Reducing Risk by Managing Software Related Failures in Networked Control Systems[⋆]

Girish Baliga[*], Scott Graham[◇], Carl A. Gunter[†], and P. R. Kumar[‡]

Abstract— **Managing risk is a central problem in the design and operation of networked control systems, and due to the increasing role and growing complexity of software in such systems, managing software related failures is becoming a central challenge. Even simple programming errors can cause catastrophic failures [1]. Hence, it is vital to contain risks due to software related failures in such systems.**

Our main thesis is that most software related failures can be managed through relatively simple and generally applicable strategies, and such strategies can be effectively developed and reused with suitable support from software infrastructure such as middleware. We describe mechanisms in Etherware, our middleware for control over networks [2], for containing software failures, and demonstrate the effectiveness of these mechanisms through experiments in a vehicular control testbed.

I. INTRODUCTION

As we continue to incorporate more advanced technologies and their associated control systems into our daily lives, there is a corresponding increase in both their capabilities to serve us, and their capability for damage when things go wrong. For example, an automated city-wide traffic system could enable much more efficient transportation in such highly congested environments as cities, perhaps avoiding waits at traffic lights and reducing energy waste. At the same time, we may also be able to better utilize the large capital investments in roadway systems. The realization of such a system would probably require automated vehicles capable of maintaining close and predictable formations through the use of various distributed sensors and complex control software. Unfortunately, a single programming error can result in fatalities very easily.

The role of software is becoming increasingly important in control systems, and the complexity of software design and implementation is contributing significantly to the risks in such systems. At the present state of understanding of software engineering, it is almost impossible to design software without errors [3]. Formal methods for program specification and verification [4] can help detect many errors in control software. However, there could still be errors in the model of software itself that could lead to failure. Hence, the presence of software errors in such systems must be a basic assumption in system design and analysis.

Since we may not be able to eliminate all software errors, we instead approach the problem from a different direction. We *contain* the failures caused by software errors, and *insulate* other system components from their impact. We accomplish this by modular software design, reduction of dependencies between various software components, employment of fail-safes and analytical redundancy through collation, and design of control hierarchies with the ability to perform supervisory overrides during critical failures.

Since most of these strategies are generally applicable to networked control systems, we advocate their incorporation into software infrastructure such as middleware that can be easily reused. This not only reduces software development cycle times, but also improves system safety as reused software has the benefit of diverse application and testing. Indeed, this has been one of the central goals for Etherware, a middleware that we have developed for control over networks [2], so that it effectively supports our strategies for management of software failures.

The rest of the paper is organized as follows. We begin by considering various classes of software related failures in Section II, and present our strategies for managing these failures in Section III. In Section IV, we describe mechanisms to support the implementation of these strategies in Etherware, and demonstrate the effectiveness of these mechanisms in Section V. We conclude in Section VI.

II. SOFTWARE RELATED FAILURES AND RISK

In this section, we consider various kinds of software related failures that can occur in networked control systems, and analyze the risks due to these failures.

A. Programming errors

Programming errors are a major reason for software failure. They may occur due to incorrect assumptions, faulty design, algorithmic errors, or coding errors. Since verifying the correctness of software is impossible in general [5], it is hard to detect all errors in software before they cause failures during system operation.

Even simple software errors such as assuming incorrect storage size for a programming variable can cause catastrophic system failures [1]. Hence, the risk due to programming errors can even be far greater than those due to

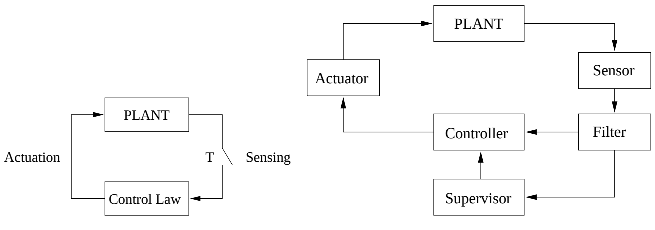
[⋆] This material is based upon work partially supported by DARPA/AFOSR under Contract No. F49620-02-1-0325, AFOSR under Contract No. F49620-02-1-0217, USARO under Contract No. DAAD19-01010-465, NSF under Contract Nos. NSF ANI 02-21357 and CCR-0325716, and DARPA under Contact No. N00014-0-1-1-0576. Prof. Gunter's research is also supported by NSF CCR02-08996, NSF CNS 05-24695, ONR N00014-02-1-0715, and ONR N00014-04-1-0562.

