Technical Report
CMU/SEI-95-TR-005
ESC-TR-95-005

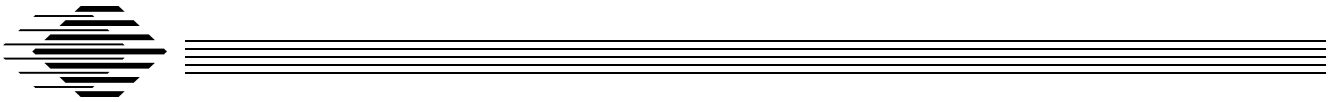
A Software Architecture for Dependable and Evolvable Industrial Computing Systems

Lui Sha, Ragunathan Rajkumar, Michael Gagliardi

July 1995

Technical Report
CMU/SEI-95-TR-005
ESC-TR-95-005
July 1995

A Software Architecture for Dependable and Evolvable Industrial Computing Systems



Lui Sha
Ragunathan Rajkumar
Michael Gagliardi

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: (703) 767-8222 or 1-800 225-3842.]

Table of Contents

A Software Architecture for Dependable and Evolvable Industrial Computing Systems

Abstract: The downtime of a large industrial operation is often prohibitively expensive and a failure of a mission critical system could have disastrous consequences. Lacking an effective approach to mitigate the risks in system upgrades or to introduce third party supplied open system components, many industrial systems and defense systems are forced to keep outdated computing hardware and software.

A paradigm shift is needed, from a focus on enabling technologies for completely new installations to one which is designed to mitigate the risk and cost of bringing new technology into functioning systems. Innovative technology is needed to support the task of *technology insertion*. Quickly and reliably turning unparalleled American innovations into industrial competitiveness and defense technological superiority is of strategic importance.

The Simplex architecture has been developed to support safe and reliable online upgrade of hardware and software components in spite of errors in the new modules. This paper gives a brief overview of the underlying technologies.

1. Introduction

Computers and computer networks have revolutionized the production of goods and delivery of services. Nevertheless, the computing infrastructure often introduces formidable barriers to continuous process improvement, equipment upgrades, and agility in responding to changing markets and increased global competition. Consider the following anecdotal scenarios from industry.

Process improvement: A research department developed a process modification that improved significantly the product yield and quality. With a relatively minor modification of the processing sequence, and new set-points for key process variables, the improvements were demonstrated on a pilot plant. Nevertheless, these improvements were never implemented in the plant, because the line manager persuaded management that it couldn't be done cost effectively. Although the required software modifications are simple logic modifications, the process sequence is controlled by a set of networked PLCs (programmable logic controllers) coordinating several valves, sensors and PID loops with several hundred lines of ladder logic code. The technician who wrote the PLC programs left the company. The last time a modification was attempted on the code, it took the process down completely, costing thousands of dollars in downtime. The line manager wanted no part of installing the so-called process improvements developed by the research department.

Equipment Upgrades: Over the years, aging equipment has been replaced with new production technology so that the factory now has a hodgepodge of old and new equipment with

controllers from five different vendors, each with its own programming interface, data structures, and data communication protocol. One of the older process machines failed recently and the best replacement was from yet another vendor. It had a considerably shorter cycle time, improved reliability, and a more sophisticated control computer. As with previous upgrades, however, installing the new equipment required yet another development and integration effort with yet another proprietary computing environment. It was particularly costly to create custom interfaces to communicate with the other equipment in the system and there was no way to predict the timing effects. Consequently, the only way the equipment could be installed safely was to perform extensive tests during factory downtime. In the end, integration cost several times the capital cost of the new equipment, and worse yet, it took several times longer to install the equipment than was originally estimated.

As illustrated by the preceding scenarios, existing integrated computer systems do not provide support for evolution in general and for safe online evolution in particular. Current system architectures do not tolerate errors in the insertion of a new technology or modification. If there is an error, the operation of the factory will be adversely impacted. It is extremely difficult to avoid all problems when new technology is inserted into systems whose design did not anticipate those new technologies, especially when the modification is done by someone other than the original system integrators.

