NSTX-U Digital Coil Protection System Software Detailed Design

Keith G. Erickson, Gregory J. Tchilinguirian, Ronald E. Hatcher, and William M. Davis

***Abstract*—The national spherical torus experiment (NSTX) currently uses a collection of analog signal processing solutions for coil protection. Part of the NSTX upgrade (NSTX-U) entails replacing these analog systems with a software solution running on a conventional computing platform. The new digital coil protection system (DCPS) will replace the old systems entirely, while also providing an extensible framework that allows adding new functionality as desired. The development of the DCPS was a multidiscipline engineering effort. The fact that long-trusted yet presently inadequate protection mechanisms were being replaced with a first-of-a-kind system at NSTX-U has led to a carefully crafted, full-featured software design. Real-time concurrent RedHawk Linux provides the deterministic environment in which the software runs, and the software architecture follows a unified modeling language design with industry standard patterns.**

***Index Terms*—Digital coil protection system (DCPS), Linux, national spherical torus experiment upgrade (NSTX-U), real time, RedHawk, real-time operating system, unified modeling language (UML).**

I. Introduction

THE national spherical torus experiment (NSTX) [1] is currently undergoing a multiyear upgrade [2]–[4] that will expand the realm of possible scientific goals [5], [6]. An increased pulselength, new divertor coils, and doubling the field capacity (which quadruples the magnetic loads) all contribute to an increased need for protection of the hardware [7]. These protection systems serve as the last line of defense to shut down the power supplies before they cause damage. The existing hardware-based systems, while highly reliable, are costly to reconfigure and upgrade.

Thus, NSTX upgrade (NSTX-U) will replace the older coil protection system with a new digital computer-based solution that enables a flexible and extensible protection system at a lower cost. Historically, however, general purpose operating systems, commercial computers, and high level programming languages have been ill-suited for protecting equipment. Determinism, latency, throughput, and failure rate are only a few of the factors that tend to preclude choosing a fast GNU/Linux system in favor of an embedded device [8]. To overcome this ambiguity, the digital coil protection system (DCPS) computing system uniquely combines new technologies and modern design techniques to provide the flexibility of software without compromising the inherent safety of embedded hardware. The chosen technologies have matured to the point that they are more effective in the embedded world today than in years past.

Among other things, DCPS will include the following.

1) AMD Opteron based x86_64 architecture.
2) Concurrent RedHawk Linux based on RedHat enterprise Linux 6 [9].
3) Supermicro H8DG6-F motherboard with a dual 16-core CPU setup and 64-GB registered error checking and correcting RAM.
4) C++11 programming language with strict adherence to the standard [10].
5) Object oriented design techniques following the unified modeling language (UML) 2.4.1 standard [11].
6) Industry standard design patterns.
7) Commonly available analog input cards from general standards (16AI64SSC) and a digital input/output card from Adlink (7296).
8) 200-μs cycle time on input data.

II. NSTX-U DCPS System Requirements

A. Fault Logic

DCPS has two concurrent outputs: a fault signal, and a heartbeat. It will continually send a heartbeat signal to an external device to validate its own health, but will not normally output a fault signal. The loss of said heartbeat or the existence of said fault signal signifies a degraded ability to prevent damage to the system, and thus triggers an immediate NSTX-U shutdown. This two factor approach ensures the ability of DCPS to operate in a failsafe manner.

B. Coil Protection

DCPS will protect NSTX-U during a plasma attempt, or pulse, by running a collection of algorithms [12], [13] against the plasma current and the 16 magnetic coil currents every 200 μs. A fault occurs if the result of any algorithm exceeds a preprogrammed minimum and maximum limit. A fault also occurs if the system determines that a fault could occur before the next time cycle given a worst case projection. Finally, a fault occurs if a disruption before the next time cycle would cause any algorithm to exceed its limit value.

Between plasma attempts, the DCPS must monitor all of the currents in the system and ensure that they remain at zero. Any current in the system prior to the start of the next plasma

Manuscript received August 12, 2013; accepted April 28, 2014. Date of current version June 6, 2014. This work was supported by the U.S. DOE under Contract DE-AC02-09CH11466.

K. G. Erickson, G. J. Tchilinguirian, and W. M. Davis are with the Princeton University Plasma Physics Laboratory, Princeton, NJ 08543-0451 USA (e-mail: kerickso@pppl.gov; gtchilin@pppl.gov; bdavis@pppl.gov).

