.

There are many algorithms used for run by run control. These algorithms form the "brain" of the controller. They take the measurements collected after and possibly during the run as well as historical data and compute the next recipe that will bring the process back to target. The actual method is dependent on the algorithm. Figure 1 shows a schematic breakdown of some of the methods used for run by run control. The methods are

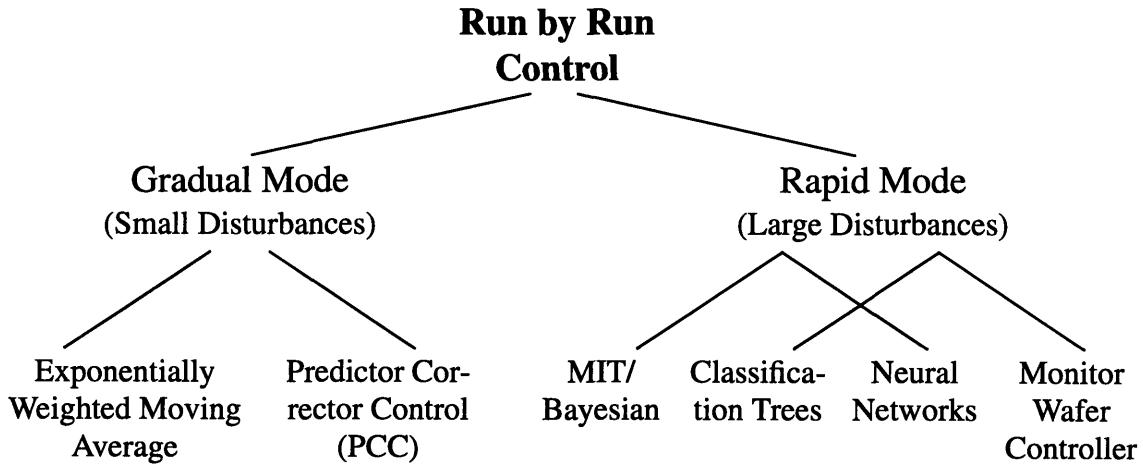


Figure 1: Run by run method hierarchy

divided into two categories, *gradual* and *rapid*, based on the nature of the disturbances. Gradual mode algorithms are designed to handle statistically small disturbances (less than 3 σ). These algorithms usually assume the process is in the correct region of operation and requires only slight changes in the recipe to account for long-term drift.

Two example gradual mode implementations are Exponentially Weighted Moving Average (EWMA) and Predictor Corrector Control (PCC). Note that the actual gradual mode control incorporates one of these filtering mechanisms along with a recipe generator (see Chapter 3). EWMA uses an exponentially weighted moving average filter to distinguish between random noise and real disturbances [Ing91]. This weighting helps eliminate some of the harmful effects that process and measurement noise can have on a system by time averaging this noise with data from previous runs. The weighting factor α can be set to adjust the extent of this averaging. The trade off is between controller response vs. controller immunity to noise. A comparison of an α weighting of 0.1 and 0.5 is shown in Figure 2. As can be seen, as α is increased, the relative weighting of previous runs (<20) is decreased.

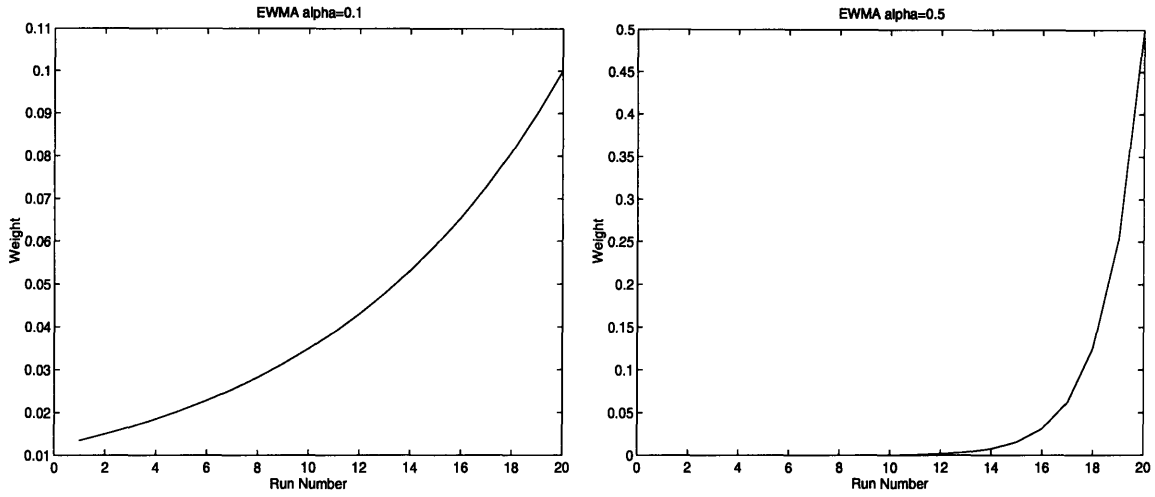


Figure 2: EWMA decay comparison

A modified version of the EWMA controller developed at Texas Instruments is *Predictor Corrector Control (PCC)* [BS93]. This algorithm more explicitly models the time dependence of disturbances. Trends in the drift are monitored over many runs and are factored into the suggested recipes. This method requires the addition of disturbance state to the process model, but has shown improved performance over EWMA when persistent drift is present. An example cause of such drift is equipment aging. The drift monitored in the PCC algorithm also uses EWMA filtering to ensure that the overall trend of the drift is more resistant to process noise.

The other major class of run by run algorithms is rapid mode. Rapid mode algorithms are designed to work under the assumption that significant disturbances have occurred and the controller must take aggressive (rapid) steps toward pulling the system back to target. These disturbances are greater than 3 σ but less than a value which suggests that the process has moved to an uncontrollable state. In the later case, techniques such as model building or expert systems may be utilized to bring the process back under control. These systems are usually used in conjunction with a gradual mode algorithm to provide both quick response with stable control. Deciding when to switch from gradual to rapid modes is the key to most rapid mode algorithms. The two rapid mode implementations discussed here use different approaches to determine when a system is "out" of the gradual mode regime and needs rapid adjustment. Other approaches that blur the distinction between rapid and gradual mode control, including a classification tree approach [Ira90] and the

monitor wafer controller [MSC94] are not discussed in detail here.

One approach to rapid mode is the MIT Bayesian method [Hu93a]. In this method Statistical Process Control (SPC) charting techniques are used to signal rapid alarms. These techniques, developed originally for manual quality control, provide a set of definite conditions which signal when a process is out of control. Once an alarm has been triggered, Bayesian statistics are performed on subsequent runs to determine the likelihood that the alarm is a true shift in machine operation. This is used to prevent outliers from triggering false alarms and causing incorrect adjustments based on one data point. An example of a real and false alarm is shown in Figure 3.

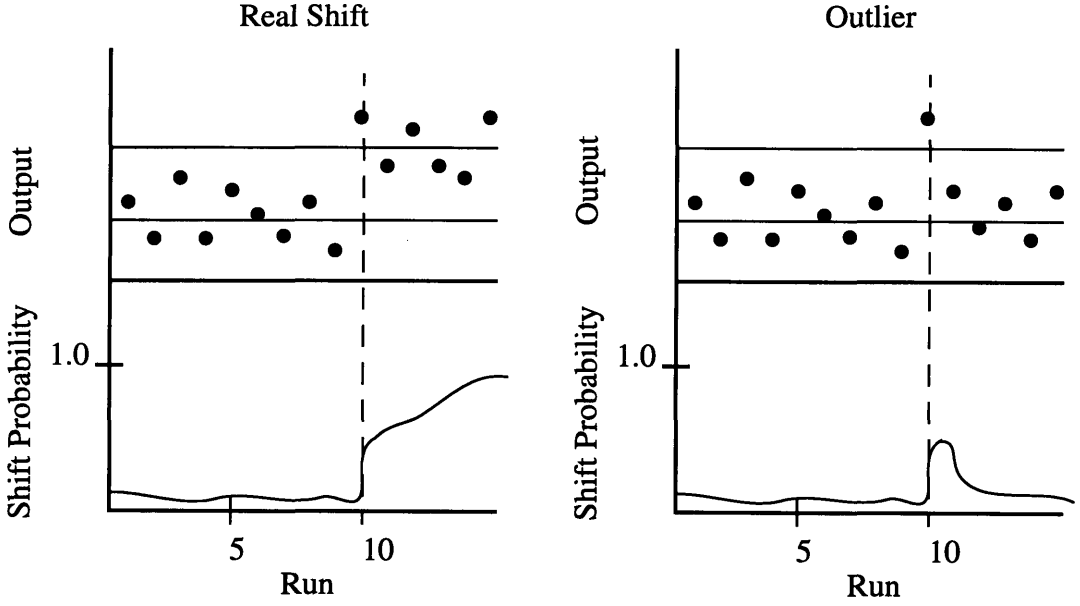


Figure 3: Bayesian shift weighting

Another rapid mode implementation involves the use of neural networks [KS94]. In this case, a neural network is trained on the inputs and outputs present after a shift has occurred. The controller then uses this information to immediately compensate for disturbances after a shift rather than waiting for convergence as in the case of an EWMA controller. Gradual mode control may then be applied once the shift has been compensated for. Experiments have shown that this method does reduce process variation when compared to a pure gradual mode, although comparisons with other rapid mode algorithms were not performed.

The concept of a clean and open interface to run by run control algorithms has not been as well studied as the algorithms themselves. In an effort to remedy this, SEMAT-

ECH as part of its attempt to define an all encompassing Computer Integrated Manufacturing Application Framework [Oco94],a has defined the functional interface that a control module would have along with its interaction with the rest of the system. Figure 4 shows the proposed information model. This model shows that a controller will have zero or one

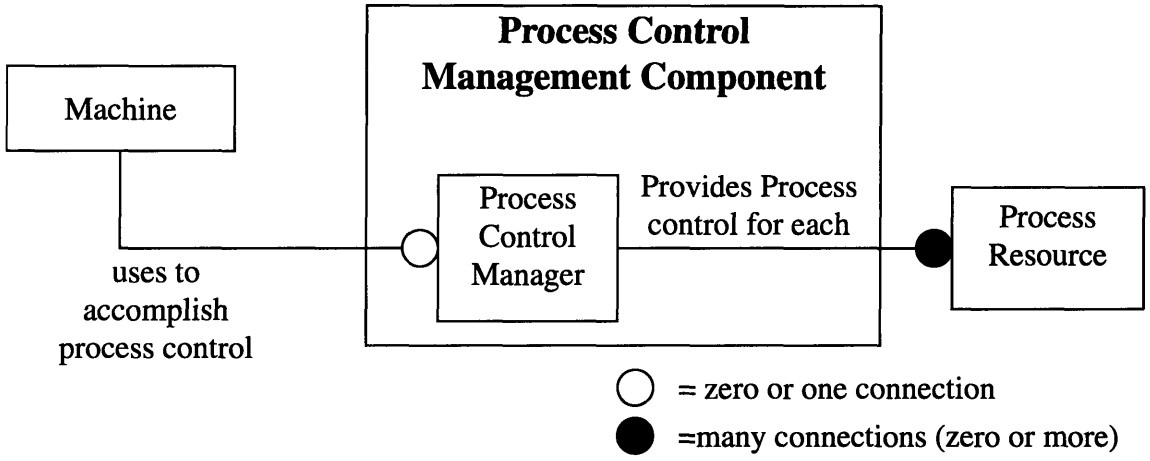


Figure 4: Process Control information model

machines to control as well as one or many process resources that it uses to get the information needed for control. SEMATECH has also presented a communication model that uses messages to transfer information between functional modules. Figure 5 shows the control communication model. This model demonstrates the need for a well-defined interface between various separate resources. By defining the behavior of routines via the messages they can respond too, a valuable level of abstraction can be obtained. Using this abstraction, vendors can implement components with vastly different internal structures yet be externally compatible.

In addition to the work done by SEMATECH, there have been several attempts at defining the form of the data needed for run by run control as well as presenting the controller in a friendly and easy to use manner [Hu93b, SBHW94]. Conversion of the MIT gradual mode control algorithms to the C language has also been performed [KM93].

Due to the developmental nature of many of the algorithms discussed, standardization of interface and data has not been extensively performed. This makes it difficult to compare different algorithms without essentially rewriting the desired modules. It also leads to difficulties when integration of these routines into larger systems becomes necessary.

Before the actual methods and abstractions developed for run by run control are dis-

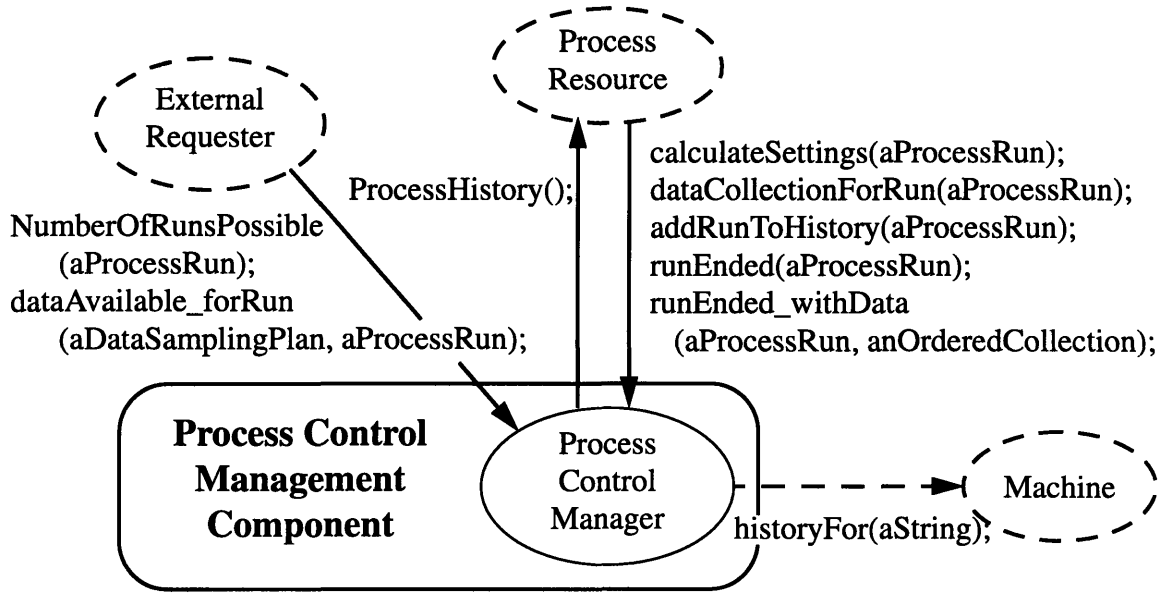


Figure 5: Process Control communication model

cussed, it is useful to examine why and how run by run control can be integrated into a manufacturing process. Although semiconductor manufacturing is the target application for much of the run by run research, the run by run method of process control is in no way limited to this area. The next section will touch on some of the issues faced when considering run by run control, and will present an example integration procedure.

1.2 RbR Control Application Issues

When evaluating control technologies to be tested and possibly integrated into a production facility, certain criteria are often examined. The actual process being examined is often of secondary importance if it fails to satisfy these constraints. These constraints do not represent all of the issues used to judge a control strategy, but are a representative subset:

- Cost
- Capability for Integration with existing equipment
- Effectiveness
- Degree of support

1.2.1 Cost

Many times cost is the primary barrier to adopting a new or modified processing method.

Ideas are often considered based on the financial risk they pose to the company. This cost is not only the cost of equipment needed by the process, but also the integration cost. In addition to one time charges, there are also execution costs. Control may require additional measurements, and thus impose additional labor, time, and materials expenses. All costs must be considered to conduct a complete analysis of the consequences of a control strategy.

1.2.2 Capability for integration with Existing Systems

It is no longer an option to refit an entire production facility to implement a control strategy. Large investments must be protected by easing this integration. For this reason, robust interfaces to process control are needed to allow existing systems to be controlled with little or no modifications. Often, due to a lack of equipment automation and communication, this must be done as a "suggestion" based strategy where the human operator becomes the interface between the machine and the controller.

1.2.3 Effectiveness / Performance

Adoption of a control strategy is futile if it does not provide an improvement over the existing process. This intuitive fact becomes a major barrier to technologies that are new and have process-specific benefits. This can be a fatal barrier if the control process is not flexible enough to allow a relatively painless "trial" period to show feasibility.

1.2.4 Flexibility

Another quality sought in a control strategy is its flexibility. This not only involves changes in the product produced by a particular process, but also changes in the process itself. A controller must be flexible enough to allow these changes with as little effort and cost as possible. Flexibility is also beneficial in that a similar control strategy can be used for a variety of processes.

1.2.5 Degree of Support

Companies are unwilling to blindly adopt a "black box" control strategy. It is important that in one form or another support for the controller and its integration into the system can be readily obtained. For this reason, companies are looking for products that embrace

open standards. By adopting an open standard, a company can better guarantee that multiple vendors can offer competitive products that all provide the same functionality.

1.3 Example Integration

Once run by run control has passed the initial phase of acceptance, it is necessary to proceed carefully toward an efficiently integrated run by run strategy. This is by no means a one step process. It requires breaking the problem up into chronological goals that when completed represent a safe and predictable path to integrated run by run control. These steps are:

- Defining the problem
- Creating a model
- Configuring a controller
- Testing the model and controller
- Developing an integrated system

The intent of the following fictitious example is to show the steps toward integrated control, not to provide insight into controlling a certain process. For this reason, a very simple process is examined: Yummy Bake cookie manufacturer is interested in integrating run by run control into an existing soft-center cookie production line. The machine to be controlled is the baking oven. The desired output of the system is a "good" cookie.

1.3.1 Defining the Problem

Many times the most difficult phase of a control strategy is determining what needs to be controlled. The solution to this can be complex. A single machine may have many characteristics that are important to the manufacturer, but choosing which and how many can be controlled can be a long and difficult process. It is also often an iterative process, where initially several parameters are targeted, and experiments are done on each to determine which are suitable for control. It is also important to note that in many cases parameters which are suitable for control may not be adjusted directly, similarly process qualifiers i.e. qualities that indicate the "goodness" of a process may not be directly measurable. In these cases indirect measurement (virtual sensors) and actuator techniques may be required to achieve the desired effect.

Initially, the goal of a "good" cookie was not very helpful. Luckily, after some market studies, this is further refined into two measurable outputs, center hardness and outer color. The controllable inputs to the oven are the temperature, cook time, and rack height. It is important to note that these inputs and outputs may not be the optimal choices and may prove to have poor control characteristics. At this stage it is not critical, they are starting points. Their suitability as parameters will be determined by both the model design and later control tests.

1.3.2 Creating a model

The intent of this overview is not to develop a revolutionary scheme for controlling cookie production, but rather to outline the steps needed to develop a run by run control system. For this reason, it is assumed that through experience and design of experiments (DOE) a three input, two output model can be obtained.

Although it is not covered here, it is essential that a suitable model be found for the process. Poor models lead to poor control. The controller by its nature has the ability to find an optimal operation point, but it is limited to a local minimum search. If the model of the system suggests an operating point sufficiently far away from the optimal value, the controller may be unable to push the process into a more favorable environment.

1.3.3 Configuring a Controller

Once a model has been constructed, it is necessary to configure a controller based on this model. Configuring the controller involves choosing an available control algorithm and modifying it to work with the model. This includes not only inputs and outputs, but also ranges, weights, as well as many other constraints. These are the parameters of the controller. It is essential that a controller be as configurable as possible: this allows models to be entered with no special programming or controller modification. Often it is possible to evaluate multiple control strategies using the same basic control framework. This aids in the final selection of an algorithm and adds to the flexibility of the controller.

1.3.4 Testing the Model and Controller

It is difficult to introduce a new control technology to a proven process. A trial period is needed to see if the control system is warranted for a particular process. This trial period

must be made as inexpensive in time and effort as possible. This is due to the real possibility that many model/controller combinations will need to be evaluated before committing to any one scheme.

There are two methods used to evaluate and show feasibility. First, simulations can be performed to mimic operation of the equipment and controller. These provide useful and inexpensive results, but often fall short of securing commitment to the idea. For this reason, these simulations are often used as a first cut toward the eventual adoption of an algorithm. The second method, real testing, is the method of choice for examining new ideas before they are put into the production line. This testing must also fit the time and effort criteria for it to be used.

