A DACS State of the Art Report
February 2003

Embedded Software Maintenance

Data & Analysis Center for Software

Unclassified and Unlimited Distribution

http://iac.dtic.mil/dacs

Embedded Software Maintenance

A DACS State-of-the-Art Report

Produced by Fraunhofer Center for Experimental Software Engineering Maryland and
The University of Maryland

By

Mikael Lindvall, Seija Komi-Sirviö, Patricia Costa and Carolyn Seaman

Prepared by:

Data and Analysis Center for Software
775 Daedalian Dr.
Rome, New York 13441-4909

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* 31 January 2003	**2. REPORT TYPE** N/A	**3. DATES COVERED** *(From - To)* N/A
4. TITLE AND SUBTITLE A State of the Art Report: Embedded Software Maintenance		**5a. CONTRACT NUMBER** SPO700-98-4000
		5b. GRANT NUMBER N/A
		5c. PROGRAM ELEMENT NUMBER N/A
6. AUTHOR(S) Mikael Lindvall, Seija Komi-Sirviö Patricia Costa and Carolyn Seaman		**5d. PROJECT NUMBER** N/A
		5e. TASK NUMBER N/A
		5f. WORK UNIT NUMBER N/A
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Fraunhofer Center for Experimental Software Engineering Maryland and The University of Maryland, College Park, Maryland		**8. PERFORMING ORGANIZATION REPORT NUMBER** DACS SOAR 12
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Technical Information Center (DTIC)/AI 8725 John J. Kingman Rd., STE 0944, Ft. Belvoir, VA 22060 and Air Force Research Lab/IFED 32 Brooks Rd., Rome, NY 13440		**10. SPONSOR/MONITOR'S ACRONYM(S)** DTIC-AI and AFRL/IFED
		11. SPONSOR/MONITOR'S REPORT NUMBER(S) N/A

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
```
Available from:  DoD Data & Analysis Center for Software (DACS)
PO Box 1400, Rome, NY  13442-1400
```

14. ABSTRACT

The goal of this report is to describe the state-of-the-art of embedded software maintenance and provide a glimpse of state-of-the-practice embedded maintenance practices. Section 2 introduces the area of embedded software based on embedded systems, the implication of embedded software and some of its characteristics. Section 3 describes the more general area of software maintenance, different process models and characteristics. Section 4 analyzes embedded software maintenance and investigates how the characteristics of embedded systems affect software maintenance and then describes the typical problems and issues in maintenance. This section also links typical problems to the case studies that can be found in Appendix A. Section 5 discusses potential solutions. One solution, impact analysis, addresses problems that are identified in many case studies; we discuss impact analysis in Section 6. References are listed in Section 7. Appendix A contains details of the case studies. In Appendix B, we list some of the resources that are available for readers interested in further studies of embedded software maintenance.

15. SUBJECT TERMS
Embedded software, software maintenance, software maintainability, software reliability

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT unclassified	**b. ABSTRACT** unclassified	**c. THIS PAGE** unclassified	UL	68	**19b. TELEPHONE NUMBER** *(include area code)* 315-334-4900

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Table of Contents

List of Figures

List of Tables

Acknowledgement:

We would like to thank the following people for making this report possible:

- Frank Bomarius, IESE, for interviews with companies.
- Forrest Shull and Ioana Rus for ideas.
- Victor R. Basili for supporting this work.
- Rikin Thakker for helping with the research.
- Jen Dix for proof reading.

1 Introduction

Due to the dramatic evolution of computer technology, computer power per dollar increases more than 1,000 times every ten years [Davis, 1993]. The result is faster, less power-consuming, and smaller computers that are not only placed on everybody's desktop, but are also built into many apparati and machines [Davis, 1993]. These built-in computers are referred to as *Embedded Systems*. These often small, built-in computers rule the marketplace: approximately 80 million PCs are sold every year as compared to circa 3 billion embedded CPUs [Koopman, 1999]. The embedded market is growing, while the PC market is mostly saturated.

A wide range of embedded systems exists on the market. Many products that feature embedded systems are small, relatively inexpensive and have a short lifetime of a few years before the next-generation product replaces them. Examples are mobile phones and other home electronics. In contrast, large, expensive and complex capital equipment, such as telephone switches, automobiles, airplanes, and industrial machines, are often safety-critical and in operation for several decades. All embedded systems, however, form part of a larger product. The buyer is primarily interested in the benefits the product brings and not in the embedded system. The fact that the product is based on computer technology is, however, not a major selling point, or even obvious to the buyer [Koopman-a, 1996], [Koopman-b, 1996].

The emergence of embedded systems in products of virtually all domains has resulted in a dramatic increase in products incorporating *Embedded Software*. The most recent generation of embedded systems relies heavily on embedded software. As a matter of fact, many of their features, which used to be controlled by electronics or mechanical components until a few years ago, are now software controlled. Current trends reveal that functionality that traditionally was implemented in hardware is now implemented in software. The Electronic Industries Association predicted, for example, a growth of software costs of 680% as compared to a growth of hardware costs of 340% for U.S. Department of Defense's (DOD) embedded systems development between 1980 and 1990 [Davis, 1993].

The emergence of this wide spectrum of embedded systems, and the increasing use of software for implementing the functionality, has led to increasing demands for more sophisticated *Embedded Software Maintenance*.

Regular software maintenance is time-consuming, cost-intensive, and error-prone. The costs for maintaining a system after it has been put into operation are usually much higher than providing similar functionality when software is originally developed. As a matter of fact, 40 to 75 percent of the total life-cycle costs of software are consumed in the maintenance phase [Lientz and Swanson, 1980].

Maintenance of embedded software is much more expensive than maintenance of non-embedded software. A study performed at the DOD found that maintenance of embedded software costs approximately $110 per line of code, while the cost for non-embedded software is $5.60 per line of code [Clark, et. al., 1999]. The growing use of embedded software and the demand for ways to decrease the cost of embedded software maintenance motivates further study.

1.1 Lack of Research

Despite its importance, a search of the literature on the topic of embedded software maintenance uncovers a lack of systematic investigation, let alone results, in that field. This impression is backed by [Sahin and Zadeli, 2001]. The difficulty of finding literature on embedded software maintenance can be related to the fact that the term embedded is rarely used.

"Personal computers are easy to describe. The English word 'computer' is enough to put that very image in the mind of at least a billion people; a quarter as many own one. Embedded systems are completely different. Few other than engineers have heard the term; most don't even notice that their microwave or cell phone has a computer at its heart. Even the designers of such systems can't always agree on what exactly the category includes. The company Cisco doesn't think of itself as an 'embedded system design company', even though many of the engineers who work there may consider that they do just that" [Barr, 2002].

It is hard to draw a sharp line between embedded and non-embedded systems. Many of the issues in regular software maintenance are likely to be similar to the ones experienced in embedded software maintenance. Therefore, the basic techniques and solutions are related. However, embedded software possesses characteristics that demand special treatment.

1.2 Software Maintenance vs. Embedded Software Maintenance

Early work on software maintenance defined software maintenance narrowly as correcting errors and broadly as expanding and extending software functionality (i.e., as continued development) [Chapin, et. al., 2001]. This viewpoint is reflected in the four dimensions of software maintenance [Lientz and Swanson, 1980]:

- *Corrective*: repair of discovered faults
- *Adaptive*: environmental adaptations (e.g., upgraded hardware, embedded COTS)
- *Perfective*: functional enhancements due to new and/or revised requirements
- *Preventive*: changes to increase maintainability of software

These dimensions give structure to the "reactive" part of maintenance activities, exercised once a product is delivered, to maintain and to renovate as necessary. This model primarily refers to the system owner or user point of view of maintenance [Chapin, et. al., 2001]. The model was quite successful in the past, since typically few changes occurred once software was deployed: In the past, electronic control units were replaced, not “refurbished-in-place” with updated software. With reactive maintenance becoming more and more complex and expensive, software developers are seeking more “proactive” approaches to maintenance.

1.3 Organization of This Report

The goal of this report is to describe the state-of-the-art of embedded software maintenance and provide a glimpse of state-of-the-practice embedded maintenance practices. Section 2 introduces the area of embedded software based on embedded systems, the implication of embedded software and some of its characteristics. Section 3 describes the more general area of software maintenance, different process models and characteristics. Section 4 analyzes embedded software maintenance and investigates how the characteristics of embedded systems affect software maintenance and then describes the typical problems and issues in maintenance. This section also links typical problems to the case studies that can be found in Appendix A. Section 5 discusses potential solutions. One solution, impact analysis, addresses problems that are identified in many case studies; we discuss impact analysis in Section 6. References are listed in Section 7. Appendix A contains details of the case studies. In Appendix B, we list some of the resources that are available for readers interested in further studies of embedded software maintenance.

2 Embedded Software

We believe that in order to understand how to best manage embedded software maintenance, one needs first to understand the characteristics of embedded software maintenance. This means understanding what characterizes embedded systems and embedded software, as well as the commonalities and differences between regular software maintenance and maintenance of embedded software.

