03199

LA-UR-09-XXXXX

Approved for public release; distribution is unlimited.

Title: Software Archeology: A Case Study in Software Quality Assurance and Design

Author(s): John MacDonald, Jane Lloyd
PMT-4
Los Alamos National Laboratory

Cameron J. Turner
Colorado School of Mines

Submitted to: 2009 ASME IDETC/CIE Conference
San Diego, CA
August 30-September 2, 2009

Proceedings of IDETC'09:
ASME 2009 International Design Engineering Technical Conferences and
Computers and Information in Engineering Conference
San Diego, California, August 30-September 2, 2009

DRAFT DETC2009-86776

SOFTWARE ARCHEOLOGY: A CASE STUDY IN SOFTWARE QUALITY ASSURANCE AND DESIGN

Cameron J. Turner
Colorado School of Mines
1500 Illinois Street
Golden, Colorado 80401
cturner@mines.edu

John M. MacDonald
Los Alamos National Laboratory
PO Box 1663, MS E530
Los Alamos, New Mexico 87544
jmac@lanl.gov

Jane A. Lloyd
Los Alamos National Laboratory
PO Box 1663, MS E530
Los Alamos, New Mexico 87544
jlloyd@lanl.gov

Abstract

Ideally, quality is designed into software, just as quality is designed into hardware. However, when dealing with legacy systems, demonstrating that the software meets required quality standards may be difficult to achieve. Evolving customer needs, expressed by new operational requirements, resulted in the need to develop a legacy software quality assurance program at Los Alamos National Laboratory (LANL). This need led to the development of a reverse engineering approach referred to as software archaeology. This paper documents the software archaeology approaches used at LANL to demonstrate the software quality in legacy software systems. A case study for the Robotic Integrated Packaging System (RIPS) software is included to describe our approach.

1. INTRODUCTION

In an ideal world, quality would be engineered into software during the design process just as it is engineered into hardware during design. While modern designs often apply this level of rigor to software as well as to hardware, this has not always been the case. Software was often created so that the system would work, with little thought given to its design or quality. As long as the system configuration (both hardware and software) remained constant and those responsible for the design remained available to deal with problems, the lack of detailed design documentation is not a significant problem. But when changes become necessary or the original people responsible for the software are lost to other programs, the quality of the software becomes important.

At LANL, the need for a legacy Software Quality Assurance (SQA) effort became apparent with a regulatory change [1] that required new and legacy systems to be brought into compliance with a Quality Assurance Plan (QAP) that is compatible with the ASME-NQA-1 [2] standard. For new systems, the integration of this requirement does not substantially change the design process. However, for legacy systems, the choices were simple:

1. Retire the system; or
2. Replace the system with a new compliant system; or
3. Bring the legacy system into compliance by establishing a SQA pedigree for the system including the software.

If the system was not to be retired, establishing a quality pedigree for the hardware components of new and legacy systems often meant reviewing the existing procurement documentation or replacing the existing components with a equivalent pedigreed component. However, software components, particularly custom developed legacy software algorithms presented a different challenge. If system retirement is not an option, the choice becomes one of either redeveloping the software from a blank slate or attempting to reverse engineer the software to develop a quality assurance pedigree.

The case study used in this paper uses the Robotic Integrated Packaging System (RIPS) as an example. RIPS is the product of nearly 15 years of development and is a one-of-a-kind system with customized hardware and software involving mechanical, electrical, chemical, nuclear and robotics engineers, as well as material scientists, chemists, and nuclear physicists in its design. Retirement of the system is not an option. Wholesale replacement would result in unacceptable project delays and severe budgetary impacts that would probably result in not just the termination of the RIPS project, but of several other associated projects. The only option available was to bring the system into compliance with the new regulations by developing an quality pedigree for the system.

While the system hardware either had sufficient documentation to establish a quality pedigree or could be easily replaced with pedigreed components, the system software is highly customized with limited documentation. Further

complicating the SQA effort, was the fact that none of the original software developer remained on the project, and only two of the five could be contacted. Unfortunately, the remaining individuals had limited knowledge of the system software beyond their own contributions and had not been involved with the software for several years. Consequently, their recollections of the software were of limited value. However, both individuals indicated that they were not aware of a SQA effort during the software design and development.

So, the task for the team assigned to this project was, given a product, in this case a software package, develop a quality pedigree for the system software using the available design information, user manuals, source code, and the integrated system to bring the RIPS system into compliance with currently regulatory requirements.

2. NECESSITY OF SOFTWARE QUALITY ASSURANCE

Many program managers have asked "Why is software quality assurance a necessary component in many engineering systems?" The best answer is that SQA can reduce project costs by preventing hardware/software conflicts or errors, facilitating software changes and upgrades, while ensuring that the customer expectations are met by ensuring that the software implementation is complete.

For a new system, SQA can be readily integrated into the design process. At its core, SQA is nothing more that a systematic documentation of the design process. However, during the development of a research and development system, a rigorous approach to SQA is rarely the priority of the scientists and engineers involved. LANL has many such systems and vast amounts of software code developed for research projects with little or no SQA documentation.

Even if the code is well-developed and thoroughly tested, it may be difficult to establish a quality pedigree if the design process was not well-documented. The LANL experience is that with legacy systems, the documentation is often lacking and some level of effort is therefore necessary to establish a quality pedigree. Often, this effort requires the recreation of the design process so that design documentation can be developed to support the system. Ralph Johnson of The University of Illinois at Urbana-Champaign terms this process Software Archeology [3].

2.1. PURPOSE OF SQA

SQA programs attempt to ensure that the needs of the customer(s) are met by the software. These needs can be described as expected, targeted and unexpected as shown by the Kano diagram [4] shown in Fig. 1. The expected requirements are often unstated by the customer – they are "expected" to be present in the software and their absence is a major source of customer dissatisfaction. The targeted needs are those that the customer intends to satisfy through the use of the software. These needs are also expected, or the customer will be dissatisfied with the software, but their presence is not necessarily a source of customer satisfaction. Unexpected needs are software features that meet needs of which the customer is unaware. The presence of unexpected software features is a major source of customer satisfaction.

In a software archeology effort, all three types of customer needs must be identified. The targeted needs are relatively apparent, they are the primary or core purpose of the software. However, most software packages include many functions that do not apparently meet the targeted need of the software. For instance, is the user interface an expected need (i.e. implicitly required by the customer) or an unexpected need (i.e. implemented by the programmer to enhance customer satisfaction). Understanding the needs underlying the software functions is vital to properly developing the SQA pedigree.