The need for a simulator and tester that do not require the complete integration of an untested system into a process motivates the development of a stand-alone graphical interface to the run by run controller. This interface allows the user to configure the controller with a test model and both simulate as well as run real test data through the controller. Using this interface, the existing process can be tested without the need for full integration.

Using this interface the model developed for the oven control problem can be first simulated to determine if it is worthy of testing, and then tested in the factory on a sample run of cookies. In this type of control, the human operator becomes the actuator of the control mechanism. In this way real results can be obtained with a minimum disruption to the existing process. As an added benefit, maintaining a human buffer between the control decisions and the actual equipment can often catch unwanted and potentially dangerous suggestions made by an incorrectly formulated problem or unanticipated system behavior. It also allows the operator to become more familiar with the control system. Figure 6 shows a sample interface screen

1.3.5 Developing an Integrated System

Once the control components have been designed and configured, the collection of modules must act as a system. This is essential to the long-term adoption of a control system. Short-term tests may be done with a loosely coupled system, but for production purposes, control must be at worst an added routine to the process engineer and at best seamlessly

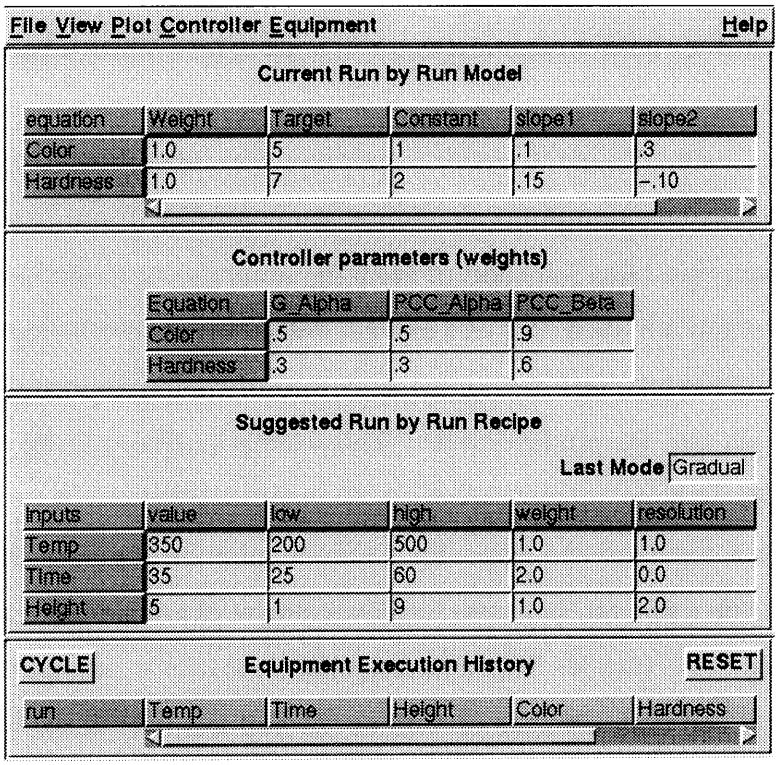


Figure 6: Interface screen

integrated into the process. Tedious control management leads to both errors and resistance from the operators.

The complete system is shown in Figure 7. Control is done in a run by run manner. The steps of the run by run process can be enumerated as follows:

1. Bake cookie (Or cookie batch) with controller settings.
2. Measure cookie quality.
3. Using measurement data, generate new settings.
4. Repeat steps 1 through 3.

The power of the run by run control scheme is that it continually attempts to improve the output of a process without requiring complex models. It also benefits from not requiring *in-situ* control mechanisms that are often not an option for existing systems. These benefits, however, are not unique to the run by run control server. What the control server provides is a client/server approach that allows portions of the system to be modified or replaced without affecting the rest of the system. The controller is based on a messaging system, where a module is only required to adhere to a given message format, and is not bound to any particular internal structure. This, for instance, allows different control strat-

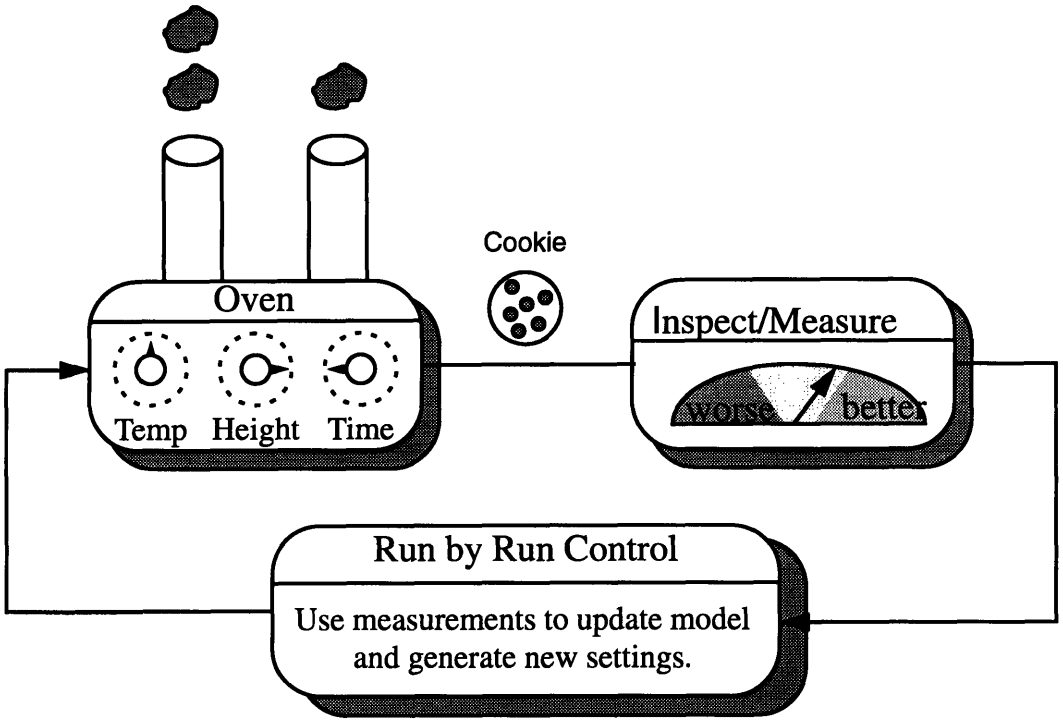


Figure 7: Cookie production with run by run control

egies to be implemented into the controller with no change to the other system components. This concept also scales well in that additional resources, such as an archiving database for run histories, can be added and accessed via messages.

1.4 Organization of Thesis

This section has presented an overview of some of the issues faced and addressed by the run by run server. Specifics were intentionally left out. The specific interfaces and programming paradigms are important to the eventual success of the run by run controller, but the ideas they are designed to address are even more important.

The following chapters describe in detail the underlying structure of the run by run controller as well as other system issues. Chapter 2 discusses the integration of run by run control into existing systems, and further details the graphical interface discussed earlier. The underlying algorithms used in the run by run controller are described in Chapter 3 as well as the parameters used for control. Chapter 4 details the messaging and general interface used by the run by run control module as well as the issues that it attempts to address. Results obtained by actual tests using the run by run control server are discussed in Chap-

ter 5. These tests were performed both with and without the graphical interface. Chapter 6 summarizes the thesis and points out important research needed for further advancement of run by run control.

Chapter 2

RbR Integration

Integration into existing systems is the primary goal of this research. The use of the run by run controller in a wide range of applications is the benchmark by which to measure progress. Only through interaction with real systems can the integration tools written for the run by run controller be tested. Valuable information about the controller interface as well as the underlying algorithms has been gained through these interactions. Currently three systems have been investigated for communication with the RbR controller:

- The CIM RbR User Interface and Simulation Environment (CRUISE)
- The Computer-Aided Fabrication Environment (CAFE) [McI92]
- The Generic Cell Controller (GCC) [MM92]

2.1 The CIM RbR User Interface and Simulation Environment (CRUISE)

In addition to interaction with existing systems, a control client has been developed that is used to test and debug the controller as well as to provide valuable simulation data. The environment also provides a valuable stand-alone version of the controller that can be used as a manual form of run by run control.

This environment is written in Tcl/Tk [Ous94], an interpreted language designed for extending application programs and creating graphical user interfaces (GUIs). The advantages of using an interpreted language like Tcl as a base for the environment are two-fold. First, rapid program development and modification are provided as a direct result of inter-

preted code. Second, the debugging information available in the programming environment can be a powerful tool when developing models for the interface, or testing communications with the controller.

In addition to its use as a debugging client, the environment provides a valuable suite of simulation tools. These include the following:

- tabular representation of historical data,
- graphing ability,
- a machine simulator with noise models,
- dynamic modification of controller parameters.

The remainder of this section will briefly explain each of these features and discuss the benefits of their inclusion into CRUISE. Appendix B contains a user's manual for CRUISE.

2.1.1 Tabular Representation of Historical Data

Simulation of control algorithms is important for both testing and development. There are many methods and conventions used for both. An archiving facility is needed to provide historical data for both comparison with other data and for graphing to identify trends or anomalies.

CRUISE has options to display, graph, store, and compare historical data. These functions are integrated into the environment and provide an intuitive interface to the data. Storage schemes for the data include an internally recognized format as well as a format suitable for exporting data to packages such as Matlab™ and Mathematica™. Data comparison is done by graphing two simulation histories simultaneously.

2.1.2 Graphing Ability

The RbR environment has the ability to provide a graph of any time series data present in the tables. This allows both inputs and resultant outputs to be analyzed. Postscript output of graphs is also supported. As stated earlier, graphing of two different histories simultaneously is supported to facilitate comparisons.

2.1.3 Machine Simulation with Noise Models

One of the major components of CRUISE is the equipment simulator. This simulator is designed to provide both valid data for RbR Server testing, but more importantly, realistic machine simulation data. This can be a valuable tool when developing a control algorithm or comparing algorithms.

In order to provide realistic simulation data, the environment has a number of model customization features. A simple first order polynomial model without noise that exactly matches the control model may be used to complement the model used in the RbR Server. This would represent control of an accurately modeled and well behaved process. The operator, however, can adjust the equipment model so that it differs from the model used for control and thereby simulate model mismatch.

In addition to first order polynomial models, custom higher order models are also supported. These "freeform" models allow the operator to enter useful effects such as cross product terms and higher order terms. The syntax of entry is ASCII based, and models are limited to functions strictly of input variables. Exponential and trigonometric functions are not supported in direct form but may be approximated by Taylor series expansion. The following shows the algebraic representation for a two input one output system:

$$Y = 10 + 0.1X_1 + 0.5X_2 + 0.01X_1^{\,2} + 0.1X_1X_2$$

This system cannot be entered into the environment in this form. It must be converted into a simple ASCII string that can be easily parsed by the environment. The following is an acceptable version of the same input:

```
Y = 10 + 0.1*X1 + 0.5*X2 + 0.01*X1*X1 + 0.1*X1*X2
```

Noise simulation is essential when modeling a real system. The RbR environment supports the addition of noise to the value obtained from the simulation model. The simulator supports two types of noise. The first and simplest is Gaussian distributed noise. This is useful for testing due to its simplicity and easily analyzed behavior. The simulator also supports more complex autoregressive moving average (ARMA) noise. This type of noise is often more realistic due to its time-dependant nature. ARMA noise takes on the following form:

$$N_t = \alpha N_{t-1} + w_t + \beta w_{t-1}$$

where:

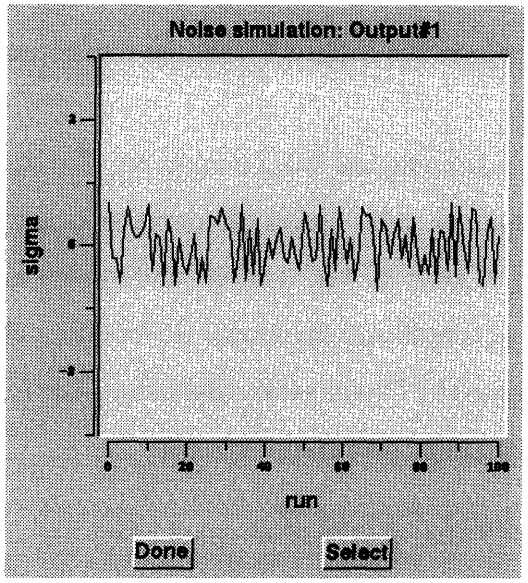
N_t = total noise for run t,

w_t = white (Gaussian) noise for run t,

α, β = noise model coefficients.

Figure 8 shows a comparison between Gaussian and ARMA based noise. The shape of the ARMA noise is heavily dependent on α and β and in fact becomes white noise with $\alpha=\beta=0$.

Gaussian Noise



ARMA Noise
(α=0.5, β=0.5)

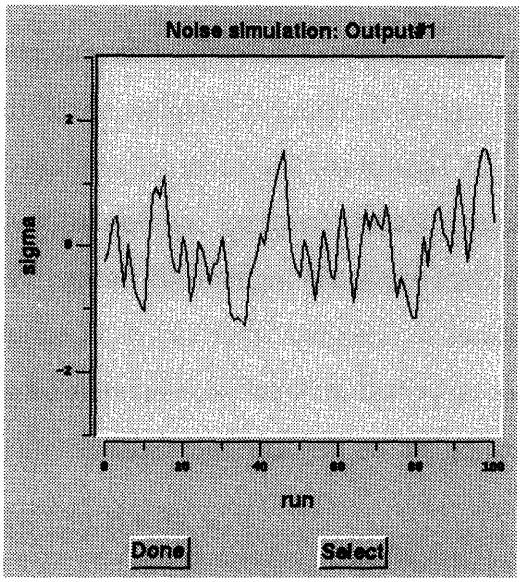


Figure 8: Gaussian vs. ARMA noise

2.1.4 Dynamic Modification of Controller Parameters

Often it is desirable to change characteristics of a simulation during execution. An example would be subjecting a system to a process shift. The simulator fully supports such actions. It not only allows the process engineer to change the form and parameters of the noise model, but also to dynamically change underlying models between runs.

Dynamic modification of models aids both simulation and prototyping. First, simulation is enhanced by allowing machine models to be changed. This aids the simulation when trying to simulate complex events such as machine cleaning and maintenance which may considerably modify the characteristics of the machine. Second, prototyping is

enhanced by permitting the control engineer to directly modify the controller's internal model without interrupting the simulation. In this way, the engineer can become part of the control process. This simulates a more complex (human) form of control. By examining the effects on the system, it can be determined whether steps should be taken to analyze the control actions to develop a controller that exhibits similar behavior to the control actions provided by the user.

2.2 The Computer-Aided Fabrication Environment (CAFE)

CAFE is a comprehensive object oriented integrated circuit fabrication facility management system. It was designed at MIT to address the many issues arising from fabrication, including recipe management and processing. CAFE currently can store process data in its database and chart this data for manual control, but has no mechanism for integrating this control into the CAFE system. Integrating run by run control into the CAFE system is an excellent test of the flexibility of both CAFE and the RbR control server.

Although much of CAFE is linked at compile time to its functional core, this was not the method of integration used. Compiling the run by run controller into CAFE would represent a break from the ideals of modularity. Modularity was maintained by making a small external program that could be called by CAFE to enact control. To keep the interface as simple as possible, the communication between CAFE and the external program is done in two steps. First, the external program is called with all data given as arguments on the command line. This is a simple way to send modest amounts of static data without complex messaging systems. The external program then interacts with the RbR control server via messages over TCP/IP and gets the suggested recipe for the next run. The program then prompts the user to decide among three possible alternatives:

- Use recipe stored in the CAFE database.
- Use run by run control suggestions.
- Enter custom recipe.

Once the user has selected a recipe, the recipe is returned as a return value to CAFE for further action and the external program exits. This intermediate program could be eliminated in the future in favor of a direct client/server paradigm. Figure 9 shows a typical screen generated by the control program.

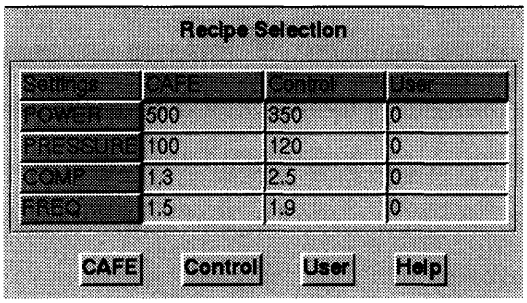


Figure 9: CAFE run by run interface

2.3 The Generic Cell Controller(GCC)

The GCC development effort is a project conducted at the University of Michigan that is designed to integrate modules from various vendors into a single control framework [MM92]. The goal of the project is to integrate these modules into a productive and cooperative tool controlled by the GCC. The RbR controller is one such module.

Integration with the GCC proved to be less abstract than was originally hoped. For the sake of expediency, the computational engine of the run by run controller was converted into a subroutine that could be called by the GCC. This provided a quick way to integrate the run by run controller with the GCC. It has, however, demonstrated some of the pitfalls that code integration can fall into, namely that software updates require at least a relink of the GCC and at most a rewrite of the functional interface to the controller. It has also led to a split in the development of the run by run server and the code used by the GCC module. Figure 10 shows the current model of integration as well as the desired future model being pursued at the University of Michigan.

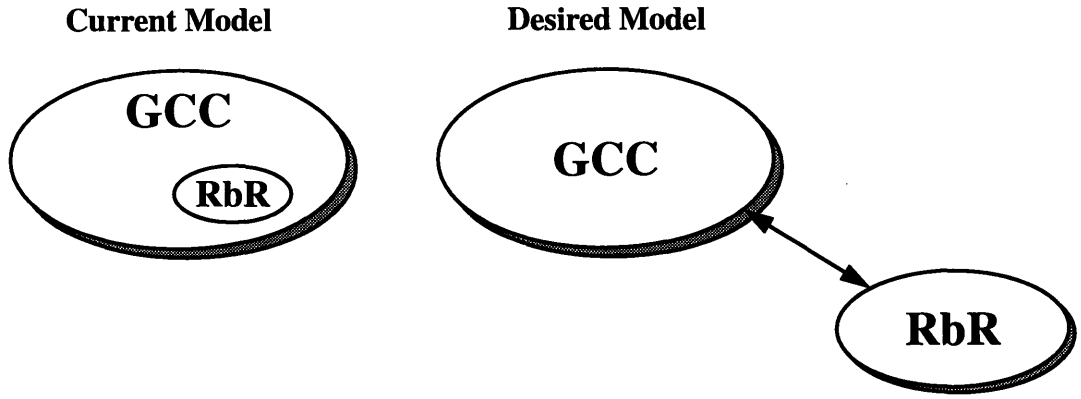


Figure 10: GCC and RbR interface models

Performance of the run by run controller as a module of the GCC has been useful to show feasibility of control in chemical mechanical planarization (CMP) (see Chapter 5). This program was meant as a first step toward automated control. As the project evolves, so will the communication link between the run by run controller and the GCC. As with any communication specification, work is needed to ensure that the communication specification used is acceptable at both ends and can grow as the complexity of the data grows.

Chapter 3

RbR Implementation

There are many different approaches to process control. The focus of this research is to implement existing algorithms for run by run control in a clear and well documented manner. It is also desired that the methods used to implement existing algorithms be designed so that future algorithms can use the same basic interface. This enables the RbR controller to become a test-bed for algorithms by allowing their rapid integration into the existing system.

Currently, the RbR controller has support for two methodologies for solving multiple input multiple output (MIMO) first order polynomial control algorithms. This may at first seem limiting, but it is assumed that run by run control will be applied to a relatively stable process, subjected to noise and drift. Once this has been established, the controller does in effect a piecewise linear approximation over many runs. Using this strategy, complex models can be linearized around an optimal point and given to the controller to maintain that point.

More formally, the controller uses an m-by-n (inputs-by-outputs) linear model with an additional constant term[1].

$$y = Ax + c$$

1. Equations will use the following notation: Arrays will be capitals, vectors will be lower case, indexing within a vector or matrix will be lower case with subscripts. In addition, the special subscript "t" will be reserved for time, or run number information.

where:

y = System output,
x = Input (Recipe),
A = Slope coefficients for equation,
c = Constant term for linear model.