2.1 Embedded Systems

"An *embedded system* is a part of a product with which an end user does not directly interact or control."[1]

Other characteristics are:

- Embedded systems consist of computers affiliated with products for implementing control, communication, usage and other intelligent functions
- Products with embedded systems include modems, disk drives, digital cellular phones, radios, audio CD players, music synthesizers, videodisk players, sonar, radar, confocal microscopes, Magnetic Resonance Imaging (MRI) medical instruments and systems, video telephones, automobiles, industrial machines, airplanes and missiles
- Embedded systems utilize mechanics, electronics, and hardware and software technologies that are closely related to each other
- Embedded systems often have no real keyboard and a limited display
- Embedded systems often have real-time requirements, i.e., correctness is partially a function of time
- Many embedded systems must be robust, i.e., their behavior must always be controlled, even during system failure

The conceptual structure of *Embedded Systems*, using a telecom network and mobile phones as an example, is illustrated in Figure 1 [Kuvaja, et. al., 1999]. Embedded computer systems consist of software and hardware (electronics and mechanics). *Embedded products* include *embedded computer systems* directly incorporated into electromechanical devices, called the *target environment.* When the target environment is distributed, it causes complicated control and communications problems. *The use environment* of the embedded product includes *end-users, operators* and *standards*. End-users of the embedded product range from non-technical to highly technical people who use the embedded product for their own purposes. Operators are those who manage, support and utilize the embedded product. One example of an operator is a telecom operator who manages and controls a telecom network for mobile phones. Standards set

[1] www.embedded.com

a basis and regulate the use of the embedded product in its use environment. An example is a mobile communication standard, such as GSM.

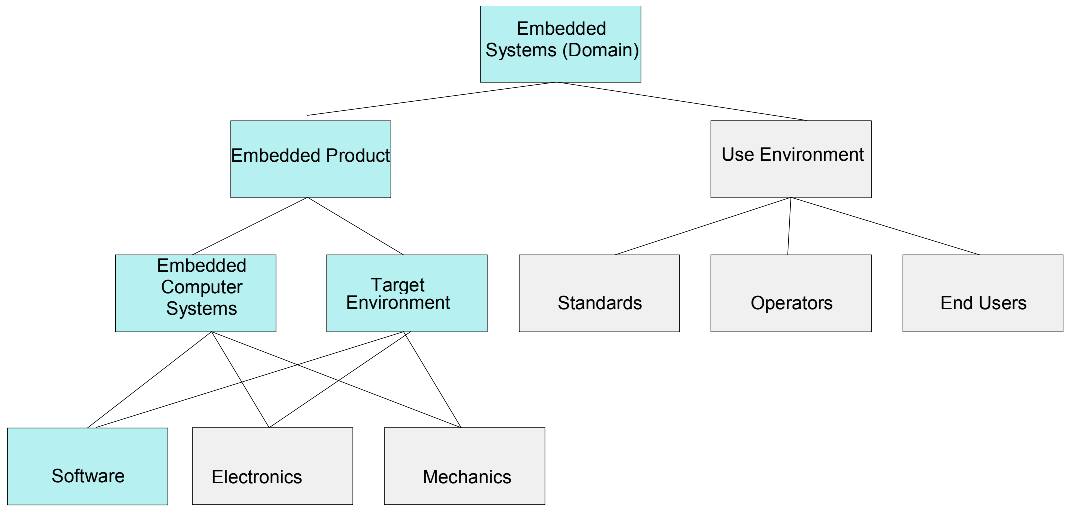

Figure 1: The Conceptual Structure of Embedded Systems [Kuvaja, et. al., 1999]

There are many different kinds of embedded systems. They can be divided into four application types [Koopman, 1999]:

General Computing

- Applications similar to desktop computing, but in an embedded package
- Examples are video games, set-top boxes, wearable computers, and automated tellers

Control Systems

- Closed-loop feedback control of real-time systems
- Examples are vehicle engines, chemical processes, nuclear power, and flight control systems

Signal Processing

- Computations involving large data streams
- Examples are radar, sonar, and video compression

Communication & Networking

- Switching and information transmission
- Examples are telephone systems and the Internet

2.2 Embedded Software

Embedded system design focuses on the implementation of a set of functionalities satisfying a number of constraints. The choice of implementation determines which functionality will be implemented, either as a hardware component or as software running on a programmable component [Sangiovanni-Vincentelli and Martin, 2001]. The complexity, coupled with constantly evolving specifications, has forced designers to look at implementations that are flexible and can be changed rapidly. Since hardware manufacturing cycles do take time and are expensive, interest in software-based implementation has risen considerably. Increased processor performance and decreased size and cost have shifted functionality to software [Sangiovanni-Vincentelli and Martin, 2001].

Table 1 shows, as an example, the growth of embedded software in mobile telephones during the past fifteen years [Karjalainen, et. al., 1996]. Consequently, considerable product development efforts are expended on software. The flexible nature of software has made it a core technology for implementing customer specific features. It is estimated that by the year 2010 there will be over ten million professionals developing embedded software [Seppänen, et. al., 1996].

Table 1: Increase in the Amount of Code Embedded in Mobile Phones [Karjalainen, et. al., 1996]

Generation	System Type	Example System	Software Size
1984: 1st	Analogue	Nordic mobile phone system (NMT)	Some Kbytes
1988: 2nd	Analogue	NMT	Tens of Kbytes
1992: 1st	Digital	Global system for mobile telecommunications (GSM)	Hundreds of Kbytes
1996: 2nd	Digital	GSM	About One Million Bytes

Most embedded software has a development environment that is different from the target environment. The development environment enables development of embedded software on a regular workstation. The development environment often has simulators and other means for verifying that the software works as intended. When the software is "finished," it is uploaded to the target environment. Typically, the software cannot be debugged or changed in the target environment. Necessary changes are completed in the development environment, and a compiled version is again uploaded to the target environment. The difference in target and development environment makes embedded software maintenance much more complicated.

3 Software Maintenance

All software maintenance, whether it is embedded or not, deals with error correction or enhancements. The area most dealt with in the literature, is corrective maintenance. Maintenance related to further development of the system, which we call enhancements or evolution, is not as well covered in literature. These two types of maintenance will be described first, and then more extensive models will be discussed.

3.1 Corrective Software Maintenance

When a system does not conform to specifications, a trouble report is issued. The maintenance programmer must investigate the trouble report, map the erroneous behavior to the internals of the system, localize the primary and secondary changes to be made to fix the problem, accomplish the changes and test the system to ensure that (a) the problem is removed, (b) no new problems are introduced, and (c) the system is still functioning according to the intentions of the system.

Corrective software maintenance can be characterized as follows [Weiderman, et. al., 2002]:

- It is a fine-grained, short-term activity focused on (possibly a large number of) localized changes. Good examples include the Y2K problem and the Euro conversion
- The structure of the system remains relatively constant, and the changes produce few economic and strategic benefits
- There is a tendency to respond to one software requirement at a time. There are few economies of scale that accrue from software maintenance

3.2 Enhancements and Evolution

When specifications are changed, i.e., a new requirement is added or an existing requirement is changed, the programmer has to understand the new requirement and how it relates to the set of existing requirements, how the existing requirements relate to existing user functionality, and where to insert the new user functionality into the existing user functionality. The primary place for the change is determined by mapping the user functionality to the system and finding the place to insert the new code. Secondary changes, changes in the surroundings of the system, must then be determined. When the changes are implemented, the system is tested to ensure that (a) the new requirement is implemented, (b) no new problems are introduced, and (c) the system is still functioning according to the system intentions.

System evolution can be characterized as follows [Weiderman, et. al., 2002]:

- It is a coarser grained, higher level, structural form of change that can make the software system potentially easier to maintain
- It allows the system to comply with broad new requirements and gain whole new capabilities
- Instead of changing software only at the level of instructions in a higher level programming language, changes are made at the architectural level
- System evolution increases the strategic and economic value of the software by making it easier to integrate with other software and making it more of an asset than a liability

The purpose of the maintenance process as defined by ISO/IEC 15288 is to sustain the capability of the system to provide a service. It monitors system performance, records problems for analysis, makes corrective and preventive actions and confirms restored system capability. IEEE Std. 1219-1998 defines software maintenance as a post-delivery activity as follows:

> "*Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.*"

3.3 Software Maintenance Process Models

Maintenance can be approached from many directions. To ease the discussion of maintenance and related maintenance processes, a domain factor model of software maintenance was proposed [Kitchenham, et. al., 1999] (see Figure 2).

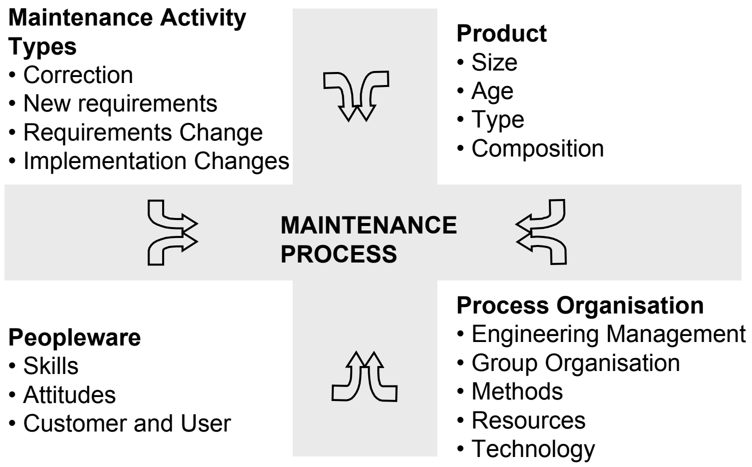

Figure 2: Domain Factors Affecting Software Maintenance

3.3.1 V-model Type Maintenance Processes

A more detailed analysis of maintenance activities results in six types that affect the maintenance process [Harjani and Queille, 1992] (see Table 2). The V-model type maintenance process model [Harjani and Queille, 1992], which is primarily applicable to corrective changes, is illustrated in Figure 3. The model presents the basic maintenance activities and their sequential order.

