10/3-1-96 DSD

SANDIA REPORT

SAND96-8214 • UC-405
Unlimited Release
Printed February 1996

Process Control of Large-Scale Finite Element Simulation Software

Paul Spence, Larry Weingarten, Kevin Schroder, Dave Tung, Don Sheaffer

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94551
for the United States Department of Energy
under Contract DE-AC04-94AL85000

SF2900Q(8-81)

35

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

SAND96-8214 • UC-405
Unlimited Release
Printer February 1996

Process Control of Large-Scale Finite Element Simulation Software

P. A. Spence
Thermal and Plasma Processes Department

L. I. Weingarten
Structural and Thermomechanical Modeling Department

K. Schroder
Exploratory Systems Department

D. M. Tung and D. A. Sheaffer
Telemetry and Digital Signal Processing Department
Sandia National Laboratories
Livermore, California 94551-0969

ABSTRACT

We have developed and demonstrated a methodology for coupling finite element software with process control strategies. We have linked controllers to both the Sandia-developed TACO thermal analysis code and the commercially available ABAQUS thermal-mechanical analysis code. Communication between the physically-based simulations and the process controllers was achieved using subroutine calls and UNIX system calls. The ability to couple control design software with large-scale analysis software creates a virtual environment for the concurrent design and testing of closed-loop systems. We have applied this coupled software to the development of a rapid-thermal-processing (RTP) chamber and its controller. The controller was designed using data from a detailed finite element thermal model of the RTP system. We have evaluated both the hardware design (i.e., sensor placement, lamp housing, and wafer support) and the controller design (linear quadratic Gaussian) using closed-loop simulations.

ACKNOWLEDGMENT

The authors wish to thank Chuck Schaper of Microelectronics Control and Sensing, Inc. (MCSI) and Curtis Elia, John Ebert, and Abbas Emami-Naeini of Integrated Systems, Inc. (ISI) for their contributions in controller design.

This work was funded through the LDRD program.

CONTENTS

LIST OF FIGURES

Process Control of Large-Scale Finite Element Simulation Software

1 INTRODUCTION

Application of physically-based computational models to the development of advanced manufacturing equipment can lead to improved system performance while reducing the development time. Typical computational simulation software is designed to run in an open-loop mode, however, many manufacturing processes are operated under closed-loop (feedback) control. To enhance our ability to simulate these closed-loop processes, we have developed a methodology for driving our general-purpose finite element codes with feedback control algorithms. We have demonstrated this approach using two independent finite element analysis codes: the Sandia-developed finite element thermal analysis code TACO [1], and the commercially available thermal-mechanical finite element code ABAQUS [2]. Control design for this work was done using both the MATRIXx control design software by Integrated Systems, Inc. [3] and the MATLAB/SIMULINK control design software by The Math Works, Inc. [4].

The link between the analysis codes and the control software required the development of input/output (I/O) interfaces for data transfer. Our approach differed in how the interfaces were implemented since we possessed the source code for one analysis package (TACO) and did not for the other (ABAQUS). We were able to successfully link both analysis codes with controllers; however, we found that having the source code available provided a greater flexibility in what parameters we were able to control.

We have developed and tested two communication techniques for coupling controllers with the analysis codes. The first (and simplest) technique is to compile the control algorithm as a subroutine (either FORTRAN or C) then link it with the analysis code. This approach is facilitated by automatic source code generation capabilities available with both the MATLAB and MATRIXx control development software packages. The second communication technique allows both the analysis code and the control-development code to run as independent processes using UNIX system calls (sockets) to pass data between them. The advantage of this approach is that the independent processes can run on different computing platforms (e.g., a CRAY and a SUN workstation) and the full power of the controller-development software is available for interactive modifications and evaluation.

We have applied this closed-loop simulation capability to two different models. A rapid-thermal-processing reactor (which is used in a number of processing steps to fabricate semiconductor integrated circuits) was modeled using the TACO thermal analysis code. A controller was developed and linked to the TACO simulation to regulate the power to lamp heaters such that the temperature of the silicon wafer followed a prescribed trajectory. A second model was developed of a simplified diamond chemical-vapor-deposition (CVD) reactor using the ABAQUS thermal-mechanical code. For this model, a controller was developed and linked to the ABAQUS simulation to control the temperature of a substrate by regulating the incident heat flux.

2 METHODOLOGY

Closed-loop analysis requires the development of both a system model and a process controller. The system model must include actuators that can be regulated by the controller and sensors to supply control feedback. It may not be necessary to model the physical sensors, however, the system model must be capable of predicting the state of parameters that are to be measured and controlled in the actual system. If a process controller does not exist for the system, then it can be designed using data from the model. Closed-loop simulations are performed by coupling the system model and the control algorithm such that the model runs in a transient mode passing data to and from the controller at each sample time. The sample rate for the process is determined prior to the design of the controller. The simulation and controller must operate with the same clock. At each sample time, the model predicts the value of sensed parameters (e.g., temperature) and sends that information to the controller. The controller receives the sensor information and updates the actuator values (e.g., heater power) which are returned to the model. The model then continues marching in time (using the updated actuator values) until the next sample rate is reached.

2.1 Software Link Between TACO and Control Application

To create a general link between the TACO finite element code and arbitrary process control algorithms, modifications were made to the TACO source code. A TACO interface subroutine was added that identifies the "process sensors" and updates the actuators (controllable parameters). For the RTP simulation, the sensor output is represented by the computed temperature at five finite-element-mesh nodes located on the wafer and the actuators are the power inputs to each of the five lamp zones. From within the interface subroutine, a call is made either directly to a control subroutine or to a communication subroutine that opens a socket for data transfer with the control development software. The current value of each of the "process sensors" in the simulation is passed to the controller and new values for the actuators are sent back. The actuator array is updated by the interface subroutine and returned to the main TACO code. An example of an LQG controller subroutine is listed in Appendix A. The socket communication subroutines are listed in Appendix B.

All variable parameters supported by TACO have a curve number associated with them in the TACO input file. To flag a parameter to be regulated by the controller, the associated

curve number in the input file is set to a value less than or equal to -800. TACO parameters that can be controlled include material properties (heat generation, thermal conductivity and specific heat capacity) and boundary conditions (convective heat-transfer coefficient, ambient air temperature, surface emissivity, contact resistance, temperature, and heat flux).

Modifications were also made to the TACO time-step algorithm. The closed-loop simulations are performed with a predetermined sample rate. The sample rate determines the frequency that the controller updates the actuator values. The new time-step algorithm provides variable time-step control while insuring that no sample time is overstepped. A simple method to insure that the simulation time steps coincide with the controller sample rate would be to use a fixed (constant) time step set equal to the controller sample interval. In the event that large changes in the actuator values are required, however, the fixed time step can result in loss of computational accuracy. The variable time step approach determines the necessary time step size to maintain computational accuracy. Large parameter changes during the simulation may result in several computational time steps between calls to the control interface subroutine to update actuator values. The sample rate for the process simulation is entered as the maximum allowable time step value (DTMAX) in the TACO input file.

2.2 Software Link Between ABAQUS and Control Application

It is important to show that the techniques used to link TACO with process control algorithms can be extended to commercially available general purpose finite element codes. These are codes which have worldwide use, but, in general, users have no access to the main source code. Due to their large customer base, usage of process control in these codes has greater potential. To demonstrate the technique, the ABAQUS commercial code was utilized. Recent work by researchers at the Massachusetts Institute of Technology also utilize ABAQUS coupled with control algorithms to simulate a metal-forming process [5,6].

Even though users normally do not have access to the source code of the program delivered to customers, most commercial codes provide the capability of allowing the use of a predefined set of user subroutines. In the case of ABAQUS, these subroutines give the user many capabilities. Examples are user-defined friction algorithms, non-uniform flux distributions, new elements, and new material constitutive models. The procedure to implement process control in ABAQUS is in many ways similar to that outlined above for

TACO. This involves writing two user subroutines in addition to the control subroutine. One of these subroutines, UVARM, obtains the "process sensor" variable after each increment and provides it to an actuator subroutine. The variables read in UVARM can be any of the element integration point variables. These include variables such as temperature and strain. The actuator subroutine, usually one which can modify the model loading, receives this variable and in turn provides it to the control subroutine. Based on the value of this variable, the control subroutine returns a modifier for the actuator or loading. Examples of the ABAQUS user subroutines for controlled simulations are shown in Appendix C. The single-input single-output LQG controller used with ABAQUS is not included, however, it is in many ways similar to the multi-input multi-output LQG controller shown in Appendix A.

3 APPLICATION OF THE DEVELOPED METHODOLOGY

Two computational analyses are presented to demonstrate the controlled simulation capability. A brief description is given of the application of the ABAQUS code to simulate the closed-loop thermal response of a diamond CVD process, followed by a detailed description of controlled TACO simulations applied to the design and evaluation of an RTP system.