This is a pervasive problem in industrial and defense systems requiring high availability, real-time performance, and safety. Indeed, existing practice in mission critical computing mandates an extremely conservative attitude towards the introduction of new technology into functioning systems. While such a conservative attitude is completely justifiable and reasonable, it has the unfortunate side effect of undermining the productivity, agility, and quality of our industrial base. It also threatens to erode the technological superiority of our defense in the post cold war era, where much fewer new systems are being developed.

A paradigm shift is needed, from a focus on enabling technologies for completely new installations to one which is designed to mitigate the risk and cost of bringing new technology into functioning systems. Technology is needed to support *evolving systems*. Industry needs a computing infrastructure in which upgrades will be safe and predictable, with negligible down-time.

The Simplex Architecture is a software architecture based on open system components. It has been developed to address these concerns. The architecture was initially developed to support the safe online upgrade of feedback control and radar systems, regardless of whether there were faults in the modeling, design, and implementation of new software or computing hardware [1, 4]. This paper provides an overview of this architecture, which supports the safe evolution of the application software online. Simplex architecture also supports the safe online addition or removal of computing hardware and system software. The rest of this paper provides an overview of the technology foundation of the Simplex Architecture. Chapter 2 gives an overview of the basic requirements of the Simplex architecture. Chapter 3 gives a very brief overview of the technological foundation and Chapter 4 provides an overview of the software architecture. Finally, Chapter 5 gives the summary and conclusion.

2. Basic Requirements

The design of the Simplex Architecture addresses three basic requirements. Firstly, *an architecture that is designed to support system evolution must itself be evolvable.* Trade-off decisions such as flexibility vs. efficiency, change from application to application. Within an application, decision weights change as technology and user needs evolve. For example, in many early PC applications, e.g., spread sheets, most of the available computing resources were devoted to problem solving rather than user-interface needs. Today, a significant percentage of the computing power used by an application is devoted to graphical user interface. A trade-off decision can, at best, be optimal at some particular given environment and time. Thus, it is important to have the mechanisms to evolve both the application software as well as the Simplex Architecture itself.

Secondly, *application codes should be protected from changes to non-functional quality attributes and vice versa.* It is important to minimize the impact of changes on application programs when the system evolves. Interfaces between components and subsystems must be defined in such a way that tradeoffs between non-functional quality attributes such as dependability and performance can change over time without the need for modifying application programs. For example, when a new software component is first introduced, it may need to be embedded in an application unit that has built-in safety checks and detailed operational monitoring. This may impact performance adversely. When the risk of a component decreases, some of the risk mitigation measures at runtime can be reduced accordingly. These changes should be made without any modification to the application program, since the semantic (functionality) of the application remains unchanged.

Thirdly, *the architecture must provide application independent utilities to meet timing constraints, to tolerate hardware and software failures, to monitor system status, to configure the system, and to provide users with templates and standardized transactions to manage changes in general and online changes in particular.* To support system evolution, we provide users with commonly used system utilities so that they can focus on the semantics of their applications.

3. Underlying Technologies

In order to allow to change software modules during runtime, an advanced real time resource management technology is needed. In Simplex architecture, real time process management primitives are built upon generalized rate monotonic theory [5]. In order to tolerate both hardware and software faults, to manage changes during development, and to support the evolution of dependable systems after their deployment, a new theoretical foundation is needed to allow for well formed diversity among redundant components. The theory of analytic redundancy provides us with such a foundation [6]. In the following, we give a brief review of these two subjects.