[*] Google Inc, 1600 Amphitheatre Pkwy, Mountain View, CA 95054, USA `email: baliga@google.com`.

[◇] Air Force Institute of Technology, Wright-Patterson AFB, Ohio 45433, USA. `email:Scott.Graham@afit.edu`.

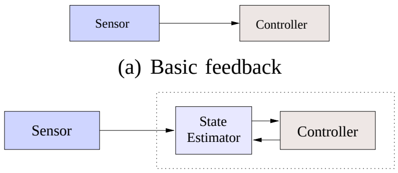
[†] Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA.

[‡] CSL and Dept. of ECE, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA. `email: prkumar@uiuc.edu`.



(a) Digital Control System (b) Software components

Fig. 1. Component based software design



(a) Basic feedback

(b) Enhanced feedback

Fig. 2. Local temporal autonomy

non-software errors such as signal noise. The inability to predict the occurrence of software errors makes managing their impact far more complicated.

B. *Passive failures*

Passive failures are failures that result in a loss of some system functionality. This includes failures of operational software, computing nodes, and communication links. Such failures are a consequence of imperfect software and hardware.

The risk due to a passive failure is usually the loss of functionality provided by the associated components. Hence, the threat due to these risks is determined by the criticality of the components that have failed.

C. *Active failures*

Active failures involve incorrect operation of software that could lead to faulty system operation. These are usually caused by programming errors, algorithmic errors, or even unexpected inputs, and can be fairly hard to detect.

Since active failures still produce what might be perceived as valid output by other parts of the system, their impact can propagate, and even lead to system-wide failures. Since active failures are much harder to detect, and can cause much more damage, they usually pose a higher risk than passive failures.

D. *Byzantine failures*

Byzantine failures are caused by willfully malicious agents operating within the system. Not only do they perform undesirable operation, but they may also be specifically designed to avoid detection. For instance, a rogue car in an automated traffic system can cause significant damage. Hence, the risk due to these failures can be significant.

III. MANAGING SOFTWARE RELATED FAILURES

In this section, we consider various strategies for managing the software related failures described in Section II.

A. *Component based design*

Programming errors are inevitable in complex software, and must be accounted for in system design. In particular, their impact must be contained as much as possible. This is usually accomplished through modular design, where the various software modules are designed in a decoupled fashion so that errors in one module do not cause failures in other modules.

Component based design of networked control systems software is a modular approach where the software is designed as a set of interacting modules, each implementing a well defined functionality. For instance, the software for the control system shown in Figure 1(a), can be designed as in Figure 1(b) with Sensor, Filter, Controller, Supervisor, and Actuator as software components. With such design, programming errors in a component such as the Supervisor can be contained and other connected components such as the Controller can be insulated from the resulting failures.

In the rest of the paper, we will assume component based design of software, and by the term component, we will mean software components as in Figure 1.

B. *Local temporal autonomy*

While many programming errors can be contained through component based system design, component failures such as passive failures can still affect other connected components. For instance, a failure of the Sensor in Figure 2(a) can affect Controller as it directly depends on receiving periodic and timely feedback from the Sensor.

Local temporal autonomy is the property of software components to tolerate failures of other connected components for a short period of time. This allows sufficient time for the failed components to recover, and resume normal operation. The State Estimator in Figure 2(b) provides reasonably accurate, though deteriorating, estimates of system state to the Controller so that it can operate for some time after a Sensor failure. This local temporal autonomy of the Controller prevents the failure of the Sensor from cascading into larger system wide failure.

Although local temporal autonomy is useful as a buffer against failure, it cannot be employed indefinitely. However, if it can operate safely long enough for the Sensor to be recovered from failure, the failure can be isolated from the rest of the system. Such techniques can be used to provide local temporal autonomy to most components in networked control systems, and significantly improve system robustness in the presence of software related failures [6].

In general, local temporal autonomy insulates components from passive failures of other connected components, and enables recovery from such failures while maintaining system stability.

C. *Collation*

Longer component disruptions may not be manageable through local temporal autonomy. In such cases, simple failsafes can be enforced until the disrupted components can be recovered. For instance, during a disruption of the Controller in Figure 1(b), the Actuator may enforce a simple fail-safe to maintain the plant in a safe state.