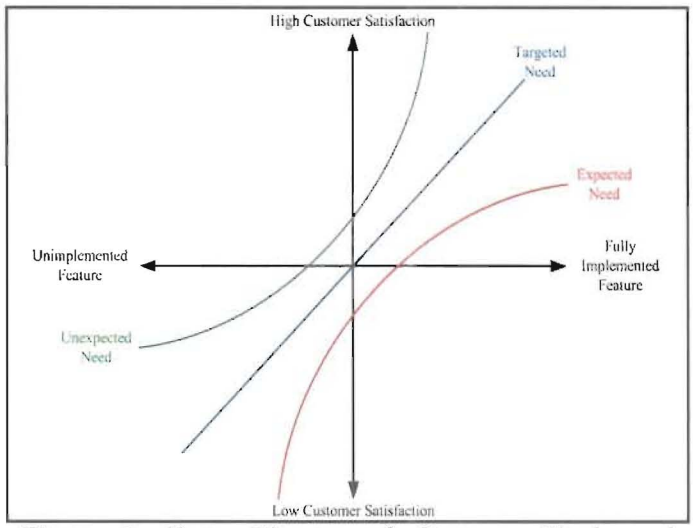

Figure 1. Kano Diagram of Customer Needs and Satisfaction versus Implementation Quality [4].

The quality of the software is a product of the quality of the design process [5]. Similarly, the quality of the pedigree is a product of the quality by which the design process is documented. If the goals in the design of a product (including a software product) include meeting the needs of the customer in terms of functionality, usability, reliability, performance, and supportability, then the goal of an SQA program is to demonstrate that those needs have been addressed throughout the design process. Not doing so adequately may significantly add to the costs of maintaining software [6] and maintaining the associated documentation of the software [7]. A lack of documentation does not mean that the design process was flawed. Instead, it simply means that the quality of the software is unknown. In an integrated system composed of hardware and software components, unknown component qualities greatly affect the operating risks associated with the system. The potential risks associated with unqualified software are the underlying basis for the new SQA policies and procedures at LANL.

2.2. IMPACT OF A LACK OF SQA

Work in nuclear facilities is generally risk-adverse. Therefore, it is not surprising that the initial impact of the new SQA requirements began with the nuclear programs at LANL. The risks of the legacy software had to be assessed and mitigated by establishing an SQA pedigree for software which could not be retired, replaced, or redeveloped.

Software failures are increasingly reported in the press. Often, the cause of these failures is a lack of SQA. The

Software QA/Test Resource Center website [8] maintains a listing of some of the more significant software failures attributable to a lack of SQA. A few of the more interesting highlights include:

- In January 2009, regulators banned a health insurance company from selling policies due to computer bugs that resulted in erroneous denials of coverage or outright cancellations in coverage to certain patients. These errors threatened the health and safety of beneficiaries.
- A January 2009 news report indicated that a major IT consulting company has spent four years correcting problems caused by an inadequately tested software upgrade.
- In August 2008, more than 600 airline flights were delayed due to a software glitch in the FAA air traffic control system.
- A lack of software testing was blamed for problems that led to privacy breaches into the records of several hundred thousand customers of a large health insurance company in August 2008.
- In December 2007, inadequate software testing of a new payroll system was blamed for $53 million of erroneous payments to employees of a school district.
- An April 2007 subway rail car fire was caused by the failure of a software system to perform as expected in detecting and preventing excessive power usage in the new passenger cars. The subway system had to be evacuated and shut down for repairs.
- A March 2007 recall of medical devices was blamed on a software bug that failed to detect low power levels in the devices.
- A September 2006 news report indicated that insufficient software testing led to voter check-in delays during the primary elections in that state.

There are dozens of additional examples of a lack of control over the development, use or maintenance of software that led to unintended failures. Clearly the need exists for producing better quality software. Software development failures also have been documented within the US Department of Energy (DOE) Laboratory Complex and at LANL. A series of software failures in DOE facilities, all attributable to a lack of SQA led to the implementation of formal SQA requirements at LANL. These new requirements apply not only to new and legacy software systems and led to the development of methods to establish an SQA pedigree for legacy systems.

2.3. A RELATIONSHIP BETWEEN SQA AND DESIGN

Figure 2 describes a typical product design lifecycle from the early problem identification process through design, production and the eventual retirement. The associated SQA process from ASME Standard NQA-1 [2] is shown alongside for comparison. The NQA-1 standard is further discussed in Section 3.

SQA naturally fits within a product development process such as that described in Fig. 2. This structure is appropriate for the concurrent development of a software system with a new product. This should not be surprising, but should be expected. SQA is not an additional complexity to be added to the design process, but rather, SQA is a documentation of a structured design process. Properly done, SQA adds very little effort to a design effort, but instead documents the decisions made during that process.

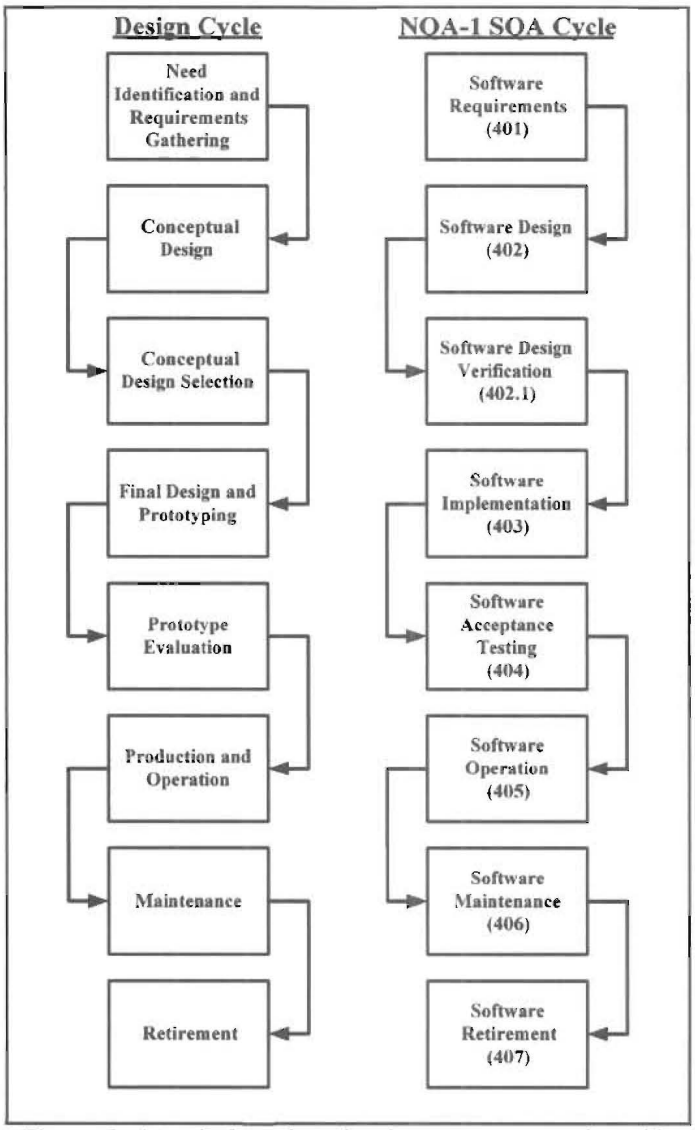

Figure 2. A typical product development process alongside the ASME NQA-1 software development process. [2]. The appropriate sections of the standard also are indicated.

