NSTX-U Digital Coil Protection System Software Design*

Keith G. Erickson, Gregory J. Tchilinguirian, Ronald E. Hatcher, William M. Davis
Princeton University Plasma Physics Lab, Princeton, NJ USA 08543-0451
Corresponding author e-mail: kerickso@pppl.gov.

***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 multi-discipline 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 UML design with industry standard patterns.

Keywords—NSTX-U; DCPS; Linux; RedHawk; RTOS; real-time; UML

I. INTRODUCTION

The National Spherical Torus Experiment (NSTX) [1] is currently undergoing a multi-year upgrade [2,3,4] that will expand the realm of possible scientific goals [5,6]. An increased pulse length, 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 hardware-based systems, while highly reliable, are costly to reconfigure and upgrade.

As a result, NSTX-U will replace the older coil protection system with a new digital computer bsed 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 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.

*This work supported by U.S. DOE Contract No. DE-AC02-09CH11466

Among other things, DCPS will include:

- AMD Opteron based x86_64 architecture
- Concurrent RedHawk Linux based on RedHat Enterprise Linux 6 [9]
- Super Micro H8DG6-F motherboard with a dual 16-core CPU setup and 64 GB Registered ECC RAM
- C++11 programming language with strict adherence to the standard [10]
- Object Oriented design techniques following the UML 2.4.1 standard [11]
- Industry standard design patterns
- Commonly available analog input cards from General Standards (16AI64SSC) and a digital input/output card from Adlink (7296)
- 200 microsecond 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 15 magnetic coil currents every 200 microseconds. A fault occurs if the result of any algorithm exceeds a preprogrammed 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 attempt will similarly result in a fault that prevents the pulse from occuring [14].

C. Finite State Machine

Central to the DCPS framework is a 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 utilized 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 role, it remains ill-equipped for a sub-millisecond hard real-time device. The ability of the kernel to meet a deadline is predominenately 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 milliseconds. Created for a workstation operating system, the Linux scheduler tends to favor servicing many simulatneous processes in a fashion that shows 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.

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. 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 approaches, described in the following section.

B. Kernel Modifications

There are two main approaches to solving this problem investigated for DCPS: a kernel modification developed by Ingo Molnar called PREEMPT_RT [16], and a technique to aid

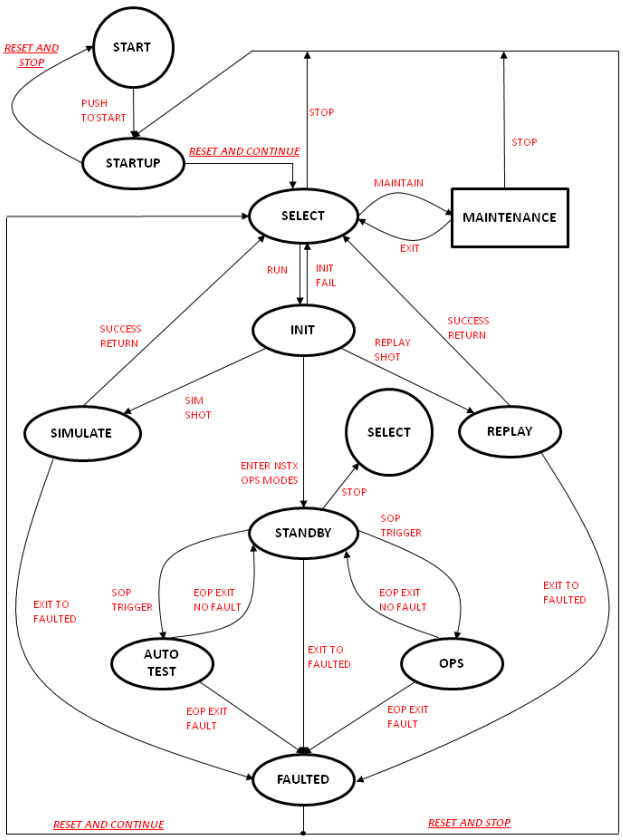

Fig. 1. DCPS Finite State Machine

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 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 *micro*second inner loop. Obviously, the real-time process will completely miss its deadline (effectively, it will miss it 1,000 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.

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 non-real-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 spinlock 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.