Active failures, however, can have a more serious impact on system operation, and more sophisticated strategies are needed to address them. Simplex [7] is an elegant architecture for managing active failures of controllers. In Simplex, two controllers are employed: a simple robust controllers, and a complex and possibly defective controller. The complex controller usually has better performance and operates the system most of the time. However, a supervisor constantly monitors the state of the system and can quickly switch to the simple controller if the system approaches some measurable level of instability due to active failures in the complex controller.

The Simplex architecture can be generalized to a Collation [8] based approach to software failure management. In addition to providing a Simplex like functionality, Collation can be employed iteratively in an operational system allowing for on-line testing of new components. With a reliable backup component, perhaps a Controller, continually guarding against a new component failure, it is safe to evolve an operational system. When the newer component (Controller) has been evaluated to be safe, it may in turn serve as the reliable backup to a future new Controller which may either operate more efficiently or provide additional functionality. For large networked control systems, wholesale replacement of the system may be cost prohibitive. Collation affords the ability to evolve, leading to greater overall system stability throughout the lifetime of the networked control system.

D. *Security overrides*

Byzantine failures require a more security oriented approach to minimize and contain the impact of malicious agents. Networked control systems can be designed with multiple layers of control hierarchy [8], where lower level controllers control local tasks, and higher level controllers supervise larger sub-systems. For instance, in an automated traffic system, the car controllers control individual cars, while traffic supervision components control the flow of traffic in larger traffic domains.

During normal operation, lower level controllers exert discretionary control to perform local tasks, while higher level controllers perform more coarse grained supervision. However, when a malicious agent is introduced in the system, the local controllers may not have sufficient information to operate effectively. For instance, when a rogue car is introduced in an automated traffic system, individual car controllers may not be able to respond effectively. In fact, the situation may be made worse by local greedy response, causing traffic jams and system instabilities. In such situations, there must be mechanisms for higher level controllers to intervene and exert mandatory control to best address the threats. For instance, the traffic supervisory components could take over control of individual cars to slow them down and allow emergency response vehicles such as police cars and ambulances to respond more effectively.

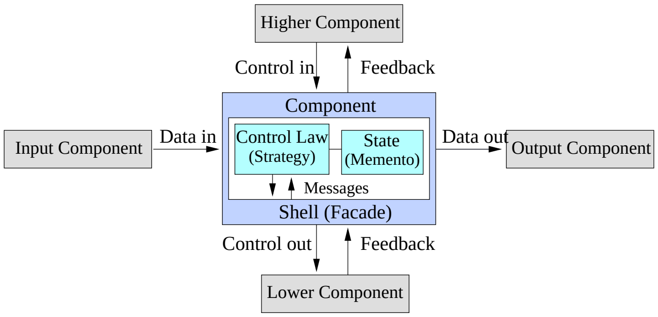


Fig. 3. Generic component in an Etherware based application

An important point to consider during security overrides is that such overrides must preserve low-level safety mechanisms. For instance, the mandatory traffic supervision must still defer to low level safety mechanisms such as collision avoidance, else the mandatory control might actually increase the risk of failure as we demonstrate in Section V-C.

IV. MIDDLEWARE BASED MECHANISMS

The software fault management strategies outlined in Section III are fairly general, and can be reused across most networked control systems. Such reuse not only reduces the time and effort required during system design, but also improves the overall software quality as reused software is more likely to be well designed and tested in operation. However, this requires well designed software infrastructure that provides mechanisms to support effective reuse. In this section, we provide a brief overview of Etherware, a middleware for networked control systems, which we have developed as software infrastructure. In particular, we describe Etherware mechanisms that support the development and reuse of software fault management strategies from Section III.

A. *Etherware*

Etherware is a message oriented component middleware for networked control systems. It is software infrastructure for such systems, and supports the development of component based software for such systems.

A generic component in Etherware is shown in Figure 3. It consists of a control law that implements the functionality of the component, and a state object that specifies the current software state of the component. These constitute the software that need to be implemented by the control software engineer. These objects are then encapsulated within a *Shell*, which provides a uniform communication interface to the other components in the system. Components participate in control hierarchies (represented vertically) and data flows (represented horizontally) as shown in Figure 3.