Table 2: Types of Changes

Type	Description
User Support	Activities for providing answers to users' information requests and correcting misunderstandings
Corrective Maintenance	Activities for error correction in software without making any changes to the requirements
Evolutive Maintenance	Activities for adding new functionalities in response to new or changed functional requirements
Adaptive Maintenance	Activities to adapt software to changes in the operational environment
Perfective Maintenance	Activities for improving non-functional requirements such as execution time
Anticipative Maintenance	Activities to anticipate future problems and increase software's robustness or render it easy to modify if the changes are realized in the future.

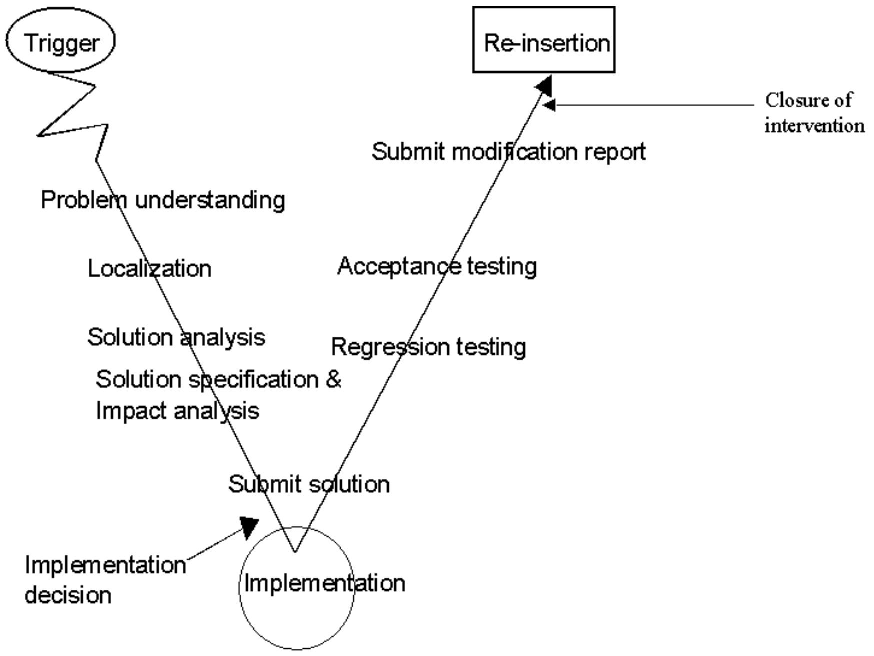

Figure 3: The V-like Software Maintenance Process Model [Harjani and Queille, 1992]

3.3.2 The AMES Maintenance Process Model

The AMES model suggests a higher-level view of maintenance processes. It identifies three layers: strategic, management and technology [Mäkäräinen, 2000]. The strategic layer makes decisions regarding the future of the product and determines customer or user relationships. The activities include marketing, budget allocation, training and process improvement. The management layer plans activities and tracks progress, makes implementation decisions, manages problems, and initiates and closes changes. The technical layer implements the changes. The three-layer maintenance process model is presented in Figure 4.

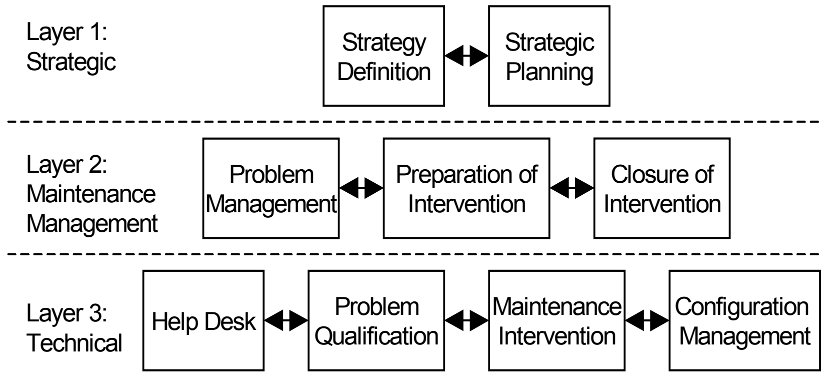

Figure 4: The AMES maintenance process model [Mäkäräinen, 2000]

3.4 Maintainability

It is too late to worry about maintainability when software is released. This is especially true for embedded software, which is often hard to change once it is delivered. Maintainability must instead be built into the system from the very beginning, during planning and design phases. Design decisions in early phases have been found to have a larger impact on maintainability than implementation algorithms [Rombach, 1987]. The lack of design is illustrated in Figure 5 [Pressman, 1992]. If the design phase of software development is ignored (or of poor quality), it has an enormous negative effect on all following activities, including maintenance.

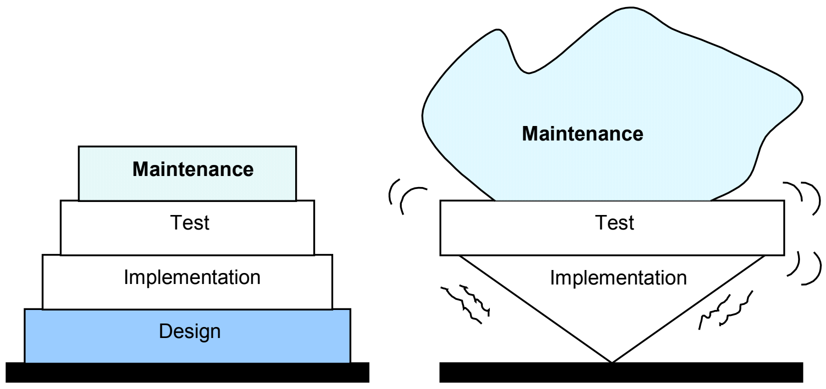

Figure 5: The Relationship Between Design and Software Maintenance [Pressman, 1992]

4 Embedded Software Maintenance

There are a number of reasons for the high costs and low productivity of implementing changes in any type of software:

- Maintenance personnel often maintain systems they did not develop. In addition, staff members are often relatively inexperienced and unfamiliar with the application domain. Maintenance has a poor image among software engineers. It is seen as a less advanced process than system development and is often allocated to the most junior staff.
- The software systems being maintained may have been developed many years ago without modern software engineering techniques. They may be unstructured and optimized for efficiency rather than understandability. The software system being maintained is often poorly documented or the provided documentation is inconsistent with the developed code. The documentation may, therefore, be an unreliable aid for understanding and changing the system. The code was often written before the documentation. Frequently, a separate group, one that does not include any of the designers, writes documentation. Usually, programmers consider the pre-implementation documentation so vague that it is nearly useless.
- The structure of the changed software system tends to degrade. This makes the system harder to understand and makes further changes difficult because the system becomes less cohesive.
- Changes made to a software system in the past may introduce new faults that trigger further change requests. New faults may be introduced because the complexity of the system may make it difficult to assess the effects of a change. Relationships between software entities are not documented explicitly. Ripple effects of changes cannot be controlled.
- Change processes are often not guided. Process descriptions for activities, like change, version, or release management, do not exist.

4.1 Challenges in Embedded Software Maintenance

In principle, the embedded software maintenance process does not differ from the non-embedded software maintenance process. It has the same main phases, but, due to its special nature, additional factors must be taken into consideration.

Maintenance needs differ depending on the type of software. Different embedded software may further produce application-independent software maintenance needs. It is possible to distinguish between three classes of embedded software [Taramaa, et. al., 1996]:

1. Product software not unique to the application (to be used in several kinds of products)

2. Special system software for the operating system, communication and device control
3. Application software (specific to the application)

Using this embedded software taxonomy, the following concerns related specifically to embedded software maintenance are identified [Taramaa, et. al., 1996]:

1. A new version of non-unique product software may be incompatible with the rest of the software
2. System software may be so specific that a change of operating system or hardware might require extensive modifications or a total rewrite of other code
3. Product software often operates only on a specific hardware platform, so that hardware changes make the original software incompatible
4. Application software may become incompatible with the special system software and hardware when switching to new computing platforms

Specific characteristics of embedded systems that characterize and set limits for software maintenance are [Kuvaja, et. al., 1999]:

1. Software is closely connected to hardware
2. Product design is constrained by the implementation technology
3. Different technologies used in the same embedded system are developed in parallel
4. Different low-level and high-level software implementation technologies are used
5. Real-time processing and data management are used, often based on a distributed system architecture
6. Reliability requirements are crucial
7. Device autonomy is used
8. Maintainability and extendibility through new functions, technologies and interfaces often required
9. Cost of the product is important in mass-markets
10. Short development lead-time (time-to-market) is required

In addition to the embedded software-specific characteristics set by the hardware connection, all the issues that make maintenance difficult in general are, unfortunately, valid also for embedded software maintenance.

Reasons that complicate embedded software maintenance are [Mäkäräinen, 2000]:

- Age of software
- Loss of design knowledge
- Loss of original requirements
- Accumulation of problems and change needs
- Lack of design for change
- Time pressure
- Diversity of tools and information

- Poor image of change function
- Poor maintainability of just-released software
- Code decay
- Few tools and methods
- Verification across several product versions
- Research focus on developing new software, not managing existing systems

Four embedded software characteristics that impose particular difficulties in performing changes in this type of system are [Mäkäräinen, 2000]:

- **Concurrent systems engineering.** Hardware and software components are developed concurrently. The requirements for both the hardware and software components change while they are being developed, creating more difficulties in the process of managing the changes. Also, problems in testing and the location of errors are very likely since the hardware environment where the software will run will be available late in the development process.
- **Sharing of software parts in several products or product families.** Since customizing products with software is more cost effective than with hardware, it is typical to have a common hardware with different versions of software to customize the products. Since new products are often based on previous versions of the system, a lot of problems in change management exist.
- **Primitive software engineering environments.** Embedded systems often have time, memory or power-consumption constraints. The compilers for high-level languages do not optimize the code enough to meet those constraints. Consequently, software is always developed in assembly language and in very primitive software engineering environments, with almost no support for testing, code measurement, etc. As a result, the code is very hard to understand. Also, the tight relationship between hardware and software make it harder to locate the problems.
- **High reliability demands.** For some embedded systems, it is impossible to make any changes to the system after delivery. Some systems might be in an inaccessible environment, while other systems are released in a huge volume, making changes after delivery very difficult and costly.