R. E. Hatcher, deceased, was with the Princeton University Plasma Physics Laboratory, Princeton, NJ 08543-0451 USA (e-mail: rhatcher@pppl.gov).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TPS.2014.2321106

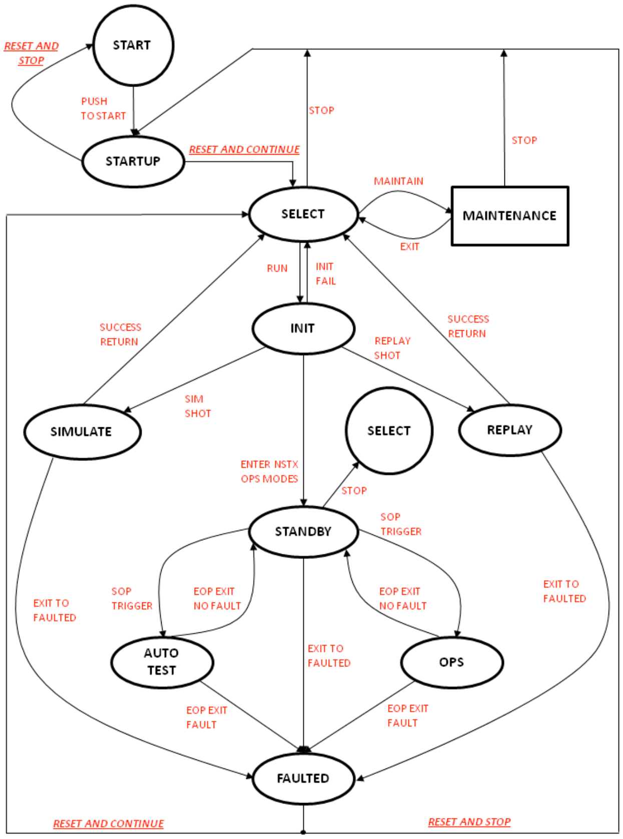

Fig. 1. DCPS finite state machine.

attempt will similarly result in a fault that prevents the pulse from occurring [14].

C. Finite State Machine

Central to the DCPS framework is the finite state machine shown in Fig. 1 that contains ten possible states reflecting the system being in one of four modes: plasma operations, auto test, simulate, and maintenance. Plasma operations, as the name implies, is the real operating mode for actually protecting NSTX-U. Auto test is for attaching an external simulator, whereas simulate is for running internal simulations. While the first three modes represent different ways to run the system, the last mode, Maintenance, is for modifying the runtime characteristics of DCPS [15].

III. DCPS System

A. Operating System Choices

Embedded systems historically used hardware dedicated to a specific task, as opposed to a more general purpose platform. While GNU/Linux is growing in popularity outside of its traditional server/workstation role, it remains ill-equipped for a sub-millisecond hard real-time device. The ability of the kernel to meet a deadline is predominantly a measure of process dispatch latency (PDL). The Linux kernel by itself has no latency guarantees at all, and PDL delays due to interrupt handling easily exceed 100 ms. Created as a workstation operating environment, the Linux scheduler tends to favor servicing many simultaneous processes in a fashion that delivers quick response time to a physical user. An embedded device by comparison would rather preempt I/O tasks-like disk activity in favor of servicing the real-time application and meeting the timing deadline at stake.

There are two mainstream real-time versions of Linux that overcome the PDL deficiencies: RedHat MRG and concurrent RedHawk. DCPS uses the latter, as it includes unlimited support, real-time I/O drivers, and a NightStar toolset that enables the DCPS development team to monitor, tune, and debug the system with orders of magnitude less effort compared with the conventional tools. Without NightStar, system tuning becomes a much more arduous task requiring many iterative test and check cycles with invasive recompiling and reconfiguring. Using the provided tools, however, it is possible to dry run hundreds of scenarios in several hours. Applying this approach to a prototype version of DCPS on NSTX-U, for example, reduced timing analysis efforts for a two man-week task to a matter of minutes. DCPS will therefore take advantage of these experiences and the benefits that RedHat MRG cannot provide.

Both systems provide deterministic capability using different techniques, described in the following section.

B. Kernel Modification Methods

There are two main approaches to solving the determinism problem investigated for DCPS: a kernel modification developed by Ingo Molnar called PREEMPT_RT [16], and a technique to aid scheduling concerns called CPU shielding [17], [18]. The kernel modification approach tends to be system wide, while the CPU shielding approach focuses on just the real-time processes. It is therefore a less intrusive and more forward compatible choice, which is a large driving factor in the DCPS design.

In kernel preemption, the technique used in RedHat MRG, the user application has the ability to preempt a kernel thread scheduled for the same CPU on which it is currently trying to run. Without this patch, the kernel is strictly not preemptible. The user application is at the mercy of any possible kernel event, such as an asynchronous interrupt request (IRQ) that needs to run for 50 milliseconds during the real-time event's 50-μs inner loop. Obviously, the real-time process will completely miss its deadline (effectively, it will miss it 1000 times in a row). For a protection system, this is catastrophic. The PREEMPT_RT patch modifies the kernel and allows the user application to stop the IRQ from running so that it can service its own event instead. Unfortunately, an IRQ has to run eventually, and no amount of preemption will alleviate the unending kernel tasks that keep a stable system operational. The patch is system-wide, greatly changing the entire scope of the kernel runtime metrics. Because of the intrusive nature of the change, the patch, by the author's own admission, will reduce overall system throughput and kernel response times. It trades total performance for the ability to preempt kernel tasks.