However, if there is an existing or legacy software product available, a reverse engineering or redesign structure should be considered. Based on the reverse engineering process description of Otto and Wood [9] an equivalent SQA reverse engineering process is defined in Fig. 3. In this case, SQA is a much more resource intensive effort than is the case in a blank-slate design. Design processes have to be replicated in order to fill-in for the missing documentation. Inevitably, this a more problematic and uncertain approach than an original design process. However, for legacy systems, it may be the only possible choice.

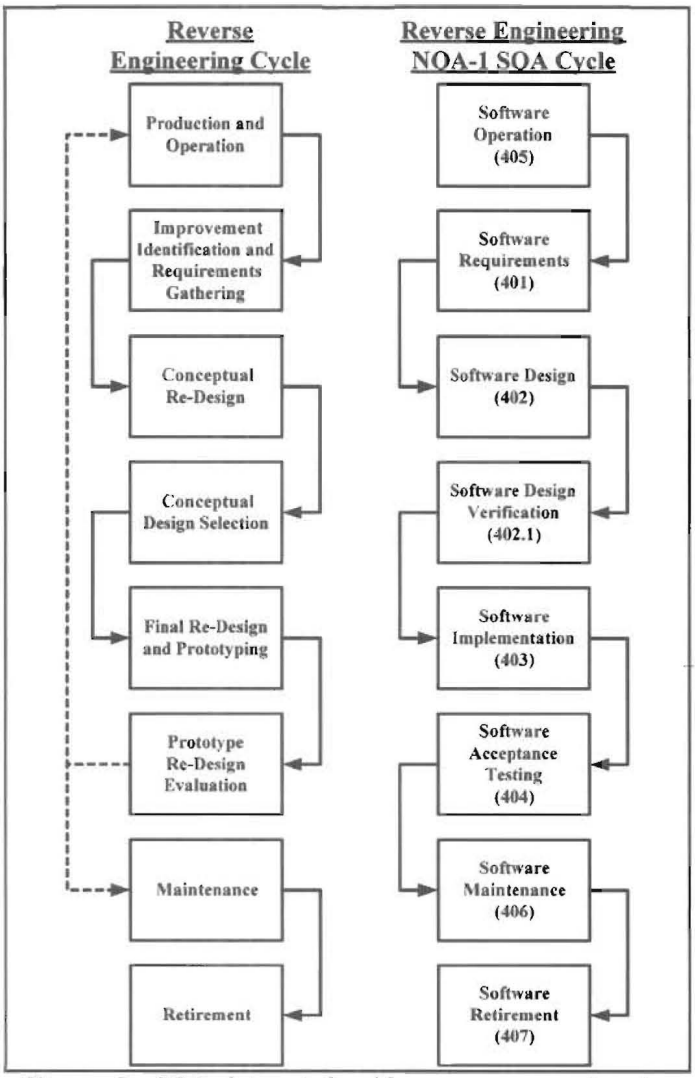

Figure 3. SQA integrated with a reverse engineering process.

The initial goal of a reverse engineering process is to establish the underlying needs that justified the product. In this case, we are interested in the expected, targeted and unexpected needs previously identified by the designers. Each need should be represented by a form embodied within the software. If programming is concerned with modeling reality with source code, reverse engineering software is concerned with obtaining a representation of reality from source code [10].

2.4. REVERSE ENGINEERING SOFTWARE

Software is commonly reverse engineered for a variety of purposes, including:

1. Coping with complexity;
2. Generating alternative system views;
3. Recovery of lost information;
4. Detect side effects;
5. Synthesize higher abstractions; and
6. Facilitate software reuse [6].

In this application, our purpose is clearly to "recover lost information," in this case information that establishes the quality pedigree of the software. However, the process also facilitates reuse and modification of the software, and uncovers potential interactions and side-effects within the system.

The seminal definition of software reverse engineering is provided by Chikofsky and Cross [11] as "the process of analyzing a subject system to (i) identify the system's components and their inter-relationships and (ii) create representations of the system in another form or at a higher level of abstraction." Effectively, this means that software reverse engineering can be classified in terms of redocumentation and recovery of the original design [12]. These are our purposes in this project.

The tasks involved included:

- Redeveloping the original customer needs that led to the original software development (recovery);
- Translating those needs into requirements and specifications (redocumentation);
- Mapping the requirements and specifications into the functional form of the design (recovery);
- Developing appropriate testing procedures to confirm that the requirements and specifications are met in the software as implemented (redocumentation); and
- Producing appropriate maintenance, upgrade and retirement plans and procedures (redocumentation).

The similarities in activities between quality assurance and design procedures are striking. Specific techniques used at LANL to reverse engineer legacy software will be noted in the case study in Section 4. The common fear of most engineers when faced with a new quality assurance program that there will be additional effort and the design process will suffer is probably unfounded. What is required is a simple documentation of the activities that already occur. In short, SQA is simply good engineering practice.

One final aspect of SQA that is not necessarily apparent is the importance of testing. This testing is beyond integrated system tests, although those certainly play a role in testing the software. Testing should demonstrate the "correctness, completeness, security, and quality of the software product against a specification" [13]. Several functions commonly included in LANL software may include access authorization protection, whose functionality was defined in the customer needs (as a expected need) but were not explicitly tested during the system acceptance tests because they were not identified as targeted customer needs. SQA analysis revealed these needs and led to their inclusion in software acceptance testing.

3. SOURCES OF SQA STANDARDS

Several professional organizations have arrived at standards for SQA programs. Among them are the American Society of Mechanical Engineers (ASME), Nuclear Quality Assurance Level 1, referred to as NQA-1 [2][1]. NQA-1 forms the basis for most of the relevant DOE and LANL standards and requirements for SQA. In addition to NQA-1, relevant

[1] Note that there are more recent versions of NQA-1, however, the DOE Orders specifically reference NQA-1-1997, and so therefore the SQA program is based on this version.

IEEE computer engineering standards such as IEEE 1228-1994 [14], Department of Defense standards such as MIL-STD-882D [15] and standards from the American Society for Quality (ASQ) were used to further refine the meanings of the ASME standards.

3.1. REGULATORY DRIVERS

DOE SQA programs are driven by regulations in 10 CFR 830.122 [2]. This code specifies a Quality Assurance Plan (QAP) and indicates that the QAP must address management, performance, and assessment criteria. Additional requirements are imposed for software if its location or use may affect the safety and/or security of a facility. Professional standards including ASME-NQA-1 have been codified into this code.