3.1 Closed-Loop Simulation of a Simplified Chemical-Vapor-Deposition Diamond Reactor using the ABAQUS software

A simplified thermal model of a diamond CVD reactor was formulated to demonstrate the controlled simulation procedure. The model consists of an axisymmetric graphite substrate 1.0 inch thick and 7.0 inches in diameter. The substrate is subjected to a heat flux of Gaussian distribution centered on the upper surface. The substrate is initially at 35 °C and throughout the subsequent loading the lower surface is maintained at 35 °C. The temperature at the center of the upper surface is quickly ramped to 1080 °C and maintained for the duration of the simulation. This is accomplished by modifying the peak flux in the Gaussian distribution by the control subroutine. As noted in section 2.2, the temperatures are read by the UVARM subroutine and passed to the subroutine which imparts the flux loading (DFLUX). The DFLUX subroutine calls the control subroutine, subsys_1, which modifies the loading based on the present temperature ("process sensor" variable) of the upper surface central coordinate. The results are shown in Figure 1. The temperature initially overshoots by about 50 °C, but quickly decreases to the desired temperature of 1080 °C. The time step used for this simulation is 1 second.

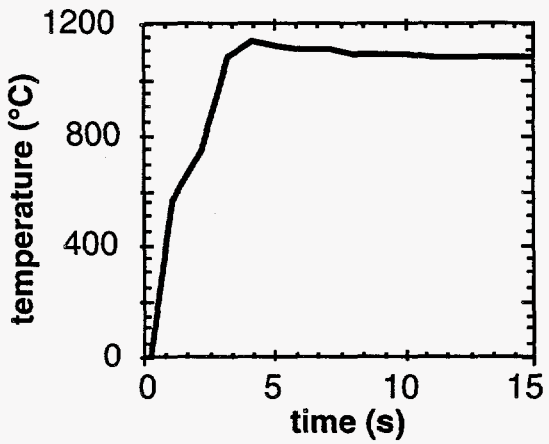

Figure 1. Temperature of substrate during controlled simulation.

3.2 Closed-Loop Simulation of a Rapid-Thermal-Processing (RTP) System using the TACO Software

A concurrent-engineering approach was applied to the development of an axisymmetric rapid-thermal-processing (RTP) reactor and its associated temperature controller. This reactor is developed for commercial use by the microelectronics equipment supplier CVC Products Inc. (CVC). Using a detailed finite element thermal model as a surrogate for actual hardware, we have developed and tested a multi-input multi-output (MIMO) controller. Closed-loop simulations were performed by linking the control algorithm with the finite element code. Simulations show that good temperature uniformity is maintained on the wafer during both steady and transient conditions. A numerical study shows the effect of ramp rate, feedback gain, sensor placement, and wafer-emissivity patterns on system performance.

3.2.1 System Description

The CVC RTP reactor is an axisymmetric design with five independently controlled lamp zones that heat the back side of a 200 mm wafer. Figure 2 shows a schematic representation of the reactor geometry. Each lamp zone contains an array of tungsten-halogen bulbs arranged in a circular pattern. Between each lamp zone is a radiation partition which limits "cross-talk" between the zones for improved control characteristics. The wafer rests face-up on a support that is attached to a rotation mechanism. Reactant gases are delivered through a multi-zone showerhead manifold from the top of the reactor. The face of the showerhead is polished for high reflectivity, creating a condition that approaches the behavior of a black-body cavity, and thereby reduces the sensitivity of the system to wafer front-side emissivity variations. A quartz window separates the lamp housing from the reaction chamber and the wafer. The chamber walls are water cooled. Deposition of reactants on the wafer back side and the

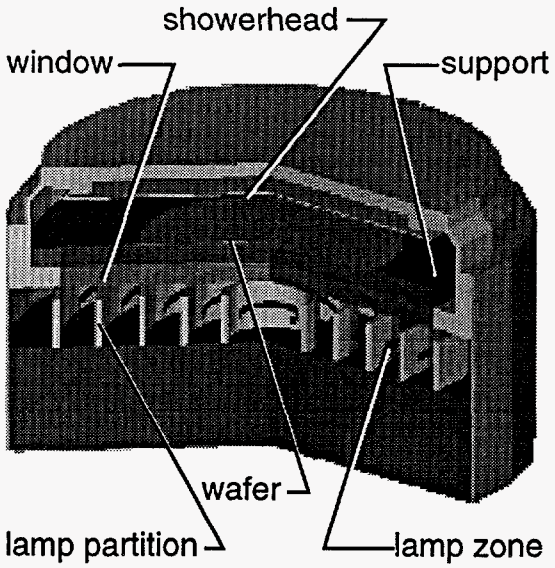

Figure 2. Model of a rapid-thermal-processing (RTP) reactor

window during processing is prevented by the wafer support ring. The reaction chamber is designed with a Modular Equipment Standards Committee (MESC) compatible interface.

3.2.2 Thermal Model

We have developed a finite element thermal model for design and evaluation of the CVC RTP system using the Sandia-developed TACO software. Radiant heat exchange between enclosure surfaces is based on the net radiation method [7]. View factors for the enclosure radiation exchange are computed using the VIEWC software [8]. The thermal simulations for this RTP system have approximately 1000 elements and 400 radiation surfaces. The model includes the silicon wafer, lamps, semi-transparent window, chamber walls, and showerhead gas injector. Heat is removed from the model through convective boundary conditions that account for air cooling inside the lamp housing and for water cooling on the outer chamber walls. Heat input to the model is through volumetric heat generation (W/m^3) in the lamp zones. An annular ring approximation is used to represent the discrete lamps of each zone. The heat generation is controlled independently for each lamp zone. A more detailed description of the model and its application to the design of the CVC RTP reactor are discussed by Spence, et al. [9] and Kee, et al. [10].

3.2.3 LQG Controller Development

System Identification

The availability of a thermal model of the CVC RTP reactor provided us with the opportunity to begin controller design before the hardware was fabricated. Using the model in place of the actual hardware, temperature responses at five discrete radial points on the wafer were predicted for a series of excitation signals to the control inputs (lamp zone powers) of the simulation. The temperature response points were chosen to correspond with the location of the pyrometers. Due to the nonlinear behavior of the system, it was important to characterize the system over the entire operating range (i.e., 500 °C to 1100 °C). Obtaining system response data at numerous operating temperatures through a series of open-loop simulations is a very time-consuming process. We were able to expedite the system identification process by performing the step-test simulations in two stages. First, one set of response data was obtained at a nominal operating temperature. Using these data, a simple (yet stable) controller was designed for the system. This

controller was programmed to drive the simulation under closed-loop operation to a specified temperature then switch to an open-loop step test sequence. Following the open-loop test sequence, the controller switched back to closed-loop operation and brought the system to the next temperature where the open-loop step-test was repeated. With this controller driving the simulation, a complete system identification was done automatically. This controller was designed with an interface that allows the test sequence parameters (i.e., test temperatures, step size and duration, ramp rate between temperatures) to be easily varied. Figure 3a shows the power inputs to five independently controlled lamp zones for seven consecutive step tests. At each test temperature, the lamp zones are sequentially pulsed with a step increase in power. The power excursions between each of the open-loop test sequences in Fig. 3a are a result of the closed-loop controller as it drives the simulation to the next test temperature. Figure 3b shows the wafer temperature response corresponding to the lamp power inputs shown in Fig. 3a.

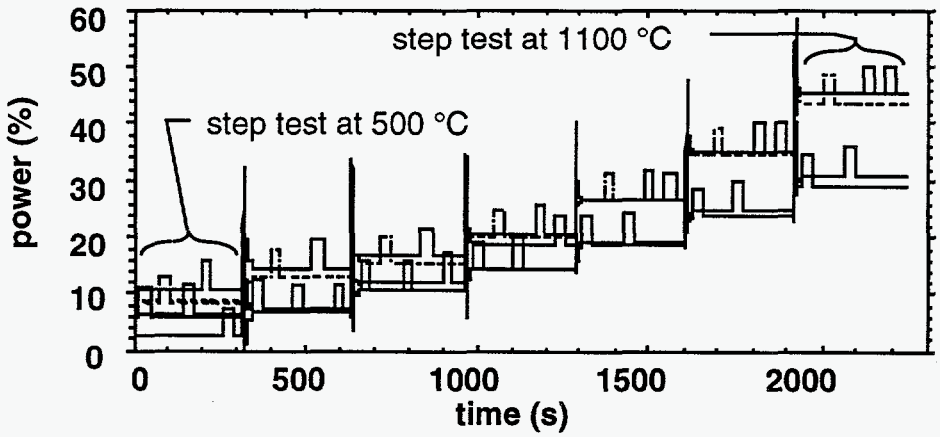

Figure 3a. Lamp powers for an automated system ID between 500 °C to 1100 °C.