Conversely, with a simplistic CPU Shielding approach, the user application and the kernel share the available CPU cores, dedicating certain tasks to specific cores. The user application receives one or more reserved cores on which the kernel cannot schedule interrupts or other kernel threads. Instead, the kernel stays on nonreal-time application cores where it is free to tie up CPU time without affecting the real-time determinism. The user application on its dedicated cores can effectively latch the CPU in a spin-wait or sleep in a blocking idle state without fear of another thread taking control. This effectively reduces dispatch latency to near zero, as there is never any resource contention in terms of processor allocation. The benefit to this approach is that the core allocation for kernel and user space remains fixed, defined at the beginning. The kernel always has a place to run its own interrupt routines, and the user always has a place to run its real-time loop. There is never any preemption one way or the other, and therefore there is no performance sacrifice. This creates a far more copasetic relationship between the kernel and the user.

Since the DCPS computer has a large core count, it is easy to organize the various tasks such that each main thread receives a dedicated core shielded from all other system activity. Operations such as I/O card interaction, watchdog monitoring, and algorithm processing can all operate on an isolated core with no overhead. Doing so still leaves spare cores to conduct normal system operations, such as handling IRQs, running user shells, and managing background services.

Of course, neither of the two options, shielding or preemption, removes the need for appropriate real-time programming techniques that manage resources outside of CPU cycles. Memory allocation, bus contention, I/O, and system calls all still pose a threat to determinism. However, CPU shielding greatly reduces the difficulty associated with these tasks.

C. Deployment Model

DCPS consists of two processes, a core and a client, that communicate over a standard SSL encrypted TCP/IP socket. The core is a multithreaded process written in C++11 that actually runs the coil protection mechanisms. The client is a separate process written primarily in Qt 5/C++ and possibly running on a different machine that connects to and controls aspects of the core. Communication between the client and core uses Google's open source project, Protocol Buffers, for object serialization and ZeroMQ for the socket transport library. These two complementary technologies are efficient, well maintained, and compatible with current object oriented programming languages (C++, Python, and Java). Since the core and client languages are predominantly C++, this presents an easy way to communicate between the two distinct applications.

DCPS incorporates three physical computers to operate. The main computer that runs the core is a fast 32-core commercial off the shelf solution running the tuned RedHawk operating system. The client connects to the core from a terminal that will typically be in the main NSTX-U control room, but can theoretically reside anywhere the virtual network rules allow. The third computer is the database storage machine, a highly protected, highly restricted machine hidden behind multiple security layers. This machine houses all of the protection data required to operate NSTX-U and DCPS. While read access to this data will be readily available, write access will instead require multiple levels of authentication combined with physical access restrictions.

D. System Inputs

DCPS receives three kinds of input from external sources. There is a 5-kHz clock signal and several discreet clock events driving individual interrupt lines on the real-time clock and interrupt module (RCIM). There are several digital inputs on a digital I/O card to handle resetting and overriding faults. Finally, there are 64 differential analog input channels spread across two cards. The signals consist of statuses, triggers, and most importantly the instantaneous currents in each coil as well as the plasma current. There are two channels for each current, duplicated for redundancy and sent to different cards.

E. System Outputs

There are far fewer outputs from the system compared with the vast and varied array of inputs. Primarily, outputs consist of two types of failure signals: a level 1 fault, and a watchdog timer. In practice, the level 1 fault line is actually four-independent fault lines, one for each coil system employed in NSTX-U. These only trigger high when a protection algorithm trips a limit value. Otherwise, they stay low. The watchdog output is on the other hand a regularly oscillating signal, alternating high and low with every successful real-time cycle completion. Missing this heartbeat signifies the unreliability of DCPS to protect NSTX-U, and thus an immediate shutdown occurs.

Aside from the primary outputs, there are also a few status outputs to signify various modes in which the DCPS might operate. For instance, with the external AutoTester attached, a corresponding output alerts other external devices that the DCPS software thinks it is in a test mode. This is useful as a sanity check to prevent crossover between real operations and testing.

IV. Software Design

A. Design Methodology

Any moderately complex software application requires accurate documentation and developer coordination. The object modeling group created the UML [11] as an effective way to communicate software designs between various stakeholders: customers, end users, designers, engineers, developers, and so on. DCPS documentation fully exploits UML version 2.4 to both identify the users and describe the software requirements, code design, and eventual deployment.

Use of modeling such as UML encourages the subsequent application of reusable design patterns that are standard in the industry. These patterns provide building blocks to form more complicated structures without reinventing commonly used foundations. They are typically language agnostic, preventing the overall design from dictating the eventual implementation.