This matrix notation can be expanded into the familiar simultaneous equations. Each output represents a target of control, and each input represents an adjustable parameter in the recipe.

$$y_1 = a_{11}x_1 + a_{12}x_2 + \ldots a_{1m}x_m + c_1$$
$$\ldots$$
$$y_n = a_{n1}x_1 + a_{n2}x_2 + \ldots a_{nm}x_m + c_n$$

The controller operates under the assumption that the underlying process is locally approximated by the first order polynomial model, and that this polynomial model can be maintained near a local optimal point solely by updating the constant term c. In order to allow maximum flexibility for algorithmic development, the computational engine of the RbR controller has been divided into two parts:

- Model update,
- Recipe update.

3.1 Model Update

Updating the model is the first step in the control process. Currently this entails updating the constant term used in the polynomial model equation. The algorithms used in the RbR control server do not update the slope coefficients. This step of the control process determines how aggressive the controller is, as well as the its ability to handle different conditions such as drift.

Two similar control algorithms are implemented in the RbR controller:

- EWMA gradual mode,
- Predictor corrector control (PCC).

3.1.1 EWMA Gradual Mode

EWMA gradual mode is the simplest method of model update used in the RbR controller. As its name implies it filters historical data with an exponentially weighted moving average (EWMA) filter to prevent over-control. A single weighting factor α is used.

$$c_t = \sum_{i=1}^{t} \alpha (1-\alpha)^{t-i} (y_i - Ax_i) \quad (1)$$

Although (1) would provide the desired EWMA weighting, it also requires data from all previous runs. Luckily this can be simplified using the additive nature of the series to generate an iterative expression for the constant term update:

$$c_t = \alpha (y_t - Ax_t) + (1-\alpha) c_{t-1}$$

Using an EWMA filter to smooth the control action on a linear process has been shown [SHI91] to have very promising results. The simplicity of the algorithm also makes it a natural starting point for a run by run control strategy.

3.1.2 Predictor Corrector Control (PCC)

PCC is an expansion on the EWMA gradual mode that adds an explicit model for drift. Drift is present in many VLSI processes that can "age." Examples include pad wearing on a chemical mechanical planarizer, or build-up on the wall of a plasma etcher. PCC uses two parameters, α and β, to weight noise and drift respectively. EWMA weighting is used for both the constant term update and for the drift estimation.

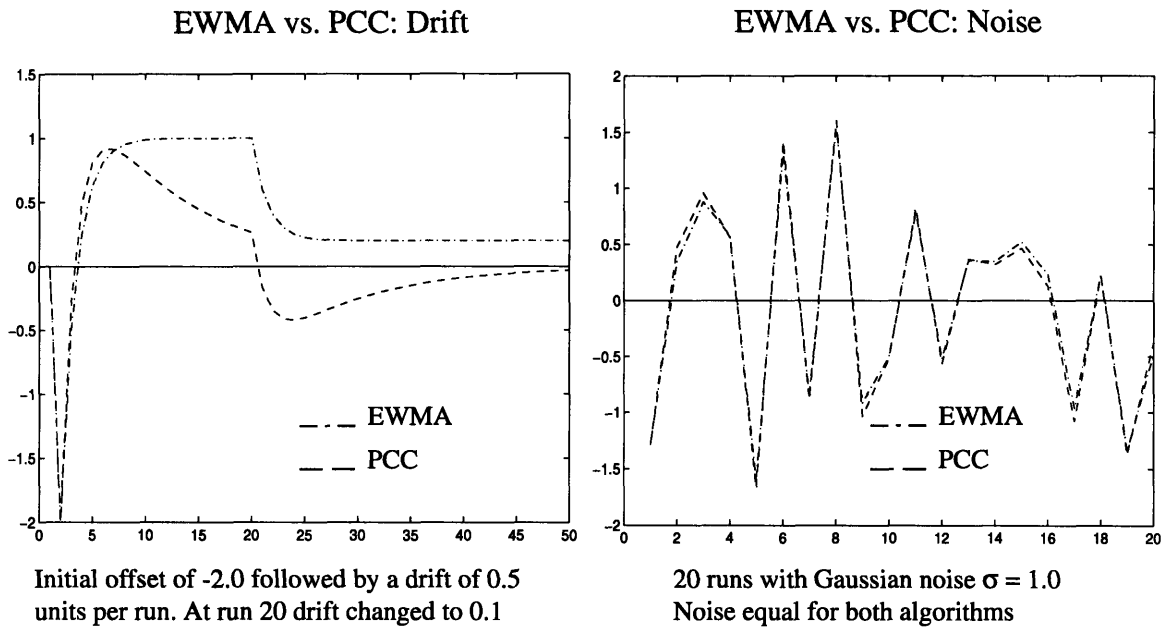
$$n_t = \alpha (y_t - Ax_t) + (1-\alpha) n_{t-1}$$
$$d_t = \beta (y_t - Ax_t - c_{t-1}) + (1-\beta) d_{t-1}$$
$$c_t = n_t + d_t$$

Where:

n	=	Estimation of noise for run.
d	=	Estimation of drift for run.
A	=	Slope coefficients for model.
y	=	Measured output of the system.
x	=	Input (recipe).
c	=	Constant term for model.
α	=	EWMA weighting for noise estimation.
β	=	EWMA weighting for drift estimation.

Simulations of PCC vs. EWMA on processes with and without drift show that PCC provides better drift response with no noticeable penalty when drift is absent. Changes in the drift-rate, however, can lead to potential overshoot based on the time averaging of the PCC drift estimator. Figure 11 shows a comparison between PCC and EWMA control

under both drift and noise conditions [BS93].



Initial offset of -2.0 followed by a drift of 0.5 units per run. At run 20 drift changed to 0.1

20 runs with Gaussian noise $\sigma = 1.0$ Noise equal for both algorithms

Figure 11: PCC vs. EWMA

3.2 Recipe update

Once a suitable constant term has been chosen for each of the equations separately, the task of determining a new recipe arrives. This solution must take into consideration many conditions and constraints that affect the process. Although in the actual controller the final recipe is calculated in the presence of all conditions and constraints, they will be discussed separately. First the algorithm used for fitting a solution to the numerous outputs will be discussed followed by the parameters that can affect this solution.

3.2.1 Curve Fitting

At the heart of the RbR recipe algorithm is a matrix least-squares routine. Least-squares is a method for determining the optimal solution (curve fit) for an overdetermined (#outputs > #inputs) system. The method has the favorable property of providing the "best" solution even if an exact solution does not exist. In this case, "best" refers to the solution which minimizes the squared error between itself and the exact solution. Care must be taken when formulating the problem. The absolute scale of the inputs can cause certain inputs to be favored over others when an optimal solution is chosen. This is beneficial when used to

modify the behavior of the controller, but is not desirable if it is not controlled (see input/ output weights discussed in section 3.2.2). To prevent unwanted bias, all inputs are normalized (as shown in equation 2) to between -1 and 1 before any computation.

$$x_n = \frac{x - \frac{(x_{max} + x_{min})}{2}}{\left(\frac{x_{max} - x_{min}}{2}\right)} \tag{2}$$

Where:

x_n = Normalized recipe.
x_{min}= Lower bound for recipe.
x_{max}= Upper bound for recipe.

Based on the formulations of the problem and possible boundary constraints, the least squares solution can take on three forms:

- Exact solution
- Overdetermined
- Underdetermined.

Figure 12 illustrates examples of the three possible forms of the solution to a control problem. Each of these must be solved in a different manner. For the underdetermined case (Figure 12a), the system has two inputs (x_1 and x_2) and one target output (3). This problem would normally lead to an infinite number of solutions (represented by a line). Since all solutions are "correct" it would serve the purpose of the algorithm to simply pick one of the values. This extra degree of freedom is instead used to bring the solution as close to the previous recipe as possible minimizing least squares distance. This not only reduces the solution to a singe value, but it also has the positive effect of minimizing the extent of the changes to the input parameters of the system.

Figure 12b illustrates the effects that two conditions have on a two input system. These constraints which are represented by two lines, create a problem that has only one solution. This solution (represented by the intersection of the two lines) satisfies both conditions exactly. Due to the lack of freedom in the problem, the previous solution information is not used.

With the addition of a third condition, an overdetermined problem arises (Figure 12c).

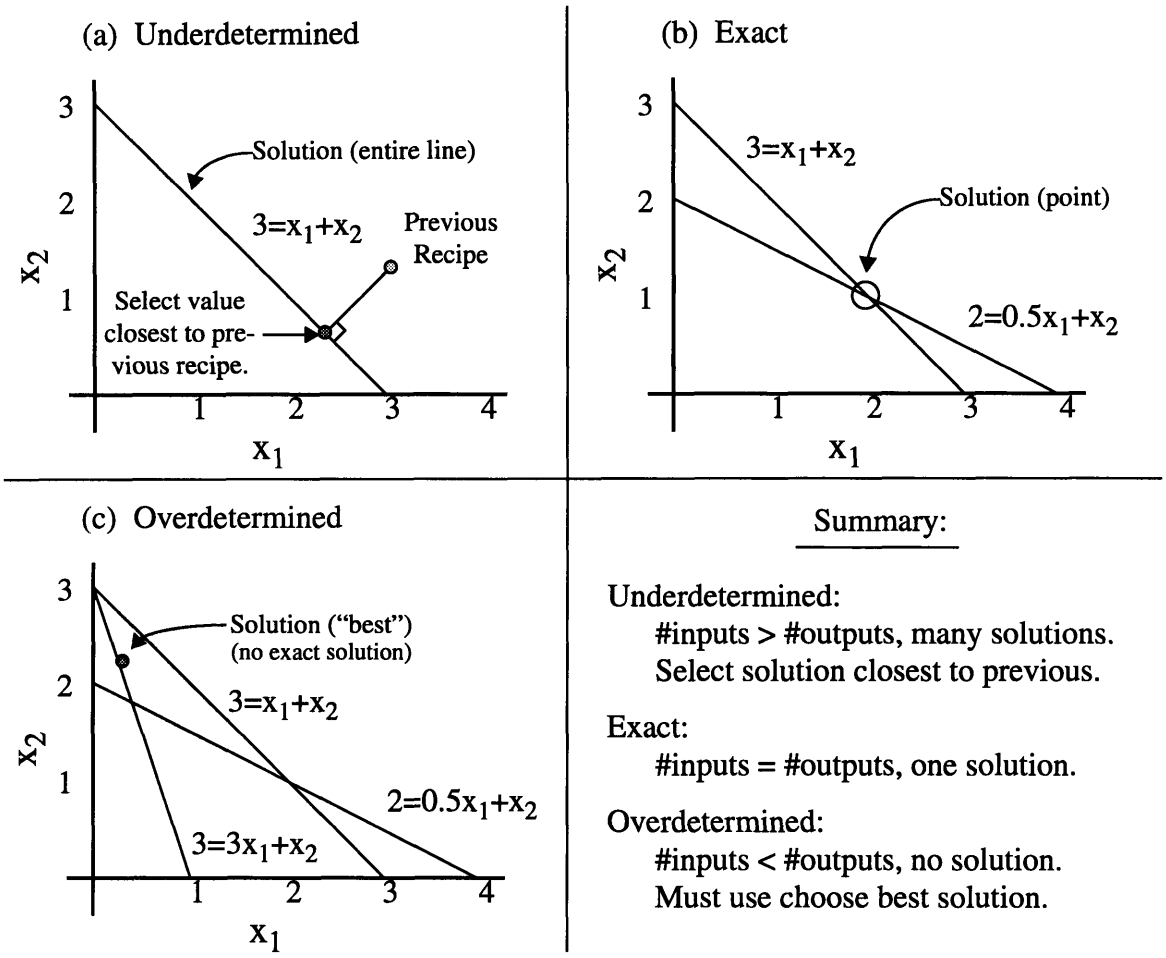


Figure 12: Three possible solution domains

In this formulation there are more conditions than degrees of freedom in the inputs, so there is no exact solution. A least squares algorithm is used to minimize the error between the target for each output and the final solution. Again the previous solution is not used to avoid further constraining the problem.

In order to provide a flexible environment that can handle any input/output combination, the controller first determines which case is occurring, then generates the solution accordingly. This could require three separate computational routines, but luckily this can be reduced to one. Since the least squared solution is guaranteed to be the best, it can be used to solve the exact case directly and the underdetermined case with some preprocessing of the data. The mathematical formulation of the recipe update problem for each of these cases is described in more detail below.

3.2.1.1 Exact Solution

If the number of inputs (n) to a system is equal to the number of outputs (m) then there is exactly one solution that satisfies the desired outputs. The calculation of this solution is straightforward.

$$y = Ax + c$$
$$x = A^{-1}(y - c)$$

There are two uses for the symbol y in the equations used for control. First, it represents the output of the system. This is what is measured as the real value of the system output. This value is primarily used to update the constant term c, discussed earlier. Second, it is used to denote the target that is desired for that output. This second use is how it is used in the remainder of this chapter. The two meanings are similar in that they are the real and ideal value of the system output.

3.2.1.2 Overdetermined

There are two events that could lead to an overdetermined problem. The first is that the problem was formulated with fewer inputs than outputs (n<m). Second, the controller could have originally been underdetermined or exact, but input bounds forced certain inputs to be locked, thus decreasing the number of controllable inputs.

Once an overdetermined case is encountered, a least-squares error fit is applied. This ensures in a least squares sense that the solution places the output as close as possible to the target.

$$y = Ax + c$$
$$A^T(y - c) = A^T Ax$$
$$x = \left(A^T A\right)^{-1} A^T (y - c)$$

3.2.1.3 Underdetermined

In contrast to the overdetermined case, the underdetermined case is encountered when the number of inputs exceeds the number of outputs (m>n). This is often the case in a process. Several inputs can be modified to help maintain a certain output, so the possible solutions are infinite.

Although being able to reach target is always desirable, the choice of the "best" solution from the set of all possible solutions must be done in a consistent manner. Again we turn to least squares. This time, however, instead of merely obtaining an answer that hits the target, we can also select an answer that is closest to the previous recipe while still exactly solving the original problem. In this way we can ensure both that our output is guaranteed to be correct, and that the associated inputs are modified as little as possible.

The actual formulation of the problem is a little more complex than the other cases. It involves the use of a Lagrange multiplier (λ) to take the two constraints and merge them into a single equation. This is not the only method of obtaining the result, but is consistent with published results [KM93].

$$min\left\|x - x_0\right\|^2\Big|_{Ax = b}$$

$$L = \frac{1}{2}(x - x_0)^T (x - x_0) + \lambda^T (Ax - b)$$

$$\frac{dL}{dx} = (x - x_0)^T + \lambda^T A = 0$$

$$x - x_0 = -A^T \lambda$$

$$Ax = Ax_0 - AA^T \lambda = b$$

$$AA^T \lambda = Ax_0 - b$$

$$\lambda = \left(AA^T\right)^{-1} (Ax_0 - b)$$

$$x = x_0 - A^T\left(AA^T\right)^{-1} (Ax_0 - b)$$

Where:

b	=	y-c (Measured output - Constant term).
x_0	=	Recipe from previous run.
λ	=	Lagrange multiplier.
A	=	Slope coefficients for model.
L	=	Equation to minimize.

3.2.2 Parameters and Constraints

In addition to satisfying the constraints given by the equations themselves, the controller must also consider additional constraints and parameters before a final solution can be found. This is what often separates a theoretical solution from a "real" answer. The difference being, the theoretical solution is often solved in a vacuum with no thought to actual

application-based limitations, whereas a real answer represents the best possible theoretical result given all of the constraints present in the working equipment as well as other preferences. For this reason, the controller provides many parameters that can both constrain and bias the recipe generation in a predictable manner. Although these parameters can complicate an otherwise simple control approach, they can also provide valuable operator influence that complements the bare controller. These additional parameters are listed below:

Constraints:

- Input bounds
- Input resolution

Bias parameters

- Output weights
- Inputs weights

3.2.2.1 Input Bounds

The RbR controller was designed to control actual machinery, and as a result must account for limitations in the range of possible settings that an input can have. One way to achieve this is to simply determine the optimal recipe without input bounds, then fix all outputs that exceed their bounds to the closest valid setting. This approach provides the necessary constraints, but allows the controller to provide a less than optimal setting to the equipment. It is important that the final recipe is chosen in the presence of these constraints. The RbR controller uses an iterative approach to achieve this. Figure 13 shows the approach used.

This approach differs from the one-pass approach in one key area. After the variables have been modified to respect their maximum ranges, these variables are removed from the system and the process is repeated. This reduces the possibility for a non optimal solution, but does not guarantee one either. It is provided as a computationally cheap alternative to a full optimization which can at least guarantee valid if not optimal results.

3.2.2.2 Input Resolution

A major problem faced when applying run by run control to a real process is input resolu-

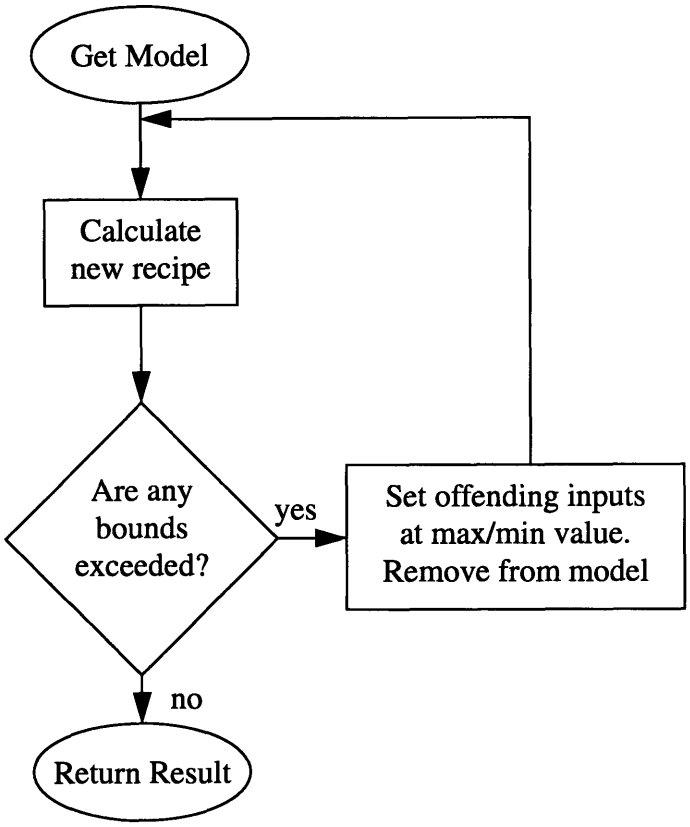


Figure 13: Input bounds algorithm

tion. Even a perfectly modeled system can suffer from this. Control decisions based on infinite resolution must be rounded by the operator (or the machine) to acceptable increments. This can often lead to unacceptable control, and as a side effect gives false information back to the controller algorithm: namely that the suggested recipe was used when in fact a rounded version was used.

As a first step to addressing this problem, a simple iterative method is proposed to provide resolution control. Inputs are ordered from least to most adjustable (using their input weights see section 3.2.2.4) and then sequentially rounded and removed from the equation. The remaining inputs are then adjusted to obtain the best solution for the new problem. This is repeated until all inputs have been rounded. Figure 14 shows a diagram of the method used.