3.1. Real-Time Scheduling

Generalized rate monotonic scheduling (GRMS) theory guarantees that the deadlines of tasks will be met, if the total CPU utilization is below some threshold and if certain rules are followed. GRMS is supported by major national standards including Ada9x, POSIX real-time extension, and IEEE Futurebus+. The following is a synopsis of generalized rate monotonic scheduling for uni-processor. For an overview of this theory for distributed systems, see [5], for system developers who want detailed practical guidelines see [2]. The name rate monotonic scheduling comes from the fact that this algorithm gives higher priorities to tasks with higher frequencies.

A real-time system typically consists of both periodic and aperiodic tasks. A periodic task τ_i is characterized by a worst-case computation time C_i and a period T_i. Unless mentioned otherwise, we assume that a periodic task must finish by the end of its period. Tasks are *independent* if they do not need to synchronize with each other. By using either a simple polling procedure or a more advanced technique such as a sporadic server [7], the scheduling of aperiodic tasks can be treated within the rate monotonic framework. In each case C units of computation is allocated in a period of T for aperiodic activity. However, the management and replenishment of the capacity is different in each case. The scheduling of periodic tasks with synchronization requirements can be analyzed as follows [3].

> **Theorem 1:** A set of *n* periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasings, if $\forall$ i, $1 \le i \le n$,
>
> $$\frac{C_1}{T_1} + \cdots + \frac{C_{i-1}}{T_{i-1}} + \frac{(C_i+B_i)}{T_i} \le i(2^{1/i}-1)$$

where B_i is the duration in which task τ_i is blocked by lower-priority tasks. This blocking is also known as priority inversion. Priority inversion can occur when tasks have to synchronize. Priority inversion can be minimized by priority inheritance protocols [3]. The effect of this blocking can be modeled as though task τ_i's utilization is increased by an amount B_i/T_i. Theorem 1 shows that the duration of priority inversion reduces *schedulability*, the degree of processor utilization at or below which all deadlines can be met.

The Simplex Architecture assumes that application tasks are scheduled according to the

rules of GRMS and the operating system in use supports either the priority inheritance protocol or the priority ceiling protocol and is free of unbounded priority inversion in the management of shared resources. Both POSIX.4 OS specifications and Ada 9x runtime specifications incorporate priority inheritance protocols and satisfy this assumption.

3.2. The Basic Concept of Analytic Redundancy

Many industrial systems and defense systems have stringent reliability requirement. The standard practice is to replicate software on redundant computers and to use the majority of the outputs. Unfortunately, replication provides no defense against design or implementation errors. Furthermore, voting schemes based on the replication approach suffer from the upgrade paradox. That is, if only a minority is upgraded, the change will have no effect on the resulting system, no matter how good the upgrade. On the other hand, if the majority is upgraded and there is a bug, the system will fail.

To allow for safe evolution of a dependable computing system, we must go beyond replication and allow for well formed diversity among members of a fault tolerant group. What is well formed diversity? There are three forms of redundancy: replication, functional redundancy, and analytic redundancy. Given the same input, two functionally redundant systems will produce the same output. Functional redundancy permits the use of different but mathematically equivalent algorithms. For example, many different programs can be written to solve a given set of linear equations. While functional redundancy permits internal diversity that is not visible at the input and output level, analytic redundancy permits diversity that is visible at the input and output level.

Two analytically redundant systems need not produce identical results. However, the diversity permitted by analytic redundancy is well formed in the sense that both of them satisfy the given set of requirements. As an every day example, baby formula is analytically redundant to mother's milk with respect to the model of basic infant nutritional needs. However, they are not the same and a healthy mother's milk is known to be superior. The oxygen mask is analytically redundant to the pressurized air system of a high flying airplane in the sense that both of them can deliver the needed oxygen to keep passengers alive. The Taylor series expansion of a continuous function at the vicinity of a point is analytically redundant to the function if we only need a good approximation.

The notion of analytic redundancy is captured by a consistency model. In the context of plant control applications, our consistency model is designed to deal with *externally* observable events only. There is a set of consistency constraints on the input data streams to the state machines, and a set of consistency constraints on the output streams from the machines, and a set of consistency constraints on the state of the plant under control. That is, we permit well formed diversity on inputs, outputs and on how a plant is controlled.