Components interact with each other by exchanging messages addressed using component *profiles*, which are meaningful descriptors of components. For instance, a sensor profile could specify that the associated sensor is a color video camera operating at a frequency of 50 Hz, and covering

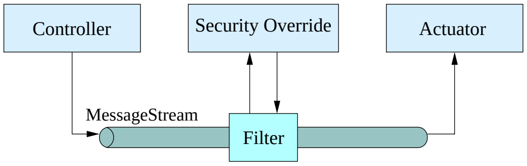


Fig. 4. MessageStreams and Filters

a certain geographic area. Other components could then communicate with the sensor using only this information, and obtain meaningful feedback without having to worry about low level network details such as IP addresses and ports.

Component based design insulates components from failures of other components, and promotes local temporal autonomy by allowing continued operation while the failed components recover. Message based communication decouples the components so that they are not dependent on the operational states of other components or intervening communication links. This exposes (disguises) most passive failures as simple message delays that can be addressed in a uniform fashion. Finally, profile based addressing of messages allows components to be restarted, upgraded, or even migrated as the low-level identity and network location of components is not exposed (and hence not required to be known).

B. *Message streams*

Etherware based components communicate by exchanging messages. However, most communication in networked control systems consists of streams of messages, where all messages in a given stream need to be delivered with similar guarantees. For instance, sensory feedback messages may need to be delivered with low delay, while supervisory control messages may need to be delivered with high reliability. Etherware supports *message streams* as a uniform mechanism for specifying such requirements. For instance, a control message stream from a Controller component to an Actuator component is shown in Figure 4.

Message streams constitute a simple mechanism to address passive failures. Etherware supports efficient restart mechanisms that allow failed components to be restarted in a timely fashion. Message streams to failed components are maintained across such restarts, and the connected components can continue operating without being aware of such restarts. Further, failures of computing nodes or communication links are exposed to components as a closing event of a message stream. This allows components to track such failures without needing to constantly monitor the system.

Multicast streams are multi-source multi-receiver message streams supported by Etherware. Such streams are useful for efficient delivery of multi-recipient messages such as system-wide feedback. These streams insulate components from passive failures as well.

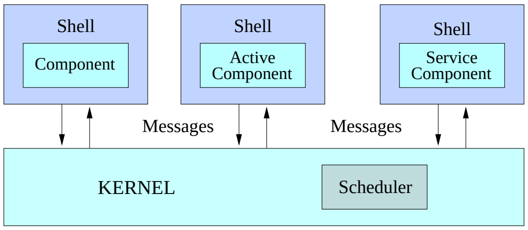


Fig. 5. Process architecture of Etherware

C. *Message filtering*

Etherware also supports filtering of messages through a message stream. For instance, if the Controller in Figure 4 has an active failure, then its controls to the Actuator can be overridden using an appropriate Filter. Similar filters can be used to filter all messages to or from a given component.

Message filtering can also be used to enforce mandatory control while reacting to Byzantine failures as described in Section III-D. For instance, a supervisor could override the discretionary control of low level car controller using an override as shown in Figure 4.

D. *Etherware architecture*

The architecture of Etherware is based on the microkernel concept in operating systems [9]. Specifically, the architecture of Etherware in an operating system process is shown in Figure 5. An Etherware process has a *Kernel* that manages all components in the process, and delivers messages between them. In particular, the Kernel can tolerate component failures and supports component restarts, upgrades, and migration while being operational. This allows passive failures to be tolerated quite easily, and insulates the rest of the system from such failures. The Kernel also has a *Scheduler* that determines the priorities for message delivery.

The Kernel is a simple and robust module which is fairly well tested and reused. All other Etherware functionality and services have been implemented as service components. This allows the system to tolerate failures in most of the Etherware software as well, and hence insulates the system from such failure. Further details about the Etherware architecture and services can be obtained from references [2] and [6].

V. EXPERIMENTS

In this section, we demonstrate the effectiveness of Etherware mechanisms for software fault management. Specifically, we present experiments to show how effective restarts and overrides can help reduce software related risk in control systems.