10 CFR 830.122 [2] resulted in DOE Order 414.1C [16], which is specific to quality assurance, safety software, and software defined as computer programs, procedures, and associated documentation and data pertaining to the operation of a computer system within DOE nuclear facilities. LANL translated this order into a LANL LIR 308-00-05.1 entitled "Software Quality Management" revised on December 29, 2006. This document has been further superseded by additional procedures and requirements.

At each level, the details of SQA implementations become increasingly specific. The SQA program developed for the ARIES project and used as the basis for the legacy work on RIPS, has been successfully audited several times and is in accord with all of the relevant DOE and LANL procedures.

3.2. THE ARIES APPROACH TO LEGACY SOFTWARE

The Advanced Recovery and Integrated Extraction System is a program that has been active at LANL since the mid-1990s. The program runs a series of gloveboxes, many of which contain integrated automation and processing systems [17, 18] for which extensive customized software was created. The ARIES glovebox lines convert nuclear materials from retired nuclear weapons into forms suitable for packaging for long-term storage, international inspection and for reuse as mixed-oxide (MOX) reactor fuel [19]. Because of the potential to reuse ARIES material in nuclear reactors, the need for an SQA program was recognized by the ARIES project long before other programs at LANL realized the need.

However, ARIES still had hundreds of thousands of lines of code for which the necessary documentation of the software quality was incomplete. To correct this deficiency, the ARIES program embarked on an aggressive software reverse engineering program to establish a defensible SQA pedigree for its legacy software systems. This program quickly became known as a software archaeology effort and is the basis for the case study in the following section.

This project used the ASME NQA-1 standard as a baseline for what information needed to be identified, documented and retained for both legacy and new software systems. These requirements are shown in Fig. 4. Some elements are only required of new software. Others are optional, and their need is determined during the development of the initial software project plan through a risk analysis.

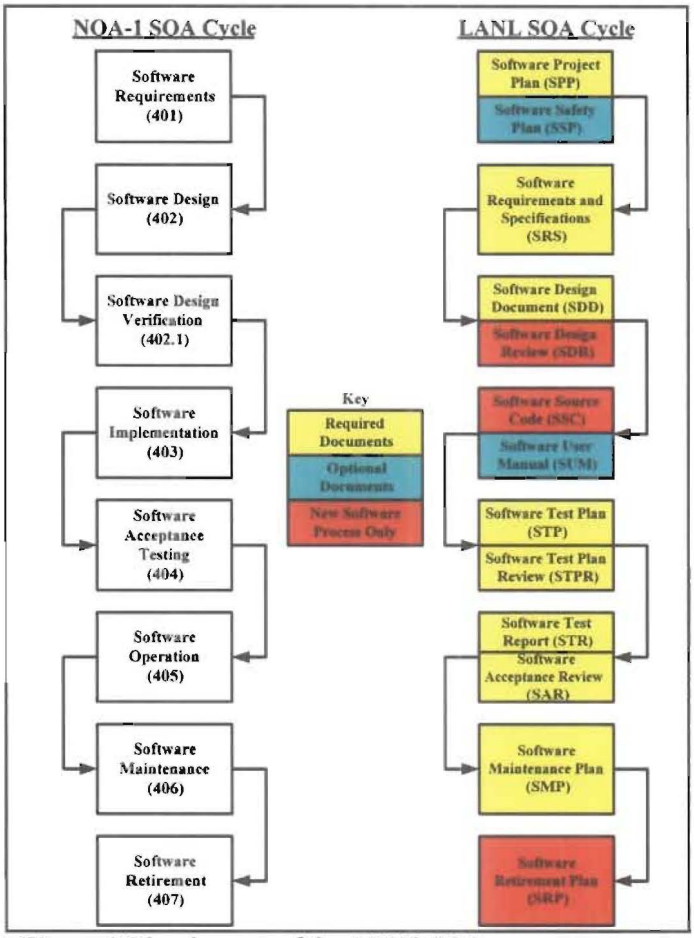

Figure 4. The elements of the ARIES SQA program.

The ARIES SQA plan has been through several internal and external audits and has earned glowing reviews each time. Furthermore, the program did review issues which had not been previously identified during system integration, acceptance testing or system operation. Most importantly, the project has increased confidence in our end-users that our product is produced to meet the required specifications.

4. SOFTWARE ARCHEOLOGY: RIPS CASE STUDY

The RIPS module is one of six major processes that make up the ARIES glovebox line. RIPS is responsible for packaging nuclear materials produced by the modules into stainless steel cans that meet the DOE 3013 packaging standard [20]. The cans are automatically packaged with two robotic systems and uses five independent subsystems to complete the process. These systems are controlled by no less than six separate computer systems and interface with six additional "intelligent" instrumentation subsystems [17, 18].

RIPS is the result of nearly 15 years of R&D by a team of about 50 people. Currently, only five individuals remain on the project and about the same number remain accessible to the project. More significantly for this project only two of the original five programmers remain accessible to the project. Most of the design knowledge for the module has been lost to retirements and career changes. A significant amount of the design history has been lost.

The RIPS glovebox, Fig. 5, is divided into three chambers called the hot side (which is radioactively contaminated), the cold side and the fluid processing side. Materials to be packaged in RIPS arrive in the hot side in a crimped convenience can. One robot then loads the convenience can into a 3013 stainless steel can which is then welded shut in a helium atmosphere. The welded can is inspected, leak checked and placed in an electrolytic decontamination chamber to be radioactively decontaminated (see Fig. 6 and 7).

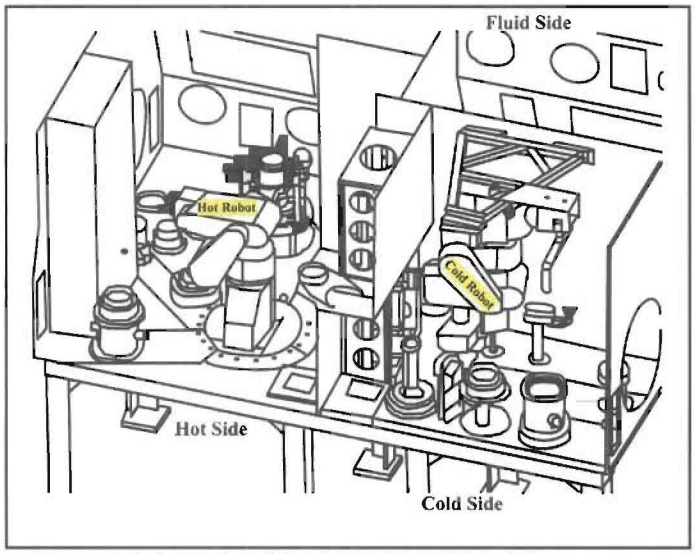

Figure 5. Schematic of the ARIES RIPS Module.

Within the electrolytic decontamination chamber, the surface of the can is electropolished which removes contamination from the surface of the can. The chemicals used to electropolish the can are recycled in the fluid processing chamber of the glovebox and the removed contamination is collected and removed from the system. Once the process is complete, the can is transferred to the cold side of the glovebox for final processing.