There are six discrete components that make up the DCPS software (Fig. 2): system management, data management, algorithm management, monitor, security, and the user-interface. Each component is an individual entity with a

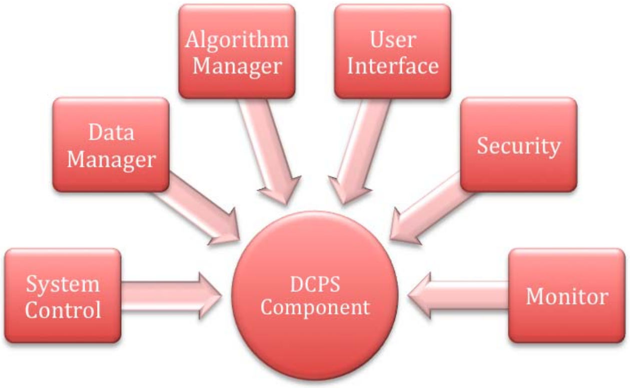

Fig. 2. DCPS component layout.

separate implementation, usually exposing itself to the remaining components via the Façade design pattern.

B. System Manager

Orchestrating the effective interaction of independent components requires that something guarantees each component is working correctly. The system manager component (SMC) starts, stops, and monitors each of the other components. It manages the state transitions of each component in accordance with the overall DCPS finite state machine model, and brokers the communication infrastructure between the components. Finally, it monitors the activity of each component for purposes of accurately reporting the heartbeat that reflects the internal integrity of the system.

The SMC implements the Façade design pattern to provide a single application programming interface (API) to the other components in the system. Through this façade, each component can report its state or communicate system changes. The SMC itself controls its own state through the same API in a self-reflective manner.

C. Data Manager

The data manager component (DMC) component is the largest of the six, both in scope and complexity. It handles two forms of data: the three types of hardware I/O, and the software database backend.

1) Hardware I/O: At the lowest level, it receives and sends all of the input and output across the PCIe I/O cards that connect DCPS to the outside world. This includes initializing and configuring each card and running several threads to continually move data on and off the various cards. The input side is a combination of analog signals, digital signals, and interrupts from the RCIM.

The link that synchronizes reads between all inputs across two card types and three cards total is the RCIM. The NSTX-U facility clock strobes the RCIM in synchronization with the rest of the NSTX-U system. The RCIM has software hooks to trigger user space code without requiring kernel space interrupt handling routines, translating into dispatch latencies in the order of 2 μs in heavily loaded testing scenarios. The user code then polls each input card simultaneously, and eventually makes the data available to the rest of the system.

All the analog channels require postprocessing at multiple levels. First, the DMC must perform baseline subtraction and calibration for each channel. This removes integration error and magnetic co-interference. Then, there is an auctioneering process that compares each set of duplicated currents and chooses the larger of the two. The design model errs on the side of caution, assuming that a larger current is a more stressful condition for the machine. This final set of auctioneered, calibrated, and subtracted set of currents is the main data set that the DMC provides to the rest of DCPS.

The digital signals are much simpler in both scope (fewer used channels) and complexity (no postprocessing), however, one card shares both input and output. The card supports 96 total channels divided in half for 48 input and 48 output channels. While the total count is a lot, DCPS currently only uses a small number of both inputs and outputs. The rest remain for future expansion. Nevertheless, the simple nature of digital input is such that once read, they require no postprocessing.

2) Software Database: The software side of the DMC consists of a database backend, MDSPlus, and a service oriented front end for the rest of the system to abstract out the inner workings of MDSPlus. The database stores preshot data to configure the pulse and postshot data to record events during the pulse. Preshot data mostly consists of the configuration information for the algorithms, such as limit values, coefficients, and algorithm scheduling. Postshot data encompasses everything required to recreate the pulse in a simulated environment, as well as any debugging or logging information and intermediate calculated algorithm values required to diagnose issues that may arise during a pulse.

a) MDSPlus security concerns (concurrency): MDSPlus is inherently insecure in its handling of precious data. Even when exploits are accidental and not malicious in nature, MDSPlus makes it easy to affect the integrity of shot data. For instance, a user creating a new shot for testing purposes can easily overwrite data belonging to someone else by merely typing the wrong number in. No checks exist, for instance, to ensure that a user is within his own sandbox.