A. *Vehicular control testbed*

Our study of networked control systems has focused on the vehicular control testbed shown in Figure 6. The testbed consists of radio controlled cars operated by control software executing on a network of laptop computers. Visual feedback is provided by a pair of ceiling mounted cameras, processed by dedicated desktop computers, and is available as feedback

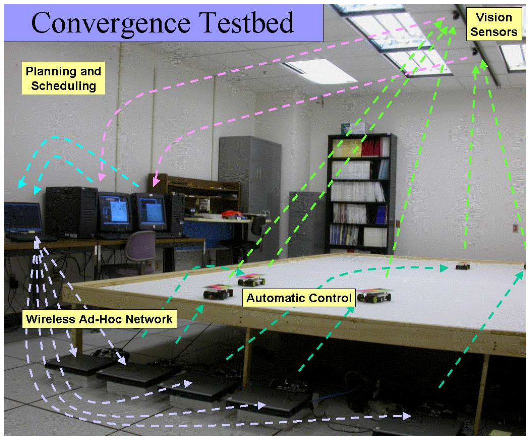


Fig. 6. Traffic Control Testbed

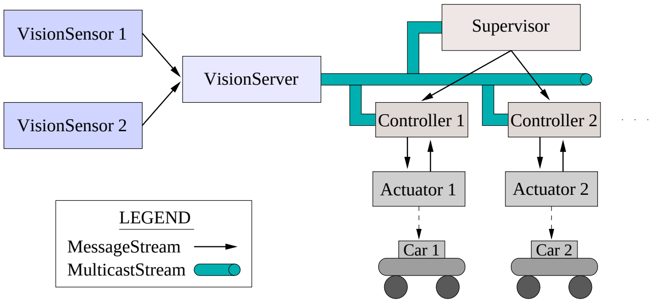


Fig. 7. Etherware based software architecture of the testbed

after appropriate data fusion. Vision images are processed and controls are actuated at a frequency of 10 Hz.

The testbed is operated by Etherware based component software whose architecture is shown in Figure 7. Each car is operated by an Actuator based on controls specified by a corresponding model predictive Controller. The way-points for the Controllers are specified by a higher level Supervisor. The communication between these components is through message streams as shown in the figure. Feedback is provided through a multicast stream by a Feedback Server, which accumulates sensory information from the Vision Sensors.

B. Component restarts

In the first experiment, we demonstrate the effectiveness of fast component restarts in addressing component failures. In this experiment a single car is operated in an oval trajectory, and failures are injected into the car Controller causing it to be restarted subsequently. The deviation of the actual car trajectory from the desired trajectory, as a function of time, is shown in Figure 8. Restarts are indicated by pointers, and the accompanying numbers indicate, in milliseconds, the time for each restart.

For the first 60 seconds, the car operated without restarts and tracked the trajectory with an error of less than 50mm. The first restart occurred at about 70 seconds into the experiment, and was followed by two other restarts in the next 20 seconds. The last three faults were also handled by the restart mechanisms in Etherware. The plot in Figure 8 indicates that the error in the car position during these restarts was well within the system error bounds during normal operation.

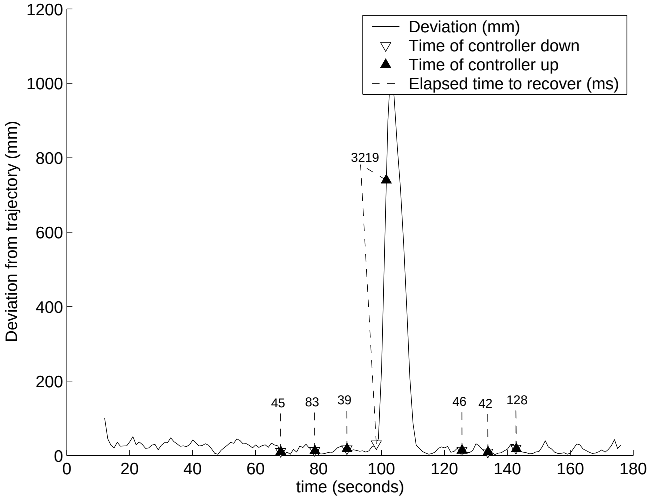


Fig. 8. Etherware based software architecture of the testbed

To illustrate the effectiveness of these two mechanisms, the entire Etherware process managing the Controller was restarted at about 100 seconds after the start of the experiment. As shown in Figure 8, the subsequent restart of the Etherware process with the Controller took about three seconds. During this time, the actual car trajectory accumulated a large error of about 0.8 meters with respect to the desired trajectory. This clearly illustrates the necessity for efficient restarts. Furthermore, even though the Controller restarted after three seconds, additional error was accumulated before recovery. This was so because the Controller had to reconnect to the other components, rebuild the state of the car, and bring it back on track. This demonstrates the improvement that has been achieved by the efficient restart mechanism in Etherware.