3.2.2.3 Output Weights

It is often the case that the desired target of a process cannot be reached given the constraints of the system. If this is the case, a decision must be made as to the relative impor-

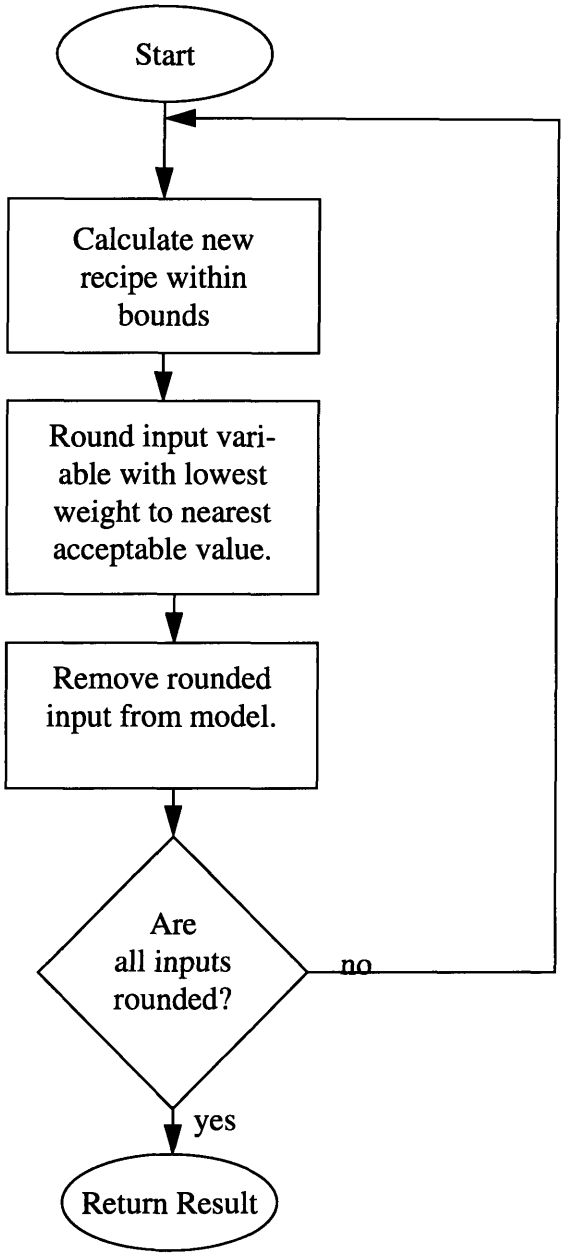


Figure 14: Input resolution methodology

tance of each output. The default is, of course, equal weighting, but this may not be desirable. For example, if a process has two outputs, thickness and uniformity, the operator may want optimal thickness with a secondary requirement of good uniformity. The weights could also be set inversely proportional to the variance of the output variable. This would put greater importance on those variables with low variance e.g. those that can be more accurately controlled. The controller accomplishes this by applying an output weighting matrix *W*.

$$W = \begin{bmatrix} w_1 & 0 & 0 \\ 0 & \dots & 0 \\ 0 & 0 & w_m \end{bmatrix}$$

where: $w_1 \dots w_m$ are the relative weights for outputs 1 ... m.

The system equation for the output becomes:

$$\begin{aligned} Wy &= WAx + Wc \\ W(y-c) &= WAx \\ (WA)^T W(y-c) &= (WA)^T WAx \\ \left(A^T W^T WA\right)^{-1} A^T W^T W(y-c) &= x \end{aligned}$$

The weighting works by biasing the magnitude of certain outputs so that when a least squared solution is calculated, outputs with higher weights are penalized the solution the most so they are set closer to their target than other outputs. Application of output weights in an exact or underdetermined system has no effect on the output; in both cases there is no reason to sacrifice one output to obtain another, therefore all outputs are reached.

Other related bias terms are the model update weights. These weights (a for EWMA, a and b for PCC control) determine the aggressiveness of the controller for each of the outputs. These parameters can be used to minimize the effects on certain noisy outputs while increasing the affect of more stable outputs. The result is a system that can quickly adapt to changing conditions while being resistant to process noise. An added benefit of these parameters is that they affect the system regardless of its condition (i.e. underdetermined, exact, or overdetermined).

3.2.2.4 Input Weights (Input Adjustability)

Although the inputs to the system are normalized to ensure consistent operation, weights can also be applied to these inputs to add yet another level of control. Input weights enable the user to set the adjustability of the inputs. By this it is meant that input variables weighted heavily are adjusted with greater magnitude relative to lightly weighted variables.

Application of the weighting is achieved by adjusting the normalized input variables so that the least squared distance incurred by each variable (distance of new x_t from x_{t-1}:

where t is the run number) is adjusted by its input weight. This should not be confused with the output weights discussed earlier for they have exactly opposite behavior. The input weighting has no effect on both overdetermined and exact solution problems. In those cases, the inputs are not factored into the calculation of the error for the final solution, so the magnitude of the inputs which is the key to their weighting is irrelevant. In the underdetermined case, where all outputs are met, the recipe is determined with the added constraint of being as close to the old recipe as possible. This can be biased by the relative weighting of the inputs. Inputs that are weighted heavily are forced to be the least adjustable due to their relatively large effect on the error calculation for the recipe. A matrix V and its inverse V^{1} are used to apply the input weighting.

$$V = \begin{bmatrix} v_1 & 0 & 0 \\ 0 & \dots & 0 \\ 0 & 0 & v_n \end{bmatrix}$$

where: $v_1 \dots v_n$ are the relative weights for inputs 1 ... n.

Once a weight matrix has been defined, it must be applied in such a manner as to ensure that the formulation of the problem leads to a correct solution. In order to achieve this, the weight must be applied to both the recipe and the slope term. First, the application of the weight term V to the recipe x modifies the least-squared error generated by these inputs when determining the solution closest to the previous solution (see section 3.213). The side effect of this weighting is that the new output generated by these inputs is not consistent to the original problem formulation. To remedy this, the slope term A is weighted with the inverse of the recipe weight. The system equation for the output becomes:

$$y = Ax + c$$
$$y = \left(A \cdot V^{-1}\right) \cdot (V \cdot x) + c$$
$$y = A^{*} x^{*} + c$$

This new formulation can be used in place of the original variables to provide the necessary weighting. The problem is then treated as before (see section 3.2.1.3), but with the

new scaled values.

$$min\left\|x^* - x_0^*\right\|^2\Big|_{A^*x^* = Ax = b}$$

The solution, however, is based on these scaled values, so it must be scaled back to the original domain.

$$x = V^{-1} \cdot x^*$$

3.3 Future Improvements

The RbR controller was designed to enable the incorporation of a wide range of algorithms. This initially was done to allow the user to select from a variety of control methods or test different algorithms with similar data sets. Although this has remained true, there has been an unexpected reward to implementing multiple algorithms in the same control framework. Since the solution generated by the controller is broken down into two parts, model update and recipe update, the algorithms used for the model update can all benefit from advances in the recipe update stage. As discussed earlier, a number of constraints and bias terms have been implemented to occur within the recipe update phase of the control action.

Future work on the control algorithms will be divided into the two phases of the solution. New constant (and possibly slope) update algorithms will be examined to give the user more freedom when controlling a variety of systems. The constraints and bias terms will also be expanded and improved. Many of these parameters have not been extensively tested, and could be improved.

Chapter 4 discusses the interfaces to the controller for communicating with other sources as well as the abstractions used internal to the controller. It is this internal data abstraction which is key to enabling expansion of the controller. Future algorithms can be added to the controller without requiring major code revision or changes to the other algorithms.

Chapter 4

RbR Interface Specification

In order to enable the integration of run by run control into a variety of existing systems, key issues must be resolved. The goal is to define and implement specifications for both data abstraction and communications that will allow the RbR controller to communicate with various clients with minimal modification of code. In order to accomplish this, the task of defining an interface specification for the RbR controller has been broken down into several parts.

First, a generic data abstraction is presented that attempts to define a concise representation of the data needed for run by run control. Once this is completed, the actual mappings to this abstraction used by the RbR control server are presented. A communication model is also defined to communicate the data stored within the data model between dissimilar modules. Finally, an example implementation of the data and communication model is shown. Each of these is discussed in more detail in this chapter.

4.1 Generic Data Definition and Abstraction

In order to implement algorithms effectively, a controller must have the data required by these algorithms. This can lead to a very tight and undesirable relationship between the controller and the data provider. Small changes to the algorithms often require changes in the data provided to the controller. This leads to the need for a clear abstraction of the data used by the controller. The goal is to provide a framework by which many different algo-

rithms can query the same data store without requiring specific modifications to the underlying routines. In order to achieve this, the data used for the controller is broken down into three levels as shown in Figure 15:

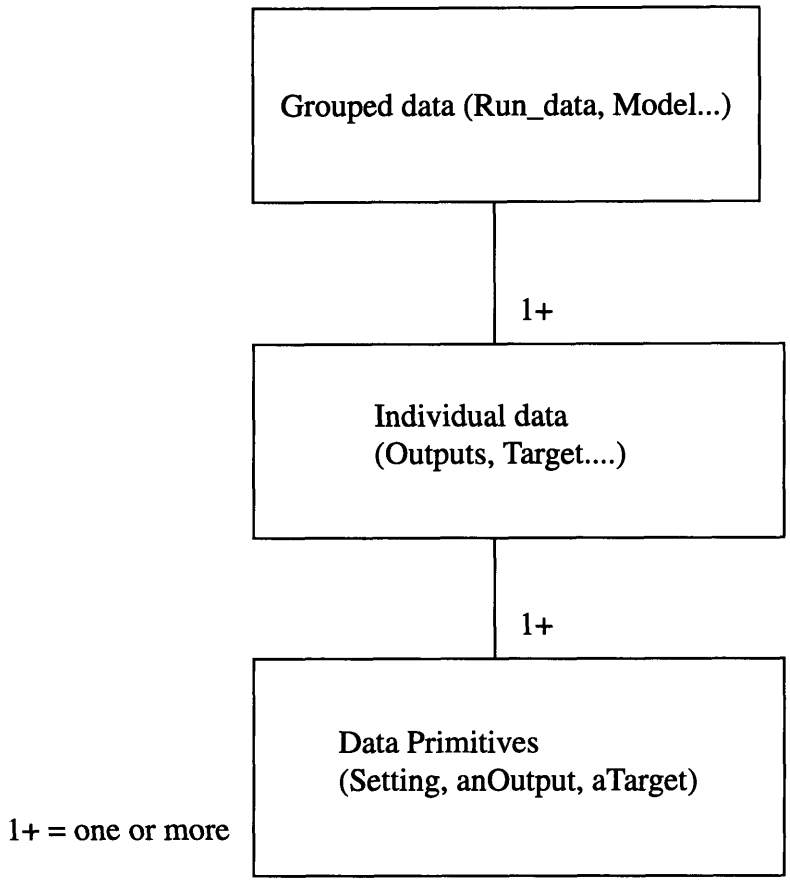


Figure 15: RbR data abstraction

- Grouped data level.
- Individual data level.
- Data primitive level.

Figure 8 shows the data hierarchy used for the RbR controller. The data groups become increasingly more specific and complex as they approach the Grouped data level. The reason for this is to allow various algorithms within the controller to use the same base of primitive data. By building specific groups needed for specific algorithms, the advantages of structured data are achieved without the loss of generality at the most basic level. The levels are further explained below.

4.1.1 High-level grouped data:

At the highest level of data abstraction are the data groups. These groups contain well defined sets of data that are related in a logical way. This data is then acted upon by various algorithms that are designed to provide run by run control. Each algorithm can request a given data group that contains the data needed for correct operation of that algorithm. Data objects can be expanded and modified as the needs of algorithms change with little or no changes to the underlying data from which they are constructed. This allows functionality to be added without the cost of code revision.

4.1.2 Medium-level individual data:

Individual data is often the finest grain of data needed in the operation of the RbR controller. This level of data represents one complete data item. These items can be scalars, vectors, or multi-dimensional arrays; what is important is that they are a single unit of data. They can be referred to by one name and represent a single piece of information. This differs from a set of data which can be represented as a vector or array but represents a collection of different data items.

4.1.3 Low-level data primitives:

At the lowest level are data primitives. These primitives can be of any form or dimension, but share one common attribute: they cannot be referred to by a unique name. They are part of larger data items. The reason for this level of data is that many times the physical form of the data is primitive in nature and must be processed to create higher levels of abstraction.

4.2 RbR High-level Data Mapping

The RbR controller can be thought of as a collection of control algorithms that share the attribute of run by run operation. Currently, the two algorithms used in the controller, EWMA and PCC, require almost identical data sets. This has led to the data mapping used in the RbR controller. This is by no means the only possible mapping; in fact, as more algorithms are introduced, various groups of data will be developed to satisfy their needs. The important thing to note is that the medium and low level data items contained in these groups are generic and can be used equally well in other groups as more algorithms are

added. These high level data abstractions are discussed next.

4.2.1 Constraints

All parameter constraints are contained in this group. It is assumed that these bounds do not change over the course of time, or that changing the constraints would invalidate the use of previous data.

Data:

Rmin	The lower bounds of the Recipe input data.
Rmax	The upper bounds of the Recipe input data.
Target	The desired output of the system.
iweight	The input weights for the system.
oweight	The output weights for the system.
resolution	The input resolution for each input.
Scale_flag	Boolean signalling normalized data.

4.2.2 Run_data

The run_data represents a snap-shot in time of the result of machine operation. This includes input settings data for the machine, and the resulting outputs.

Data:

Recipe	Settings used to configure the machine before each run.
Output	Measured outputs gathered after previous run.

4.2.3 Model

The run by run controller currently uses a first order polynomial model as its basis for control. The dynamic elements of that model are stored in this data group. If the model were to be expanded to other higher-order terms, they would be named and stored within this group.

Data:

Slope	This represents the coefficients for each term in the first order polynomial model. Since the controller supports Multi-Input-Multi-Output (MIMO) models, this object is often two dimensional.
Intercept	The constant offset of the first order polynomial model. There is one intercept associated with each output in the model.

4.2.4 Algorithm_params

The parameters for the two algorithms used in the RbR controller are stored in this group. It may seem wasteful to carry information about an unused algorithm when using another, but since currently the number of algorithm parameters is small, it is easiest to lump them

together into one group. If more complex models with many parameters are later added, specialization of the algorithm_params group can be made to accommodate them.

Data:

g_alpha	Alpha parameter (forgetting factor) used in the RbR EWMA mode algorithm.
pcc_alpha	Alpha parameter used for the Predictive Corrective Control (PCC) algorithm.
pcc_beta	Beta parameter for PCC.

4.2.5 Controller_flags

The Controller_flags group contains various flags that select which control method to use. This group could also include initialization information common to all control methods.

Data:

enable_rapid	Boolean to enable Rapid Mode.
select_gradual_pcc	Selects between EWMA, PCC, and None. These can not be run simultaneously.

4.3 Communication Model

The RbR controller is arranged in a client/server configuration using TCP/IP sockets [Car94]. This RbR Control Server has the ability to accept multiple requests for control action from various clients. The key to the integration of this server into existing systems is the communication model. Figure 16 shows an overview of the RbR Control Server as well as a few possible connections.

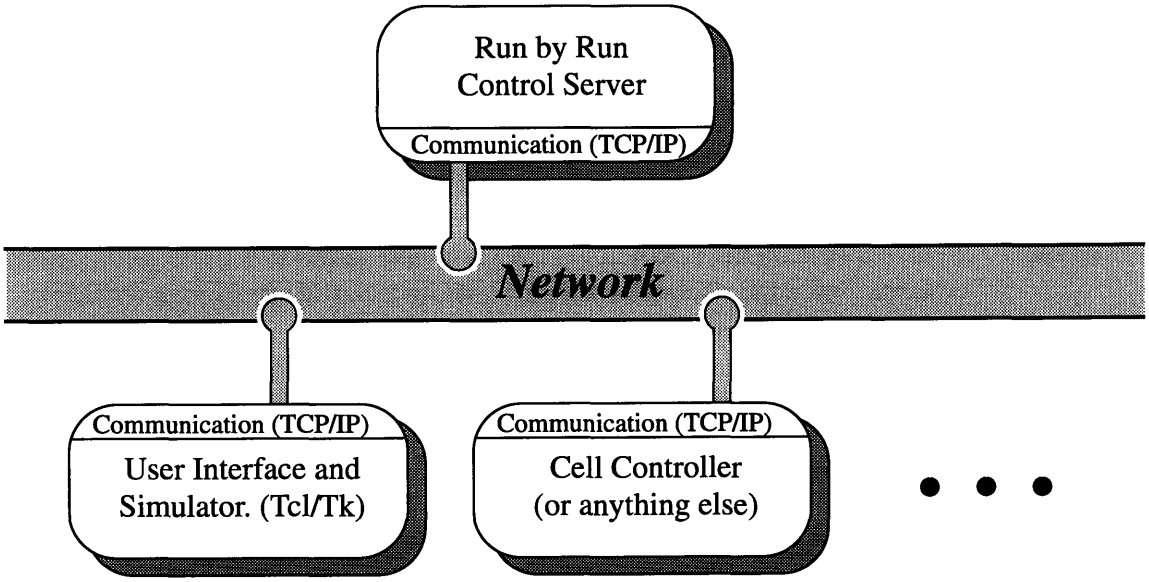


Figure 16: RbR Server and possible connections (clients)

Although sockets were chosen as a robust method of communication with the server, the underlying messaging system is independent of the media. The advantage of this is that the same core controller can be used for both client/server operation, as well as embedded operation. The communication model is based on the data model and has a one-to-one mapping with its components. This makes integration of the communication modules into the controller much more straightforward.

Figure 17 shows an overview of the communication model. The model is arranged as a set of functional modules that communicate via messages. The Control Requester is the external client that uses the RbR Controller for recipe suggestions. The Database is an external storage area for data not contained within the Control Requester. The other three components (RbR "Brain," Group Builder, and Data Builder) are actually contained within the RbR Controller, but are separated here to emphasize the abstraction used by the controller. The RbR "Brain" consists of the algorithms used for the recipe generation process and operates on a group-based data level. The Group and Data Builder provide valuable translation services to the RbR "Brain."

Communication between the modules follows the following pattern. First, the Control Requester queries the RbR Controller for recipe information (via calculateSettings). Next, the RbR "Brain" must retrieve group-based data information it needs for operation. To do this, the "Brain" sends a get_group() message to the Group Builder requesting pieces of data. The Group Builder then retrieves the information needed for each group of data. This may require multiple queries using get_data() to the Control Requester or Database to get all the data needed. If this level of interaction is not supported, then the Data Builder is called. The Data Builder preforms an operation similar to the Group Builder, but at a single scalar level, piecing vectors together out of scalars. Once these operations have been performed the "Brain" can perform the requested control action and return the result. This result is again translated to the level needed by the Control Requester; a set of modules (not shown) provide the sending counterpart to the ones discussed above. The net result is a communication model that provides both abstract data models with compatible interaction levels, customizable to the data interface requirement of particular systems.