We also recognize these other characteristics of embedded software systems that have an impact on the process of maintenance:

- **Hardware constraints.** Embedded products often have hardware constraints of weight, size, cost, power-consumption, memory, etc. As a result, code is written with optimization in mind rather than understanding, making it harder to manage changes.
- **Development environment is different than target environment.** Frequently, developers do not have access to the target system for various reasons. Development is therefore done in a development/simulation environment that has a different behavior than the target environment, since some aspects of the target

environment are hard to simulate. Also, the target environment is often a specialized environment that requires well-trained developers.

4.2 Main Issues Found in Case Studies

In order to better characterize embedded software maintenance, we analyzed case studies that cover the following product areas: Consumer electronics, DSL-Modems, Mobile phones, Airplanes, Satellites, Field appliances, Automobiles, and Industrial machines. The case studies gave us insight into common problems, ideas and solutions in embedded software systems. Some case studies were found in the literature and we conducted others. The case studies report how maintenance is performed in practice, identify typical problems practitioners have, and, in some cases, describe how they solved those problems. The case studies can be found in Appendix A.

This section presents the main issues identified in the case studies. It is important to notice that while the problems were mentioned or reported by specific organizations, we believe that almost all these problems are common in all embedded systems development. Some of the problems are related only to embedded systems, while others are also observed in the maintenance of non-embedded software systems. However, common problems in other types of systems are accelerated in the embedded systems environment. Another important note is that these problems are observed at this date, some will be eased with future technological advances, while others will always exist.

4.2.1 Unstable requirements

Some of the organizations (telecommunications systems and space instruments) mention unstable requirements as a primary problem. Requirements are added and changed frequently; this causes problems in the projects. In an environment where the change of requirements is the rule, it is common that the changes are not effectively communicated to all partners, causing problems in the integration phase and making software maintenance more complex.

4.2.2 Technology changes

One common problem for some organizations (consumer electronics, automobiles, airplanes and space instruments) is related to changes in technology.

In the consumer electronics and automobile organizations, technology changes are related to changes in the development tools. In the consumer electronics case, new versions of the development tools are delivered several times during the maintenance phase of the software. Problems occur if the software has not been changed in any way for a long time and there have been several new compiler releases during that time. It is difficult to use a new version of a tool when the maintenance activity needs to be performed quickly. In the case of the automobile company, it is not enough to put software under configuration management. Tools, compilers, and documentation need to be under configuration control. This can be especially complicated if a contractor did part of the work.

In the organization that develops airplanes, technology changes are related to changes in the technical configuration, causing changes in the software. To avoid this situation, development teams try to stick to a given technology as long as possible. In addition, the selected technology has to be available for a long time.

The organization that develops space instruments reports problems with version control of code. No automated configuration management tools have been used during the prototyping phase. Frequently, one person conducted the programming of a module and when someone else made changes in the code, version control of the code became very difficult.

4.2.3 Location of errors

Many organizations (consumer electronics, telecommunications systems and automobiles) mention that error location is very time-consuming. In the consumer electronics organization, the reason is that the person who has to find the error is not the one who reported it. In environments where location is very time consuming, it makes the whole maintenance phase become longer and more complex. The "organization developing automobiles" case study demonstrates that their chief problem lies in determining whether the hardware or software caused a failure, because (a) failure reports of the software do not contain sufficient information to allow the cause of a failure to be identified clearly; (b) system failures may be difficult to reproduce; and (c) it cannot be objectively judged whether the hardware caused the failure or whether the software was not "intelligent" enough to compensate for existing hardware limitations.

4.2.4 Impact of change

Identifying the impact of changes poses another common problem. Once the problem is localized or a new change is requested, it is necessary to decide which parts of the software will be modified to fix the problem or implement that change. Changes cause ripple effects that need to be determined to avoid introducing new problems. Some of the organizations (consumer electronics, telecommunications systems, telephone switches and automobiles) report that locating changes and identifying ripple effects are very time consuming tasks.

In the consumer electronics industry, modifications of the core part of the software will also affect other products using the same core, making it even harder to identify the impact of the change. In the "telephone switches" systems, the problem lies in the fact that it is very hard to understand how the switches work. Very few people understand the whole system, and it is very hard to determine the impact of a suggested change. In the "automobiles" industry, the problem is related to the fact that hardware and software interact so closely that it becomes difficult to determine the ripple effects in such an environment.

4.2.5 Need for trained and specialized people

In some industries, the maintenance process is complicated by the difficulty in understanding the software programs. In the case of space instruments, the hardware-oriented code is hard to understand, requiring additional time for familiarization.

In the case of airplanes, the complexity lies in the fact that many different computer systems are involved. The computer systems communicate with each other and run at different processor speeds. Data is moved back and forth between these systems and it can easily seem as though the data was transferred correctly when actually, in real time, not enough time was devoted for the memory cells to stabilize. This makes it very easy for mistakes to occur. In addition, most of the systems, including hardware, are developed for a specific purpose, which creates problems with staffing. A long training process is, for example, necessary for new people not used to the environment.

In the "telephone switches" case, a majority of the development environment (hardware and software) is non-standard. As a result, little knowledge and experience sharing can occur between the company and the outside world. Likewise, it takes a long time to familiarize new people due to the fact that they often have little or no experience in similar environments.

4.2.6 Lack of documentation

The lack of documentation causes further problems. In the case of consumer electronics, the development phase does not produce much documentation. The maintainer is responsible for updating the documentation manually if needed. Thus, documentation is often not up-to-date. The "automobiles" industry also reports that, technically, maintenance requires that documentation is being used and updated, but the documentation is never good enough and it shows the obvious, not the relevant. The organization also reports a lack of documentation (or knowledge sharing) that would enable the organization to remember how to use the languages and tools that form part of older products. In addition, when systems are built on several modeling styles, knowledge gets lost when old styles are forgotten.

In the Space Instruments case, the specifications and requirements for the software have changed frequently. Most changes have been made directly to the source code. Sometimes, document updates have been omitted because of frequent changes that resulted in inconsistent documentation.

The telecommunications systems organization reports a problem related to the lack of traceability and design rationale. Traceability information between a change request document and the modified document is often missing. The original motive and effort spent for each modification is not recorded. The modification is usually recorded, but the reason why it was undertaken is not. Project plans are changed according to new requirements, but no statement is included that explains why the change to the plan was implemented.

4.2.7 Simulation environment versus target environment

Some industries face challenges related to the fact that development occurs in a simulation environment, and then the software is uploaded to the target environment. Often, it is not possible for every developer to have access to a prototype of the target environment, and development has to rely on simulation environments that behave differently than the target environment.

In the "mobile phone systems" organization, the problem is that the behavior in the simulation environment differs from the target environment. Limitations are typically related to real-time aspects that are hard to simulate. One such aspect is that the network requires that a call be answered within a certain time. Another problem is that memory allocations are done in a different way in the simulated environment compared to the target environment. This makes it hard to detect memory problems early. It is difficult to debug the software once it is downloaded to the cell phone because any changes to the software cause the system to behave differently.

In the "airplanes" systems, the largest problem reported notes that real-time aspects do not show in the simulation environment. Some of these aspects can be spotted in more advanced flight simulators, but the most intricate ones do not manifest until operation on the airplane.

In the case of the telephone switches organization, equipment is very expensive and the developer does not have access to the switch, i.e., the target environment.

4.2.8 Hardware constraints

Other challenges faced in the development and maintenance of embedded software systems are due to hardware constraints. In the "mobile phone systems" for instance, the lack of memory is a problem because mobile phones have to be small and are sensitive to any increase in cost, weight, and size. This results in code optimized for fitting into a small memory instead of for readability.

In Digital Subscriber Line (DSL) modems, cost, size and weight sensitivity put severe constraints on feasible solutions. Adding more memory (for example, redundant memory to make sure upgrades do not go wrong) adds to the cost of the product. Adding hardware also adds to the size and weight of the product, which also can be a problem due to such limitations.

4.2.9 The testing process

Another problem reported by some of the organizations has to do with the difficulty or deficiency of testing and review processes. In the consumer electronics industry, testing is done manually without any support. Testing takes too much time, and the coverage of the testing procedures is not known. Subsequently, the most serious errors are often found when the product is already in the field. In telecommunication systems, regression testing is difficult because of the complexity of the system.

In space instruments, a lot of time is spent in testing because of the requirement for high reliability of the software. No systematic method to ensure that the coverage of the case software test is complete and sufficient has been used.

4.2.10 Other systems specific problems

Some organizations reported other types of problems that are specific to their organization and applications.

The telecommunication systems organization reported a code decay problem. The software is reused repetitively in many subsequent projects, causing the software to decay. It is, however, hard to detect such decay.