Figure 6. Handling the convenience can on the hot side.

Figure 7. Handling the welded 3013 can.

On the cold side, Fig. 8, a second robot conducts a radiation survey on the surface of the can, and conducts a second leak check to confirm that the can remains sealed. Once these checks are completed, the can is released from the RIPS module and taken to the next process in the ARIES process.

Figure 8. RIPS Cold Side Activities.

The operation of the RIPS system is controlled by a master PC, which can delegate system control to either robot, the welding subsystem, the electrolytic decontamination system, or the two leak check systems. When control is handed off to the subsystems, the master computer "locks-out" the uninvolved systems until control is returned by the subsystem completing its assigned task. The master computer and each of these subsystems include software which needed to be evaluated. In addition, the radiation checks use three additional "intelligent" instruments to survey the surface of the can. These instruments also included software that needed to be addressed.

Due to the nature of the RIPS system, complete details of the hardware and software are not included in this paper. The information provided is as complete as possible; however, some data is not presented as completely as may be desired by the readers. The authors believe that the level of detail provided is adequate to provide the interested readers guidance in the processes used to develop the SQA pedigree for the RIPS system and to apply this approach to other legacy systems.

4.1. SOFTWARE PROJECT PLAN

The Software Project Plan (SPP) is the initial step in the reverse engineering process used by ARIES at LANL. The purpose of the SPP is to document the original customer needs for the software and to establish a plan to complete the SQA process. Particular attention is paid in this initial phase with the identification of expected, targeted and unexpected needs. The main techniques used to develop the system needs included interviews with the system operators, reviews of the existing design documentation and user manuals, and a systemic review of the system software to catalog each feature. Features that did not clearly map to a particular customer need were identified and their purpose was defined. This final set of functions led to the identification of many of the unexpected customer needs.

The SPP identified four targeted customer needs for the system, including:

- the automated welding of the 3013 containers;
- the automated verification that the weld sealed the container;
- the automated decontamination of the outer surface of the container; and
- the automated verification of satisfactory decontamination and that the container remains intact.

In addition, this analysis led to the identification of several expected customer needs, including for example:

- a graphical user interface to the main operating program;
- a graphical user interface to the decontamination program;
- localized control that prevented independent operation of subsystems without authorization of the main control system;
- error detection of a set of limited error conditions with associated error recovery procedures; and
- access authorization control to override automated routines.

Furthermore, the development of the SPP identified several unexpected customer needs, including for example:

- multiple access levels enabling increasing control of subsystem interfaces with reduced safety margins;
- database parameter control;
- centralized data collection and package report preparation;
- system expandability for program modifications; and
- logging of system activity.

During the original system acceptance testing, only the functionality associated with the targeted needs was systematically tested. Several element of the expected needs and the unexpected needs were also tested, but the tests were not explicit. For instance, the access authorization was tested, but not intentionally. So the system operators were confident that it worked, but there was no documentation of an access denial. Similarly, while system logs were created of the acceptance test activities, generation of a system log was not a requirement for the acceptance test.

Beyond the identification of the customer needs for the RIPS system, the SPP provided the opportunity to identify the system components and their connections. At this point in the project, an important issue was identified concerning the presence of "intelligent" instrumentation within RIPS and to what level of granularity within the system should the SQA process be terminated.

These intelligent instruments contain internal software, often in the form of firmware, which processes the sensor data and provides a result to a local user interface and to the main system through a network connection. The issue is whether or not an adequate quality pedigree existed for these instruments or whether a separate quality pedigree needed to be developed.

For instance, RIPS uses several thermocouples and a thermocouple controller to monitor temperatures in the electrolytic decontamination system. The thermocouple controller uses firmware to convert the thermocouple voltage to a temperature readout which is supplied to the system software.

The quality issue is whether the data supplied by the thermocouple is correct. If the firmware is in error, the system will not provide correct data and may cause errors in other system components. Furthermore, is the firmware controlled and cannot be modified without knowledge of the project.

On one hand, the firmware is software which needed to be validated. On the other hand, this software can only be modified by the vendor, and since the systems are located in a secure facility, the firmware configuration is controlled. Furthermore, the instruments were subject to a calibration plan, which also served to verify that the firmware and the sensor are functioning correctly. Consequently, it is our conclusion that the quality issues in the intelligent instruments are controlled through the existing project change control and calibration requirements. Therefore, these intelligent instruments have an adequate quality pedigree to support their continued use within the system and do not need to be reverse engineered.

Consequently, the SPP allowed for the system hardware configuration and major software components to be diagrammed, initially resulting in the diagram shown in Fig. 9. Figure 9 is obviously lacking in significant detail, including the nature of the network connections, specific hardware components and interfaces, and software communication protocols. However, it represents an initial high level conceptual system diagram that was refined as the reverse engineering project proceeded. This diagram is equivalent to an early system sketch.

Finally, the SPP required an analysis of the safety significance of the software. The safety analysis was conducted with a formalized questionnaire completed by the responsible system engineers that resulted in a determination of the level of safety significance of the software. RIPS was determined not to be safety significant software and therefore did not require a Software Safety Plan (SSP). It also was determined that the software maintenance plan could be integrated into the system maintenance plan which was also under development at the time. Finally, the retirement plan development was deferred since there are no immediate plans to retire this system.

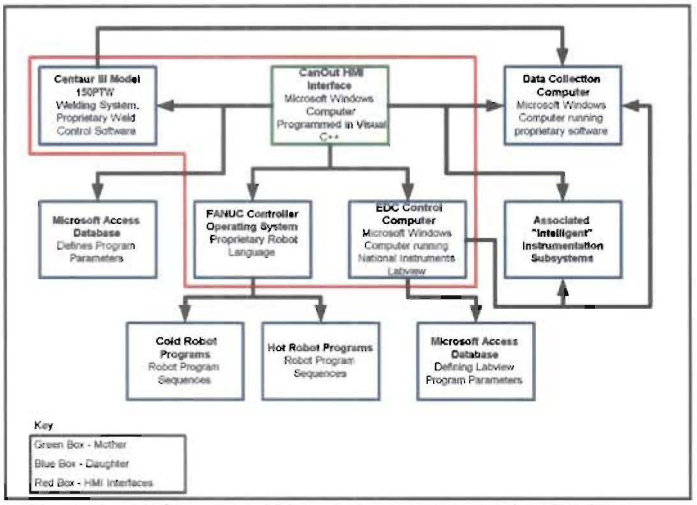

Figure 9. Software Architecture uncovered with the SPP.

So, in summary, the primary design techniques used in to develop the SPP included:

- customer interviews and surveys;
- literature reviews (of existing design documents and user manuals);
- functional analysis of the existing software (to generate additional customer needs); and
- conceptualization sketches of the system architecture yielding a high level black box model of the system.