DCPS employs several mechanisms to bolster the security situation. First is the forced atomicity of shot tree creation. Historically, during testing, a user would create a new test shot manually using a numerical series outside normal operations. This has undefined behavior, however, when two users try to use the same number. Different projects have developed different methods to address the issue, including assigning number ranges to specific people. However, no scheme stops an accidental typographical error from destroying someone else's data. Therefore, NSTX-U instead has a scheme by which a user can atomically request a new test shot, and have that number atomically transferred to the test program without user interaction. The atomic nature of the request prevents any two users from receiving the same number.

b) MDSPlus security concerns (data access): Another MDSPlus concern is the data store itself. It proved challenging to maintain usability under the current permission system that MDSPlus employs. Instead, DCPS will use a secondary data server hidden behind the main MDSPlus host that serves data for all of NSTX-U. This secondary server provides two functions: extra security due to restricted permissions and data hiding, and contingency against network failures, as the main DCPS program has a direct patch cable link on a secondary Ethernet interface to the data server itself. This dual path access ensures that during an actual test shot, any transient network issues will not affect the real-time operation of DCPS and thus NSTX-U.

Because the operational DCPS has direct access to the secret data server, and knowing that it is possible to run simulation versions of DCPS elsewhere in the networking infrastructure, the DMC abstracts out the identification of and connection to this server. This abstracted nature reduces the complexity of the code and of the user interaction, since neither requires knowledge of the actual route taken to access the secret server.

c) Database contents: There are two main components to the data that DCPS stores in MDSPlus. The first is the preshot data known as parameter data. The second is postshot data, consisting of every conceivable piece of interesting data from a test shot.

The parameter data are highly controlled data representing all of the settings required to protect the NSTX-U coils. This includes algorithm limits, coefficients, threading priorities, and which algorithms to run, among many other settings. Changing this data in an adverse way could prevent the ability of DCPS to protect the coils and possibly damage the system. Therefore, it is vital that the data be under close scrutiny and tight controls. Alongside software restrictions and physical separation, NSTX-U will also employ strict operating procedures as another layer of protection. Finally, the DMC will contain hard-coded values on a per-algorithm basis to prevent truly outlandish limits and coefficients.

The postshot data contains mostly time-based data collected for every 200-μs cycle. This includes the result of every algorithm calculation, intermediate calculated values, faults, all of the raw input data, the calibrated version of the input data, and more. All of this resides in the MDSPlus tree following a strict organization that allows easy retrieval for numerous different offline analysis programs.

D. Algorithm Manager

The algorithm manager component (AMC) controls the core of the DCPS protection mechanism. For every 200-μs time step, the AMC processes a complete set of algorithms. Each algorithm checks against two predetermined limit values, a minimum and a maximum, and potentially generates a fault. At the conclusion of each time step, the AMC sends all faults to the DMC for output to the hardwire control system, which ultimately will terminate the pulse. There is built-in monitoring to ensure that algorithms do not exceed an allotted run time, and a method to adjust the runtime characteristics of the algorithm processing allocation before the pulse.

1) Algorithm Manager Design: An algorithm factory (using the factory pattern) hides the algorithm instantiation, and thus the algorithm type, from the rest of the AMC. It employs a strategy pattern to bind the calling API of a given algorithm instance to a standard signature shared by all algorithm types. This hides any differences in the underlying algorithms, and allows the dispatcher to remain algorithm-agnostic.

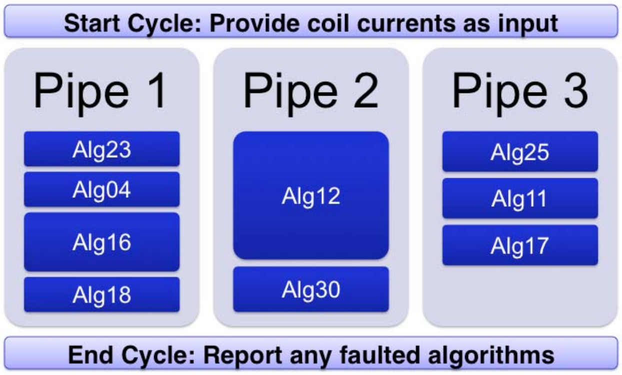
Start Cycle: Provide coil currents as input
Pipe 1
Alg23
Alg04
Alg16
Alg18
Pipe 2
Alg12
Alg30
Pipe 3
Alg25
Alg11
Alg17
End Cycle: Report any faulted algorithms

Fig. 3. Algorithm pipelining scheme.

Each strategized algorithm instance created by the factory runs in a pipeline, possibly shared with other algorithms. The pipelines employed here are object pools, another design pattern, locked to a thread running on a dedicated core. Based on preshot data (parameter data) from the DMC, the AMC assigns each algorithm to a specific pipeline created from the object pool, as shown in Fig. 3.

This unique combination of four standard design patterns, factory, strategy, Façade, and object oool, results in a system that can allocate and dispatch arbitrary tasks to processing queues without any internal knowledge of the task itself. This is a powerful generic tool with application outside of DCPS.