Note that our consistency model does not include predicates on the internal states of controllers so that programs with different specifications and internal states can be used to

satisfy the same consistency model. On the other hand, we include the states of the plant into our model since the states of the plant must be observable in order to be controlled and the observed state transitions of the plant allow us to evaluate the controllers.

We assume that the plant states are observable and controllable. We partition the states of a plant into three classes, namely, operational states, mishap states, and hazardous states. Mishap states are those states of the physical system that incur damages or injuries. Operational states are those required for the correct execution of the application. Hazardous states are those that are neither operational states nor mishap states. Hazards are conditions that cause incorrect state transitions that may result in mishap states. A function is said to be critical if an incorrect implementation of it can generate hazards. When the state variable is continuous, the safety margin guarding a given set of mishap states is the minimal distance between the given mishap states and operational states. For example, the safety margin of falling off a cliff is the shortest distance to the edge. When the state variable is discrete, a useful definition of safety margin is the minimal number of hazardous states a system must go through before it reaches a mishap state when starting from an operational state. An everyday example of a discrete hazardous state is the "push down" state of a child proof cap of a medicine bottle. One may also assign weights to hazardous states and the distance becomes a weighted sum. The weights are used to model how strong is the barrier effect of various hazardous states.

In our consistency model, the constraints on the plant states are divided into two classes: safety specifications (constraints) and performance specifications. Safety specifications are predicates that exclude mishap states. The most common form of safety specifications are the specifications of the boundary conditions of the operational states. Getting out of the boundary of operational states leads to the activation of safety devices, e.g., too large a current triggers the circuit breaker, and the skidding of wheels during the braking of a car triggers the brake pumping actions of the anti-lock brake system. The class of performance specifications (constraints) is usually stated in terms of response time[1], steady state errors and tracking errors[2]. In controller design, another important consideration is stability and robustness with respect to perturbation to the plant control and variations in the parameters of the sensor, plant and actuators. Some of the controller design objectives can be in conflict. For example, a fast response time calls for higher gains but too high a gain may lead to instability and increased sensitivities to noise and system parameter variations. The different tradeoff decisions lead to the use of different control technologies.

Since plant states can be sampled by each controller independently, the general form of input constraints is simply a statement of the data sampling requirement for each controller. We can, of course, specify that identical inputs must be provided to all the controllers. However, this is generally not needed from the viewpoint of system observability. We will do so only if such specification is easy to meet and meaningful.

---

[1]This is the response time of the plant under control, for example, the time it takes a car to reach 60 miles/hour from resting condition.

[2]This measures how closely the plant following the specified set points.

The consistency model also includes the optional specification of the correlations between the output of a reference controller and the output of the new controller. Even though the different control algorithms give different outputs, the outputs are often correlated. The correlation can be quite strong when the system state is far away from the set point since any reasonable algorithm will use a large force to push the system towards the set point. If a strong correlation between outputs exists (conditionally) for the given application, it can be used as an additional diagnostic tool to monitor the execution of the complex controller.

3.3. Model Based Voting

To tolerate both hardware and software failures and to support the safe modifications of dependable systems, the voting protocol must permit well formed diversity. In a multi-computer fault tolerant group, model based voting is a means to support well formed diversity. Instead of directly comparing the output values step by step, a computer that controls a device is monitored by others to see if it observes the consistency model. Model based voting allows us to introduce new hardware and software technologies into the system safely. New upgrades will be accepted as long as it improves functionality or performance while observing the consistency model. In addition to supporting system evolution, modeled based voting also allows us to tolerate combined hardware and software failures. The basic requirement for a triplicated fault tolerant group using model based voting is as follows:

- tolerate an unlimited number of application design and implementation errors in complex controller software.
- tolerate active failures and malicious attempts within one computer.
- automatically reconfigure the system into a duplex standby system after one of the computer fails.
- support the safe online change of hardware, system software or application software.