4.2. REQUIREMENTS & SPECIFICATIONS

The next phase of the SQA process for RIPS involved the development of the initial general customer needs defined in the SPP into a detailed set of engineering requirements and specifications. These included both requirements for system functionality during normal and abnormal operations as well as requirements for data interfaces and network connections between the various hardware systems. All of this information was documented in a Software Requirements and Specifications (SRS) document.

This analysis led to the development of a Requirements Traceability Matrix (RTM), organized hierarchically into 33 major categories corresponding to the final listing of customer needs and encompassing more than 500 individual requirements. Each requirement was also associated with a particular component from Fig. 9. This matrix was used in subsequent documentation. A sample of the content of the RTM is shown in Table 1.

Detail in the RTM was added through a combination of system operations and studies of the documentation and source code. The desired level of granularity in the RTM was that each action occurred within a single component of the software system and within an identifiable module of code. It was possible that during this study that additional customer needs could be identified, which is indicative of an iterative design process and would not surprise the authors. While no additional needs were uncovered during the RIPS study, several customer needs were refined to better correlate with the underlying details within the software.

As the process proceeded, the RTM was modified. Specific algorithm names were identified, such as the Hot Robot Program, G_MC_DE, which moves the robot into position to grasp the material can (MC) in the decontamination location. Many of these program names were not associated until the Software Design Document, described in Section 4.3 was completed. The RTM also began to reveal interfaces between software levels, such as that associated with requirement 32.2 where the H_Robot class in the top level software package (called CanOut) interfaces with the Hot Robot to call the individual programs defined by requirements 32.2.1 through 32.2.8. Thus, requirement 32.2 is dependent upon the successful integration of the eight subrequirements. The interfaces between the two software packages are represented in their own set of requirements.

In constructing the RTM, a detailed diagram of the operation of the CanOut software system was generated. This flow chart was instrumental in the development and organization of the RTM and in the development of testing procedures. The testing data is documented in the last three columns of the RTM. The flow chart for CanOut is shown in Fig. 10.

In addition, as detailed task requirements and specifications were developed, the information within Fig. 9 was further refined to include network structures, communication protocols, data flows, software versions and hardware interfaces. The functionality of these interfaces and data flows also are represented within the RTM. Notably, these requirements and specifications were not the result of the targeted customer needs for the system, but instead evolved from the expected and unexpected customer needs that appeared as the system was integrated.

The SRS document used design methods and techniques that included:

- task studies and decomposition;
- process and software process observations;
- process and software flowcharts (developed similarly to function structures and incorporating a grammar of movements and actions specific to the system)
- reviews of the system bill of materials and wiring diagrams; and
- software class diagrams and automated code review technologies.

4.3. SOFTWARE DESIGN DOCUMENT

The Software Design Document (SDD) associated each requirement and specification with particular software and hardware components. It is at this phase of the project where specific hardware was called out for the system, network connections were identified and data exchange protocols were specified. This phase is not that dissimilar from how an SDD might be created for new software with one exception. In this case, no new software was written. Instead, the source code was reviewed to determine which module(s) were responsible for each requirement. This information was used to refine the RTM by associating specific requirements with code modules.

Table 1. Example Section of the Requirements Traceability Matrix from the Error Recovery Customer Need.

Requirement Source	Requirement ID Number	Requirement Description	Top Level Software Module					Requirement Type	Software Testing		
			CanOut	Hot Robot (HR)	Cold Robot (CR)	Welding Computer	Decon Computer		Test ID	Test Source	Test Result
...	... ...	...	...	...	...	...	...	...	...	...	...
LANL510.0	32	Error Recovery	multiple routines					Rollup - Functional	T32	Hot AT & STP - retest	Pass
	32.1	Maintenance Mode Accessible	MM_menu class					Rollup - Functional	T32.1	Hot AT & STP - retest	Pass
	32.2	HR Recovery Programs Functional	H_Robot class	interface routines				Rollup - Functional	T32.2	Hot AT & STP - retest	Pass
	32.2.1	Get MC at Decon		G_MC_DE				Expected	T32.2.1	Hot AT & STP - retest	Pass
	32.2.2	Confirm Robot in Stow Position		ISATSTOW				Expected	T32.2.2	Hot AT & STP - retest	Pass
	32.2.3	Leave MC at Staging		L_MC_MS				Expected	T32.2.3	Hot AT & STP - retest	Pass
	32.2.4	Reverse MC Orientation		MC_TNOVR				Expected	T32.2.4	Hot AT & STP - retest	Pass
	32.2.5	Put MC at Staging		P_MC_MS				Expected	T32.2.5	Hot AT & STP - retest	Pass
	32.2.6	Take MC from Decon		T_MC_DE				Expected	T32.2.6	Hot AT & STP - retest	Pass
	32.2.7	Take MC from Staging		T_MC_MS				Expected	T32.2.7	Hot AT & STP - retest	Pass
	32.2.8	Move HR to Safe Position		TO_SAFE0				Expected	T32.2.8	Hot AT & STP - retest	Pass
	32.3	CR Recovery Programs Functional	C_Robot class		interface routines			Rollup - Functional	T32.3	Hot AT & STP - retest	Pass
	... ...	...	...	...	...	...	...	...	...	...	...