2) Pipeline Synchronization: Each pipeline of discreet tasks must execute in parallel, yet also synchronize between each time step. Traditional multithreading involves keeping the work queue of each thread full to maximize work output, and waiting on a mutex lock when there is no more work to allocate. This method is not ideal for a cyclical real-time mechanism that requires both determinism and synchronization. The pipelines must be deterministic in that they must all start without delay at the beginning of each time step. They must synchronize with each other so that they all start at exactly the same time. To accomplish this, the AMC has one manager thread, multiple worker threads each containing a single pipeline, and a bidirectional synchronization mechanism to create a concurrency barrier both at the beginning and at the end of every time step.

At the start of every time step, the manager thread sends a notification to each worker thread to start processing their work queues and waits for a response. The workers then send notifications back to the manager to indicate completion of the cycle, and wait for another notification to start. Waiting in this context implies a spin-wait that prevents releasing a CPU when no work remains. Since each worker thread and the manager thread has exclusive access to a single CPU, this allows instantaneous start up once the new cycle begins. The RedHawk tools discussed earlier ensure that absolutely nothing else runs on these CPUs, including operating system interrupt handlers and the system timer.

The manager thread does not actually distribute work to each worker thread using this design. The workers assemble their task queues before the test shot in a fixed fashion such that the work processed by each pipeline does not change throughout the entire shot.

3) Algorithm Types: There are currently five types of algorithms that the AMC can handle, however, the design is such that adding new algorithm types is easy and expected for future growth. Each algorithm type can have any number of algorithm instances, each with its own set of coefficients and limit values. Some algorithms require the outputs of other algorithms as additional input, resulting in a dependency tree that prevents running.

a) Current predictor: The first algorithm that always runs in a cycle is the current predictor. There is no limit value for this algorithm. Instead, it provides two sets of currents for all future algorithms: the now currents, and the predicted currents. First, it auctioneers between the redundant input currents and takes the highest of the two. This is the current in each coil for the currently executing time step that the remaining algorithms will use

$$I_{\rm PD} = I + \underline{L}^{-1}\underline{\rm M}{\rm I}_{\rm PL}. \tag{1}$$

Then, it applies an influence matrix to those currents to determine two possible predicted currents. In this phase of the algorithm, the objective is to predict what the current would be should a disruption occur before the next opportunity to execute a time step. The two possibilities depend on the shape of the plasma cross section, which is initially limited to either a circular plasma or an elongated plasma.

b) Action integral:

$$A_k = A_{k-1} + I_k^2 \Delta t \tag{2}$$

$$A_F = A_k + I_k^2 \tau/2. \tag{3}$$

Action integrals estimate the conductor temperature rise in the coils, commonly referred to as $\int I^2(t)dt$. There are two action integrals: the total action for the current time step k based on the action from the previous time step $k-1$ (2), and an estimate of the additional action that would accumulate if a fault were to occur and the current were to decay exponentially from the present state (3). In this context, Δt is the time between each time step, and τ is the L/R time constant of the circuit under consideration.

Since it does not make sense to compute action for postdisruption currents, this algorithm only uses the currents for the currently executing time step as its input current vector.

c) Forces and moments (torques):

$$X = wI\Sigma_j(C_j I_j). \tag{4}$$

This is the first of the more general algorithms that the DCPS will execute. This same formula will calculate radial force (F_r), vertical force (F_z), and torque (T) for both the currents in currently executing time step as well as the predicted postdisruption currents. X represents each of those results. For each X, there is a separate set of coefficients C and an overall weighting factor w. I is the current in the coil for which we are calculating X, and I_j is the vector of all of the currents in the system.

d) Derived type I:

$$Y_A = K + \Sigma\left(C^I I + C^A A + C^{Fr} F_r + C^{Fz} F_z + C^T T\right). \tag{5}$$

The first of the derived type algorithms is a weighted sum of all previously calculated values. There is a separate set of coefficients (C^x) for each value type, and each corresponding X in the $C^x X$ products ranges over all of the coils in the system.

e) Derived type II:

$$Z = \surd(Y_A^2 + Y_B^2 + \cdots + Y_J^2). \tag{6}$$

The second of the derived type algorithms is a square root of the sum of squares of all previously calculated derived type I algorithms. This is not currently used, but is available for future growth.

E. Security

The security component (SC) provides a service to the rest of the system, in contrast to the several manager components that operate independently processed tasks. It defines and enforces a set of permissions that restrict user actions given a combination of user type, system state, and other key factors. Other DCPS components use the SC to check if a requested user action is permissible at a given time. For instance, it might harm the system if a user switched to test mode during a pulse. Likewise, it would be counter intuitive to allow every user to modify the algorithm run list.

Also unlike other components, the implementation of the SC spreads across disciplines. Parts of the security model incorporate tools outside of the source code. For instance, to group users into eight user types with inheritable hierarchies, standard UNIX groups fit well inside the existing security infrastructure of the laboratory. Processes already exist to control user group mappings, requiring authorization, sign off, and auditing. Therefore, the SC provides a gateway to access the standard UNIX group permissions via pluggable authentication module instead of providing its own custom set. Similarly with network access, the model integrates existing network security infrastructure in terms of virtual networks and firewalls to reduce the number of devices that can try to access the operational DCPS software.