These requirements can be fulfilled by a leadership protocol known as the Simple Leadership Protocol (SLP). While the detailed description and analysis of this protocol is too long for this paper, the basic idea is quite simple. Under a leadership protocol, a member of a fault tolerant group is selected as the leader who controls the plant. The rest are said to be (registered) voters. The leader is monitored by the voters. If the majority of the voters consider that the leader has violated the consistency model, the leader will be impeached and a new leader is selected from the voters. SLP was described in detail and its properties of were proven in [6]:

> **Theorem 2:** In a triplicated fault tolerant group using SLP, in the event of consistency model violation is detectable at time *t*, the leader will be impeached no later than (t + $2T_{max}$), where T_{max} is the longest sampling period used in the fault tolerant group.

Remark: Theorem 2 does not assume that tasks in the three computers are synchronized. In fact, synchronizing the execution of the tasks in the three computers will not reduce the worst case delay of leader impeachment. Finally, the sampling frequencies used by members of the fault tolerant group need not be the same.

Theorem 3: Under SLP, the triplicated fault tolerant group can tolerate an unlimited number of application level software errors in the complex controller software.

Theorem 4: Under SLP, the system can tolerate active failures and malicious attempts confined within one computer of a triplicated fault tolerant group.

Theorem 5: Under SLP, upon the failure of the leader in a triplicated fault tolerant group, the system automatically reconfigures into a duplex system.

4. Overview of The Architecture

4.1. Basic Structure

Software architecture is a set of definitions and rules that define the components of a software system, their interfaces and rules for their interaction. The Simplex Architecture has been designed to support safe online upgrades of industrial computing systems. Thus, it is mainly concerned with runtime components[3]. Compile time abstractions such as definitions of objects and methods or data structures and functions will not be visible to the Simplex Architecture. The online change of compile-time definitions is through the online replacement of processes that use these definitions. On the other hand, runtime issues such as the constraints on the request of memory blocks, the protocols used to schedule processes and threads, and the run-time support for exchange of messages will be part of the Simplex Architecture definition. Under the Simplex Architecture, the basic components available to users to structure the runtime aspects of their applications are replacement units, application units, processor and system configuration modules, and upgrade transactions.

The basic building block in the Simplex Architecture is the *replacement unit*, one or more processes with a communication template that facilitates the replacement of one unit with another online. Replacement units are designed in such a way that they can be added, deleted, merged or split online by a set of standardized upgrade transactions. Using the replacement unit as the basic building block allows a uniform approach to support not only the evolution of the application architecture but also the Simplex Architecture itself. In fact, the replacement unit used in the architecture can itself be refined and upgraded online.

From an application perspective, an industrial system typically consists of a set of cooperating autonomous subsystems, for example, the coordination of the controls of different workcells in a plant. In the Simplex Architecture, an autonomous subsystem is implemented as sub-system module, which consists of a set of specialized replacement units and we will examine its structure later in this section. The sub-system module provides the options to implement software fault-tolerance mechanisms that are important during the upgrade process.

[3]There are, however, certain guidelines regarding the preferred approach to structure compile time abstractions. They are, however, not the focus of this paper.

4.2. Logically Interchangeable Resources

Two or more resources can be defined to be runtime inter-changeable with respect to the execution of a sub-system module. For example, if sub-system module M can perform its functions on either processor A or processor B, then processors A and B are logically inter-changeable with respect to module M, even though processor A and B may be different physically in terms of speed and memory size. A processor and network membership service can be defined as part of the Simplex Architecture to determine which processors and networks are currently available in the system. If processors A and B are logically inter-changeable, and processor A fails, the sub-system modules on processor A may then run on processor B.