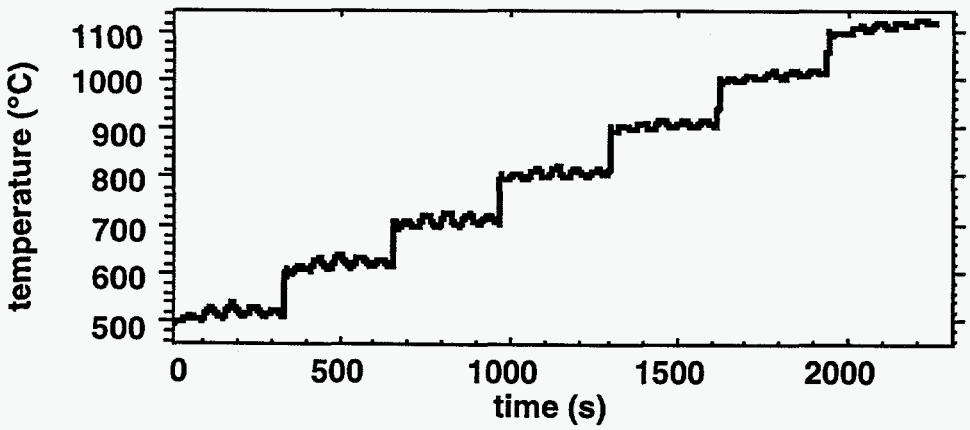

Figure 3b. Temperature response at wafer for excitation signal shown in Fig. 3a.

Linear Control Model

The response data we obtained from the finite element model was used to develop a control model of the RTP system. Using the state-space approach, a process is represented by a system of first-order differential equations. The least squares method is used to derive a plant model of the form

$$\dot{x} = Ax + Bu \tag{1}$$

$$y = Cx + Du \tag{2}$$

where x is the state vector, u is the input (power) vector, and y is the measured output (temperature) vector. The control model is developed in two stages. First a high-order model is developed. Next, a reduced-order model is obtained by eliminating the unimportant modes thereby reducing the number of states in the high-order model. The reduced linear model for our system has 5 states. Figures 4a and 4b show how the linear models compare with the nonlinear finite element model in predicting the wafer response to a step change in the power of the center lamp zone. The plots show that both the 5-state and the 38-state linear models are in good agreement with the finite element model. Reducing the model from 38 states to 5 states generates a minimal decrease in accuracy.

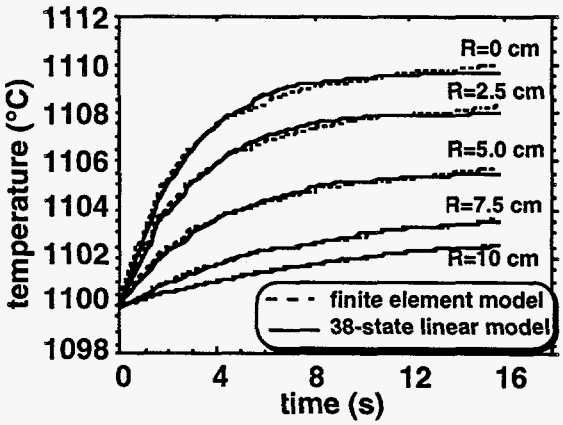

Figure 4a. Comparison of FE model and 38-state linear model in predicting response to a step change in power of the center lamp zone.

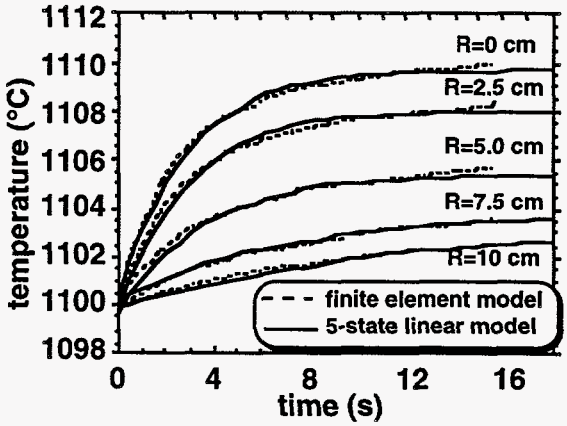

Figure 4b. Comparison of FE model and 5-state linear model in predicting response to a step change in power of the center lamp zone.

Design Process

We have designed a linear quadratic Gaussian (LQG) controller for the CVC RTP reactor. The LQG control strategy is well suited for RTP because of its applicability to multivariable and time-varying systems [11]. The control design process includes design of a linear-feedback controller (regulator) and design of a state estimator. The state estimator gives estimates of the states from the observed outputs. The regulator drives the states of the system while maintaining them within specified limits. The design of the regulator requires an optimal gain matrix, K_r, to be computed that minimizes a specified cost function, V. The cost function is expressed as the integral [12]

$$V = \int_0^\infty \left[x^T(\tau) Q(\tau) x(\tau) + u^T(\tau) R(\tau) u(\tau) \right] d\tau \qquad (3)$$

where Q and R are symmetric weighting matrices. The goal in designing the regulator is to minimize system response to noise or disturbances while avoiding saturation of the control signals. This balance is achieved through a somewhat trial-and-error process of selecting the weighting matrices (Q and R) that give the desired performance.

Since the regulator requires that all states of the system be available, an estimator (i.e., Kalman filter) is also required. The goal is to find an estimate of the state vector which minimizes the error between the actual state vector x and the estimated state vector $\hat{x}$. An optimal state-estimator gain matrix is calculated for the dynamic system. This gain matrix is derived by minimizing the expected mean square of the error between the measured output, y, and the output from the estimator, $\hat{y}$. The estimator model accounts for the fact that there may be some process noise within the system model itself as well as some noise inherent in the device used to measure the outputs. The resulting state equation for the estimator is [13]

$$\dot{\hat{x}} = (A - K_e C)\hat{x} + Bu + K_e y \qquad (4)$$

where K_e is the optimal state-estimator gain matrix. Combining the equations for the plant, the regulator, and the estimator results in the following equation for the LQG controller [13]:

$$\dot{\hat{x}} = (A - K_e C)\hat{x} + (K_e D - B)u + K_e y \tag{5}$$

where

$$u = -K_r \hat{x}. \tag{6}$$

Figure 5 shows a schematic of the controller design process utilizing a finite element model. The optimization loops represent the iterative process used to adjust the control design parameters (i.e., weighting matrices). At the first level, the control parameters are optimized using the high-order linear model to represent the plant. Next, the controller is linked with the finite element model. At this level, both the control parameters and the hardware design can be modified to optimize closed-loop performance. The final step is to optimize the controller on the actual plant (i.e., the RTP reactor).

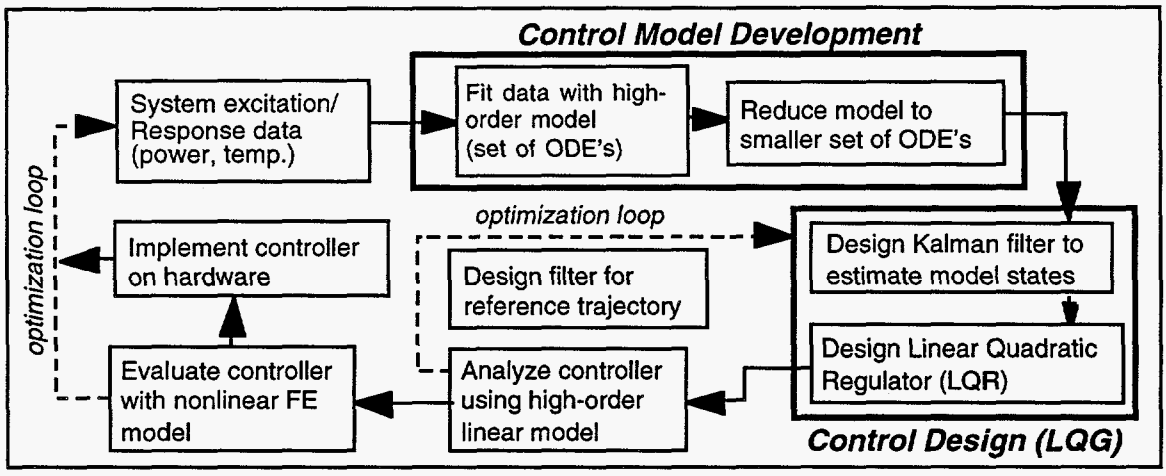

Figure 5. Schematic of the control design process utilizing a finite element model for response data and closed-loop evaluation.

3.2.4 Closed-Loop Simulations

The LQG controller was linked with the finite element model to evaluate the behavior of the closed-loop system. Running controlled simulations allows concurrent evaluation of both the controller design and the hardware design. We have used the closed-loop model to simulate a ramp from 800 °C to 1100 °C. Figure 6a shows the temperature history of the five "sensor" points on the wafer during a controlled simulation. The model does not include actual sensors; rather, specific points on the wafer that would be monitored by sensors in the actual hardware were designated as the sensor points for the simulations.

The sensor points were located at the wafer center, the wafer edge, and three equally distributed intermediate wafer points (R=2.5 cm, R=5.0 cm, and R=7.5 cm). Note that the five temperatures track so closely that they cannot be distinguished from each other in Fig. 6a. The reference temperature trajectory specified for the simulation calls for a smooth curve at the start and finish of the ramp to minimize the temperature tracking errors and the power spikes that will occur for trajectories with a discontinuity in the slope. Figure 6b shows the power history for each of the five lamp zones corresponding to the ramp shown in Fig. 6a.