F. Monitoring

The monitoring component (MC) provides an interface for the rest of DCPS to report status to the outside world via several means. It can log debugging information to a file, populate EPICS displays, or send feedback to the user-interface component (UIC).

Typically, logging implies writing out successive lines of text to a file to aid in tracing the order in which events occur. There are different levels of log details such as error, warning, informational, and one or more levels of debug with increasing verbosity. RFC5424 from the Internet Engineering task force defines eight logging levels which the MC will implement. The various levels allow filtering based on the characteristics of a given test. For example, an error indicates an identified problem causing a failure, whereas a warning is something that might be a concern but is not catastrophic. Informational messages help tagging events in a timeline (initialization complete, shot started, etc.), and debug messages only serve a transient purpose while a developer traces down a problem. Debug messages tend to be more intrusive to real-time operations, either because of a high frequency or because of overhead associated with crafting the specific line of text.

Logging on a real-time system presents a challenge due to the nondeterministic nature of writing out files. Whether the application stores files on a local disk, a network mount, or some other medium, writing to the files still requires kernel system calls that disrupt deterministic real-time processing. There are two approaches that, when combined, alleviate this challenge.

For the first approach, the MC will do any output in a low priority thread on a dedicated CPU. Conversely, the input will arrive in a high priority thread that queues the writes to the low priority thread. This separation between priorities provides a mechanism that keeps short tasking at a high priority and long tasking at a low priority to optimally allocate the system resources.

The second approach involves short circuiting disabled log entry function calls to avoid unnecessary processing. For instance, consider a highly system intensive code path containing a call to the logging API with a logging level of debug. If creation of the logging string is intrusive, the logging API should not only prevent recording the eventual log string, but it should also prevent creating the string in the first place, thus saving the overhead of building a string that it will never use.

G. User Interface

The UIC is the primary means by which a DCPS user performs all possible actions. It is likewise the primary component of the DCPS Client. User actions include starting and stopping the system, adding and modifying algorithms, changing the runtime mode, and building simulation scenarios using a waveform editor. The UIC is unique in DCPS from a deployment standpoint, as it can run on a physically separate computer. It communicates to the rest of the core system via a secured socket administered by the SC.

Since the UIC is the bridge connecting a physical user to the rest of DCPS, it naturally is the largest customer for the SC. The UIC continually asks the SC for permission to allow actions, and modifies the display accordingly. For instance, during a pulse, buttons to change the DCPS Core mode turn gray and stop accepting input. Though it does not preclude additional security checks further downstream, this extra layer of security does inhibit many potential errors that might otherwise arise oscillating.

The UIC implementation uses the Qt widget framework to streamline GUI design and refocus efforts from coding details to graphical window content and purpose. Editing Qt windows and their contents is minimally invasive, and enables a dynamic communication between the developer and customer. Communication between the physical nodes combines a transport package called ZeroMQ with object serialization software called ProtoBufs. These two technologies handle serializing arbitrary objects into a string of bytes, moving those bytes through sockets between the computers, and deserializing them back into the same objects on the other side.

V. Conclusion

NSTX-U will replace the existing coil protection solution with a software-based DCPS. It will make use of concurrent RedHawk to achieve real-time performance on a GNU/Linux system, as it outperforms RedHat MRG in determinism, throughput, and overall development cost. The software design is flexible enough to allow dynamic changes to runtime characteristics, and extensible enough to provide an avenue for future growth in the form of new algorithms and algorithm types.

DCPS will naturally expand in the future to accommodate plasma goals. Future work further includes adding a regression tester that will automatically validate new changes against a database of previously fixed bugs to reduce the probability of reintroducing the same bug again. Additionally, DCPS can possibly expand its reach from coil protection to machine protection. Finally, in the short term, parts of DCPS will run on the plasma control side with stricter limits to enable controlled shutdowns instead of the current method of simply turning the power supplies off.

Acknowledgment

C. Neumeyer created the top level system requirements document for the overall NSTX-U DCPS [14]. R. Hatcher created the software requirements document [15] from which this design derives. P. Sichta and S. DeLuca created a new external timing unit to drive DCPS in synchronization with the rest of the NSTX-U operation [19].