There are two possible integration paths while using the communication model. First, if the controller has been established and a system wishes to connect without disrupting

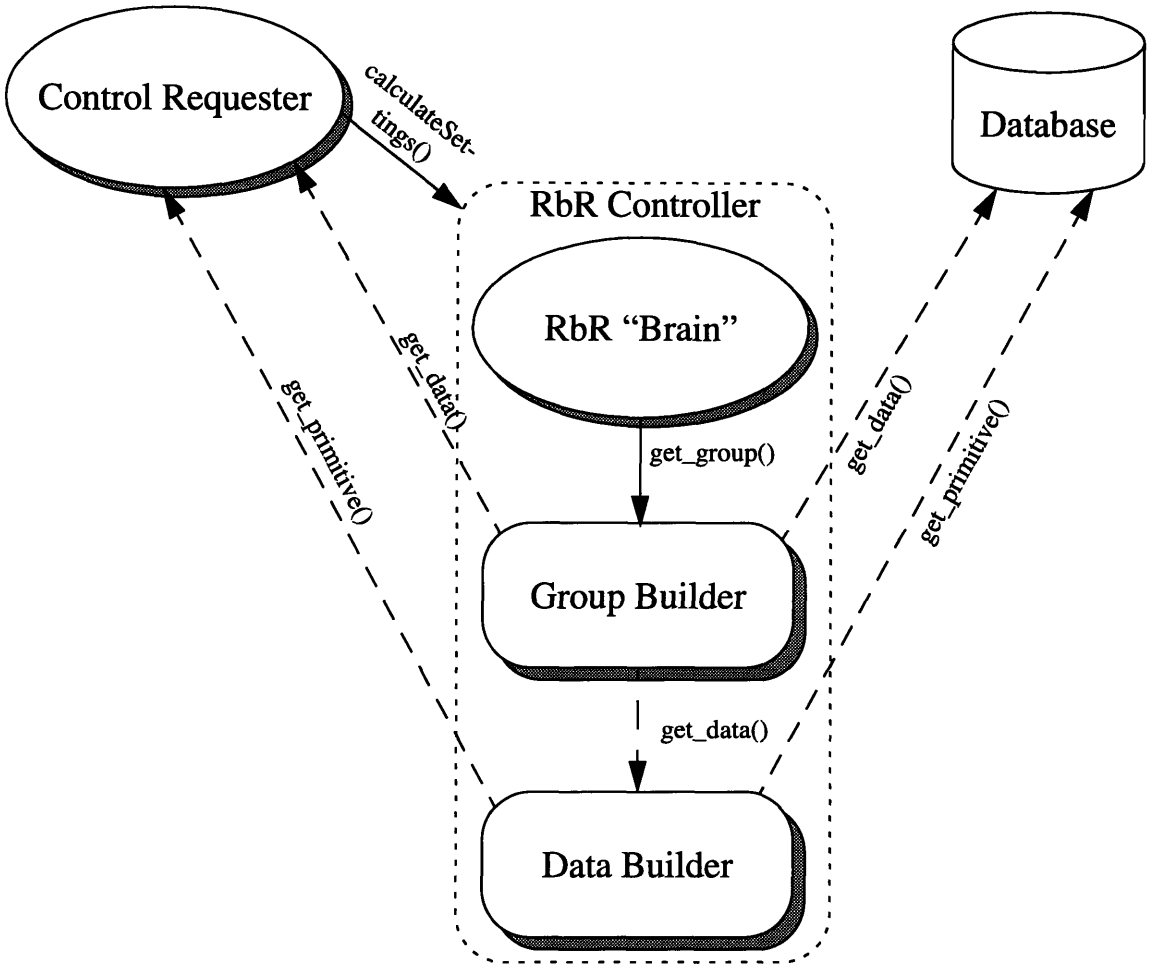


Figure 17: RbR communication model

existing clients, then it must comply with the established communication level. To do this, the client must both request and provide data in the manner defined by the controller. Since there are only three possible levels of communication in this model, it should be straightforward to determine and implement the correct level of interaction. The second integration path involves writing a control server to service a number of previously established clients. In this scenario, the "Brain" of the controller can remain constant, but the two builder routines must be written so that they can correctly interpret the data level presented by each of the valid clients and act accordingly. Once this has been established, additional clients can be added by either conforming to the existing methods, or modifying the builder routines to recognize and correctly translate data between the client and the control server.

The remainder of this section describes the specific routines used to access the data at

the three levels. Other issues such as desired behavior and data scoping are also discussed. The section ends with a short pseudo coded example using some of the communication routines.

4.3.1 Desired Functional Behavior

It is important that the basic behavior of the data accessor routines be defined. This behavior goes beyond the mere prototype for enacting the function, but rather centers on the state of the data once it is retrieved or sent via the routines. By defining certain behavior ahead of time, incorrect assumptions that could lead to possible errors can be avoided. Although the list of important behavior can be extensive, some of the more prevalent pitfalls are discussed below.

Memory management is often implemented incorrectly. A program that incorrectly manages memory can often run flawlessly but then crash unexpectedly and even worse unpredictably. This need for robust memory management leads to some of the behavior required of the data routines. As a matter of convention, the routines used in the RbR controller to both store and retrieve data leave de-allocation of variables to the calling functions, independent of who originally allocates the storage. This simplifies use of the routines by maintaining a consistent interface to their use.

In addition to memory constraints, the access of data at all levels must also be standardized. To this end, the concept of the access "key" has been developed. Both storage and retrieval of data items is based on a key. A key in its simplest sense is a unique name. This name is used as an identifier to access data between two sources. The use of a key allows a single routine to access all forms of data. This is very powerful when many varying types of data must be passed between the RbR controller and a data source.

A problem with using a key is that it implies a global namespace. This means that all data is effectively floating in a large sea, with no grouping structure. This can be undesirable when large amounts of data from various sources must be accessed. An example would be a query to a large database containing several machines that all share the same statistics, naming each variable uniquely could lead to confusion, while identical naming schemes would be impossible. To remedy this, a function is provided which takes a format string and a variable number of arguments to build a scoped key. The key then assumes a

dual role, both as a name identifier, and as a scope specifier. In a simple implementation, the key will be a name specifying an object in a global namespace, while in a complex implementation, the key will be a formatted string that is used to scope the namespace. An example follows:

```
char *key_mask="%s:%s:%s:MyOffice";
char **key;
char *data;
char *country="USA";
char *state="MASS";
char *city="CAMBRIDGE";
make_key(key,key_mask,country,state,city);
get_data(key,data);
```

Where:

```
error_type make_key(char **key,char *mask,...)
   key......Output: Processed key is returned.
   mask.....Input: Format string for key.
   "..."....Input(s): Argument(s) to fill in format fields.
```

In this function, the name "MyOffice" is given scope by country, state and city. This provides an easy to use yet powerful and dynamic naming convention. The format of the scoping is left to the implementation, and in simple cases, where global naming is not an issue keys may be simple names with no scope formatting.

The remainder of this section will define the prototypes for the interface modules. They are arranged in hierarchical order, highest to lowest. Although the specification is meant to be generic, an ANSI C language interface is used as an example since this is the language used in the controller itself.

4.3.2 Data Group Level

At the highest level of interaction are the data group functions. These access complete groups of related data and store them in a provided variable. The structure of this data is very specific and well defined. There is freedom, however, as to the size of any one of the data elements. This size is stored within the variable and can be accessed when needed. There are only two functions needed at this level, one to receive data groups from the outside, and one to send them. This is done by *get_group* and *put_group*.

```
error_type get_group(char *key,void **data)
   key......Input
   data.....Output

error_type put_group(char *key,void *data)
```

```
   key......Input
   data.....Input
```

Both *get_group* and *put_group* use the key concept to access data that is to be passed between the data source and the controller. The data is both sent and received using the *data* argument. An attempt has been made to make both the sending and receiving routines exact counterparts. This leads to a more standard interface that is easier to remember and understand. Also, as discussed earlier, allocation of these groups is done by the *get_group* command, but de-allocation (freeing) must be done by the caller. For this reason, every group object has complete information about its internal size. This allows them to be safely destroyed.

4.3.3 Individual Data Level

As discussed earlier, it is often difficult to request entire groups from the data source. For this reason, simpler data items are used. These data items can then be arranged into groups to form the grouped data items used at the highest level.

To retain a consistent interface to all types and dimensions of data, a simple one dimensional variable length vector has been chosen as the basic unit of communication. This requires that scalars be packed as length 1 vectors, and more importantly that higher dimension variables be packed into a vector and then later expanded. This may seem limiting, but the additional overhead of communicating arbitrary dimension data is enough to warrant a universal coding scheme.

```
error_type get_data(char *key,enum dtype,void **data,int
*length)
   key......Input
   dtype....Input
   data.....Output
   length...Output

error_type put_data(char *key,enum dtype,void *data,int
*length)
   key......Input
   dtype....Input
   data.....Input
   length...Input
```

Again both *get_data* and *put_data* share identical calling arguments. At this level there is less agreement or knowledge about the structure of the data being accessed. The *dtype*

and *length* arguments have been added to remedy this. *Length* is simply the length of the vector being sent or received. *Dtype* is an enumerated type representing one of the primary data types (int, float, char, long). The use of *dtype* as an input to the *get_data* routine is to allow further scoping for the data being requested (e.g. to differentiate between an integer named "foo" and a float named "foo").

4.3.4 Data primitive level

At the lowest level of the data abstraction is the data primitive interface. This makes very few assumptions about the outside data other than that it exists and can be referenced via a key. These functions are designed to return only a single value upon execution. As with the data primitives, higher level data can be made by grouping many data primitives together.

```
error_type get_primitive(char *key,enum dtype,void *element)
   key......Input
   dtype....Input
   element..Output

error_type put_primitive(char *key,enum dtype,void element)
   key......Input
   dtype....Input
   element..Input
```

4.4 Example Implementation

There are many possible levels at which the RbR controller can be integrated into a system. One example is to control the oven discussed in chapter 2. For this application (baking cookies) concepts of data groups such as settings and measurements are foreign to the existing process. The original equipment was designed for simple single value data communication, yet group data is what is used by the control algorithms. All that is available are three input controls (temperature, time, rack height) and two outputs (center hardness and outside color). This represents an individual data interface to the system. Even though the data is scalar by nature the data is not considered primitive; each data item represents a complete piece of information. Figure 18 shows the interface between the controller and the system for this fictitious example.

The symbols used in this example are summarized in table 1. One complete run of the

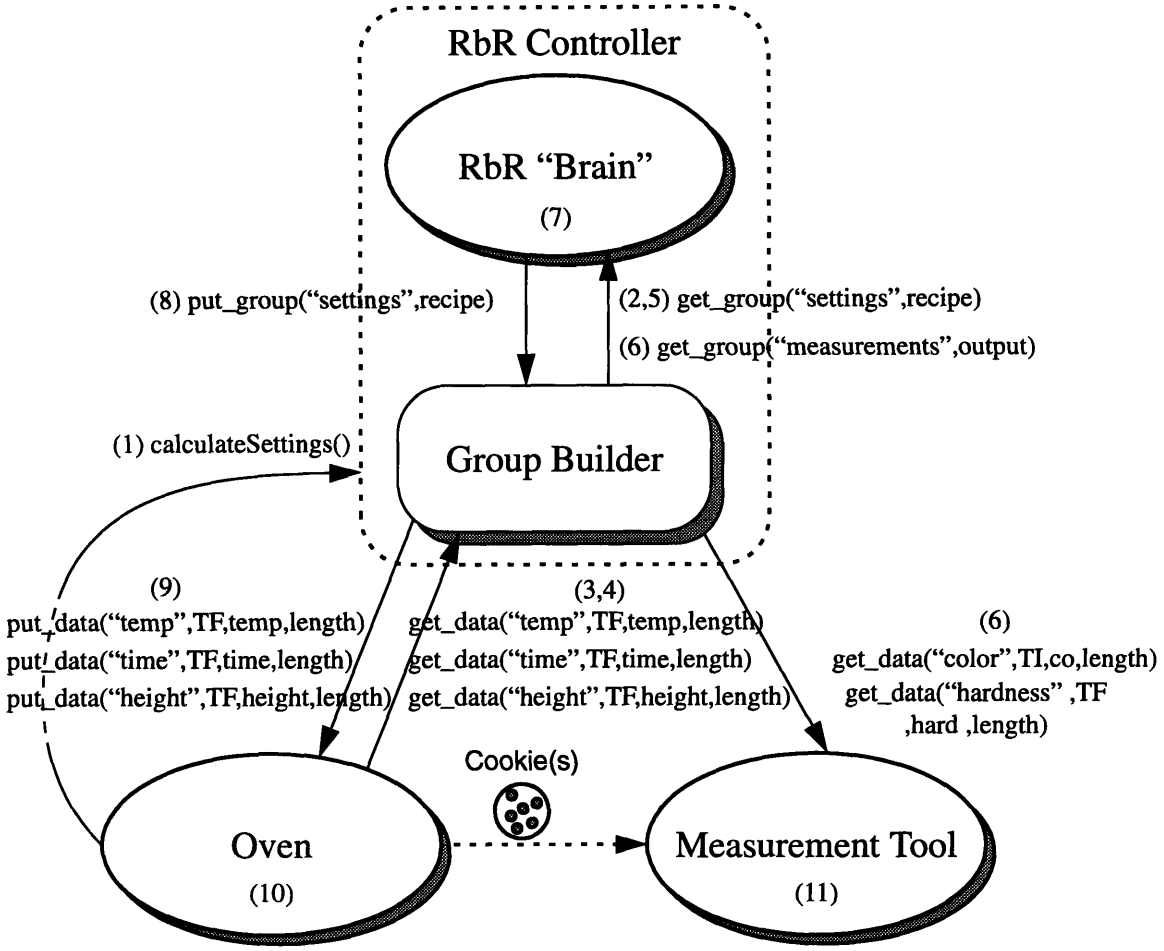


Figure 18: Example RbR controller communication

controller involves several steps shown in table 2.

Key	Variable	Length	Type	Description
settings	recipe	3	group	Settings for oven.
measurements	output	2	group	Measurement data.
temp	temp	1	float vector	Temperature of oven.
time	time	1	float vector	Time to cook cookies.
height	height	1	float vector	Height of oven rack.
color	color	1	int vector	Color of cookie.
hardness	hard	1	float vector	Hardness of cookie center.
(none)	length	1	integer	Generic length variable.

Table 1: Example variable definitions

Key	Variable	Length	Type	Description
Note: TF and TI are enumerated types "type float" and "type int", used for the dtype arg.				

Table 1: Example variable definitions

Step	Description of Action
1	Oven calls calculateSettings() to request control from RbR Controller[a].
2	RbR "Brain" calls *get_group()* to get "settings".
3	Group Builder must actually put groups together, so for "settings" it calls *get_data()* three times for the data `temp`, `time`, and `height`.
4	Oven responds to *get_data()* for "temp", "time", and "height" keys and sends their values.
5	Group Builder combines `temp`, `time`, and `height` into `recipe` and returns to RbR "Brain"
6	Steps 2-5 repeat for "measurements", except the Measurement Tool is queried.
7	RbR "Brain" performs control calculations.
8	RbR "Brain" calls *put_group()* for "settings".
9	Group Builder unpacks `recipe` into `temp`, `time`, and `height` variables and calls *put_data()* three times.
10	Oven responds to *put_data()* calls and uses `temp`, `time`, and `height` to run oven.
11	Finished cookie is sent to Measurement Tool.
12	Process Repeats

Table 2: Steps for one complete "run"

a. To avoid confusion, the following convention will be used for keys and variable names:"aKey", `aVariable`.

The key point of the example is that the structure of the data external to the controller is not important, as long as it can be accessed in some way. By using "builder" routines, the controller can maintain a high level interface to the data. This allows one controller to be used for various applications. If later, the oven is updated to allow downloading of a complete control setting, then the controller could interact with the oven at a higher level. From the perspective of the control algorithms, however, the result would be the same.

Chapter 5

Results of Run by Run Control

As the primary goal of this research is to allow run by run control to be easily integrated into systems, it is useful to examine the results of this integration. The two primary test-beds for MIT run by run control using the control server (or its algorithmic core) were set up at SEMATECH and Digital Equipment Corporation (DEC). All tests were done on the chemical mechanical planarization (CMP) process. Appendix A presents a brief overview of the process.

5.1 SEMATECH Tests

As part of the SEMATECH J88D project, two tests were conducted on an industrial CMP machine. The purpose was to show feasibility of both run by run control and of the Generic Cell Controller (GCC). The two tests were performed in October 1994 and December 1994 respectively.

5.1.1 SEMATECH: First Test

The first SEMATECH CMP control experiment proved to be a valuable test of the Run by Run Control Server. Although originally it was planned to use the GCC to provide a control framework which included the algorithmic core of the RbR Server, the GCC was still under development when the actual test was done. This required a stand-alone version of the RbR controller to be developed to perform the test without the GCC.

As an example of the flexibility and open structure of the RbR Server, a simple text input/output client was written in under one week. This simple interface provided the link to the Control Server that was used in the control experiment. Figure 19 shows a sample output from the text-based client.

```
Enter type of control:(0=gradual 1=pcc1 2=pcc2) 0
Number of inputs to system: 2
Number of outputs to system: 2
Do you want to modify model parameters?
(Enter y/n): y
Enter gradual alpha:(0.000000) .5
Enter 2 constant terms separated by spaces:
0.000   0.000
1.0 2.0
Enter 2 targets separated by spaces:
0.000   0.000
2.0 4.0
Enter 2 slope coefficients for equation 0:
(Separated by spaces)
0.000   0.000
.1 .4
Enter 2 slope coefficients for equation 1:
(Separated by spaces)
0.000   0.000
.5 .23
Enter 2 recipe terms separated by spaces:
0.000   0.000
10 20
Enter 2 recipe lower bounds separated by spaces:
0.000   0.000
0 -10
Enter 2 recipe upper bounds separated by spaces:
0.000   0.000
50 75
Enter 2 weight terms separated by spaces:
0.000   0.000
1.0 1.0

Enter 2 system outputs
2.5 4.03

---------------------- RESULTS ----------------------
Control Mode=> gradual

Expected output after correction:
2.0 4.0

New Constant terms:
1.11 1.897

New Recipe
18.65 21.2

-----------------------------------------------------
Would you like to control another run?(y/n)
```

Figure 19: RbR text client

The results of the first SEMATECH test proved to be quite favorable. The controller was able to maintain a constant removal rate without drastic changes in the non-unifor-

mity. The test also pointed out weaknesses in the control approach. In particular, the control parameters of the machine were discretized and could not be precisely set to the suggested recipe. This led to sudden "bumps" in the recipe whenever an input crossed a discretization point. This also led to very few actual control changes.

5.1.2 SEMATECH: Second Test

The second run by run control test was done using the GCC with embedded run by run algorithms. Although the control system was better organized, the test uncovered limitations in the current algorithm. Discretization was not implemented in the controller, and proved to be needed in this multiple input, multiple output scenario. The system drifted significantly, with marginal ability to correct this trend.

The test provided a wealth of information as to the limitations of the controller as it was then formulated. Among the areas that needed work were: discretized inputs, weighted inputs, separate gradual weights for each output, and simulator development. Each of these enhancements provides another level of customization of the controller. Currently all updates have been integrated into the stand-alone controller and the updated controller will soon be integrated into the GCC.

5.2 Digital Equipment Corporation Test

Run by run CMP was examined at DEC by an MIT Leaders for Manufacturing student, Arthur Altman [Alt95]. The thrust of Altman's work was to show feasibility of run by run control when applied to CMP. He used CRUISE to both test his control models, and to control the actual machine.

The development and testing phase of the project done at DEC provided valuable feedback as to errors and limitations present in the run by run GUI. The need to store simulation data in a way that allowed simulations to be performed over multiple days was addressed and added to the environment. The need for feedforward information also became apparent due to the type of problem being examined. This too was added and subsequently improved by valuable interaction with Altman.

Due to the apparent drift examined while characterizing the system, Altman decided to use the Predictor Corrector Control (PCC) algorithm as a base for the control scheme. His

two goals where to control both final thickness and uniformity. This was done by monitoring 9 points on the wafer (see Appendix A).

Due to the lack of materials, time, and access to the CMP machine, the tests were not large enough to provide conclusive evidence of correct control but do suggest that run by run control can effectively improve CMP processing. The DEC experiments also provided the first on-line test of CRUISE. Especially informative data collected from the run included the resulting outputs for Mean Polish Rate Slope (where slope is a measure of non-uniformity). This graph, shown in figure 20, is an example of both a controlled as well as an uncontrolled process. The reason for the process moving to an uncontrollable state is that, at run four of the experiment, the control "knob" reached its limit, and thereafter the opportunity to control the process further was lost. The first few runs suggest that run by run control can maintain the process near target; runs 5-9 show that without control the uniformity measure drifts and degrades with time.

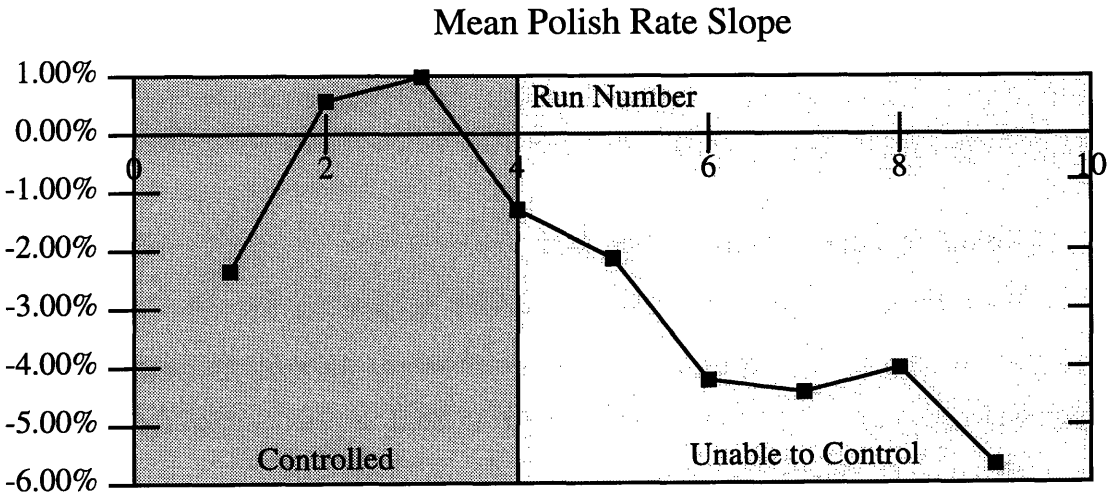


Figure 20: Output from DEC CMP experiment

Although the test performed is small in relation to those used to verify the use of a process, it does provide useful information as to the usability of the Run by Run Control and Simulation Environment. It has also been instrumental in fixing errors and structural limitations that would otherwise have gone unnoticed due to lack of use. Future tests will be needed to advance the state of the interface, but this was a very productive and fruitful start.