The "Digital Subscriber Line (DSL) modems" organization reported a problem related to the complexity of the upgrading procedure. If the upgrading procedure is incorrect, it threatens the overall functionality of the modem. The most important problem is to protect sensitive memory from being overwritten during upgrading of software. A strategy for reducing the risk of overwriting memory is to have a double, redundant memory. The new software is then uploaded to one of the memory units while the other one serves as backup. If anything goes wrong, the system falls back to the backup.

The "satellites" organization mentioned a problem related to its remote, disconnected system. It is not easy to reboot remote space systems, and software failure is more likely to cause loss of the system or the mission.

The space instruments organization faces a problem related to concurrent development. Since product parts are designed and built simultaneously by separate subcontractors, the integration of the devices cannot be accomplished until each partner has achieved a stable version of their parts.

The telecommunications organization mentioned a problem related to the lack of learning based on defect data. No systematic practices for analyzing the change data for learning or improvement purposes exist.

5 Potential Solutions

After reviewing the case studies, we identified the main issues faced by organizations when dealing with embedded software maintenance, which were presented earlier. This section indicates possible solutions to resolve some of these issues. There is no silver bullet, but ideas from this section can be used to make maintaining embedded systems easier and more efficient.

Companies can influence most of the issues listed in Section 4. Most actually result from poor software quality and management of the software engineering process itself. Some of the potential solutions described here are even based on some of the case studies themselves. Knowledge Management is, for example, a new area that could provide a general solution to the problem of losing knowledge in software organizations [Rus and Lindvall, 2002]. Another potential solution is to verify that maintenance and testing processes are in place and conducted in an efficient way. Other, more specific solutions observed in the case studies include, for example, duplication of resources, a course of action that is already used in many systems (see Appendix A, Sections A.4 and A.6 for more information).

The following sub-sections present possible solutions for improving the process of maintaining embedded software systems. We will discuss a new approach to maintenance planning, Reliability Centered Maintenance, that has been applied in numerous domains to the maintenance of heterogeneous systems, and we will also investigate technical management of external teams as ways to improve the process. We will also comment on software configuration management and impact analysis as specific activities that can help organizations improve. As impact analysis was mentioned in many case studies, we will devote a special section to it. But before we go into the solutions, let us first revisit the area of maintainability in order to lay the groundwork for possible solutions.

5.1 Maintainability

Even though the IEEE Standard on Software Maintenance 1219-1998 defines maintenance as a post-delivery activity, it acknowledges the need for overlap by stating "Ideally, maintenance planning should begin during the stage of planning for software development." [Life Cycle Working Group, 1998]

By looking at the maintenance process, one understands better what properties make a system easier to maintain. The process of maintenance can be broken down into the following sub-processes:

1. Understanding the change request
2. Understanding the system and its structure

3. Locating where to change the system in order to implement the change request (primary changes [Lindvall, 1997])
4. Implementing the primary changes
5. Determining the ripple effects (secondary changes resulting from primary changes [Lindvall, 1997])
6. Implementing secondary changes
7. Testing that the system fulfills all previous, as well as new, requirements

Due to the fact that the maintenance process is iterative, the stated order above indicates steps to be taken, rather than the order in which they always occur.

Many studies indicate that there is only a vague connection between the requirements of a system and its structure [Soloway, 1987]. This vague connection makes it hard to conduct steps 1 to 3 above because they are dependent on each other. It is impossible to fully understand a change request without comprehending the system, its structure, and where in the system the change can be made. In the same way, it is hard to determine ripple effects without implementing the primary changes, and the implementation of the secondary changes might cause new ripple effects, necessitating another set of secondary changes, and so on.

Ripple effects cause problems related to coupling between the pieces that make up the software system, e.g., modules or components [Haney, 1972], [Yau and Collofello, 1980]. Both Collofello and Haney show that tighter coupling increases the risk of ripple effects. Thus, reduced coupling between components reduces the risk of ripple effects, change propagation and the generation of secondary changes as a consequence of primary changes. Consequently, reduced coupling results in systems that are easier to maintain. Guidelines like this can be used when designing and planning architecture to render the system more maintainable.

As described earlier, maintainability can be viewed in many ways and depends on many different factors. Code that is written according to a set of well-defined design rules and guidelines is easier to understand, and therefore easier to maintain. Low complexity, low coupling and high cohesion, in theory, indicate a higher level of maintainability. A system that is easier to understand, change, and test is easier to maintain. A system with well-structured, clearly-defined subsystems that is well documented and uses clear and logical names is easier to maintain. A smaller system with equivalent functionality is generally easier to maintain than a bigger system. A system that offers fewer features is also easier to maintain.

While all of these characteristics indicate maintainability, some cannot be assessed until the system has been operational and undergone changes. There are techniques and methods that help the assessment of architecture properties that could be used to evaluate the maintainability of an architecture before the system is implemented. One example of such a method is the Architecture Trade-Off Analysis Method (ATAM) [Clements, et. al., 2001]. The ATAM method uses brainstorming sessions to perform a trade-off analysis of architectural decisions based on their impact on different quality attributes that are important for different classes of stakeholders. Maintainability could be one of the quality attributes. The method is valuable in an earlier phase of the development at the time when the design is being created. Once the architecture is evaluated, the design decisions are made and should be implemented.

Once the system is implemented, it is always a good idea to ensure that it matches the design, especially if some effort was put into making decisions in the architecture to ensure certain properties are achieved. Lindvall [Lindvall-a, et. al., 2002] [Lindvall-b, et. al., 2002] and Tesoriero [Tesoriero, et. al., 2002] present a method that could be used to (a) make sure the architecture maintains the properties desired to reach a certain quality attribute and (2) make sure the implementation conforms to the design. These assessments can be made for every version of the system when the method can be applied.

Observing the case studies, we can identify situations where the organizations made design decisions to make the system more maintainable despite the complexities and particularities of the hardware. The "mobile phones" and "DSL modems" organizations use layered architectures. The goal is to encapsulate the hardware by software so that less hardware-oriented software can be developed. This renders it easier to maintain the software as well as reuse it for similar products.

5.2 Reliability Centered Maintenance

Reliability Centered Maintenance, or RCM, is a new approach to maintenance planning that has been applied in numerous domains to the maintenance of heterogeneous systems. It is not a software maintenance approach, per se, but applies to entire systems of hardware and embedded software. RCM is defined as "a process used to determine what must be done to ensure that any physical asset continues to do what its users want it to do in its present operating context" [Moubray, 2001].

The roots of RCM appeared in the early 1960s. The North American civil aviation industry, in response to increasingly expensive and risky maintenance practices, put together a series of "Maintenance Steering Groups" to re-examine these practices.

The RCM process begins by defining users' expectations in terms of primary performance parameters such as speed, range, risk, quality, economy, etc. The next step is to identify ways in which the system can fail to live up to these expectations (failed states), followed by a failure modes and effects analysis (FMEA) to identify all the events that are reasonably likely to cause each failed state. Finally, the RCM process defines actions to manage each type of identified failure, e.g., predictive or preventive maintenance, design changes, or changes to operational procedures.

The seven basic questions that drive the RCM process are as follows [Moubray, 2001]:

- What are the functions and associated performance standards of the asset in its present operating context?

- In what ways does it fail to fulfill its functions?
- What causes each functional failure?
- What happens when each failure occurs?
- In what way does each failure matter?
- What can be done to predict or prevent each failure?
- What if a suitable proactive task cannot be found?

One of the distinguishing characteristics of RCM is the centrality of the user and the user's expectations. Not only is it users' expectations that initiate and drive the process, but also users (i.e., people who live with the system under maintenance day in and day out) make up an important part of the RCM Review Team. The idea is that such people are an invaluable source of information.

The RCM Review Team [Moubray, 2001] is a small team that includes at least one person from maintenance and one from operations. All team members should have a thorough knowledge of the asset under review and should also have been trained in RCM. This team structure allows management access to a wide range of knowledge and expertise, while the members themselves learn a great deal about how the system works. The members of a typical RCM review team include a facilitator, an engineering supervisor, an operations supervisor, a technical expert, an operator, and an external specialist (if needed).

The major outcomes of the RCM approach are [Moubray, 2001]:

- Greater safety and environmental integrity
- Improved operating performance (output, product quality and customer service)
- Greater maintenance cost-effectiveness
- Longer useful life of systems
- A comprehensive database of maintenance planning and activity
- Greater motivation of individuals
- Better teamwork

In practice, RCM (even streamlined versions of it) has often been found to be highly labor-intensive. Another problem is that the specialized team that carries out the RCM process does not always couch its recommendations to the maintenance organization in a way that is easily translatable and does not promote ownership on the part of maintainers. The cost of RCM can run as high as $40,000 a system and take as long as two years to complete, even when the process is streamlined.[2]

5.3 Software Configuration Management (SCM)

SCM supports maintenance activities by providing the means to identify the configuration of the software by systematically controlling changes of the configuration,

[2] http://www.fractalsoln.com/reliability.html#improvement.

and by maintaining traceability of the configuration throughout its life cycle [Dart, 1991], [Tichy, 1988]. It is the hardware of embedded computer systems that changes the role of SCM compared with its role in the context of a pure software system. Because embedded software is part of a physical product, SCM must be seen in a more comprehensive framework that includes interfaces with hardware and other technologies [Taramaa, 1998].

5.4 Technical Management of External Teams

Technical Management of External Teams is becoming increasingly important in the maturing software domain where, as in other domains, more and more components are purchased rather than built in-house. The decision to use external suppliers, be it COTS suppliers or subcontracted developers, is always tied to the question of whether the component to be purchased is considered to be part of the core competence of the company.