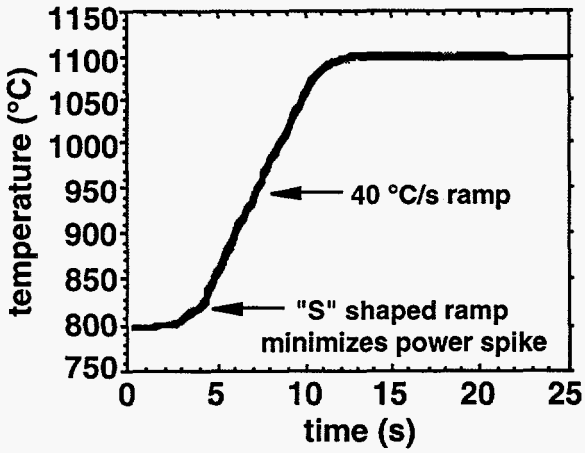

Figure 6a. Wafer temperature history during controlled simulation.

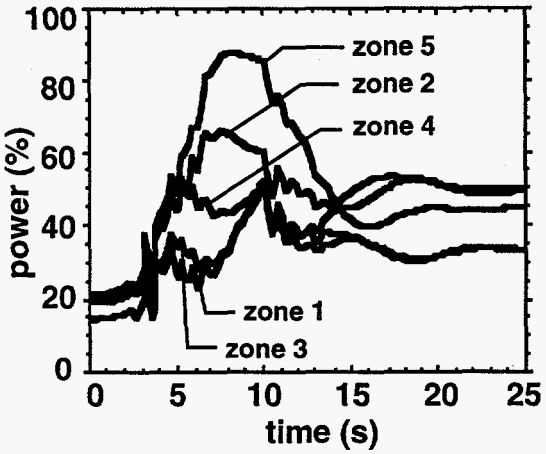

Figure 6b. Lamp zone powers during controlled simulation.

Wafer temperature uniformity is an important criterion for performance evaluation. Temperature gradients during high ramp rates can lead to stress fracture (slipping) of the wafer while temperature non-uniformity during steady conditions leads to non-uniformity of the process (e.g., chemical vapor deposition, oxide growth, or diffusion). Figure 6c shows the wafer temperature difference predicted for the trajectory shown in Fig. 6a. The dashed curve shows the maximum temperature difference as indicated by the five sensor points. A significant advantage of using a simulation for controller evaluation is that the model is not limited to information from the sensors. Temperature data is available over the entire wafer. The solid curve in Fig. 6c shows the maximum temperature difference across the entire wafer with a peak value of 6 °C which is almost twice that indicated by the sensors. The radial temperature profile corresponding to the time at which the maximum temperature difference occurs (time = 6 s) is shown in Fig. 6d. A slight overlap between the wafer and the support ring creates an annular region at the wafer edge with a slightly higher mass than the rest of the wafer. During high ramp rates, this high-mass region lags

behind the rest of the wafer resulting in the temperature dip seen at the wafer edge in Fig. 6d.

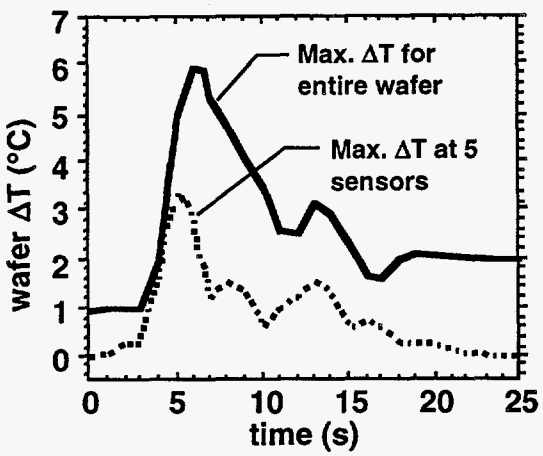

Figure 6c. Maximum temperature variation (ΔT) across the wafer during a controlled 40 °C/s ramp.

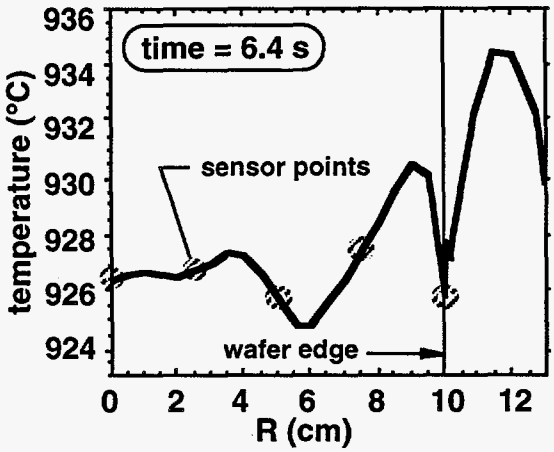

Figure 6d. Radial temperature profile on wafer at time of peak ΔT during controlled ramp.

Sensor Location

The optimal sensor locations depend on reactor design, control strategy, and process objectives. We investigate the effect of shifting the position of the outer sensor. Determining the best position requires consideration of the transient temperature uniformity requirements and the size of the exclusion region (i.e., annular area at the wafer edge containing no die).

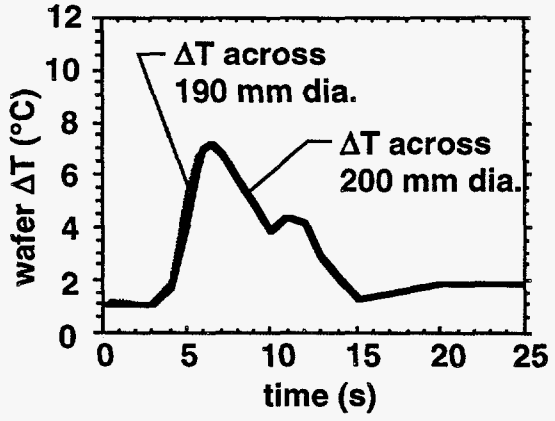

Figure 7a. Outer sensor at R=100 mm. Tight control on wafer edge increases variation at inner regions of the wafer.

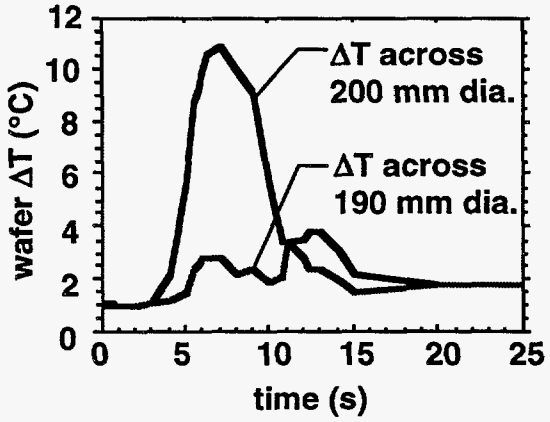

Figure 7b. Outer sensor at R=95 mm. Removing wafer edge from the active control zone significantly reduces ΔT across 190 mm of a 200 mm wafer.

The primary difficulty in maintaining wafer temperature uniformity during the high ramp rates is the wafer edge effect. If we design the controller to minimize temperature variation over a region that excludes the very edge of the wafer, then temperature uniformity over the inner portion of the wafer is significantly improved. Figure 7a shows the wafer-temperature variation for a simulation in which the temperature is controlled to the wafer edge (outer sensor at R = 100 mm).

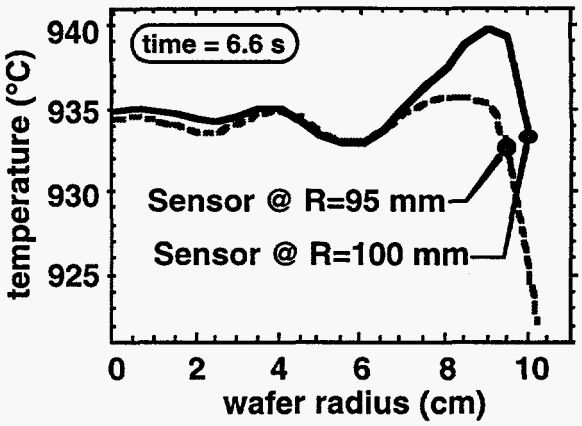

Figure 7c. Location of outer sensor has a large effect on radial temperature profile during high ramp rates (50 °C/s shown).

Figure 7b shows the wafer-temperature variation for a simulation in which the outer 5 mm of the wafer is excluded from the controlled zone (outer sensor at R = 95 mm). A comparison of Figs. 7a and 7b shows that by moving the outer sensor in from the wafer edge we degrade the overall uniformity, however, uniformity over the majority of the wafer (190 mm diameter) is improved. This effect is illustrated in Fig. 7c which shows the wafer temperature profiles during the time of worst case temperature non-uniformity during the ramp. From this analysis we can conclude two important points: (1) Design of the interface between the wafer and wafer-support ring should minimize variations in thermal mass, and (2) the position of the temperature sensors for optimal uniformity depend on the accepted exclusion region for the wafer edge.