Neither approach 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 Python 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 technologies are efficient and complimentary, well maintained, and compatible with current object oriented programming languages (C++, Python, and Java). Since the Core language is predominantly C++ and the Client language is predominantly Python, this presents an easy and efficient way to communicate between 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 (COTS) 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.

D. *System Inputs*

DCPS receives three kinds of input from external sources. There is a 5 kHz clock signal driving an interrupt line on the Realtime Clock and Interrupt Module (RCIM). There are several digital inputs on a digital I/O card to handle reseting 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.

IV. Software Design

A. *Design Methodology*

Any moderately complex software application requires accurate documentation and developer coordination. The Object Modeling Group (OMG) created the Unified Modeling Language (UML) [11] as an effective way to communicate software designs between various stakeholders: customers, end users, designers, engineers, developers, et al. 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 (See Fig. 2): System Management, Data Management, Algorithm Management, Monitor, Security, and the User Interface. Each component is an individual entity with a separate implementation, usually exposing itself to the remaining components via the Façade design pattern.

B. *System Manager Component*

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 components.

The SMC implements the Façade design pattern to provide

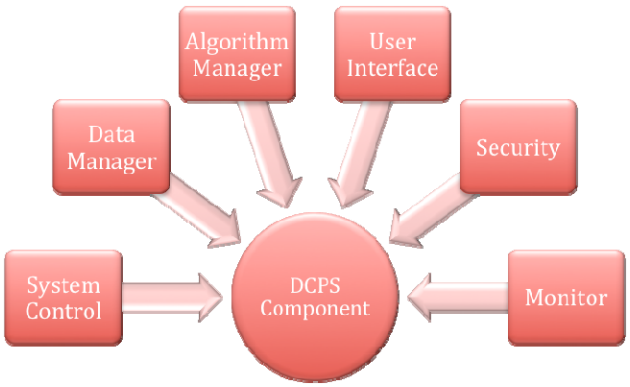

Fig. 2. DCPS Component Layout

a single Application Programming Interface (API) to the other components in the system. Through this façade, each componet 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 Realtime Clock and Interrupt Module (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 (NFC) 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 on the order of 2 microseconds 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 analog channels require post processing 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 post processing), 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 post processing.

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 pre-shot data to configure the pulse and post-shot data to record events during the pulse. Pre-shot data mostly consists of the configuration information for the algorithms, such as limit values, coefficients, and algorithm scheduling. Post-shot 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.

D. Algorithm Manager

The Algorithm Manager Component (AMC) controls the core of the DCPS protection mechanism. For every 200-microsecond time step, the AMC processes a complete set of algorithms. Each algorithm checks against a predetermined limit value, 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 take too long, and a method to adjust the runtime characteristics of the algorithm processing allocation before the pulse.

There are currently 4 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. 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.

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 pre-shot 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 Pool, 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.

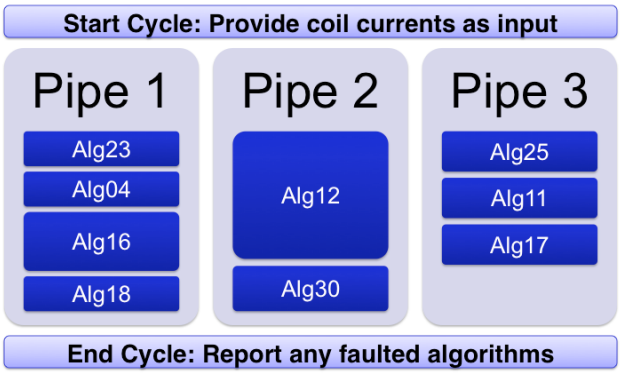

Fig. 3. Algorithm Pipelining Scheme

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. Therefore, the SC provides a gateway to access those standard UNIX group permissions 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.

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 non-deterministic 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 unneccesary 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 User Interface Component (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 doesn't preclude additional security checks further downstream, this extra layer of security does inhibit many potential errors that might otherwise arise.

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 developer and customer.

V. Conclusion and Future Work

NSTX-U will replace the existing coil protection solution with a software-based Digital Coil Protection System. 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 them. 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.