In the embedded domain, system components are either outsourced entirely, which can mean that a company loses or fails to build up a potentially vital competence, or are built in-house with contractors that work on-site. In the latter case, additional requirements for management are the selection of additional staff (e.g., free-lancers) and appropriate contracting policies that leave the intellectual property rights with the company. For subcontracting, the following issues must be addressed:

- Identification of components that are candidates for outsourcing
- Formulation of requirements that allow for tendering, bid-assessment, and controlling the subcontract during execution
- Systematic assessment and selection of components (COTS)
- Systematic assessment and selection of subcontractors
- Contracting and controlling subcontractors
- Integration of the purchased componentMetrics/measures for managing acquisition of COTS

5.5 Impact Analysis

The implementation of changes has to be planned. Planning requires an accurate estimate of the cost caused by the implementation of the changes. The cost estimate has to be precise because it is used as a basis for deciding whether the changes are to be implemented or not. An accurate estimate of the costs can be supported by a precise estimate of the impacts of changes. The process of identifying the entities that are affected in the software by a software change is called *impact analysis*. Impact analysis provides, for example, visibility into the effects of changes before the changes are implemented. Traditionally, determining the effects of software change has been something that software professionals do intuitively, after some cursory examination of the code and documentation. This may be sufficient for small software systems, but not for large ones. In addition, in a case study, Lindvall and Sandahl [Lindvall and Sandahl, 1998], showed that even software professionals predicted incomplete sets of change impacts which indicates that impact analysis is difficult. Impact analysis for embedded systems resembles impact analysis for non-embedded systems, but is expected to be even more difficult. Change is typically propagated via dependencies between software entities. Embedded systems are more complex than non-embedded systems because they have additional dependencies. Examples are semantic and time-sensitive dependencies that are not typically recorded and that result in hidden side effects. These side effects are not very well understood. As a matter of fact, the paper by Li and Feiler [Li and Feiler, 1999] was the only paper that we found that addresses this issue. Nevertheless, embedded software can apply many of the general concepts and techniques for impact analysis and, as impact analysis was identified as a critical area in many of the case studies, we expand on the topic in the next section.

6 Impact Analysis

The goal of this section is to provide more detail on impact analysis. It starts with the problem of ripple effects, which was discovered relatively early and resulted in a set of techniques based on dependency analysis as the mechanism for propagation of change. Dependency analysis research lays the foundation for impact analysis and the understanding of the problem of detecting change. Little is said, however, in that kind of literature about how to identify the initial set of changes. These *primary changes* are the changes that stem from new requirements and cause the ripple effects (or secondary changes).

As the size and complexity of software grew and many organizations defined and documented requirements prior to implementation, interest also grew in relating requirements to the software artifacts, which is also known as *traceability*. This relationship can be used for identifying primary changes. Traceability techniques and dependency analysis techniques can be combined to account for initial changes (primary changes), as well as ripple effects (secondary changes).

In this section, we also describe a framework for impact analysis, a model for impact analysis, as well as experiments with different impact analysis approaches and techniques.

6.1 Different Flavors of Impact Analysis

Impact analysis can be performed during several phases in the software life cycle, however with different objectives.

Requirements-driven impact analysis is motivated by the need to determine the cost of proposed changes before they are implemented. Requirements-driven impact analysis is, therefore, carried out very early, namely during the planning phase [Lindvall, 1997]. The objective is to predict, for each new requirement, where and how much change is needed in order to fulfill the new requirement, which can be used to calculate cost. Requirements-driven impact analysis is based on the identification of primary changes, as well as secondary changes with the constraint of not altering any source code.

Change-related impact analysis is motivated by the need to identify what additional changes are needed. It is mainly focused on determining how a change of one entity of the system affects other system entities (e.g., [Bohner and Arnold, 1996]). This is a crucial activity during the implementation of the change. The objective is to identify all entities (change candidates) that might possibly be affected by the change and inspect each of them in order to decide whether it really has to change or not. It should be noted that the primary changes are already identified at this point in time and the objective is to account for all ripple effects (secondary changes).

Test-Related Impact Analysis is motivated by the need to limit regression testing - if it is known exactly which entities are affected by a change, the unaffected entities need no regression testing [Kung, et. al., 1994]. At this point in time all changes are known and implemented. Unit testing has been carried out and it is now time to test whether the system works as a whole, that the set of new requirements is correctly implemented and that the system is still fulfilling the "old" requirements (regression testing).

For all of these different perspectives, ripple effect analysis, dependency analysis, and traceability approaches can be used.

6.2 Ripple Effect Analysis and Dependency Analysis

Impact analysis can be characterized as detecting consequences of change using some kind of dependency analysis. The model for module connection analysis [Haney, 1972] is an early and typical example of dependency analysis. It uses a probability connection matrix that subjectively models the dependencies between modules in order to determine how change in one module necessitates change in other modules. This model is probabilistic, meaning that it is probable, but not necessarily true, that change will propagate as the model forecasts.

Mechanical approaches, as described by [Queille, et. al., 1994] are, instead, based on a model of the system in terms of the static dependencies between entities, together with change propagation rules. Using a change propagation mechanism it is possible to identify changes that must be made. These techniques analyze how change propagates in order to identify secondary change as a consequence of planned or conducted primary change. Change candidates are entities in the source code that are likely to change and each candidate must be analyzed in order to determine whether it must be changed or not.

It is also desirable to identify entities that are affected by change and might behave differently as a result. Such entities might be affected by the change, but it might still not be necessary to change them. It is, however, necessary to retest the affected entity as it might behave differently, though still producing a correct result. This is exemplified in [Arango, et. al., 1996], together with a technique for explaining how changes in data representation translate into performance changes.

A different approach aiming at the same problem is slicing [Gallagher and Lyle, 1991]. The program is sliced into a decomposition slice, which contains the changed code, and a complement slice, which is the rest of the program. Slicing is based on data and control dependencies in the program. Changes made to the decomposition slice are guaranteed not to affect the complement if a certain set of rules is obeyed. Slicing limits the scope for propagation of change and makes that scope explicit. Turver and Monroe [Turver and Monroe, 1994] use the technique for slicing of documents in order to account for ripple effects as a part of impact analysis. Shahmehri, et. al., [Shahmehri, et. al., 1990] apply the technique to debugging and testing. Complex languages are hard to slice and have not been supported before, but techniques for C++ [Tip, et. al., 1996] and Java [Chen and Xu, 2001] are now being developed.

6.3 Traceability Approaches

Traceability is germane to life-cycle meta-models such as the iterative reuse model [Basili, 1990]. In this model, software includes not only the resulting source code, but also up-front documents such as requirements and design specifications, which are regarded as models at various abstraction levels of the software in service [Jacobson, et. al., 1992].

Maintenance from Basili's perspective, initially performs and documents changes of requirements, which are subsequently propagated through the analysis and design models to the source code. Proponents of this approach, such as [Pfleeger and Bohner, 1990], assume a high level of traceability, which, in practice, implies that:

- All models of the software are consistently updated
- It is possible to trace dependent items within a model (vertical traceability or intra-model traceability)
- It is possible to trace correspondent items between different models (horizontal traceability or inter-model traceability)

Traceability can be graphically represented. Software items (requirements, design components and parts of code) are nodes and the traceable dependencies are edges, together forming a traceability web. It is assumed that if tracing dependencies in the web is easy, the effort required to understand the software and assess the impact of a proposed change is decreased.

Traceability links are often provided as a feature in life-cycle oriented case tools (e.g., SODOS [Horowitz and Williamson, 1986]), but to our knowledge little has been published about how to actually use them. It is seemingly a paradox that even though it is well known that the problem of understanding the relationship between requirements and code consumes much time and money [Soloway, 1987], current practitioners are not at all convinced of the usefulness of traceability. As a reaction to the fact that traceability between requirements and code is unlikely to be found in a real project, a test-case-based method for finding a starting place (finding primary changes) for further investigations prior to changing the system was proposed in [Wilde, et. al., 1992].

6.4 A Framework for Impact Analysis

For the purpose of comparing different impact analysis approaches, a framework was defined in [Arnold and Bohner, 1993]. It should be noted that the framework describes change propagation approaches. This means that the primary changes are already determined and the goal is to analyze secondary changes or identify which entities can be affected by the changes. The approaches covered by the framework are, for example, determination by incremental compilers of which parts to recompile, as well as changes induced by a maintainer and their potential effects. The framework is useful as it reveals the underlying mechanisms in most impact analysis approaches available.

The following parts of an impact analysis approach are identified:

- A change
- The artifact object model (the system)
- The interface object model (a model of the system — its interface)
- The internal object model (a model of the system — its internals)
- The impact model (knowledge about change propagation)

We will now describe how the effects of a proposed change on the artifact object model are determined by using an impact analysis approach. The impact analysis approach is based on a model of the system, which describes dependencies between system entities. User interaction is carried out via the interface object model (interface for short). The interface is used to describe the change to be analyzed. The internal object model contains information about the objects in the system and the dependencies between them.

The difference between the interface and the internal model is analogous to the difference between a database and the different views of it. The database constitutes the underlying representation of the structure and is populated with data. The database is manipulated via views (for example, a selected set of entities, their relations, and tools for manipulating them) which constitute the interface to the database. While the internal object model captures information about the objects and their dependencies, the impact model captures knowledge about how changes propagate from object to object via dependencies. The knowledge can be expressed in terms of rules or algorithms.