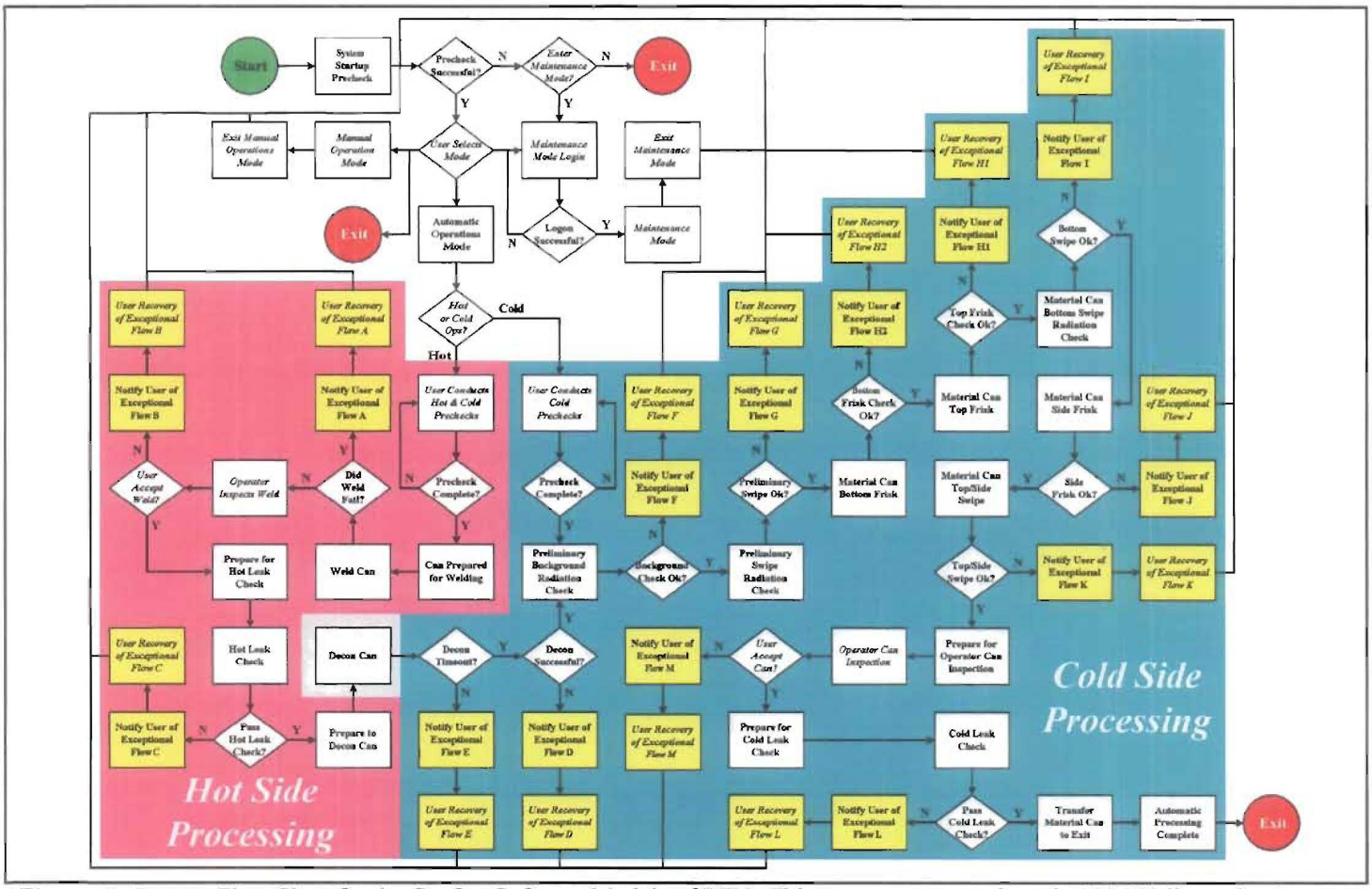

Figure 11. Process Flow Chart for the CanOut Software Module of RIPS. This represents approximately 100,000 lines of code.

The source code reviews of the RIPS software modules were performed by subject matter experts. While some automated tools were used to generate class diagrams and related programming representations, ultimately the task became to simply review the individual classes and identify functionality. This is in essence a reverse function structure, where the implementation or solution to the function is known - but the function structure is unknown.

The code review was completed to the level of detail required for integrated software testing. The function of every line of code was not identified, but the goal was to identify the major blocks of code responsible for each requirement. Thus, the SDD provides a starting point for code revisions, should changes prove necessary at some later date.

If this should be the case, the SDD also provides a format to capture new design knowledge and design changes to the software. Updating the associated software documentation is an important, but often overlooked step in software maintenance [7].

The final process used to develop the SDD was a software and requirements review by the system engineers. The goal of the review was to identify if the developed system representation captured the real system and if there were deficiencies that needed to be explained before the software would be validated, verified and placed under change control.

The design techniques used to develop the SDD included:

- system hardware reviews;
- reverse functional analysis to translate forms into functions and match the functional organization of the source code with the requirements for the system;
- formalization of the system interface and data exchange standards; and
- system engineering conceptual design review.

4.4. SOFTWARE TESTING

Perhaps the most important documents resulting from the SQA effort are the Software Test Plan (STP) and the Software Test Report (STR). The STP allows the integrated system to be tested to validate that the system performs as expected. This is the primary metric by which the quality pedigree of the system is determined [5].

At LANL, systems for use in a nuclear facility (such as RIPS) are required to pass a cold acceptance test in a non-nuclear environment and a hot acceptance test within the nuclear facility before operations may begin with nuclear materials. Both of these tests are integrated system tests that include both hardware and software. These tests were capable of revealing flaws involving the integrated system, but often did not test all of the software requirements identified for the system. The regulatory change that required a SQA pedigree for systems at LANL highlighted the need to completely evaluate the software in order to properly assess the risks associated with the system.

In the case of RIPS, when the SQA effort began, the system had already completed and passed its cold acceptance test and the hot acceptance test was underway. Consequently, an STP was prepared to be conducted following the hot acceptance test plan. The goal of the STP was not to repeat tests conducted in either the cold acceptance test plan or the hot acceptance test plan, but rather to capture any software requirements not tested by those plans. Therefore, planning for the STP began by reviewing the tests previously conducted and determining if the documented results adequately tested any of the identified software requirements from the RTM.

As a result of this review about 10-15% of the software requirements identified in the RTM were identified as either not tested during acceptance testing or were inadequately tested. Most of the requirements that were missed were relatively minor requirements from an operational standpoint of the system. For instance, the access authorization to the software was not formally tested during acceptance testing. The operators who performed the testing were certain that it worked, and could recall incorrectly entering their password and being rejected by the software, but there were no explicit requirements in either the cold or hot acceptance test plans to test the access authorization, nor was there any documentation that the function of the software worked.

With the untested requirements identified, brainstorming sessions were held with the system operators and engineers to develop appropriate and credible test cases. These test cases were organized into a logical progression and developed into the STP used for testing. Therefore, the STP was developed as an outgrowth of the RTM developed throughout the project. The final three columns shown in Table 1 shoudl now be self-explanatory.

Surprisingly, lessons were learned during the software testing. Some tests produced unexpected results which had to be explained. One example was a failure message that was reported when a fault was deliberately caused, but the message reported a more generic fault condition than what was expected by the test and the operators. All of the test results, whether indicating success or failure, were reported in the STR. Erroneous conditions required explanation and an accept-as-is or correct-with-change-order decision.

In the case of the failure message, investigation revealed that the software was reporting a correct error response. But, because of limited communication channels, two different fault conditions generated a common error message to the main software system. This hardware decision meant that the software could not distinguish between the two different failure causes as had been expected by the operators. Hence the message that indicated a failure had occurred, but did not indicate that there were two sources. The underlying design decision to piggyback the two error signals on a common communication pathway was obviously known to the original software developer - but was not documented in any of the available materials reviewed and required a much more detailed code review than was performed during the SDD to uncover its existence. This new information was not only documented in the STR, but also was used to update the Software User Manual (SUM). In this case, the test result was accepted-as-is.

Both the STP and the STR are subjected to a review process that includes system experts, software experts, and the responsible management for the operations and engineering of the system. These reviews provide a valuable check on the rigor of the STP and the conclusions drawn in the STR. The review

committees must also approve of any accept-as-is or correct decisions proposed by the responsible system engineers. This rigorous process forces the engineering underlying the system to be completely documented and produces an auditable document record of critical design decisions.