ACKNOWLEDGMENTS

Charles Neumeyer created the top level system requirements document for the overall NSTX-U Digital Coil Protection System [14]. Ronald Hatcher created the software requirements document [15] from which this design derives. Paul Sichta and Steve DeLuca created a new external timing unit to drive DCPS in synchronization with the rest of the NSTX-U operation [19].

REFERENCES

[1] Ono, M., Kaye, S., Peng, Y., Barnes, G., Blanchard, W., Carter, M., et al. (2000). Exploration of spherical torus physics in the NSTX device. *Nuclear Fusion*, *40*(3Y), 557.

[2] Neumeyer, C., Avasarala, S., Chrzanowski, J., Dudek, L., Fan, H., Hatcher, R., ... & Zhan, H. (2009, June). National Spherical Torus Experiment (NSTX) Center Stack Upgrade. In *Fusion Engineering, 2009. SOFE 2009. 23rd IEEE/NPSS Symposium on* (pp. 1-4). IEEE.

[3] Menard, J. E., Gerhardt, S., Bell, M., Bialek, J., Brooks, A., Canik, J., ... & Zolfaghari, A. (2012). Overview of the physics and engineering design of NSTX upgrade. *Nuclear Fusion*, *52*(8), 083015.

[4] Menard, J., Menard, J., Canik, J., Covele, B., Kaye, S., Kessel, C., et al. (2010). Physics design of the NSTX-U. *27th EPS Conf. on Plasma Physics P*.

[5] Menard, J., & Neumeyer, C. (2009). NSTX Upgrade Scientific Motivation and Project Requirements. *318*, 15-16.

[6] Gerhardt, S. P., Andre, R., & Menard, J. E. (2012). Exploration of the equilibrium operating space for NSTX-Upgrade. *Nuclear Fusion*, *52*(8), 083020.

[7] Dudek, L., Chrzanowski, J., Heitzenroeder, P., Mangra, D., Neumeyer, C., Smith, M., et al. (2012). Progress on NSTX center stack upgrade. *Fusion Engineering and Design*.

[8] Barbalace, A., Luchetta, A., Manduchi, G., Moro, M., Soppelsa, A., & Taliercio, C. (2007, April). Performance comparison of VxWorks, Linux, RTAI and Xenomai in a hard real-time application. In *Real-Time Conference, 2007 15th IEEE-NPSS* (pp. 1-5). IEEE.

[9] Baietto, J., Korty, J., Blackwood, J., & Houston, J. (2008). Real-Time linux: The RedHawk Approach. *Concurrent Computer Corporation White Paper (Sep)*.

[10] ISO/IEC, *ISO/IEC 14882:2011: Information Technology—Programming Languages—C++* (International Organization for Standardization, Geneva, 2011)

[11] UML, O. (2011). *2.4. 1 superstructure specification*. document formal/2011-08-06. Technical report, OMG.

[12] Woolley, R., Titus, P., Neumeyer, C., & Hatcher, R. (2011). Digital Coil Protection System (DCPS) algorithms for the NSTX centerstack upgrade. *Fusion Engineering (SOFE), 2011 IEEE/NPSS 24th Symposium on*. IEEE.

[13] Titus, P. H., Woolley, R., & Hatcher, R. (2011). Stress multipliers for the NSTX upgrade digital coil protection system. *Fusion Engineering (SOFE), 2011 IEEE/NPSS 24th Symposium on*. IEEE.

[14] C. Neumeyer, "Coil Protection System Requirements Document," NSTX_CSU-RQMT-CPS-159, unpublished.

[15] R.E. Hatcher, "Digital Coil Protection System Software Requirements Document," NSTX-SRD-13-163-0, unpublished.

[16] Gonçalves, L. C. R., & de Melo, A. C. (2008, July). Application Testing under Realtime Linux. In *Linux Symposium* (p. 143).

[17] Brosky, S. (2004). Shielded CPUs: real-time performance in standard Linux.*Linux Journal*, *121*(9), 21.

[18] Brosky, S., & Rotolo, S. (2003, April). Shielded processors: Guaranteeing sub-millisecond response in standard Linux. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International* (pp. 9-pp). IEEE.

[19] S. DeLuca, P. Sichta, & G. Tchilinguirian. (2013, June). Reconfigurable Timing Unit for NSTX-U. In *25th Symposium on Fusion Engineering, 2013. Proceedings*, unpublished.