Chapter 6

Summary

One of the greatest barriers to the integration of run by run control into VLSI manufacturing is compatibility with existing systems. To address this issue, a generic data format as well as a multilevel communication model for run by run control has been developed. The hope is that using these abstractions, the task of integration can be greatly simplified.

The abstraction used is broken down into three levels, high, medium, and low. These represent increasingly primitive representations of data. The highest level contains logical groups of related data. At the mid level, single data items are stored. These items may have many components, but are considered a single entity. The lowest level of the abstraction interacts with raw data types directly. All three levels are needed to present a complete picture of possible operation.

Complementing the data model is a three tier communication model. Each level communicates data items of a given complexity (high, medium, low). These three levels of communication are unseen by the inner core of the controller which uses high-level data exclusively. This abstraction is facilitated by modules that construct complex data groups out of primitive data items. This allows the algorithms of the controller to maintain a high-level data abstraction without requiring this level of communication from the data provider. By allowing multiple levels and complexities of interaction, the controller can connect to a variety of systems.

To demonstrate the use of the data abstractions and communication model, a run by run control server has been developed. This server uses TCP/IP sockets to allow multiple clients to request run by run control over a network. The server provides both a medium and primitive level of communication to the clients. Control data is communicated via access "keys" that denote various pieces of data. These keys can be easily expanded as more complex control algorithms are added and a single server can support multiple algorithms by querying algorithm-specific data from the client as needed.

In addition to a control server, a stand alone graphical client has also been developed. CRUISE is a Tcl/Tk based application that provides complete access to the control server. This interface provides three useful functions. First, the interface serves as an example implementation of the messaging system in the RbR Control Server and can be used to both test and demonstrate the messaging used. Second, the interface has a wide range of simulation features that can be used to both simulate real machinery as well as test new algorithms as they are being developed. Third, the interface can be used as a fully functional run by run controller interface. In this scenario, real processes can be controlled by entering data into the interface and querying the Control Server for control decisions. The interface has information graphing and storing functions that are useful when evaluating the control action for a process.

The controller has also been integrated into two manufacturing systems as an effort to examine the effects of run by run control. These systems are the MIT Computer-Aided Fabrication Environment (CAFE) and the Generic Cell Controller (GCC). CAFE is a CIM system developed at MIT that integrates the RbR Control Server via an external callable program. This program provides the user with the options of accepting the run by run control suggestion or the default recipe for the process. The GCC development effort is a project at The University of Michigan under contract with SEMATECH to examine the possible introduction of run by run (and other methods of) optimization and control into VLSI fabrication. The functional core of the RbR Server is integrated directly into the GCC.

Along with interface and integration issues, the run by run algorithms themselves have been expanded to include: input bounds, input weights, input discretization, and output weights. These parameters enhance the operation of the underlying algorithms by provid-

ing both constraints and bias terms. These "knobs" give the process engineer greater freedom to adjust the controller to provide the best possible control.

As with all applications, the RbR Control Server and the data abstraction and communication model it uses are continually evolving. The communication model will be modified as more experience is gained in the area of integration. This is needed to ensure that the controller remains as flexible as possible. The underlying data model may also change and become object oriented as object based systems become more prevalent.

The controller is designed to be expandable by allowing multiple algorithms to coexist. Both the addition of new algorithms and the further refining of existing ones will allow the controller to provide a wide range of control choices. This will allow comparisons between various algorithms without the need to redesign the implementation from scratch. This saves time and leads to more accurate results by enabling the researcher to effectively toggle between competing algorithms.

The user interface will also expand to embrace new functionality as it is added to the controller. The interface is object oriented[2] and therefore can be easily modified and expanded as advances are made. This has been the method used throughout the development of the interface. New or modified controller options are first added and then tested using the graphical interface.

The main focus of this research has been to develop a run by run controller for integration with various clients. This goal can be best met if the controller, the communication models, and algorithms themselves continually evolve to meet the changing needs of manufacturing systems.

2. The interface uses [incr tcl], an object oriented extension to the Tcl/Tk programming language.

References

[Ing91] A. Ingolfsson, *Run by Run Process Control*, MS Thesis, Operations Research, Massachusetts Institute of Technology, 1991.

[BS93] S. Butler and J. Stefani, "Application of Predictive Corrector Control to Polysilicon Gate Etching," *American Control Conference*, June 1993.

[SHI91] E. Sachs, A. Hu and A. Ingolfsson, "Run by Run Process Control: Combining SPC and Feedback Control," *IEEE Transactions on Semiconductor Manufacturing*, Oct 1991.

[WEC56] Western Electric Company, *Statistical Quality Control*, AT&T, Charlotte, NC, 1956.

[Ira90] K. Irani, J. Cheng, U. Fayyad, and Z. Qian, "Applications of Machine Learning Techniques in Semiconductor Manufacturing." *Proceedings of The S.P.I.E. Conference on Applications of Artificial Intelligence* VIII, Bellingham, WA, 1990.

[Hu93a] A. Hu, "An Optimal Bayesian Process Controller for Flexible Manufacturing." *Technical Report MTL Memo 93-711*, Massachusetts Institute of Technology, 1993.

[MSC94] P. Mozumder, S. Saxena, and D. Collins, "A Monitor Wafer Based Controller for Semiconductor Processes", Accepted for publication, *IEEE Trans-*

actions on Semiconductor Manufacturing, 1994.

[KS94] N. Kermiche and R. Su, "Run-to-Run Process Control," *Semiconductor Research Corporation*, 1994.

[OCo94] John O'Connor, *Collaborative Manufacturing System Computer Integrated Manufacturing (CIM) Application Framework Specification 1.0*, SEMATECH, Mar. 1994.

[Hu93b] A. Hu, S. Wong, D. Boning, K. Wong, *The Run by Run Controller Version 1.2*, San Jose State University, Massachusetts Institute of Technology, 1993.

[SBHW94] M. Sullivan, S. Butler, J. Hirsch and C. Wang, "A Control-to-Target Architecture for Process Control," *IEEE Transactions on Semiconductor Manufacturing*, Vol. 7, No. 2, pp. 134-148, May 1994.

[KM93] M. Kim and J. Moyne, *Multiple Input Multiple Output Linear Approximation Run-to-Run Control Algorithm -- User Manual ver 1.0*, The University of Michigan, Nov. 22, 1993.

[Tel94] R. Telfeyan, "Generic Cell Controller Module Interface Specification (GCCMIS) Version 0.3," Internal preliminary document, The University of Michigan, Aug. 1994.

[McI92] M. McIlrath, D. Troxel, M. Heytens, P. Penfield Jr., D. Boning, and R. Jayavant, "CAFE - The MIT Computer-Aided Fabrication Environment," *IEEE Transactions on Components, Hybrids, and Manufacturing Technology*, Vol. 15, No. 2, p. 353, May 1992.

[MM92] J. Moyne and L. McAfee, "A Generic Cell Controller for Automated VLSI Manufacturing Facility," *IEEE Transactions on Semiconductor Manufacturing*, Vol. 5, No. 2, pp. 379-390, May 1992.

[Ous94] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, Massachusetts, 1994.

[SHI91] E. Sachs, A. Hu, A. Ingolfsson, "Run by Run Process Control: Combining SPC and Feedback Control," *Submitted to IEEE Transactions on Semiconductor Manufacturing*, Oct 1991.

[Car94] J. Carney, "Implementation of a Layered Message Passing System," *Massachusetts Institute of Technology CIDM Memo Series*, Memo 94-15, Oct. 1994.

[Alt95] Arthur Altman, *Applying Run-By-Run Process Control to Chemical-Mechanical Polishing of Sub-Micron VLSI: A Technological and Economic Case Study*, MS Thesis Proposal, Management and Electrical Engineering and Computer Science Departments, MIT, Jan. 1995.

[Sze83] S. M. Sze, *VLSI Technology*, McGraw-Hill, New York, NY, 1983.

Appendix A

Chemical Mechanical Planarization (CMP)

As VLSI technology advances, the feature sizes of both the underlying devices and the underlying metal line widths decrease. With this decrease comes increased transistor speed and density, but also a need for more layers of metal interconnect. Thus interconnect technology is the center of much of today's VLSI research.

One of the major problems with fabricating additional layers of metal interconnect is that the

topography of the silicon wafer becomes increasingly non-planar as levels of metal are added. This coupled with the demand for increasingly smaller geometries has led to some problems previously unseen. First, due to the clarity of image needed for submicron geometries, the focal depth of lithography machines has decreased. This reduced focal depth results in some of the topography of the wafer being out of focus when other parts are in focus (see Figure 21). This is unacceptable as geometries shrink.

In addition to lithography concerns, the non-planar surface can lead to difficult processing as the aspect ratio of the valleys of the wafer become great enough that the interconnect metal is unable to fill and cover these areas. This effect can lead to circuit failure due to metal fatigue or lack of connection entirely. Figure 22 shows a typical non-planar process as well as an ideal one.

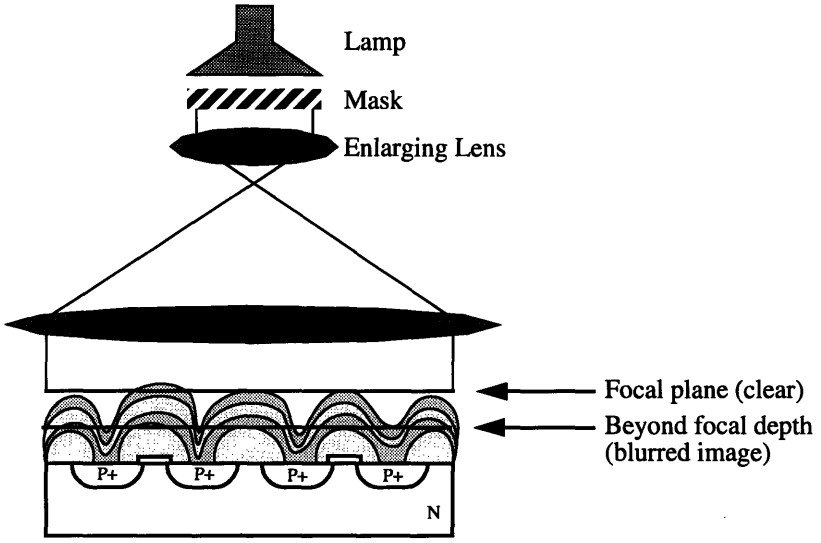


Figure 21: Enlarging lithography system

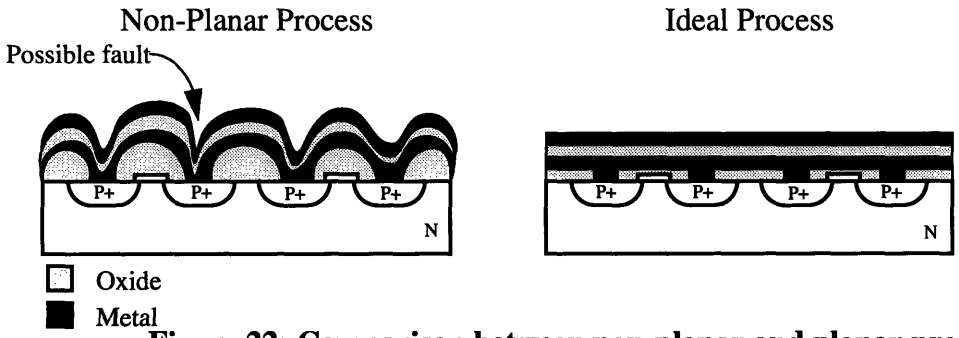


Figure 22: Comparison between non-planar and planar processes

There are many techniques used to increase planarity. Most involve applying a level of glass or oxide in an attempt to fill the valleys that can lead to trouble later on. The problem is that the peaks are also extended to some extent, so it is very difficult to achieve planarity through this process alone. The process of etching peaks from the dielectric has also been attempted; again this suffers from the inability to etch peaks while leaving valleys unchanged. CMP solves this problem by using a combination of chemical etching and physical abrasion to achieve global planarization.

CMP has its roots in the silicon wafer production machines used to polish the wafers before processing. These machines provided a wealth of information that led to the CMP machines of today. The basic process is the same for both. Wafers are loaded into a vacuum grip carrier which can rotate. This is then pressed against an abrasive pad which can also rotate (in the opposite direction). The lower pad is much bigger than the wafer, and is

continually coated with a chemical slurry by a nozzle. Figure 23 [Sze83] shows a schematic of a simple CMP machine.

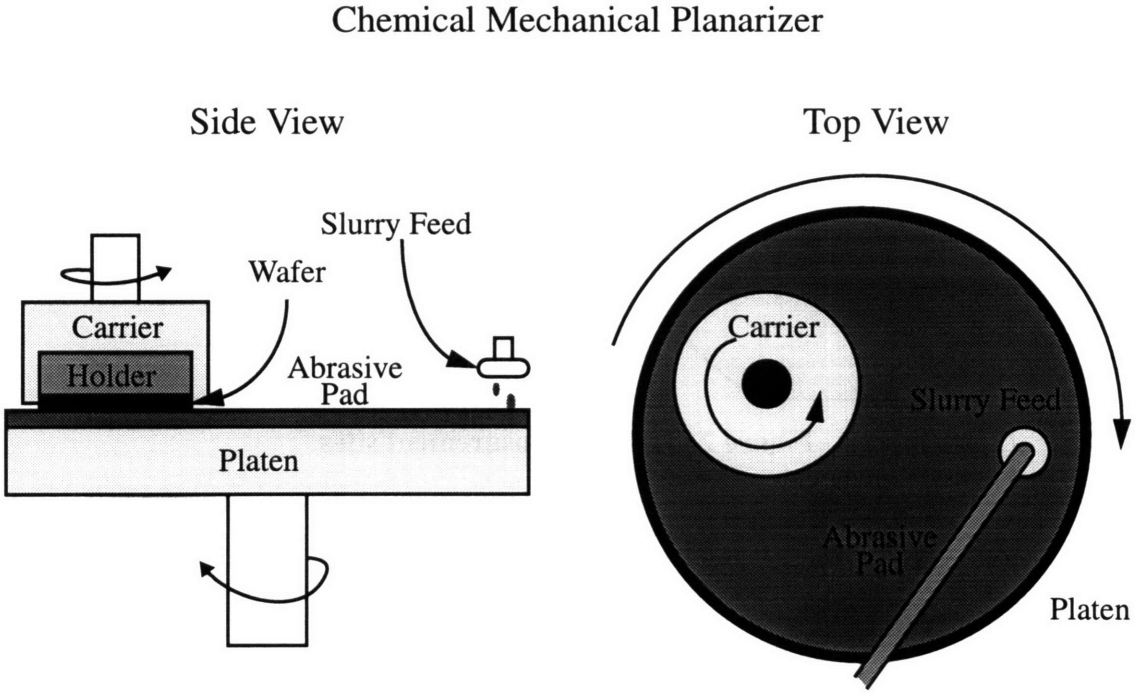


Figure 23: Schematic of a CMP machine

Through the use of CMP, near ideal planarization can be achieved. This has allowed VLSI manufacturers to increase the number of interconnect layers. It has also aided reliability by reducing the mechanical strain in metal lines resultant from non-planarity.

CMP is not without its flaws. In addition to its high cost, it has non-uniformity issues that are the center of much CMP research. Non-uniformity issues can arise both within a wafer, and between two wafers. Within wafer uniformity is measured by comparing the relative thicknesses of the wafers along various sited located radially from the center. The reason for this rather than a more uniform pattern is that CMP involves rotating the wafer which makes all sites that are radially equal the same thickness. Figure 24 shows the two methods of measurement.

Uniformity between wafers is measured by comparing the average thickness between two wafers. This measurement is related to the overall drift in a machine. This drift can have many sources, among them are pad wear, and changes in slurry composition.

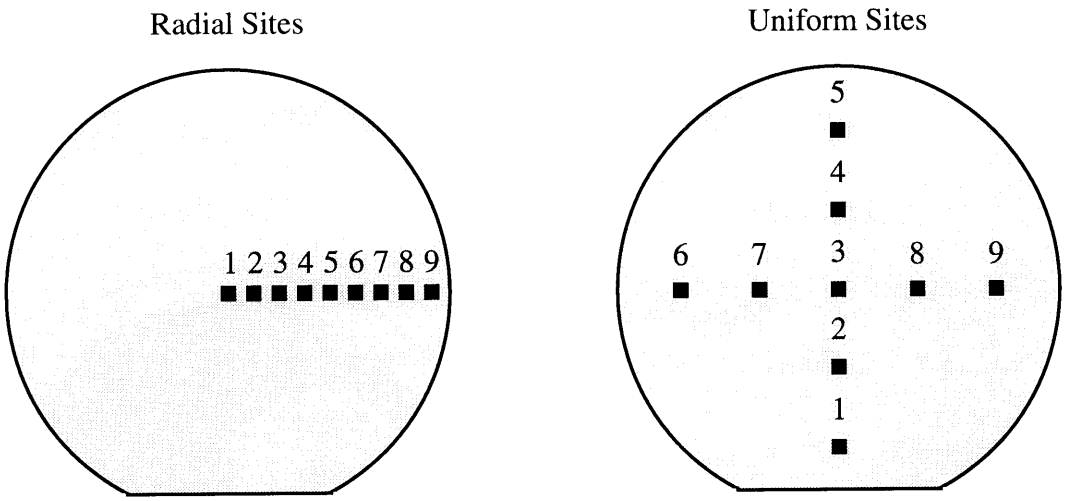


Figure 24: Wafer measurement sites

Appendix B

Run by Run User Interface and Simulation Environment Manual (CRUISE)

The following is a user's manual for the Run by Run Interface and Simulation Environment. It provides a brief overview of the main functional units of the interface as well as an example simulation. The graphics contained in the manual are actual screen shots using a screen capture program.

CIM Run by Run User Interface and Simulation Environment Manual (CRUISE)

version 2.1

February 3, 1995

William Moyne

1 Introduction

The CIM Run-by-Run User Interface and Simulation Environment (CRUISE) is a graphical tool that acts as both a user interface and a simulation test-bed for the Run by Run control server. The environment can act as a user interface to a Run by Run (RbR) Control Server in a stand-alone configuration. CRUISE is also designed to make both testing and simulation as easy and efficient as possible. As with any tool, an attempt has been made to make the environment easy to use while retaining both power and flexibility.

This manual is by no means meant to be a complete reference, but rather a quick guide to essential operations so that the user can feel confident to explore the environment further.

2 Overview of Manual

This manual provides the following:
- Explanation of various fields,
- Overview of options present on the menu bar,
- Various figures illustrating actual operation.,
- A complete simulation example,

3 Explanation of Fields:

CRUISE can be thought of as a collection of panels and a menu bar. The panels serve as both an input and output device for data. The menu bar provides a structured way to access the functions present in the environment. Figure 1 shows CRUISE along with labels denoting the panels and menu.

The main features are:

- Menu Bar ....Used to access most of the environment's functions. (Discussed below.)
- Model .....Displays current experimental model for controller.
- Weights .....Displays and sets the value control algorithm weights.
- Recipe .....Displays current suggested recipe.
- Initial Conditions ....Displays the initial state of the process.
- History.....Contains a tabulated summary of previous results.

Explanation of the panels will be left to the example simulation, the reason being that these tables are mainly for display and storage of simulation data. They also play a key roll when generating a new simulation model. For these reasons an example simulation will provide the most useful information.