Ramp Rate Effects

The push for high ramp rates is generated by the need to reduce cycle time and thermal budget. We investigate the effect of increased ramp rates on temperature uniformity and cycle time. Increasing the ramp rate will reduce the transient time between setpoint temperatures, however, temperature variation on the wafer will increase resulting in longer stabilization times. The effect that ramp rate will have on the overall cycle time depends on reactor design and control strategy.

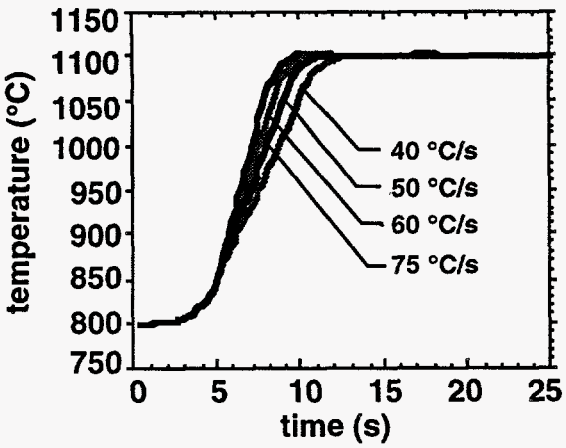

Figure 8a. Wafer temperature during controlled simulations with ramp rates of 40, 50, 60, and 75 °C/s.

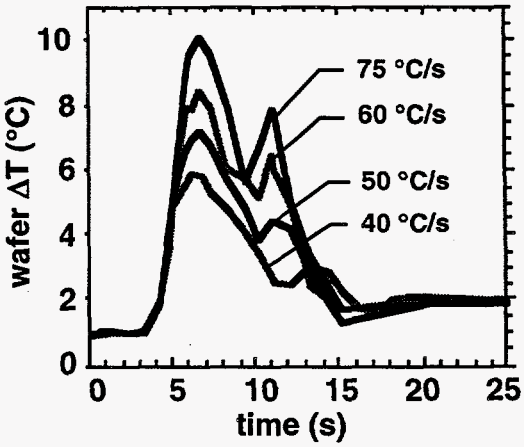

Figure 8b. Maximum temperature variation across the wafer during controlled simulations at ramp rates of 40, 50, 60, and 75 °C/s.

We have repeated the controlled ramp at a number of different rates. Figure 8a shows the wafer-temperature history of four ramps between 40 °C/s and 75 °C/s. The wafer-temperature variation (over the entire 200 mm diameter) for each ramp is shown in Fig. 8b. The challenge of maintaining temperature uniformity on the wafer becomes greater as the ramp rate is increased. Both reactor design and controller design play a role in the realizable temperature uniformity. As the ramp rate increases, the power resources required to drive the states within the desired tolerance also must increase. Also, the higher ramp rates may require a longer stabilization time which will reduce the impact on cycle time reduction. If we impose the restriction that processing can begin only after the wafer temperature variation has decreased below 3 °C, Fig. 8b shows that the higher ramp rates provide little improvement in cycle time.

Feedback Gain

Weighting parameters in the controller define the balance between setpoint tracking and the range of power control. We have included a simple tuning parameter *(α)* in our controller to adjust the feedback controller gain. In this case, $Q = \alpha\overline{Q}$ where $\overline{Q}$ is a weighting matrix. (We note that a more appropriate procedure to improve performance involves tuning the entire Q and R matrices.) As the value of α is increased, the control action for a given tracking error $\left(\left|T_{reference} - T_{measured}\right|\right)$ is increased. Figure 9a shows the temperature

variation on the wafer during a 50 °C/s ramp followed by a stabilization at 1100 °C. This simulation was run with tuning parameters of $\alpha=1$ and $\alpha=3$. Note that as we increase α from 1 to 3, the peak temperature variation on the wafer is reduced from 10 °C to 7 °C. The cost of the improvement in temperature uniformity is an increase in required power resources. Figure 9b, shows that we actually saturate power in lamp zone 5 (the outer zone) for the simulation with α equal to 3. With α equal to 1, we have no problem with power saturation, however, the controller now tolerates greater tracking errors resulting in a more sluggish response.

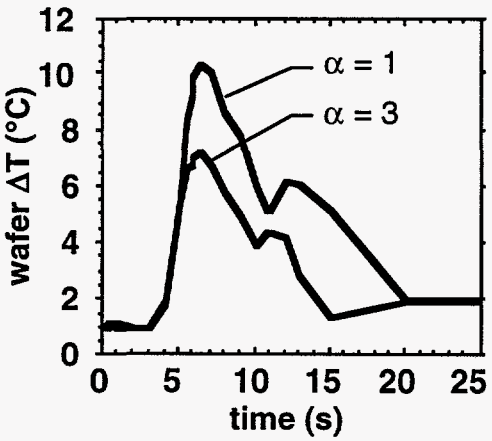

Figure 9a. Maximum temperature variation across wafer during a 50 °C/s ramp for two values of the control parameter α.

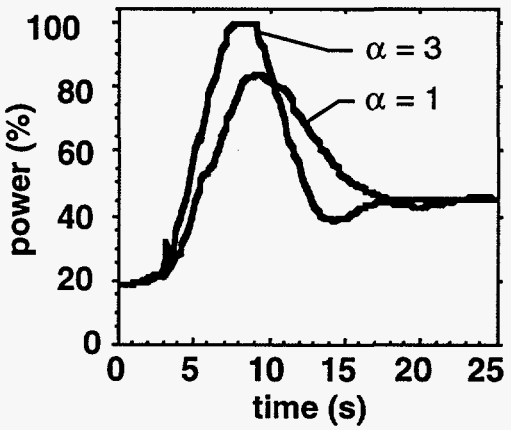

Figure 9b. Power history for the outer lamp zone during a 50 °C/s ramp for two values of the control parameter α.

4 CONCLUSION

We have developed a methodology for coupling large-scale numerical codes with process control algorithms. Closed-loop simulations were demonstrated using the Sandia-developed finite element thermal code TACO and the commercially available finite element thermal-mechanical code ABAQUS. This new capability enables us to use computational simulations for designing and prototyping advanced process-control systems. By testing control algorithms on simulators before building and testing hardware, enormous time and cost savings can be realized.

The need for a closed-loop simulation capability was demonstrated in a detailed design study of a rapid-thermal-processing reactor under development by CVC Products Inc. Using a thermal model of the RTP system as a surrogate for the actual hardware, we were able to generate response data needed for controller design. We then evaluated the performance of both the controller design and the hardware design by using the controller to drive the finite element model. The controlled simulations provided data on wafer temperature uniformity as a function of ramp rate, temperature sensor locations, and controller gain. This information, which is critical to reactor design, cannot be obtained from typical open-loop simulations.

5 REFERENCES

[1] W.E. Mason, *TACO3D - A Three-Dimensional Finite Element Heat Transfer Code,* Sandia National Laboratories, (1983).

[2] ABAQUS Standard Users Manual, Version 5.4, Hibbitt, Karlsson & Sorensen, Inc., Pawtucket, RI, 1994.

[3] MATRIXx User's Guide, Integrated Systems, Inc., CA, (1994).

[4] MATLAB User's Guide, The MathWorks, Inc., Mass., (1994).

[5] H.B. Sim and M.C. Boyce, *Finite Element Analyses of Real-Time Stability Control in Sheet Forming Processes,* J. Engng. Matls. and Tech., Vol. 114, (1992), p. 180.

[6] J. Cao and M.C. Boyce, Draw Bead Penetration as a Control Element of Material Flow, Proceedings of SAE Symposium on Sheet Stamping, Detroit, MI, (1993), p. 145.

[7] R. Siegal, J. Howell, Thermal Radiation Heat Transfer, Hemisphere Publishing, New York, NY, (1981).

[8] A.F. Emery, *View Users Manual*, Univ. of Wash., (1984).

[9] P. Spence, W. Winters, R. Kee, A. Kermani, *The Application of Computational Simulation to Design Optimization of an Axisymmetric Rapid Thermal Processing System,* Proc. of the 2nd Int. Conf. on RTP, R.B. Fair and B. Lojek (eds), Monterey, CA, Sept., (1994), p. 139.

[10] R. Kee, A. Ting, P. Spence, *Understanding and Improving Materials Processing Through Interpreting and Manipulating Predictive Models,* Proc. of the Mat. Res. Soc. Conf., Boston, MA, Nov., (1994).

[11] K. Åström, B. Wittenmark, Computer Controlled Systems, Prentice-Hall, Englewood Cliffs, NJ, (1984).

[12] B. Friedland, Control System Design - an Introduction to State-Space Methods, McGraw-Hill Book Co., New York, NY, (1986).

[13] T. Kailath, Linear Systems, Prentice-Hall, Englewood Cliffs, NJ, (1980).