When the user orders an analysis of the change, the impact analysis approach uses the definition of the change, as defined using the interface, translates it to the internal object model, and uses the knowledge in the impact model to propagate the initial change throughout the internal object model. The result, in terms of a set of affected interface objects, is presented to the user via the interface. In automated environments (e.g., incremental compilers), some of the steps are automated and invisible to the user, while in semi-automated and manual environments (e.g., case tools with some support for dependency analysis, but without knowledge and routines for change propagation) much of the initial analysis work has to be done by the user. In the latter case, the user must also be prepared to spend time on the result from the impact analysis, as objects presented as affected might be false positives and, thus, not need to change.

6.5 A Model for Impact Analysis

An instantiation of the framework described above is illustrated in this model for impact analysis. It is based on the following objects that are needed in order to conduct impact analysis [Queille, et. al., 1994]:

- A representation to model the underlying structure of the system
- Data about the system to populate the model
- A mechanism to propagate the changes through the system

The model is based on dependency graphs that can be collected automatically from the underlying source code. Impact analysis is regarded as propagation of specific modification events over the generated dependency network. The propagation of change is governed by a set of propagation rules.

A modification can either be automatic or potential. Automatic modifications are those that are certain and are inserted automatically. Potential modifications are conditional and must be confirmed before they can be inserted. A change of the interface of an object is, for example, always propagated to its users of the object as an automatic change. A change of the implementation of an object may affect the interface of the object, but not certainly. Therefore, this change is propagated as a potential change of the interface to the object itself. Note that a confirmation of the latter change would trigger the propagation of the change to the users of the object.

The process is roughly described as:

1. Identify the change set (the initial set of objects that should change)
2. Develop the forest of impacts (for each object in the change set)
3. Identify candidate impact objects using the approach described
4. Assess each candidate object to see if changes are really needed
5. If change is needed, add the object to the change set
6. If change is not needed, proceed with the next object in the change set

This process was used in an experiment. A C program with 2100 lines of code was analyzed to account for two changes: one subtle bug and one enhancement. The maintainer was not familiar with the code at the beginning of the study. The results of the case study show that the tool produced a long list of candidate objects to be assessed, in fact too many candidates, and not enough time was available to inspect them all. Instead, the semantics of the change were used to examine whether the change would invalidate each object. Twenty-five percent of the modules found were important to understand the program and the change. There were no cases where requirements impacted each other and the links to code were not precise enough to be useful for cost estimation in a mechanical way. The tool was regarded useful, even if some modifications were needed.

Detailed algorithms for producing a conservative prediction of the system-wide effects of a proposed change in an object-oriented system are provided in [Offutt and Li, 1996]. The prediction is based on a set of data members and member functions that are to be changed and calculated as the transitive relation between the initial set and all other data members and member functions in the system. The result from the prediction is a set of all data members and member functions in the system that are direct or indirect clients, children, or internally related to the changed set.

Rules on the identification of conducted changes and on the subsequent change impact identification in an object-oriented system are provided in [Kung, et. al., 1994]. It should be noted that this approach is entirely devoted to support testing and, thus, is applied after the changes have been made. By using this approach, it is possible to identify the changed parts and also discover what kind of changes are made, and find the set of parts of the system that might be affected, at run-time, by the changes. This makes it possible to limit regression testing to only the affected part of the system.

6.6 Ripple Effects In Documents

Software source code is not the only product that has to be changed in order to develop a new release of the software product. Normally, many documents are also affected by new requirements. The user manual is an example of a document that has to be updated when new user functionalities have been provided. Turver and Munro [Turver and Munro, 1994] focus on the problem of ripple effects in documentation and note that this has not been a widely discussed subject in the literature.

A logical model of documentation is defined as consisting of:

- Documentation libraries
- Volume entities
- Chapter entities
- Section entities
- Subsection entities
- Segment entities

Graph theory is used to describe the relations between the various entities and their hierarchy. The Ripple Propagation Graph is an acyclic graph consisting of documentation entities and relations.

This technique relies on dependencies between entities. The common dependency used is the consist-of dependency that, for example, says that a chapter consists of a number of sections. The conclusion is that, if a particular section is changed, then the chapter it is part of also needs to be changed. The unusual dependency used in this work is the thematic dependency, which exists between segment entities. A segment can have many themes and, likewise, many subjects can be related to the same theme. The ripple effects may be calculated based on these dependencies using a thematic slicing technique. The analyst orders a thematic slice over a document by specifying the document and the theme that is affected by the change. The algorithm produces a thematic slice showing the sub-graph representing the parts of the document that have to be checked for ripple effects. To even increase the value of the information, Turver and Munro suggests that this technique be used in cooperation with a probability connection matrix like [Haney, 1972].

6.7 Software Architecture Analysis and Impact Analysis

In larger systems, there is a need to identify how change propagates between architectural components of the system. One such approach is the Software Architecture Analysis Method (SAAM). SAAM compares and evaluates systems based on their adaptability. A software developer might, for example, be interested in adapting a software application to a specific situation. If this is possible or not, cheap or expensive, depends on the adaptability of the application. McCrickard and Abowd [McCrickard and Abowd, 1996] use the following criteria to evaluate adaptability in terms of impact factors. The higher the impact factor, the more severe the change.

SAAM involves the following activities:

- Describe candidate architectures in terms of high-level components and how they are connected
- Develop scenarios
- Perform scenario evaluations

Each scenario was "mentally" applied to each application, and all necessary changes were rated according to the table of impact descriptions, and an impact factor was consequently assigned to each change. The result from the evaluation is a table showing the impact factor for each scenario. If the investigated debugger already had a particular scenario implemented, the impact factor was zero. The table was then used to assess which debugger was most adaptable regarding the set of scenarios evaluated.

6.8 Experiments on Impact Analysis

An experiment in order to test the effectiveness of a test case-based method's ability to locate user functionality in a program was conducted in [Wilde, et. al., 1992]. The experimenters claim that traceability between requirements and code is unlikely to identify why, in a real project, there is need to find alternative solutions. A C-program consisting of 15,000 lines of code was to be modified. The task was to locate the set of subroutines in the program (total 360) that were central to each of the ten user functionalities provided by the program. In other words, the intention was to find a starting place for further investigations prior to changing the system. Two program experts' opinions about which subroutines were central for each of the ten user functions were used for comparison and evaluation of the test case-based method. The result is that the method generally identifies many more subroutines than the experts, with some overlap. For example, the method found 11 subroutines, while the experts found three. The three subroutines found by the experts were included among the subroutines found by the method. Considering that the program consists of 360 subroutines, the eight extra subroutines (false positives) should not be a problem and from that viewpoint, the method could be judged to be useful in locating a starting point for further investigation.

An experiment on the effect of design recording on impact analysis was conducted in [Abbattsista, et. al., 1994]. The objective of the experiment was to investigate how differences in design recording influence maintenance performance. The task was to modify a program in three different ways. The modifications were designed so that there was one correction, one adaptation, and one perfection. Each task was limited to two hours. Three different documentation situations were accounted for:

- No documentation at all but the source code, which also might be the common case in real projects
- Standard documentation, which means it is structured in a traditional way, for example, in the way the IEEE standards prescribe. The standard documentation is voluminous, but lacks rationale, i.e., information about why the software was designed that way.
- Documentation based on decisions and traceability links. The standard documentation is complemented with decisions and traceability. In this experiment, model dependency descriptors were used. A model dependency descriptor is based on decisions and captures information about the problem to be solved, the alternatives, the solution and the justification for it.

The subjects, 23 computer science students, were divided into 9 groups (3x3x3 factorial design) with the task of modifying an information system written in Pascal. The group modified different versions of the system (6.6 KLOC - 9.7 KLOC). All three versions were developed from the same requirements, but different design and code.

Evaluation was conducted by comparing, for each task, the estimated impact set (EIS) with the actual impact set (AIS). The completeness and accuracy of the estimated impact were calculated. Completeness and accuracy describe how the two sets (EIS and AIS) are related to each other and range from zero to one. When the two sets are disjointed both completeness and accuracy have the value of zero, which reflects a totally misestimated situation. *Completeness* increases as more of the actual set is included in the estimation. It has the value of one when the estimation (EIS) is entirely included in the actual set (AIS). *Accuracy* increases as more of the estimation is included in the actual set. It has the value of one when the actual set (AIS) is included in the estimation set (EIS). When the two sets are equal, both completeness and accuracy have the value of one, reflecting the ideal situation.

The overall results (the arithmetic mean over the three tasks for each measure) show that both completeness and accuracy increase with the amount of documentation available, thus model dependency descriptors result in the best figures. Examining the results for each task gives a more varied pattern. The results are always better when using the model dependency descriptors compared to using standard documentation. For one measure of six, using the code only gives the best result. For two other measures using the code only is better than using standard documentation. Interestingly, the time required to conduct the impact analysis task also seems to increase with the amount of information available. This can be explained by the fact that all work was carried out manually with no tool support, which is almost always required to navigate among a large amount of information.

6.9 Summary

Impact analysis is mainly based on dependencies between software entities. These dependencies are often logical, such as one software module referring to another. Logical dependencies are used by many change propagation techniques in order to figure out additional changes. Traceability is another form of logical dependency, but between requirements and code. When such dependencies exist, they can also be used to model change propagation. The techniques described in this section can be used to determine change in embedded systems, but changes based on subtle dependencies must be taken into account too. Examples of such dependencies are timing constraints. As these are often not documented, they are hard to model and result in hidden side-effects.