We have similarly demonstrated the effectiveness of Etherware mechanisms for component upgrades and migration as well [2], [6].

C. Security overrides

In the second experiment, we show the importance of message filtering based overrides in Etherware. In this experiment a rogue car operated by a human is introduced into a traffic setup with two regular and two police software controlled cars. On detecting the rogue car, the two police cars are tasked with pursuing and ”apprehending” it. The safety goal is to track the rogue car without causing any accidents, while the security goal is to minimize the distance between the police cars and the rogue car.

The police cars can be operated with local discretionary control, where each car is controlled independently, or with global mandatory supervision and security override, where all cars are controlled by a central supervisor. Safety is provided by low-level collision avoidance.

We consider the following scenarios in the experiment:

TABLE I

SAFETY MEASURE FOR THE FOUR SCENARIOS

Collision Avoidance	Security Override	Time to first collision (seconds)
No	No	2
No	Yes	10
Yes	No	No collisions
Yes	Yes	No collisions

TABLE II

SECURITY MEASURE FOR THE FOUR SCENARIOS

Collision Avoidance	Security Override	Minimum of distances between police cars and rogue car	
		Mean distance (mm)	Std. deviation (mm)
No	No	2182.7	1195.5
No	Yes	925.6	833.5
Yes	No	913.0	785.0
Yes	Yes	766.1	578.2

1) *No collision avoidance or security override:* This corresponds to local discretionary control of cars without low-level safety.
2) *Security override without collision avoidance:* This corresponds to global mandatory control of cars, also without low-level safety.
3) *Collision avoidance without security override:* This corresponds to local discretionary control with low-level safety.
4) *Security override with collision avoidance:* This corresponds to global mandatory control with low-level safety as well.

The results of the experiment are tabulated in Tables I and II below.

In the first case, since there is no collision avoidance, and the cars are operated with local discretionary control, collisions occur within the first two seconds of operation, and the police cars are quite ineffective. The situation is improved in the second case, since global mandatory supervision controls the system with better global information. However, since the global supervision is coarse grained, collisions still occur and affect the system performance.

Interestingly, in the third case, the performance actually improves by enabling low level collision avoidance, even though the cars are operated with local discretionary control. Basically, the safety mechanism prevents further system failures and improves performance.

The best improvement is achieved when both low level collision avoidance and global mandatory supervision are enabled. The safety mechanism prevents further failures, while the global supervision exploits better information and centralized control to optimize for better security. This demonstrates the effectiveness of security overrides, and illustrates the importance of safety preservation by security overrides in control systems.

The videos for these and other interesting experiments on the vehicular control testbed can be viewed at the testbed website [10].

VI. CONCLUSIONS

In this paper, we have established the importance of managing software failures to reduce risk in networked control systems. We have considered the various kinds of software related failures in networked control systems, and analyzed the risks associated with them. We have presented various strategies for managing these failures and reducing the associate risks. We have illustrated how these strategies can be implemented using software infrastructure such as middleware, and demonstrated the effectiveness of such strategies in practice on a vehicular control testbed.

REFERENCES

[1] "ARIANE 5 - Flight 501 Failure, Report by the Inquiry Board," July 1996.

[2] G. Baliga and P. R. Kumar, "A Middleware for Control over Networks," in *Proc. of the 44th IEEE Conference on Decision and Control*, Dec 2005.

[3] I. Sommerville, *Software Engineering*, 6th ed. Addison-Wesley Pub Co, 2000.

[4] C. Heitmeyer and D. Mandrioli, *Formal Methods for Real-Time Computing*, ser. Trends in Software. John Wiley & Sons Ltd, 1996.

[5] A. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proc. of the Londong Mathematical Society*, vol. 42, no. 2, pp. 230–265, 1936.

[6] G. Baliga, "A Middleware Framework for Networked Control Systems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2005.

[7] L. Sha, "Using simplicity to control complexity," *IEEE Software*, July/August 2001.

[8] S. Graham, "Issues in the convergence of control with communication and computation," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2004.

[9] A. Silberschatz, P. Galvin, and G. Gagne, *Applied Operating System Concepts*, 1st ed. John Wiley and Sons Inc, 2000.

[10] "Information Technology Convergence Lab, CSL, UIUC," http://decision.csl.uiuc.edu/ testbed/.