A Control Subroutine

(Four-input four-output LQG controller designed with MATRIXx software for an RTP reactor)

```
---- AutoCode/C Code Generator V3.3   -----
--
--  Model File   : control.rtf
--  Model Date   : 15-Aug-94 11:16
--
--  Generated File: control.c
--
--  Number of External Inputs : 4
--  Number of External Outputs: 4
--  Number of Internal States : 9
*/

#include <stdio.h>
#include <math.h>
#include "sa_system.h"
#include "sa_syntax.h"
#include "sa_types.h"
#include "sa_math.h"

#define OK                    0
#define STOP_BLOCK            1
#define MATH_ERROR            2
#define STOPPED               3
#define UCB_ERROR             4
#define TIME_OVERFLOW         -1
#define UNKNOWN_ERROR         100

#define IO_ERROR              1
#define EXIT_CONDITION        2
#define USERCODE_ERROR        3

void controller(IINFO, T, U, NU, X, XD, NX, Y, NY, R_P, I_P)
struct STATUS_RECORD *IINFO;
RT_DURATION  T;
RT_FLOAT     U[], X[], XD[], Y[];
RT_INTEGER   NU, NX, NY;
RT_FLOAT     R_P[];
RT_INTEGER   I_P[];

#define     RP_OFF 0
#define     IP_OFF 0

{

#define EPSILON               3.72529E-09
  static const RT_FLOAT       EPS = 4.0*EPSILON;

  static RT_INTEGER           GLOBAL_EXCEPTION = 0;
```

```
RT_FLOAT  BUS[40];
RT_INTEGER I;

struct STATUS_RECORD INFO;
INFO.INIT = IINFO->INIT;

if (INFO.INIT) {
 /* No R_P values to initialize */
 /* No I_P values to initialize */
}

if (INFO.INIT) {
  /* Pre-initialize all states to ZERO */
  for ( I=0; I<NX; I++ ) X[I] = 0.0;
}

/*------------------------------- State Space System  */
/* {control.controller.2} */
   BUS[4] = -0.0360997 * X[0] + 0.0559576 * X[1] - 0.0357339 *
            X[2] + 0.0111592 * X[3] - 1.354634 * X[4] +
            2.08484 * X[5] - 1.234865 * X[6] + 0.3101312 *
            X[7];
   BUS[5] = 0.037932 * X[0] - 0.0902785 * X[1] + 0.0635839 * X[2]
            - 0.0181796 * X[3] + 1.223012 * X[4] - 2.917237 *
            X[5] + 1.905031 * X[6] - 0.4231844 * X[7];
   BUS[6] = -0.0135509 * X[0] + 0.0364134 * X[1] - 0.050265 *
            X[2] + 0.0115975 * X[3] - 0.2825409 * X[4] +
            0.8602452 * X[5] - 1.370691 * X[6] + 0.2300479 *
            X[7];
   BUS[7] = 0.0053118 * X[0] - 0.0159889 * X[1] + 0.0263451 *
            X[2] - 0.0223297 * X[3] + 0.1161253 * X[4] -
            0.3819002 * X[5] + 0.67949 * X[6] - 0.5771485 *
            X[7];
/*------------------------------- Time Delay  */
/* {reference generator.clock state.1} */
   BUS[8] = X[8];
/*------------------------------- General Nested Expression  */
/* {reference generator.clock reset value.10} */
   BUS[9] = 0.0;
/*------------------------------- General Nested Expression  */
/* {control.reference parameters.4} */
   BUS[10] = 1.0;
   BUS[11] = 1073.0;
   BUS[12] = 1373.0;
   BUS[13] = 20.0;
   BUS[14] = 26.0;
   BUS[15] = 50.0;
   BUS[16] = 60.0;
/*------------------------------- General Logical Expression  */
/* {reference generator..5} */
   if( BUS[8]<BUS[13] ){
     BUS[17] = 1.0;
   }else{
     BUS[17] = 0.0;
```

```
    }
    if( BUS[8]>=BUS[13] &&  BUS[8]<BUS[14] ){
      BUS[18] = 1.0;
    }else{
      BUS[18] = 0.0;
    }
    if( BUS[8]>=BUS[14] &&  BUS[8]<BUS[15] ){
      BUS[19] = 1.0;
    }else{
      BUS[19] = 0.0;
    }
    if( BUS[8]>=BUS[15] &&  BUS[8]<BUS[16] ){
      BUS[20] = 1.0;
    }else{
      BUS[20] = 0.0;
    }
    if( BUS[8]>=BUS[16] ){
      BUS[21] = 1.0;
    }else{
      BUS[21] = 0.0;
    }
/*------------------------------ General Nested Expression */
/* {control.detrend output.97} */
    BUS[22] = U[0] - 1178.99;
    BUS[23] = U[1] - 1176.75;
    BUS[24] = U[2] - 1173.5;
    BUS[25] = U[3] - 1176.81;
/*------------------------------ General Nested Expression */
/* {reference generator.clock update.6} */
    BUS[26] = BUS[8] + 0.05;
/*------------------------------ Data Path Switch */
/* {reference generator.clock reset switch.4} */
    if( BUS[10] > 0.0 ){
      BUS[27] = BUS[26];
    }else{
      BUS[27] = BUS[9];
    }
/*------------------------------ General Nested Expression */
/* {reference generator.compute divisor.2} */
    BUS[28] = BUS[14] - BUS[13];
    BUS[29] = BUS[16] - BUS[15];
/*------------------------------ Bounded Limiter */
/* {reference generator.prevent divide by zero.99} */
    BUS[30] = MIN( 10000.0, MAX( 0.1, BUS[28] ) );
    BUS[31] = MIN( 10000.0, MAX( 0.1, BUS[29] ) );
/*------------------------------ General Nested Expression */
/* {reference generator..3} */
    BUS[32] = BUS[11] + ( BUS[8] - BUS[13] )/BUS[30]*( BUS[12] -
            BUS[11] );
    BUS[33] = BUS[12] + ( BUS[8] - BUS[15] )/BUS[31]*( 0.0 -
            BUS[12] );
    BUS[34] = 0.0;
/*------------------------------ Dot Product */
/* {reference generator..98} */
```

```
   BUS[35] = BUS[11] * BUS[17] + BUS[32] * BUS[18] + BUS[12] *
         BUS[19] + BUS[33] * BUS[20] + BUS[34] * BUS[21];
/*------------------------------ General Nested Expression */
/* {control.trend input.99} */
   BUS[36] = BUS[4] + 7.333333;
   BUS[37] = BUS[5] + 19.0;
   BUS[38] = BUS[6] + 28.0;
   BUS[39] = BUS[7] + 37.0;
/*------------------------------ Bounded Limiter */
/* {control.limit output.7} */
   BUS[0] = MIN( 100.0, MAX( 0.0, BUS[36] ) );
   BUS[1] = MIN( 100.0, MAX( 0.0, BUS[37] ) );
   BUS[2] = MIN( 100.0, MAX( 0.0, BUS[38] ) );
   BUS[3] = MIN( 100.0, MAX( 0.0, BUS[39] ) );
/*------------------------------ Collect Ext. Outputs into Y vector */
   Y[0] = BUS[0];
   Y[1] = BUS[1];
   Y[2] = BUS[2];
   Y[3] = BUS[3];
/*  ---  */
/*------------------------------ State Space System */
/* {control.controller.2} */
   XD[0] = 1.0 * X[0] - 1.0 * BUS[35] + 1.0 * U[0];
   XD[1] = 1.0 * X[1] - 1.0 * BUS[35] + 1.0 * U[1];
   XD[2] = 1.0 * X[2] - 1.0 * BUS[35] + 1.0 * U[2];
   XD[3] = 1.0 * X[3] - 1.0 * BUS[35] + 1.0 * U[3];
   XD[4] = -0.0003946 * X[0] - 0.0005087 * X[1] + 0.0000432 *
         X[2] - 0.0003761 * X[3] + 0.5924984 * X[4] -
         0.2501317 * X[5] - 0.0774687 * X[6] - 0.0551646 *
         X[7] + 0.3776521 * BUS[22] + 0.2284378 * BUS[23] +
         0.0852405 * BUS[24] + 0.0465561 * BUS[25];
   XD[5] = -0.0000606 * X[0] - 0.0006783 * X[1] - 0.0000873 *
         X[2] - 0.0004033 * X[3] - 0.2287926 * X[4] +
         0.7298877 * X[5] - 0.1640016 * X[6] - 0.0901365 *
         X[7] + 0.2284093 * BUS[22] + 0.2268715 * BUS[23] +
         0.1626117 * BUS[24] + 0.0832757 * BUS[25];
   XD[6] = -0.0001137 * X[0] - 0.0000289 * X[1] - 0.0006235 *
         X[2] - 0.0004372 * X[3] - 0.0903641 * X[4] -
         0.1557468 * X[5] + 0.7271194 * X[6] - 0.1944726 *
         X[7] + 0.0844371 * BUS[22] + 0.1622539 * BUS[23] +
         0.225916 * BUS[24] + 0.1898447 * BUS[25];
   XD[7] = -0.0000488 * X[0] - 0.000101 * X[1] - 0.0000525 * X[2]
         - 0.001069 * X[3] - 0.0473771 * X[4] - 0.0833228 *
         X[5] - 0.1878652 * X[6] + 0.5320755 * X[7] +
         0.0452525 * BUS[22] + 0.0821656 * BUS[23] +
         0.1883355 * BUS[24] + 0.416543 * BUS[25];
/*------------------------------ Time Delay */
/* {reference generator.clock state.1} */
   XD[8] = BUS[27];

Exception
  When EXIT_CONDITION :
     IINFO->ERROR = STOP_BLOCK;
     Breakexception;
```

```
    Others :
        IINFO->ERROR = UNKNOWN_ERROR;
        Breakexception;
  Endexception

}

void SR_SCHEDULER(init, t, u, nu, y, ny)

int *init;
RT_DURATION *t;
RT_FLOAT u[], y[];
RT_INTEGER *nu, *ny;

/*
This is a single rate scheduler used with MATRIXx user code blocks.
init is used to indicate that initialization is required.
Make sure to call the single rate scheduler with init=1
at least once before calling it with init=0.

init = 1 implies initialization.
init = 0 implies normal running of the ucb.

This single rate scheduler is set up to call the application
controller(), which contains application code.
*/

{
 extern void controller();

 static RT_FLOAT     x[9], xd[9];
 static RT_INTEGER   nx;
 static RT_INTEGER   ipp[1];
 static RT_FLOAT     rpp[1];
 static struct STATUS_RECORD info;
 int i;

 if (*init != 1) { /* Update the states and outputs. */
   controller(&info, *t, u, *nu, x, xd, nx, y, *ny, rpp, ipp);
 }
 else { /* Initialize the application */
   *nu = 4;
   *ny = 4;
   nx = 9;
   info.ERROR = 0; /* Set flags to initialize application. */
   info.INIT = 1;
   info.STATES = 0;
   info.OUTPUTS = 0;
   controller(&info, *t, u, *nu, x, xd, nx, y, *ny, rpp, ipp);
   info.ERROR = 0; /* Reset flags for application execution. */
   info.INIT = 0;
   info.STATES = 1;
   info.OUTPUTS = 1;
 }
```

```
 /* Propagate the states. */
 for (i=0;i<nx;i++) {
  x[i] = xd[i];
 }

}

RT_FLOAT
MIN (ARG1, ARG2)
  RT_FLOAT ARG1, ARG2;
{
  if( ARG1 <  ARG2 ){
    return  (ARG1);
  }else{
    return  (ARG2);
  }
}

RT_FLOAT
MAX (ARG1, ARG2)
  RT_FLOAT ARG1, ARG2;
{
  if( ARG1 >  ARG2 ){
    return (ARG1);
  }else{
    return (ARG2);
  }
}

RT_INTEGER
NINT (R)
RT_FLOAT R;
{
  if ( R >= 0.0 )
    return ((RT_INTEGER)(R+0.5));
  else
    return ((RT_INTEGER)(R-0.5));
}
```