It is important to note that logical inter-changeable resources provide only the necessary condition for re-allocating application units. To re-allocate an unit, one must also ensure that there are sufficient CPU cycles and memory blocks to accommodate a unit.

4.3. System Administration

Sub-system modules are managed by a processor configuration manager (PCM) (one on each processor) which is in turn managed by a system configuration manager (SCM). Both PCM and SCM are replacement units with additional functions. The SCM interfaces with the user through a user-interface manager, another replacement unit with special functions. Through this user-interface, an application system administrator defines the topology of the distributed system and the types of processors, networks, and operating systems in use. The administrator also defines the class of interchangeable resources including processors, processes, and networks. In addition, the administrator defines the system state variables to be monitored.

Having defined the system topology and inter-changeable classes for sub-system modules, the system administrator has the option to define the degree of resource failures that will be tolerated. For example, the number of processor and network failures that the system configuration manager is able to tolerate can be specified. The degree of hardware failures that can be tolerated is, of course, bounded by the number of resources in the inter-changeable classes that are needed to support the service. System administrators can specify the actual physical resources used by each sub-system module. However, it is generally preferable to leave this task to the system and processor configuration managers. This allows the system to quickly and automatically reconfigure itself in the event of hardware failures.

4.4. Sub-System Modules and Software Fault-Tolerance

A sub-system module is made of specialized replacement units: application units, a module management unit and an optional safety unit. The safety unit is intended to implement a safety controller when the physical sub-system is not fail-safe[4]. The application units are used to implement the application-specific functions of a given sub-system. If a safety unit is used, application units within a module can only communicate with one another and with the safety unit. That is, the safety unit is responsible for all communications to other application units, perhaps including device I/O. This is a necessary condition to allow the safety unit to lock the application unit in a safe operational state.

Each sub-system module also has a module management unit, which is an instance of a module management template. The module management template is a replacement unit with functions that are designed to support process management, the upgrade operation and the handling of software faults in an application unit. The separation of upgrade related operations with application functions allows us to fulfill the second requirement. Structurally, both the safety unit and the application units may be child processes of the management unit and the safety unit is just a trusted and privileged application unit.

Each sub-system module also acts as a software fault-containment unit and normally runs in its own address space(s). This provides protection against programming system faults. Temporal faults also need to be considered. For example, an application unit can burn more CPU cycles than expected because of some error condition. The protection against timing faults can be provided in one of two ways. First, one can keep track of the CPU cycles used by the application units and compare the count with an expected value. This requires OS support and consumes CPU in the form of some scheduling overhead. Secondly, one can assign to the safety units in a processor higher priority than those of the application units in a processor. This approach can be implemented in an OS that supports fixed priority scheduling and the CPU cost is in the form of bounded priority inversion to all safety units. One may use Theorem 1 to compute the impact of these two different approaches.

4.5. Upgrade Transactions

The fundamental operation provided by the Simplex Architecture to support system evolution will be the replacement transaction, where one replacement unit is replaced by another. During this replacement transaction, state information may need to be transferred from the original unit to the new replacement unit. Alternatively, the new unit may capture the dynamic state information of physical systems through input devices. Without state information, there may be undesirable transients in the behavior of the new replacement unit

[4] In a fail-safe system, when a fault or failure occurs in the system, the system fails in a safe fashion and does not cause any damage.

when it comes online. Hence, the replacement transaction of a single replacement unit is carried out in stages:

1. The new replacement unit is created.

2. New input and any state information is provided to the new replacement unit when it is ready. The new unit begins computations based on the data. However, the output of the unit is monitored but not used.