The following section describes the various capabilities available through the menu bar.

4 Menu Bar

The Menu Bar (Figure 2) has 6 basic submenus:
- File
- View

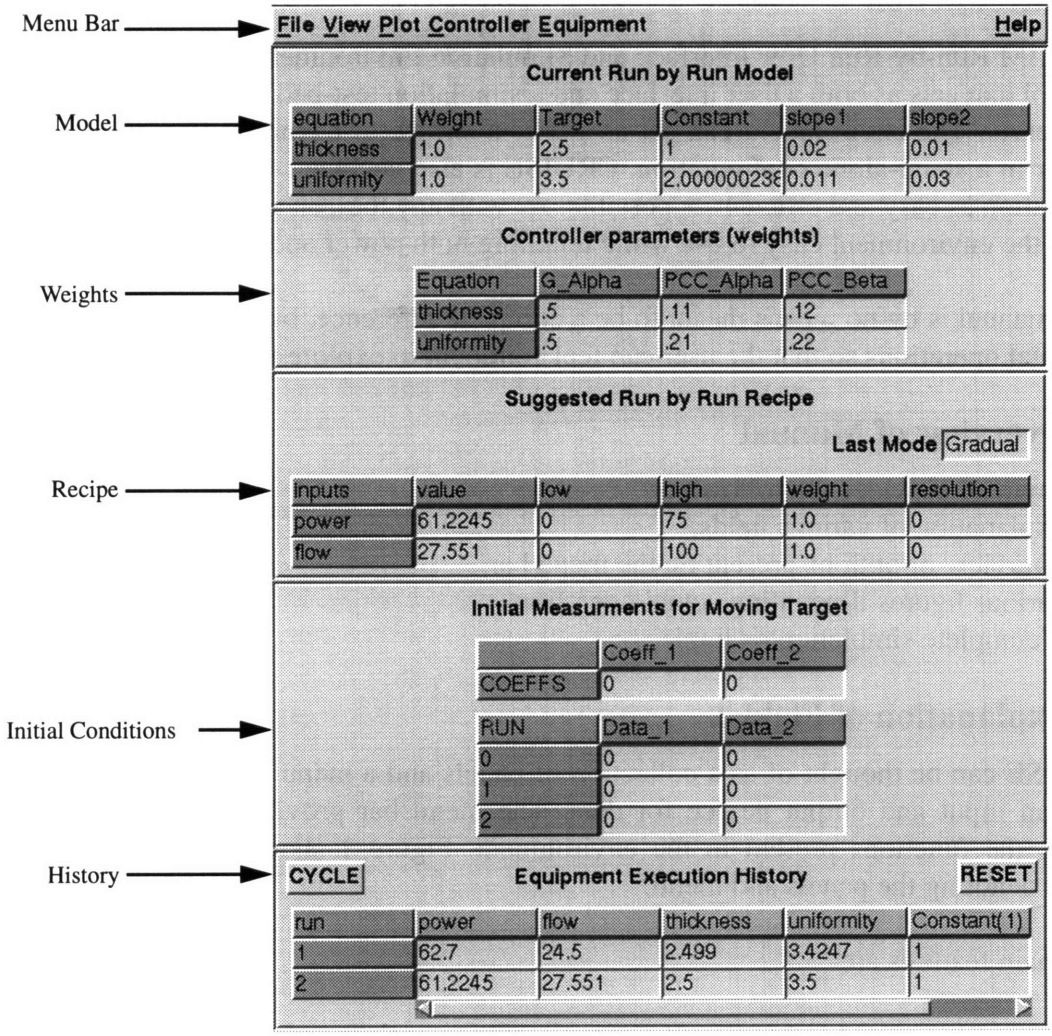


Figure 1: RbR Simulation Environment

- Plot
- Controller
- Equipment
- Help

Figure 2: Menu Bar

4.1 File

The *File* menu (Figure 3) as the name implies takes care of saving and retrieving informa-

tion from disk.

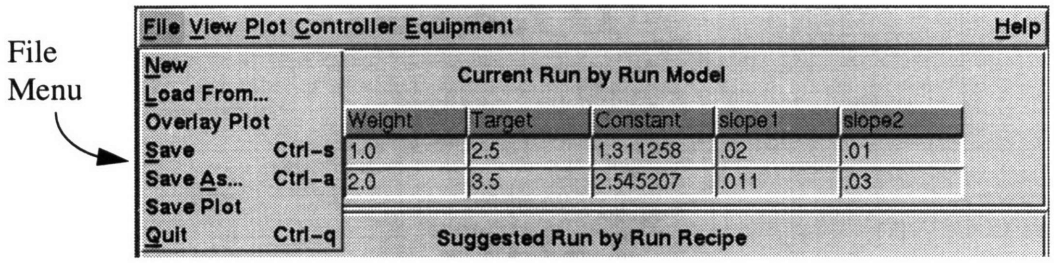


Figure 3: File Menu

4.1.1 New

Creates a new model for simulation with a given number of inputs and outputs. Figure 4 shows the new model configuration window.

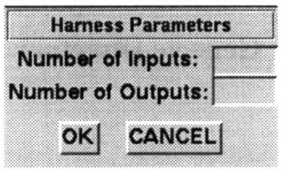


Figure 4: New Model Window

Once the size of the new model has been set. An empty model is generated that can be configured as desired.

4.1.2 Load From:

Retrieves from disk a previously configured model. A directory browser (Figure 4) is provided to aid in finding files.

4.1.3 Overlay Plot:

Retrieves previously saved plot data. This data may then be plotted together with the current data. This is useful when comparison of algorithms is done.

4.1.4 Save:

Saves the current state of the environment to the current model name.

4.1.5 Save As:

Similar to *Save*, but allows a new file name to be specified.

4.1.6 Save Plot:

Stores data from the current history of runs so that it can be accessed later using the *Overlay Plot* menu item.

4.1.7 Quit:

Exit immediately without saving model or history information.

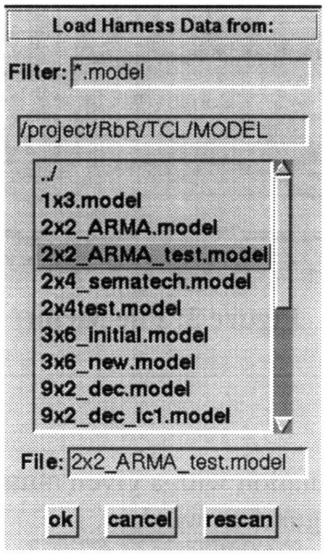


Figure 5: Generic Directory Browser

4.2 View:

The *View* menu (Figure 6) provides a convenient way to manage the representation of data in the environment.

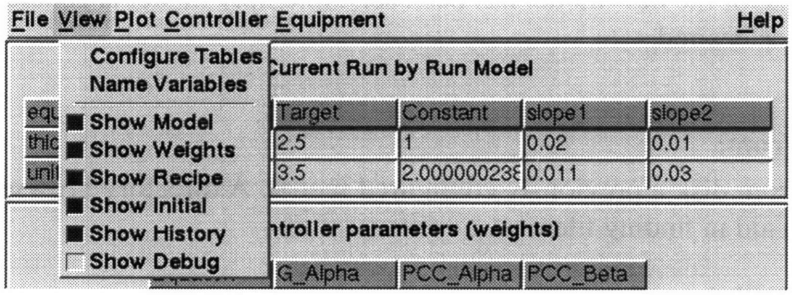


Figure 6: View Pulldown Menu

4.2.1 Configure Tables:

Lets the user control the dimension of the display tables. Figure 7 shows the three parameters:

- Maximum Rows .....Maximum row cells before adding a scrollbar
- Maximum Cols .....Maximum columns before adding a scrollbar
- Column Width.....Default width of table columns.

4.2.2 Name Variables:

Allows the user to change the default input and output variable names. Figure 8 shows the entry format. The old names are then updated automatically.

In addition to the table formatting features, the *View* menu also controls which panels are displayed. The *on* toggle buttons represent panels that are actively displayed. Panels can

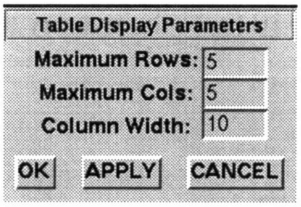


Figure 7: Configure Tables

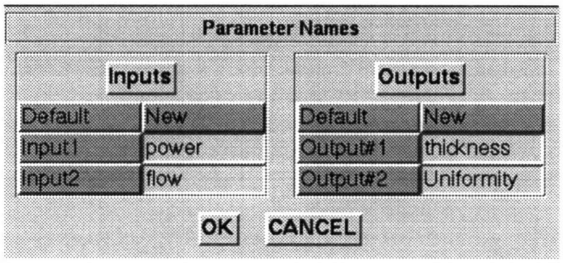


Figure 8: Name Variables

be toggled on and off by clicking on the square preceding their name. Note: The debug window is for debugging purposes only, and will not be present in the final released version.

4.3 Plot:

The *plot* menu (Figure 9) provides a convenient way to view the data stored in the environment graphically.

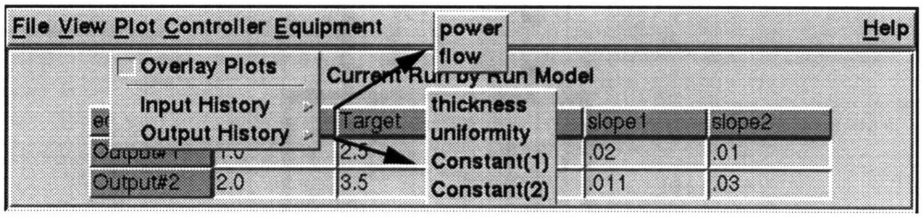


Figure 9: Plot Pulldown Menu

The options currently supported are: *Overlay Plots*, *Input History*, *Output History.*

4.3.1 Overlay Plot:

This option lets the user decide whether or not to overlay data that has been loaded in using the *File/Overlay Plot* command. When selected, the previous data are plotted with the corresponding new data.

4.3.2 Input History:

This item is actually another submenu that displays all the current input variables. As shown in Figure 9, *Input1* and *Input2* are the two input variables for the given scenario. Selecting one of these would produce the appropriate plot. Example input vari-

able plots are shown in Figure 10 (note the overlay data in the second plot):

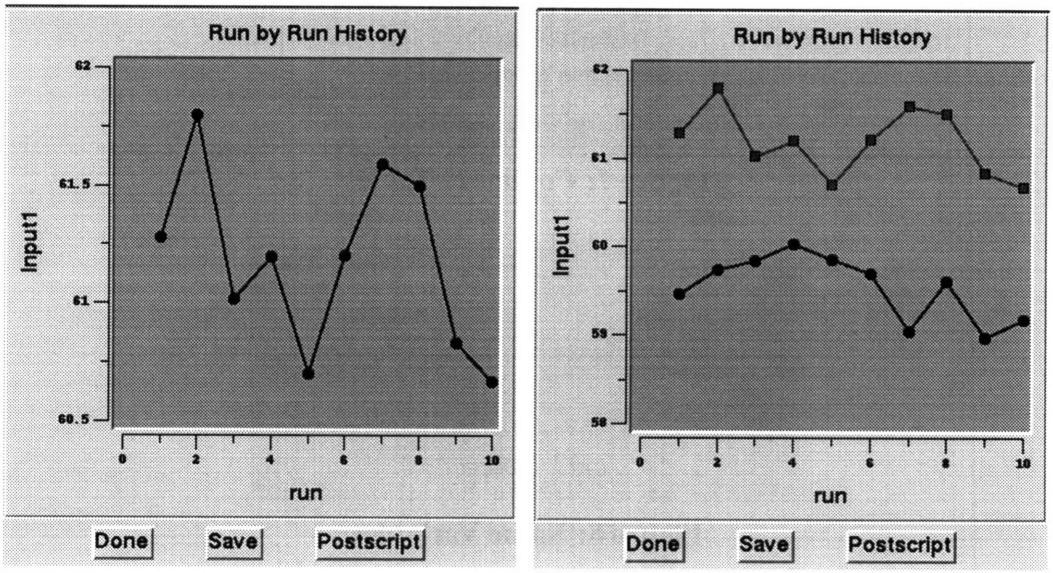


Figure 10: Input Variable Plots

4.3.3 Output history:

The system output can also be plotted, as shown in Figure 11 below:

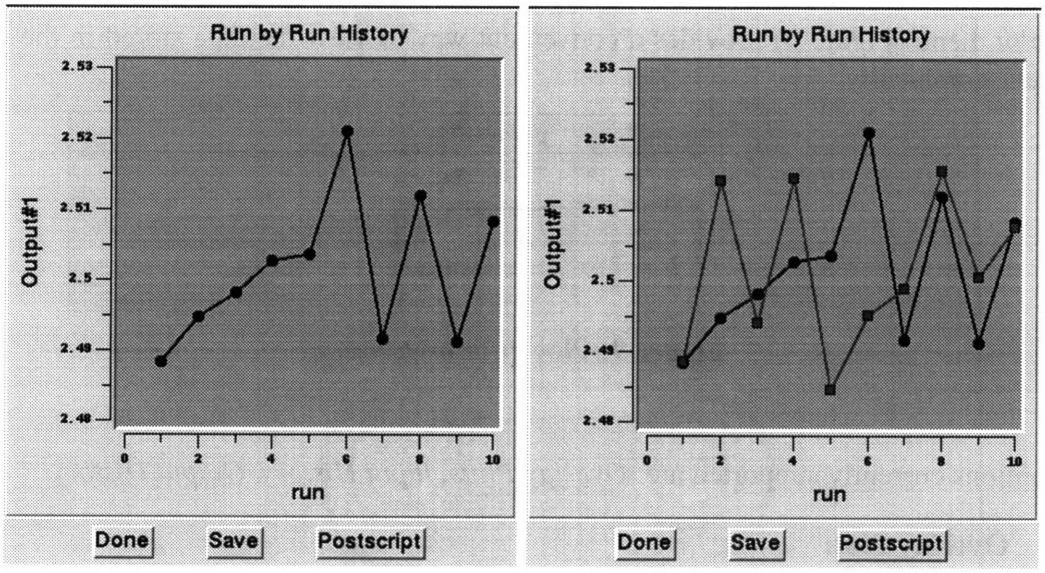


Figure 11: Output Variable Plots

Both plot types provide a menu bar (Figure 12) with the following functions:

- *Done*Dismiss graph
- *Save*Save graph data to a text file. (Useful for importing into Matlab™.)
- *Postscript*tGenerate postscript file for printing purposes.

Future plot tools will include SPC sigma guide lines, and various other statistical informa-

tion. The current implementation is for algorithm testing only.

Figure 12: Plot Menu Bar

4.4 Controller:

The *controller* menu (figure 13) provides an easy way to specify the various parameters of the control algorithms, along with some related functions. When a new algorithm is implemented, fields containing the algorithm's parameters are added to this list. Currently *Gradual*, and *PCC* modes are available.

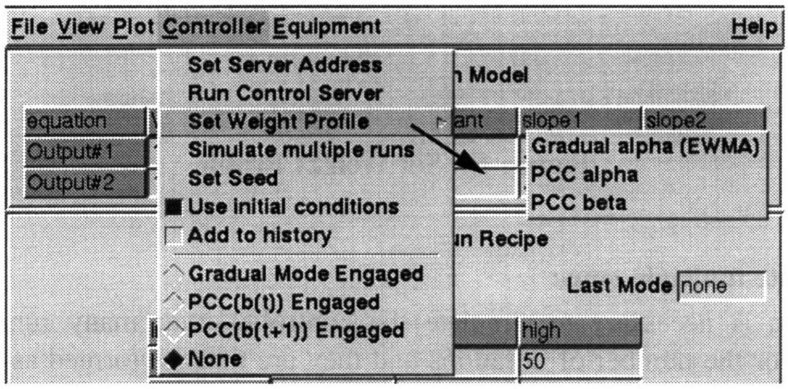


Figure 13: Controller Pulldown

4.4.1 Run Control Server:

This sends the RbR Control Server the current state of the process and requests new recipe information.

4.4.2 Set Weight Profile:

This option is a submenu that allows the user to set the parameters for the *Gradual* and *PCC* algorithms. These parameters, or weights, are used in EWMA (Exponentially Weighted Moving Average) filters present in both the *Gradual* and *PCC* controllers.

Note: The *PCC* algorithm has two EWMA filter parameters:

- *Alpha* is used to weight the noise term which is similar to the gradual controller.
- *Beta* is used to weight drift estimation.

To aid in visualizing the effect of various weights, a graphing function has been added to this option (Figure 14). Note that a weight of 100 (100%) represents 100% update on current data, while a setting of 1% relies heavily on past data. This graphically shows the exponentially decaying nature of the EWMA filter. As with most aspects of the environment, other weighing techniques could be easily added and displayed in a similar manner.

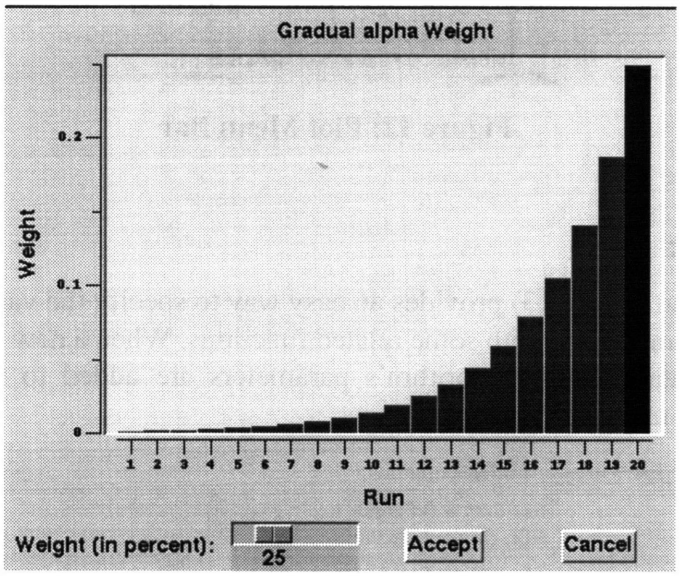


Figure 14: Set Weight Profile

4.4.3 Simulate multiple runs:

This option is necessary to simulate the controller over many runs. The user is prompted for the number of iterations and they are then performed as a batch operation.

The way the new data is added to the history is controlled by the *Add to history* option discussed below.

4.4.4 Set Seed:

It is often necessary when comparing two algorithms to test them under exactly the same conditions. This option allows the user to seed the random numbers used in the generation of noise so that multiple algorithms are tested with exactly the same data.

4.4.5 Use Initial Conditions:

This toggle controls the use of initial conditions in the model. If initial conditions are activated, the initial condition panel (see Figure 1) is used to set initial conditions and weightings of these conditions.

4.4.6 Add to history:

This toggle sets whether the environment should delete old data in the database upon arrival of new data, or add to this old data. This is used in conjunction with the *Simulate multiple runs* command.

4.4.7 Gradual, PCC(b(t)), PCC(b(t+1)) Mode Engaged:

These radio buttons represent possible algorithms that may be used for control. The reason for the radio button (radio buttons only allow 1 selection to be chosen from the

list) is that these particular routines are similar in function and cannot be used together.

4.4.8 None:

Sometimes it is useful to examine a system in "open loop," not providing any control. This option allows this.

4.5 Equipment:

The *equipment* menu (Figure 15) controls the model portion of the environment. This model is what is used to simulate the actual machine being controlled.

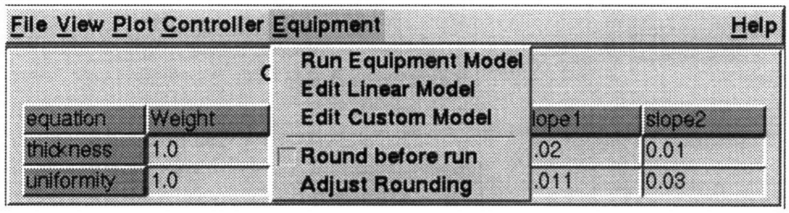


Figure 15: Equipment Pulldown Menu

Although the current implementation uses an internal model, the system has been designed with enough modularity so that as external models become available they can be seamlessly integrated into the environment.