References

[1] M. Ono *et al.*, “Exploration of spherical torus physics in the NSTX device,” *Nuclear Fusion*, vol. 40, no. 3Y, p. 557, 2000.
[2] C. Neumeyer *et al.*, “National spherical torus experiment (NSTX) center stack upgrade,” in *Proc. 23rd IEEE/NPSS Symp. Fusion Eng.*, Jun. 2009, pp. 1–4.
[3] J. E. Menard *et al.*, “Overview of the physics and engineering design of NSTX upgrade,” *Nucl. Fusion*, vol. 52, no. 8, p. 083015, 2012.
[4] J. Menard *et al.*, “Physics design of the NSTX-U,” in *Proc. 27th EPS Conf. Plasma Phys. P*, 2010.
[5] J. Menard and C. Neumeyer, “NSTX upgrade scientific motivation and project requirements,” *Process. Plasama*, vol. 318, pp. 15–16, Jun. 2009.
[6] S. P. Gerhardt, R. Andre, and J. E. Menard, “Exploration of the equilibrium operating space for NSTX-upgrade,” *Nucl. Fusion*, vol. 52, no. 8, p. 083020, 2012.
[7] L. Dudek *et al.*, “Progress on NSTX center stack upgrade,” *Fusion Eng. Des.*, vol. 87, pp. 1515–1518, Jun. 2012.
[8] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio, “Performance comparison of VxWorks, linux, RTAI and xenomai in a hard real-time application,” in *Proc. 15th IEEE-NPSS Real-Time Conf.*, Apr. 2007, pp. 1–5.
[9] J. Baietto, J. Korty, J. Blackwood, and J. Houston, “Real-time linux: The redhawk approach,” in *Proc. Concurrent Comput. Corp. White*, Sep. 2008.
[10] *Information Technology-Programming Languages-C++*, document ISO/IEC, ISO/IEC 14882:2011, International Organization for Standardization, Geneva, Switzerland, 2011.
[11] O. Uml, *2.4.1 Superstructure Specification*, document formal/2011-08-06, 2011.

[12] R. Woolley, P. Titus, C. Neumeyer, and R. Hatcher, "Digital coil protection system (DCPS) algorithms for the NSTX centerstack upgrade," in *Proc. IEEE/NPSS 24th Symp. Fusion Eng.*, Jun. 2011, pp. 1–8.

[13] P. H. Titus, R. Woolley, and R. Hatcher, "Stress multipliers for the NSTX upgrade digital coil protection system," in *Proc. IEEE/NPSS 24th SOFE*, Jun. 2011, pp. 1–6.

[14] C. Neumeyer, "Coil protection system requirements document," unpublished.

[15] R. E. Hatcher, "Digital coil protection system software requirements document," unpublished.

[16] L. C. R. Gonçalves and A. C. de Melo, "Application testing under realtime linux," in *Proc. Linux Symp.*, Jul. 2008, p. 143.

[17] S. Brosky, "Shielded CPUs: Real-time performance in standard linux," *Linux J.*, vol. 121, no. 9, p. 21, 2004.

[18] S. Brosky and S. Rotolo, "Shielded processors: Guaranteeing sub-millisecond response in standard linux," in *Proc. Int. Parallel and Distrib. Process. Symp.*, Apr. 2003.

[19] S. DeLuca, P. Sichta, and G. Tchilinguirian, "Reconfigurable timing unit for NSTX-U," in *Proc. IEEE 25th SOFE*, Jun. 2013, pp. 1–4.

Keith G. Erickson joined the fusion community in 2010, after having designed and integrated several real-time weapon control systems for the defense industry since 2004. He was with the Princeton Plasma Physics Laboratory, Plainsboro Township, NJ, USA, where he was involved in the real-time plasma and power supply control functions of the NSTX project, and is an expert at modern real-time design.

Gregory J. Tchilinguirian received the B.S. degrees in computer science and cognitive neuroscience from Rutgers University, Brunswick, NJ, USA, and the New Jersey Institute of Technology, Newark, NJ, USA.

He was with AT&T Labs, Florham Park, NJ, USA, and joined the Princeton Plasma Physics Laboratory, Plainsboro Township, NJ, USA, in 2007. He has developed systems to acquire and analyze data for NSTX and LTX, and DOE funded nuclear fusion devices.

Mr. Tchilinguirian is a member of several cyber security technical and emergency response teams, the Computational Intelligence Society, and the Nuclear Plasma Society.

Ronald E. Hatcher , deceased, received the B.S. degree in electrical engineering and mathematics and the M.S. degree in electrical engineering.

He was a Principal Engineer with the Princeton Plasma Physics Laboratory, Plainsboro Township, NJ, USA, from 1984 to 2014. He specialized in designing and analyzing power supply systems for large fusion experiments such as the S-1 Spheromak, PDX, TFTR, and NSTX. He was leading the Digital Coil Protection System project for NSTX-U.

William M. Davis received the B.S. and M.S. degrees in atmospheric sciences.

He has been with the Computer Division, Princeton Plasma Physics Laboratory, Plainsboro Township, NJ, USA, since 1980, where he is currently the acting CIO. His current research interests include data acquisition, data analysis, and data visualization. He has authored papers in these areas for work on PBX-M, TFTR, and NSTX. He is an expert Interactive Data Language Programmer.