3. The upgrade transaction manager waits for the output of the new unit to synchronize or converge to a stable point.

4. Finally, the output of the old unit is turned off and the new unit is turned on. The old unit can now be destroyed.

A two-phase protocol can be used when multiple replacement units are to be replaced simultaneously. The first phase is to wait for all the new replacement units to reach a steady state (step 3 above). The second phase is a distributed action that simultaneously switches on all the new replacement units and switches off all the old replacement units. The granularity of "simultaneity" is subject to the accuracy of clock synchronization in a distributed system. If the switching is successful, the old replacement units can be destroyed. If any switching action is not successful, the system can automatically switch back to the old replacement units and the replacement transaction can be aborted. The replacement transaction is managed by the PCMs in cooperation with the SCM.

5. Summary and Conclusion

Industrial and defense computing systems often have stringent safety, reliability and timing constraints. Failure in such systems can potentially have catastrophic consequences, and system downtimes can be expensive.

In this paper, we give a brief overview of the technology foundation of the Simplex Architecture. The architecture can be used to maintain the safety, reliability and real-time constraints of industrial and defense computing systems, despite inevitable glitches when new technologies are introduced and integrated with existing equipment. The architecture is based on open system components, and supports the safe evolution of the application software architecture itself online. It will also support the safe online addition and removal of computing hardware and system software.

Two demonstration prototypes were built and are available for demonstration. The single computer prototype uses a personal computer controls that an inverted pendulum. The controller software can be modified on the fly. Members of audience are invited to modified the control software online. Arbitrary bugs at the application level can be inserted by the audience. The demonstration shows that the control performance can only be improved but not degraded. A triplicated fault tolerant group implements the model based voting. It permits the safe online modification of not only the application software but the hardware and system software. An improved minority can gain and improved the control performance. However, no hardware or software errors at that computer can degrade the control, including malicious attempts with root privileges at that computer. The triplicated fault tolerant group can also be reconfigured online into a duplex system or a uni-processor system and vice versa as needed.

Currently, the SEI is actively working with industry partners and government agencies to mature this promising new technology.

Acknowledgement

This work is sponsored in part by the Office of Naval Research, in part by the National Institute of Standards and Technology, and in part by the Software Engineering Institute. The authors want to thank John Lehoczky, Jennifer Stephan, and Marc Bodson for the discussion of control related issues, Neal Altman and Chuck Weinstock for the discussions on software engineering issues, Bruce Krogh for providing the anecdotal scenario from industry, and John Leary for his review.

References

1. Bodson, Marc; Lehoczky, John P.; Rajumar, Ragunathan; Sha, Lui; Soh, D.; Stephen J.; and Smith, M. "Control Reconfiguration in the Presence of Software Failures". *Proceedings of the 32nd IEEE Conference on Decision and Control 3* (December 1993).

2. Klein, Mark H.; Ralya, Thomas; Pollak; Bill, Obenza, Ray; and Gonzalez Harbour, Michael. *A Practitioner's Handbook for Real-Time Analysis.* Kluwer Academic Publishers, Norwell, MA, 1993.

3. Sha, Lui and Goodenough, John B. "Real-Time Scheduling Theory and Ada". *IEEE Computer Vol. 23, no. 4* (April 1990).

4. Sha, Lui; Lehoczky, John P.; Bodson, Marc; Krupp, P.; and Nowacki, C. "Responsive Airborne Radar Systems". *The Proceedings of The Second International Workshop on Responsive Systems* (October 1992).

5. Sha, Lui; Rajkumar, Ragunathan; and Sathaye, S. "Generalized Rate Monotonic Theory: A Framework for Developing Real-Time Systems". *Proceedings of the IEEE Vol. 82, No. 1* (January 1994).

6. Sha, Lui; Gagliardi, Michael; and Rajkumar, Ragunathan. "Analytic Redundancy: A Foundation for Evolvable and Dependable Systems". *The Proceedings of the International Conference on Reliability and Quality in Design* (March 1995).

7. Sprunt, Brinkley; Sha, Lui; and Lehoczky, John P. "Aperiodic Task Scheduling for Hard Real-Time Systems". *The Journal of Real-Time Systems Vol. 1* (1989), pp. 27-60.