4.5.1 Run Equipment Model:

This item is the complement to the *run control server* discussed earlier. When activated, the current control recipe is fed into the model, and the resulting data is stored in the history table.

4.5.2 Edit Linear Model:

This selection brings up the *Equipment simulator* (Figure 16) which allows the operator to control the various model coefficients and noise terms used for simulation.

Changing the parameters of the simulation model allows the user to have separate algorithm and simulation coefficients. This can help test the effect of an incorrect machine characterization.

The RbR environment also supports a variety of noise types. Gaussian noise can be added with a given mean and standard deviation. Drift can also be added to simulate machine aging and other time dependant events. The drift term may also have a standard deviation which represents uncertainty in the drift coefficient.

Along with random Gaussian distributed noise, the simulator also supports more complex autoregressive moving average (ARMA) noise. This is often more realistic due to its time dependent nature. This noise takes on the following form:

where:

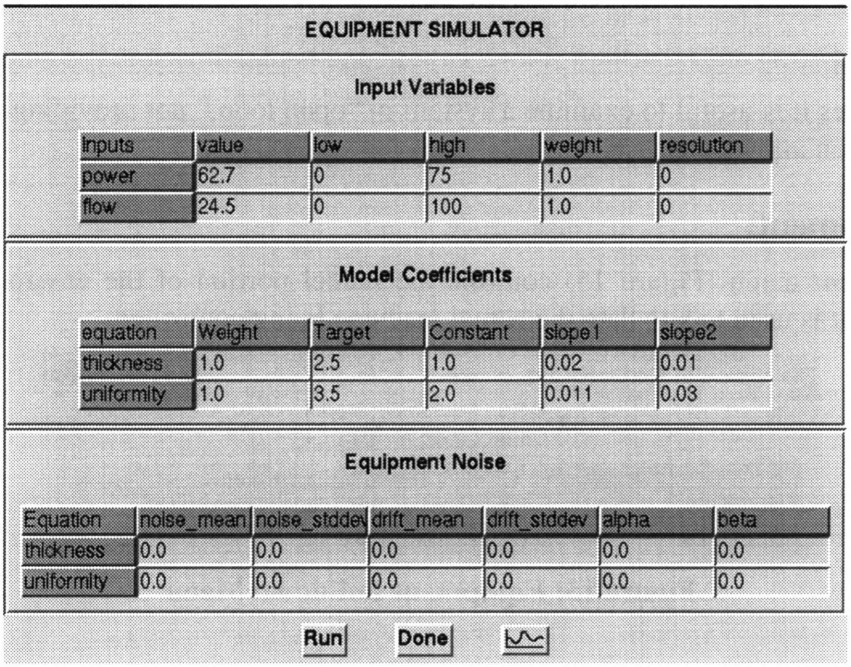


Figure 16: Equipment Simulator

$$N_t = \alpha N_{t-1} + w_t + \beta w_{t-1}$$

N_t = total noise for run t,

w_t = white noise for run t,

α, β= noise model coefficients.

To aid in visualizing the noise generator, a sample noise distribution over time can be viewed by clicking on the graph button on the tool bar of the equipment simulator. Figure 17 shows an ARMA example.

This plot is also dynamically updated as the α and β values of the model are changed. In this way the user can get a better feel for what various constants will do to the over all shape of the noise terms. The graph also has a *select* button which is used to select which output to view in a multi-output problem.

4.5.3 Edit Custom Model

Often it is not enough to simulate a real machine with a purely linear model. This option allows the user to construct higher order models so that the simulator can more accurately mimic actual machine behavior. This does not, however, mean that the controller can use these higher-order models. The controller will try to react using its linear model. Figure 18 shows the custom equation editor. The format of the equations is text based.

The *Custom* and *Enabled* pull-down menus allow the user to do a variety of operations. One of the most useful options is *check equation*. This will test the current formula against the current data and return a result or error depending on the result. This

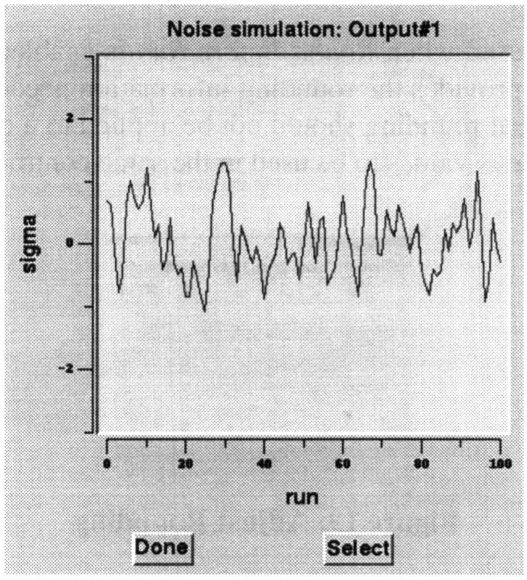


Figure 17: Noise Simulation Display

helps avoid syntax errors while entering formula. This function is automatically called when the user attempts to *save* a formula to avoid simulation problems in the future.

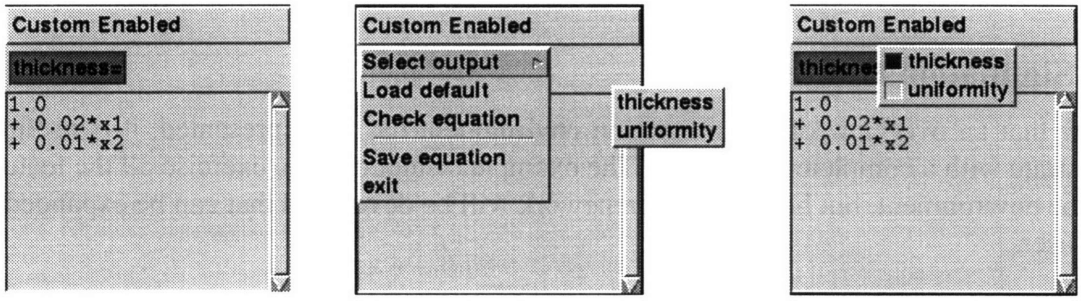


Figure 18: Custom Model Editor

4.5.4 Round Before Run

An important difference between theoretical simulations and real experiments is that in a real experiment, control over a process is often limited to a discrete set of values, while in simulation, these values or "knobs" have infinite resolution. This can lead to inconsistent behavior between simulation and actual operation. The *Round Before Run* toggle has been added to remedy this. When enabled, this toggle rounds all recipe suggests to a set precision. These values are then used in the next run of the process, thus simulating a discrete process.

The controller itself supports an internal type of recipe rounding, but this function attempts to suggest recipes that are both correctly rounded and provide an optimal output. The toggle simply rounds recipes that have already been suggested and does not attempt to generate an optimal solution using these values.

4.5.5 Adjust Rounding

The degree of precision used when *Round Before Run* is enabled is set by this option. A small tabular entry form provides the rounding information needed for each of the inputs. A value of zero signals that rounding should not be applied to a certain input. This allows both discrete and continuous values to be used in the same control action. Figure 19 shows the input table.

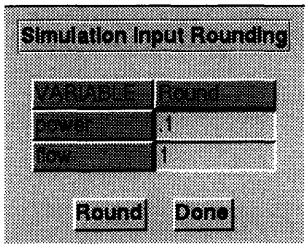


Figure 19: Adjust Rounding

4.6 Help:

The current implementation of the simulator does not provide on-line help.

5 Simulation Example

Now that an overview of the simulation environment has been presented, it is useful to illustrate with a complete simulation. The example cannot hope to exercise all the features of the environment, but hopefully a framework will be developed that can be expanded by the user.

First, an explanation of our problem:

We wish to simulate control of a new furnace we bought (or an old one!) that based on a design of experiments has the following parameters:

$$Thickness = Temp \cdot 0.7 + Time \cdot 3.0 + InitialThickness \cdot 0.5 + 150.0$$
$$Uniformity = Temp \cdot 0.08 + Time \cdot 0.1 + InitialUniformity \cdot 0.19 + 10.0$$

For a quick simulation, we set the variables as follows:

Thickness=	Thickness
Uniformity=	Uniformity
Temp=	Temp
Time=	Time
InitialThickness=	Initial condition (Not a control parameter!)
InitialUniformity=	Initial condition (Not a control parameter!)

Let us also assume the allowable ranges and optimal outputs are as follows:

Thickness=	1000.0

Uniformity= 100.0
700 < *Temp* < 1200(Degrees Celsius)
25 < *Time* < 100(Minutes)

Another assumption is that we do not want to sacrifice *thickness* for *uniformity* (we want the emphasis on thickness to be 4 times that of uniformity), so we weight the equations as follows:
$W_{Thickness}$= 4.0
$W_{Uniformity}$= 1.0

In order to start the simulation, we need an initial guess for the recipe. This is usually the operating point around which we linearized to get the equations for the model:
Temp= 950
Time= 45
InitialThickness= 100
InitialUniformity= 50
Now that we have formulated the problem, we are ready to begin.

5.1 Creating a New Model

The first step after formulation is to create a new simulation based on the problem. This can be done by selecting **File->New**. This will allow us to enter the dimensions of the problem (see Figure 4). For this problem:
Number of Inputs = 2
Number of Outputs = 2

Noise was also added to the system. Simple white noise was used as a starting point. More complex ARMA type noise could be added later once the simulation was well understood.

We now enter the various bits of data we have into the appropriate fields. (See Figure 20) Since we are using initial conditions, they must be enabled using the *Controller* menu. See section 4.4.5 for more information.

5.2 Simulation

Once the data has been entered into the environment we can begin simulation. There are many ways to construct a simulation, each based on different requirements and expectations. We will perform two basic tests:
- Initial condition test
- Noise test

These tests are designed to exercise some of the features of the environment, and to display what is considered "normal" behavior. The user should be able to start with these

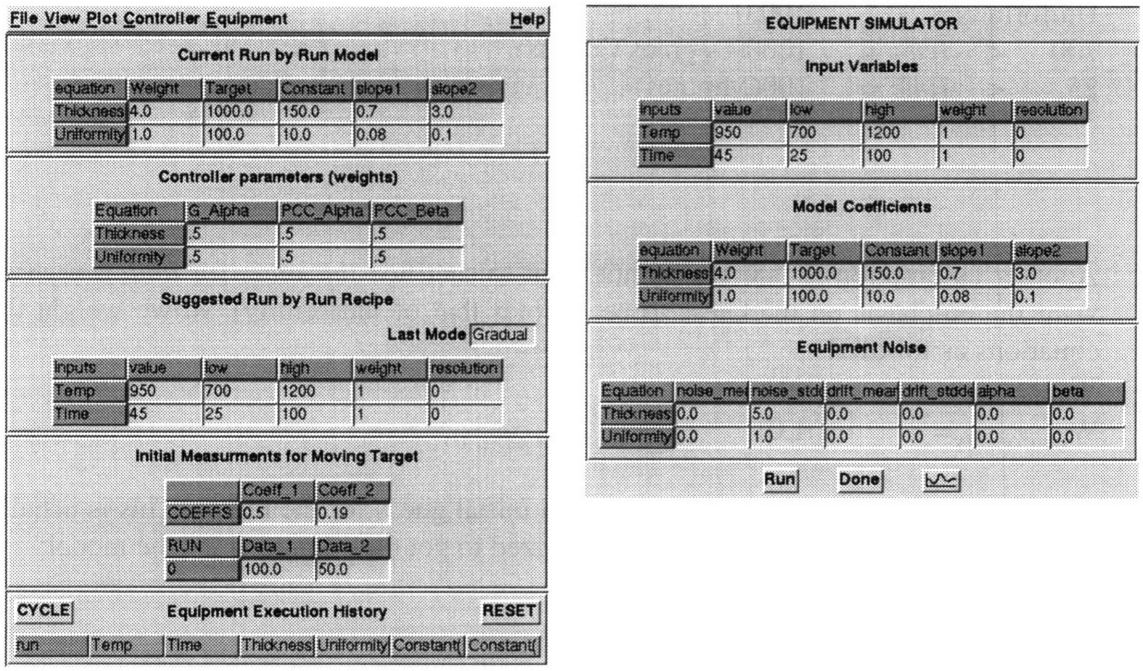


Figure 20: Simulation: Main Panels, Equipment Simulator

basic results and construct much more complex simulations that provide relevant information about a process.

These tests were performed using the *CYCLE* button on the front panel (see Figure 1). This button has two distinct modes of operation based on the state of the initial conditions. If initial conditions are disabled (See section 4.4.5) then the button performs a simulation using the current model, then queries the RbR Control Server for recipe suggestions. This represents one complete cycle of operation. The user can then cycle again.

The second mode of operation is used for initial conditions. Initial conditions require special attention because the simulator does not generate them directly. It must rely on the user to enter relevant initial conditions before it can complete a simulated run. To facilitate this, the *CYCLE* button is used to perform a two-part procedure. First, the user presses *CYCLE* and is given the results of the simulator (no control yet) and a blank row is added to the initial condition table. The user is then required to enter initial condition data in this row for this run. The *CYCLE* button can then be pressed again to complete the run.

The dual nature of the *CYCLE* button allows consistent operation of both initial and non-initial condition runs, and also helps avoid erroneous data created by simulation with insufficient data.

Noise for these simulations was added via the *Equipment Simulator* (See section 4.5.2). In addition, when comparisons between competing models was required, the *Set Seed* command (See section 4.4.4) was used. This ensured that two simulations could be performed at different times and that the noise generated for each was guaranteed to be equivalent,

hence allowing comparison.

5.2.1 Initial Condition Test

The first test is designed to test the treatment of initial conditions. It is essential that the user be both familiar and comfortable with these results so that controller response can be understood, and proper operation is ensured.

In order to isolate the initial conditions themselves, we eliminate all noise from our problem formulation. Noise will be tested later, so we will concentrate on the response to various initial conditions. The desired result is this: *"Given various initial conditions, the controller should suggest recipes that ensure the equations meet target. In addition, the controller SHOULD NOT change any model parameters. These parameters should only be affected by noise."*

We will vary the initial conditions from their initial values, to various other values, then back. The desired result is a range of recipes with no major changes in any other values. Figure 21 shows the results of the simulation.

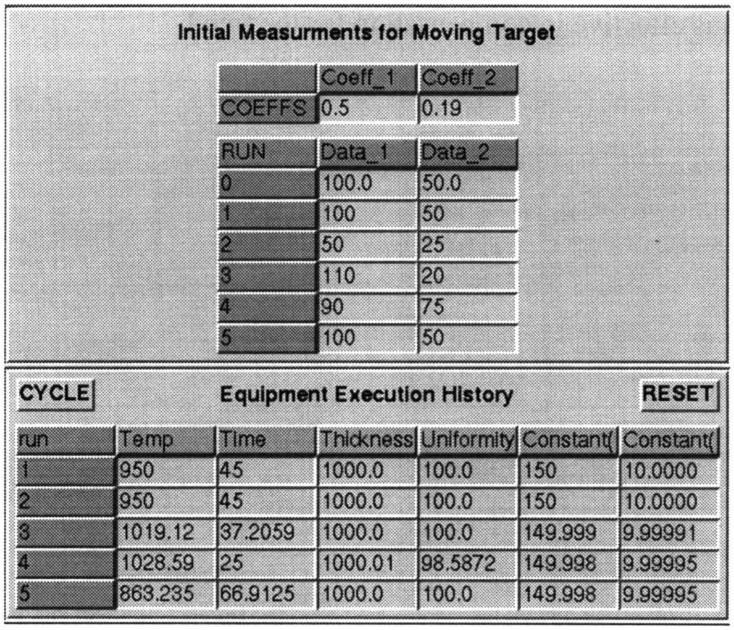


Figure 21: Simulation to Check Initial Conditions

Notice that the controller performed as expected, changing the recipe terms without changing the constants. Also notice **run 4** displays what happens when a parameter limit is met. Input2 cannot be set lower than 25, so the controller sets it to 25 and re-optimizes. The output is also effected, not being able to hit target with the added constraint.

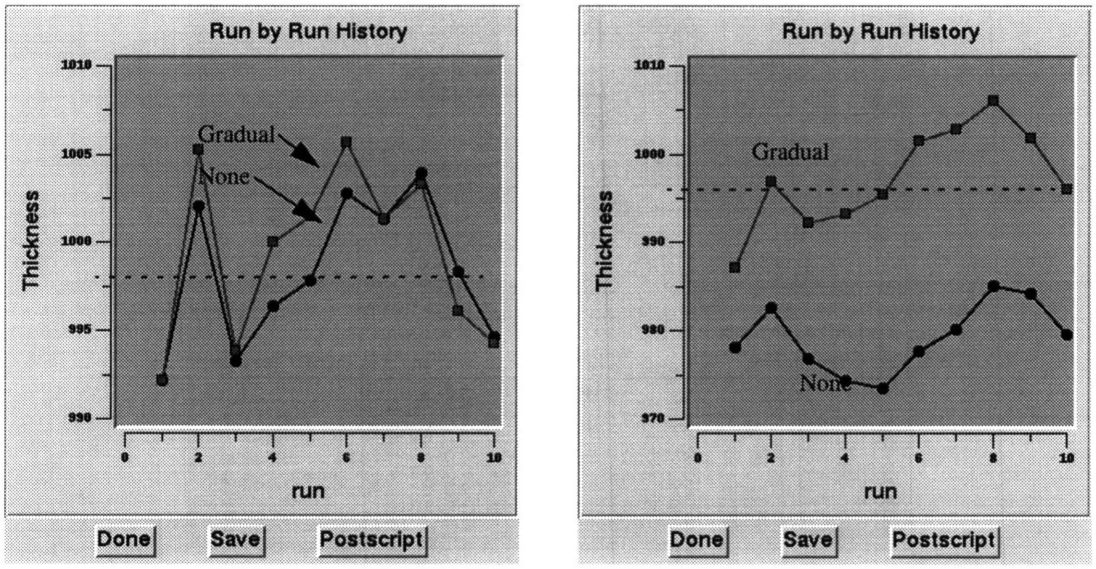
5.2.2 Noise Test

Once we have verified that the initial conditions work as expected, we can perform an

actual simulation with noise. Noise is an important ingredient to any simulation because it represents a fundamental phenomena of a real process. For the sake of simplicity, white noise will be used. The application of more complex noise models and drift can be applied in a similar manner once the simulation process is understood.

As before, it is essential that we understand the desired operation of the controller. In this simulation the desired result is this: *"Given a system with noise, the controller should suggest changes to the constant term in the model and generate recipes that allow the system to reach target when possible."*

Figure 22a shows the graphic results of the simulation. There seems to a problem though. The non-controlled case seems to be at least as good if not better than the gradual mode controller. This would lead us to believe that no control is the best course of action and in this context it may be. In contrast Figure 22b shows the results of a similar simulation but with the addition of an ARMA α weighting of 0.95 (See section 4.5.2). This addition makes the noise much more likely to exhibit trends rather than the purely random nature exhibited in the white-noise case. In such a scenario the gradual mode is effective in compensating for the trend.



A B

Figure 22: Output of Noise Simulation

5.3 Notes on Simulation

There is no right or wrong way to perform a simulation. The environment is designed to allow the user as much freedom as possible without being washed out with options. There are a few functions/styles that are useful and are worth mentioning again here. They will be briefly described below.

Possibly the most useful aid to simulation is the Save command (See section 4.1.4). There

is no limit (other than disk space) to the number of times a model/state can be saved. This is very helpful for "what if" type scenarios. The user can save the state of the controller, try something, note results, re-load the previous state, and try something else. This was used in the noise simulation above.

Another small but very useful function is the Set Seed command (See section 4.4.4). This is essential when noise is used in a simulation. Due to the nature of computer generated random numbers, given a seed, the computer will produce the same stream of random numbers for that seed every time. This allows multiple simulations to be subjected to exactly the same noise.

The environment also supports plotting any input or output to the system. This allows the user to view not only the performance of the system (via the output variables) but to also examine the input parameters to see if there is some useful trend or unwanted operation.

6 Conclusion

As with any overview, many things have been left out, and many questions remain. It is the intent of this document to provide enough background to get the user going with CRUISE.

Once familiar with the basics, the user will quickly be able to test control algorithms in an easy to use and flexible environment.