The development of an STP will also play an important role in the configuration management of the software. Once it was established that the existing software was functional and met the existing requirements; the software was baselined and the configuration frozen. A formal change process is now in place to prevent software and hardware changes without an associated software impact review and a subsequent testing after the change is implemented. Any changes will require use of the STP and Integrated Acceptance Test Plan for the system to confirm system modifications have not inadvertently effect the capabilities of the system. Subsequent testing will generate additional STRs.

For systems that were not already in the acceptance testing process, the plan is to integrate the STP into the Cold and Hot Acceptance Test Process. For legacy systems that have already completed the acceptance testing process, the STP approach used in RIPS allows those systems to be evaluated and test deficiencies identified and tested individually.

The use of the RTM as the basis for establishing the necessary tests provides a systematic means of demonstrating that the project requirements have been met. The result is a software package backed by demonstrable performance metrics and capable of exceeding customer expectations.

In the development of the STP, again typical design methods were used, including:

- review of prior test documentation;
- identification of testable requirements and test methods;
- brainstorming of credible test configurations;
- design of experiments;
- experimental testing; and
- design reviews of the resulting test plans and test results.

5. CONCLUSIONS AND LESSONS LEARNED

Even in a tightly controlled and regulated environment such as a DOE nuclear facility, there remains room for improvement. The implementation of a SQA program within the ARIES project required some creative problem solving approaches. Since adequate documentation was not produced during the initial software design and implementation, additional resources were required to reverse engineer the design. These reverse engineering efforts, colloquially called "software archaeology," revealed previously lost design decisions and features that had been lost.

These discoveries have enhanced our present understanding of systems such as RIPS. However, perhaps most importantly, the results of the SQA project have provided a quality pedigree for the software within ARIES that did not previously exist. Hardware and software upgrades can now be pursued with a much better understanding of their potential impacts upon the system, which should lead to a more predictable design and integration process. Furthermore, with an established set of software and hardware testing procedures, any upgrades or modifications to the system can be evaluated before the system is returned to operation. As a result, the risk profile of the system is much more clearly defined.

The relationship between a quality assurance program and design should be a strong one. Many of the features and requirements are similar. Strong formal design methods, when well documented are the backbone of quality assurance documentation. And if that documentation is lacking, reverse engineering methods are the central to the recovery, recreation and rediscovery of what has been lost. Good engineering design practices make quality assurance programs easy to implement and quality assurance programs can nurture good engineering practice.

ACKNOWLEDGEMENTS

This paper is approved for release by Los Alamos National Laboratory under LA-UR-09-01250. The assistance and support of Los Alamos National Laboratory and the Division of Engineering at the Colorado School of Mines is greatly appreciated. In particular, the authors would like to recognize Joe Lewis, Sonya Lee, Stan Zygmunt, Mark Swoboda, Max Evans, Bill Everett, and Paul Graham for their contributions to this project and extend their appreciation for their contributions and feedback on this paper.

REFERENCES

[1] CFR. (2002). *Quality Assurance Criteria*, 10 CFR 830.122, US Government, Washington, DC.

[2] ASME. (1997). *Quality Assurance Requirements for Nuclear Facility Applications*, ANSI/ASME NQA-1-1997 Standard, American Society of Mechanical Engineers, New York, New York.

[3] Johnson, R. (2005). "Reverse Engineering and Software Archaeology," *Software Tech News*, **8**:3, pp.7-13.

[4] Kano, N., Seraku, N., Takashi, F., Tsuji, S. (1984). "Attractive Quality and Must-be Quality," *Journal of Japanese Society for Quality Control*, **14**:2, pp. 39-48.

[5] Pai, W., Chung, C., Hsieh, C., Wang, C., and Wang, Y. (2005). "Software Testing Methodology with Control Flow Analysis," *Journal of Information Science and Engineering*, **21**, pp. 1213-26.

[6] Tonella, P., Torchiano, M., Du Bois, B., and Systa, T. (2007). "Empirical Studies in Reverse Engineering: State of the Art and Future Trends," *Journal of Empirical Software Engineering*, **12**:5, pp. 551-71.

[7] Napier, N., Kim, J., and Mathiassen, L. (2008). "Software Process Re-Engineering: A Model and Its Application to an Industrial Case Study," *Software Process: Improvement and Practice*, **13**:5, pp. 451-71.
[8] Hower, R. (2009). The Software QA/Test Resource Center, http://www.softwareqatest.com/qatfaq1.html, last accessed February 22, 2009.
[9] Otto, K. and Wood, K. (2001). *Product Design: Techniques in Reverse Engineering, Systematic Design, and New Product Development*, Prentice-Hall, Upper Saddle River, New Jersey.
[10] Deissenboeck, F. and Ratiu, D. (2006). "A Unified Meta-Model for Concept-Based Reverse Engineering," *Proceedings of the 3rd International Workshop on Metamodels, Schemmas, Grammars, and Ontologies – ATEM'06,* Genoa, Switzerland, October 1, 2006.
[11] Chikofsky, E. and Cross, J. (1990). "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, **7**:1, pp. 13-7.
[12] Canforma, G. and Di Penta, M. (2007). "New Frontiers of Reverse Engineering," *Proceedings of the 2007 ACM International Conference on Software Engineering*, Minneapolis, Minnesota, May 20-6, 2007, pp. 326-41.
[13] Geetha, B.G., Palanisamy, V., Duraiswamy, K., and Singaravel, G. (2008). "A Toolfor Testing Inheritance Related Bugs in Object Oriented Software," *Journal of Computer Science*, **4**:1, pp. 59-65.
[14] IEEE. (1994). *IEEE Standard for Software Safety Plans*, IEEE 1228-1994, IEEE Computer Society, New York, New York.
[15] DOD, (2003). *DOD Standard Practice for System Safety*, MIL-STD-882D, US Department of Defense, Washington, DC.
[16] DOE, (2005). *Quality Assurance*, DOE Order 414.1C, US Department of Energy, Washington, DC.
[17] Turner, C. and Lloyd, J. (2008). "Automating ARIES," *Actinide Research Quarterly*, pp. 32-5, LALP-08-004.
[18] Turner, C., Harden, T., and Lloyd, J. (2009). "Robotics for Nuclear Material Handling at LANL: Capabilities and Needs," MECH-86777, *Proceedings of the 2009 IDETC/CIE Conferences*, San Diego, CA, August 30-September 2, 2009.
[19] McKee, S. (2008). "ARIES at 10," *Actinide Research Quarterly*, pp. 1-6, LALP-08-004.
[20] DOE, (2004). *Stabilization, Packaging and Storage of Plutonium-Bearing Materials*, DOE-STD-3013-2004, U.S. Department of Energy, Washington, DC.