B Subroutines for Socket I/O

```
/*   socket.c                    */

/*       Include standard libraries */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

/* Need to include to get socket communication and string interface */
#include  "communicate.h"
#include "cntrl_string.h"

#define SERVERCLIENT 1

#define MAX 6

/* Global Variable */
int server = FALSE;

void SOCKET( init, t, u, nu, y, ny, fds, fdc, terminate)

int *init;
double *t;
double u[];
int *nu;
double y[];
int *ny;
int *fds;
int *fdc;
int *terminate;
{
 int result, length;
 char hostName[MAX_LENGTH], imtheserver[MAX_LENGTH];
 char buff[MAX_LENGTH],text[MAX_LENGTH];
 int j,i;

 printf("nu_skt = %d\n",*nu);
 printf("ny_skt = %d\n",*ny);
 printf("init_skt = %d\n",*init);
 sprintf(hostName,"%s","beast");
 if (*init)
 {
#ifdef SERVERCLIENT
/* First You Must open up client/server socket */
 if (!(*fds = socketInit( TRUE, imtheserver, PORT1))) /* Open Socket */
 {
  printf("Failed to open socket\n");
/*   return(TRUE);
 */
 }
 if (!(*fdc = socketInit( FALSE, hostName, PORT2))) /* Open Socket */
 {
```

```
   printf("Failed to open socket\n");
/*    return(TRUE);
 */
  }
  *ny = *nu;
  for (i=0; i<*ny; i++)  *(y+i) = *(u+i);
#else
  if (!(*fdc = socketInit( FALSE, hostName, PORT1))) /* Open Socket */
  {
   printf("Failed to open socket\n");
/*    return(TRUE);
 */
  }
  if (!(*fds = socketInit( TRUE, imtheserver, PORT2))) /* Open Socket */
  {
   printf("Failed to open socket\n");
/*    return(TRUE);
 */
  }
  *ny = *nu;
  for (i=0; i<*ny; i++)  *(y+i) = *(u+i);
#endif
  }
#ifdef SERVERCLIENT
    length = GenDblString(text, u, *nu);
    for (i=0; i<*nu; i++)printf("%f\n",*(u+i));
    printf("%s\n",text);
    if (!(result = write(*fds, text, MAX_LENGTH)))
    {
      printf("Error on socket write\n");
/*      return(-1);
 */
    }
    printf("\n\nSent Data\n\n");

    if (!(result = read(*fdc, buff, MAX_LENGTH)))
    {
      printf("Error on socket read\n");
/*      return(-1);
 */
    }
    ParseDbl(y, buff);
#else
    if (!(result = read(*fdc, buff, MAX_LENGTH)))
    {
      printf("Error on socket read\n");
/*      return(-1);
 */
   }
    ParseDbl( y, buff);
    length = GenDblString(text, u, *nu);
    if (!(result = write(*fds, text, MAX_LENGTH)))
    {
      printf("Error on socket write\n");
```

```
/*      return(-1);
 */
    }
#endif

 if (*terminate)
 {
  close(*fds);
  close(*fdc);
 }
}
```

/* communicate.c */

```
#include "communicate.h"

float byterev(float f)
{
        char fbytes[4];
        char *charptr;
        int i;

        charptr = (char *)&f;
        for(i=0;i<4;i++) fbytes[i]=*(charptr+3-i);
        return(*(float *)fbytes);
}
int fullRead(int fd, char *buff, int amount)
{
        int result;
        while (amount >0)
        {
          result = read(fd, buff, amount);
          if(result <=0) return(-1);
          amount -= result;
          buff += result;
        }
        return(0);
}

int fullWrite(int fd, char *buff, int amount)
{
        int result;
        while (amount >0)
        {
          result = write(fd, buff, amount);
          if(result <=0) return(-1);
          amount -= result;
          buff += result;
        }
        return(0);
}

/*  specify internet socket by defining TCP */
```

```
#define TCP          1

#ifdef TCP
#define SOCK_TYPE SOCK_STREAM
#else
#define SOCK_TYPE SOCK_DGRAM
#endif

/*static void fatal(char *msg);*/

static void fatal(char *msg)
{
        perror(msg);
        exit(1);
}

int socketInit(long int server,char hostName[], int port)
{
        int s;
        struct hostent *h;
        struct sockaddr_in addr;
        int addrlen = sizeof(addr);
        int i;

        /* client takes remote host name as argument, server takes no        */
        /* arguments.                                                        */

        /* create the socket                                                 */
        if( (s = socket(AF_INET, SOCK_TYPE, 0)) < 0 ) fatal("socket");

        /* Initialize the address                                            */
        addr.sin_family = AF_INET;
        addr.sin_port = htons(port);
        addr.sin_addr.s_addr = INADDR_ANY;
        if( server )
        {
                int f = s;
                int got;

                /* bind socket to PORT                                       */
                if( bind(s,(struct sockaddr *)&addr,addrlen) < 0 )
                                                        fatal("bind");

#ifdef TCP
                /* make socket listen for connects                           */
                if( listen(s,5) < 0 ) fatal("listen");

                /* Accept/wait for a connection                              */
                if( (f = accept(s,(struct sockaddr *)&addr,&addrlen)) < 0 ) fatal("accept");
                printf("f = %d\n",f);
                return(f);

#endif
```

```
        }
        else /* client */
        {
                char hname[100];
                int hnamelen = 100;
                /* lookup remote host name in etc/hosts database      */
                h = gethostbyname(hostName);
                if( h == NULL ) fatal("gethostbyname");

                addr.sin_addr = *((struct in_addr *)h->h_addr);

                /* connect socket to remote PORT                      */
                if( connect(s,(struct sockaddr *)&addr,addrlen) < 0 ) fatal("connect");
                return(s);
        }
}
```

/* communicate.h */

```
#ifndef COMMUNICATE
#define COMMUNICATE 1

/* Include Internet socket libraries */
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
/* Include UNIX socket library */
#include <unistd.h>

/* Magic port number - change if necessary */
#define PORT1                   4567
#define PORT2                   4568
#define TRUE -1
#define FALSE 0
#define MAX_LENGTH 1024

/* function prototypes */
int socketInit(long int server, char serverName[], int);
int fullWrite(int fd, char obuffer[], int nbytes);
float byterev(float f);
int fullRead(int fd, char *buff, int amount);

#endif
```

/* cntrl_string.c */

```
#include "cntrl_string.h"

int ReadLine(char *ps)
```

```
{
  char plocal[100],*pl;
  int length;
  pl = plocal;
  while ('\n' != (*pl = getc(stdin))) pl++;
  *pl = '\0';
  length = strlen(plocal);
  sprintf(ps,"%d%s",length,plocal);
  return(length);
}

int GenFltString(char *ps, float *data, int length)
{
 int i, strlength;
 char tempstr[MAX_LENGTH],dummy[40];
 for (i=0; i<length; i++)
 {
   sprintf(dummy,"%f$",*(data+i));
   strcat(tempstr,dummy);
 }
 strlength = strlen(tempstr);
 sprintf(ps,"%s%d$%d$%s","FLT",strlength,length,tempstr);
 return(strlength);
}

int GenDblString(char *ps, double *data, int length)
{
 int i, strlength;
 char tempstr[MAX_LENGTH],dummy[40];
 for (i=0; i<length; i++)
 {
   sprintf(dummy,"%lf$",*(data+i));
   strcat(tempstr,dummy);
 }
 strlength = strlen(tempstr);
 sprintf(ps,"%s%d$%d$%s","FLT",strlength,length,tempstr);
 return(strlength);
}

int GenString(char *ps, char *ps1)
{
  int length;
  length = strlen(ps1);
   sprintf(ps,"%s%d%s","STR",length,ps1);
  return(length);
}

int ParseLine(char *string)
{
 int check_length, len;
 char nolen[10];
 sscanf(string,"%d",&check_length);
 sprintf(nolen,"%d",check_length);
 len = strlen(nolen);
```

```
 strncpy(string,string+len,check_length+1);
 return(check_length);
}

int ParseFlt(float *pf, char *ps)
{
 int l1,l2, len1,len2,lt,i;
 char nolen1[10],nolen2[10], *pstr,c,c1,c2;
 sscanf(ps+3,"%d%c%d%c",&l1,&c1,&l2,&c2);
 sprintf(nolen1,"%d",l1);
 sprintf(nolen2,"%d",l2);
 lt = strlen(nolen1)+strlen(nolen2)+3;
 pstr = ps+lt+2;
 for (i=0;i<l2;i++)
 {
   sscanf(pstr,"%f%c",pf+i,&c);
   while(*pstr++ != '$');
 }
 return(l2);
}

int ParseDbl(double *pf, char *ps)
{
 int l1,l2, len1,len2,lt,i;
 char nolen1[10],nolen2[10], *pstr,c,c1,c2;
 sscanf(ps+3,"%d%c%d%c",&l1,&c1,&l2,&c2);
 sprintf(nolen1,"%d",l1);
 sprintf(nolen2,"%d",l2);
 lt = strlen(nolen1)+strlen(nolen2)+3;
 pstr = ps+lt+2;
 for (i=0;i<l2;i++)
 {
   sscanf(pstr,"%lf%c",pf+i,&c);
   while(*pstr++ != '$');
 }
 return(l2);
}

void ParseString(char *ps1, char *ps2)
{
 int l1, len1,lt;
 char nolen1[10], *pstr;
 sscanf(ps1+3,"%d",&l1);
 sprintf(nolen1,"%d",l1);
 lt = strlen(nolen1)+3;
 strncpy(ps1,ps2+lt,l1+1);
}
```

/* cntrl_string.h */

```
#ifndef CNTRL_STRING
#include <stdio.h>
#include <string.h>
```

```
#include <math.h>

#define CNTRL_STRING 1
#define MAX_LENGTH 1024

void ParseString(char *, char *);
int ReadLine(char *);
int ParseLine(char *);
int GenString(char *, char *);
int GenFltString(char *,  float *, int);
int GenDblString(char *, double *, int);
int ParseFlt(float *, char *);
int ParseDbl(double *, char *);
#endif
```

C ABAQUS User Subroutines

```
      SUBROUTINE DFLUX
     (FLUX,TEMP,KSTEP,KINC,TIME,NOEL,NPT,COORDS,JLTYP)

c     This subroutine generates new flux based on elemental temperatures

      INCLUDE 'ABA_PARAM.INC'
      DIMENSION FLUX(2),TIME(2),COORDS(3)
      DIMENSION ETEMP(4,350)
      DIMENSION U(2),Y(2)

c     t11 is the element temperature used for control

      COMMON /ETEMPE/t11,t131,ETEMP
      character*256 dirout
      save qpsave,icount

c     LXGENV is internal ABAQUS subroutine that sets up output file definition

      call lxgenv('OUTDIR', dirout, ldirout)
      open (unit=51,file=dirout(1:ldirout)//'d551')
      R=COORDS(1)
       S =.01
       tsp=1080.0
      u(2) = tsp
      if(icount.ge.kinc) go to 197
      if(kinc.lt.2) then
        qpsave = 1.98e7/2.74
        go to 197
      endif
      if (kinc.gt.1) then
       u(1) = t11

c     subsys_1 is control subroutine which calculates flux based on temperature , t11

       call subsys_1(u,y)
       qpsave = y(1)/2.74
      endif
  179 icount = kinc
  197 QP = qpsave

c     Flux distribution, FLUX(1)

      FLUX(1) = QP*EXP(-0.5*R*R/(S*S))
      FLUX(2)=0.0
      WRITE(51,102) kstep,kinc,noel,npt,qpsave,flux(1),t11
      WRITE(51,103) coords(1),coords(2),time(1),time(2),
     1 temp,u(1),u(2)
  102 FORMAT(1X,4(I5,1X),e12.5,1x,e12.5,1x,e12.5)
  103 FORMAT(1X,7(e10.3,1x))

      RETURN
      END
```

```
      SUBROUTINE UVARM(UVAR,DIRECT,T,TIME,DTIME,CMNAME,ORNAME,
     1 NUVARM,NOEL,NPT,NLAYER,NSPT,KSTEP,KINC,NDI,NSHR)

c     This subroutine reads temperatures from previous increment

      INCLUDE 'ABA_PARAM.INC'
      CHARACTER*8 CMNAME,ORNAME,FLGRAY(15)
      DIMENSION UVAR(NUVARM),DIRECT(3,3),T(3,3),TIME(2)
      DIMENSION ARRAY(15),JARRAY(15)
      DIMENSION ETEMP(4,350)

c     In this case, t11 is temperature which is to be controlled

      COMMON /ETEMPE/t11,t131,ETEMP
      character*256 dirout
      save ts131,ts11
      JERROR = 0

c     GETVRM is internal ABAQUS subroutine which holds element variables

      CALL GETVRM('TEMP',ARRAY,JARRAY,FLGRAY,JRCD)
      UVAR(1) = ARRAY(1)
       if(noel.eq.1.and.npt.eq.1) then
        ts11 = uvar(1)
        t11 = ts11
      endif
       if(noel.eq.13.and.npt.eq.1) then
        ts131 = uvar(1)
        t131 = ts131
      endif
      ETEMP(NPT,NOEL) = UVAR(1)

      RETURN
      END
```

DISTRIBUTION

INITIAL DISTRIBUTION
UNLIMITED RELEASE

0320 C. E. Meyers, 1011

0740 R.T. McGrath, 9114
0826 W.L. Hermina, 9111
0828 P. Hommert, 9100
0835 S.E. Gianoulakis, 9113

9001 T. Hunter, 8000
Attn: J. B. Wright, 2200
A. West, 8200
W. J. McLean, 8300
R. C. Wayne, 8400
P. N. Smith, 8500
L. A. Hiles, 8800
D. Crawford, 8900

9004 M. E. John, 8100
9005 W. G. Wilson, 2204

9042 C. M. Hartwig, 8345
9042 W.G. Houf, 8345
9042 M.P. Kanouff, 8743
9042 R. J. Kee, 8303
9042 E. Meeks, 8345
9042 C. D. Moen, 8345
9042 P. E. Nielan, 8742
9042 A.R. Ortega, 8743
9042 J. Robles, 8743
9042 J. Shon, 8345
9042 P. A. Spence, 8345 (10)
9042 L. I Weingarten, 8742 (5)

9054 W. J. McLean, 8300

9102 A. L. Hull, 8416
9102 D. M. Tung, 8416 (5)
9102 D. A. Sheaffer, 8416 (5)
9103 G. A. Thomas, 8111
9103 K. Schroder, 8111 (5)

9401 M. L. Callabresi, 8743
9401 T. M. Dyer, 8700

9021 Technical Communications Department, 8815, for OSTI (10)
9021 Technical Communications Department, 8815/Technical Library, MS8099, 4414
8099 Technical Library 4414 (4)
9018 Central Technical Files, 8950-2 (3)