7 References

[1] Abbattsista, F.; Lanubile, F.; Mastelloni, G.; and Vissaggio, G., “An Experiment on the Effect of Design Recording on Impact Analysis”, Los Alamitos, CA, USA, IEEE Computer Society Press. International Conference on Software Maintenance, pp. 253-259, 1994

[2] Arango, G.; Schoen, E.; and Pettengill, R., "A Process for Consolidating and Reusing Design Knowledge," *Software Change Impact Analysis,* Los Alamitos, CA, USA, IEEE Computer Society Press, pp. 237-248, 1996

[3] Arnold, R. S. and Bohner, S. A., “Impact Analysis - Towards a Framework for Comparison”, IEEE International Conference on Software Maintenance, pp. 292-301, 1993

[4] Barr, M., "Whither Embedded?," *Embedded Systems Programming*, vol. 15, no. 2, Feb.2002

[5] Basili, V. R., "Viewing Maintenance as Reuse-Oriented Software Development," *IEEE Software*, vol. 7, no. 1, pp. 19-25, 1990

[6] Bohner, S. A. and Arnold, R. S., “*Software Change Impact Analysis”,* Los Alamitos, CA, USA, IEEE Computer Society Press, 1996

[7] Chapin, N.; Hale, J.; Khan, K.; Ramil, J.; and Tan, W.-G., "Types of Software Evolution and Software Maintenance," *Journal of Software Maintenance and Evolution Research and Practice*, vol. 13, pp. 3-30, 2001

[8] Chen, Z. and Xu, B., "Slicing Object-Oriented Java Programs," *SIGPLAN Notes*, vol. 36, no. 4, pp. 33-40, 2001

[9] Clark, E.; Forbes, J.; Baker, E.; and Hutcheson, D., “Mission-Critical and Mission-Support Software: A Preliminary Maintenance Characterization”, Crosstalk 12[6], pp. 17-22, 1999

[10] Clements, P.; Kazman, R.; and Klein, M., *“Evaluating Software Architectures: Methods and Case Studies”,* Addison-Wesley, 2001

[11] Dart, S. A., “Concepts in Configuration Management Systems”, Third International Workshop on Software Configuration Management, pp. 1-18, 1991

[12] Davis, A. M., “*Software Requirements: Objects, Functions, and States”,* Prentice Hall, 1993

[13] Gallagher, K. B. and Lyle, J. R., "Using Program Slicing in Software Maintenance," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 751-761, 1991

[14] Haney, F. M., “Module Connection Analysis - A Tool for Scheduling Software Debugging Activities”, AFIPS Joint Computer Conference, pp. 173-179, 1972

[15] Harjani, D. K. and Queille, J., “A process model for the maintenance of large space systems software”, IEEE. Conference on software maintenance, pp. 127-136, 1992

[16] Horowitz, E. and Williamson, R., "SODOS: A Software Documentation Support Environment-Its Definition," *IEEE Transactions on Software Engineering*, vol. 12, no. 8, pp. 849-859, 1986

[17] Jacobson, I.; Christersson, M.; Jonsson, P.; and Overgaard, G., “*Object-Oriented Software Engineering”,* Addison-Wesley, Menlo Park, CA, USA, 1992

[18] Karjalainen, J.; Mäkäräinen, M.; Komi-Sirviö, S.; and Seppänen, V., "Practical process improvement for embedded real-time software," *Quality Engineering*, vol. 8, no. 4, pp. 565-573, 1996

[19] Kea, H.; Kraft, S.; and Stark, M., “Profile of Software at the Information Systems Center”, Software Engineering Laboratory Series, SEL-99-001A, 1999

[20] Kitchenham, B.; Travassos, G. H.; von Mayrhauser, A.; Niessink, F.; Schneidewind, N. F.; Singer, J.; Takada, S.; Vehvilainen, R.; and Yang, H., “Towards an Ontology of Software Maintenance”, Journal of Software Maintenance and Evolution Research and Practice 11[6], pp. 365-389, 1999

[21] Koopman, P., “Embedded System Design Issues (the Rest of the Story)”, International Conference on Computer Design, pp. 310-319, 1996

[22] Koopman, P., “Tutorial: Embedded System Design Issues (the Rest of the Story)”, 1996

[23] Koopman, P., “Embedded Systems in the Real World: Introduction to Embedded Systems”, 1999

[24] Kung, D. G. J.; Hsia, P.; Wen, F.; Toyoshima, Y.; and Chen, C., “Change Impact Identification in Object-Oriented Software Maintenance”, International Conference on Software Maintenance, pp. 202-211, 1994

[25] Kuvaja, P.; Maansaari, J.; Seppänen, V.; and Taramaa, J., “Specific requirements for assessing embedded product development”, International Conference on Product Focused Software Process Improvement, pp. 68-85, 1999

[26] Li, J. and Feiler, P. H., “Impact Analysis in Real-Time Control Systems”, IEEE International Conference for Software Maintenance, pp. 443-452, 1999

[27] Lientz, B. P. and Swanson, E. B., “*Software Maintenance Management”,* Addison-Wesley, 1980

[28] Life Cycle Data Harmonization Working Group of the Software Engineering Standards Committee of the IEEE Computer Society, IEEE Standard on Software Maintenance, 1998

[29] Lindvall, M., "An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Systems Evolution.", PhD Thesis No 480, Linköping Studies in Science and Technology, 1997

[30] Lindvall, M. and Sandahl, K., "How Well do Experienced Software Developers Predict Software Change?," *Journal of Systems and Software*, vol. 43, no. 1, pp. 19-27, 1998

[31] Lindvall, M.; Tesoriero Tvedt, R.; and Costa, P., "An Empirically-Based Process for Software Architecture Evaluation (Accepted for publication)," *Empirical Software Engineering: An International Journal*, 2002

[32] Lindvall, M.; Tesoriero, R.; and Costa, P., “Avoiding Architectural Degeneration: An Evaluation Process for Software Architecture”, International Symposium on Software Metrics, pp. 77-86, 2002

[33] Mäkäräinen, M., “*Software Change Management Process in the Development of Embedded Software”,* VTT Publications, pp. –185, 2000

[34] McCrickard, D. S. and Abowd, G. D., “Assessing the Impact of Changes at the Architectural Level: A Case Study on Graphical Debuggers”, International Conference on Software Maintenance, pp. 59-67, 1996

[35] Moubray, J., “*Reliability-Centered Maintenance”,* 2nd rev. ed., Industrial Press, New York, 2001

[36] Offutt, A. J. and Li, L., “Algorithmic Analysis of the Impact of Changes to Object-Oriented Software”, Los Alamitos, CA, USA, IEEE Computer Society Press, International Conference on Software Maintenance, pp. 171-184, 1996

[37] Pfleeger, S. L. and Bohner, S. A., “A Framework for Software Maintenance Metrics”, Los Alamitos, CA, USA, IEEE Computer Society Press, Conference on Software Maintenance, pp. 320-327, 1990

[38] Pressman, R., “*Software Engineering, A Practitioner's Approach”,* McGraw-Hill, 1992

[39] Queille, J.; Voidrot, J.; Wilde, N.; and Munro, M., “The Impact Analysis Task in Software Maintenance: A Model and a Case Study”, Los Alamitos, CA, USA, IEEE Computer Society Press, International Conference on Software Maintenance, pp. 234-242, 1994

[40] Rombach, H., "A Controlled Experiment on the Impact of the Software Structure on Maintanability," *IEEE Transactions on Software Engineering*, vol. 13, no. 3, pp. 344-354, 1987

[41] Rus, I. and Lindvall, M., "Knowledge Management in Software Engineering," *IEEE Software*, vol. 19, no. 3, pp. 26-38, 2002

[42] Rus, I. and Zelkowitz, M. V., “Space Shuttle Software IV&V Process Model”, Fraunhofer Center for Experimental Software Engineering, College Park, Maryland 20742, Technical Report 00-106, 2000

[43] Sahin, I. and Zadeli, F. M., "Policy Analysis for Warranty, Maintenance, and Upgrade of Software Systems," *Journal of Software Maintenance and Evolution Research and Practice*, vol. 13 pp. 469-493, 2001

[44] Sangiovanni-Vincentelli, A. and Martin, G., “A Vision for Embedded Software”, ACM Press, International Conference on Compilers, Architectures and Synthesis for Embedded Systems, pp. 1-7, 2001

[45] Seppänen, V.; Kähkönen, A.; Oivo, M.; Perunka; H., Isomursu, P.; and Pulli, P., “Strategic Needs and Future Trends of Embedded Software”, TEKES, Technology Development Centre of Finland, Technology Review 48/96, 1996

[46] Shahmehri, N.; Kamkar, M.; and Fritzson, P., “Semi-Automatic Bug Localization in Software Maintenance”, International Conference on Software Maintenance, pp. 30-36, 1990

[47] Soloway, E., “I Can't Tell What in the Code Implements What in the Specs”, The Second International Conference on Human-Computer Interaction, pp. 317-328, 1987

[48] Taramaa, J., “*Practical Development of Software Configuration Management for Embedded Systems”,* VTT Publications, 1998

[49] Taramaa, J.; Seppänen, V.; and Mäkäräinen, M., "From Software Configuration to Application Management," *Journal of Software Maintenance and Evolution Research and Practice*, vol. 8, no. 1, 1996

[50] Tesoriero Tvedt, R.; Costa, P.; and Lindvall, M., “Does the Code Match the Design? A Process for Architecture Evaluation (Accepted for publication)”, International Conference on Software Maintenance, 2002

[51] Tichy, W., “Tools for Software Configuration Management”, Teubner Verlag, International Workshop on Software Version and Configuration Control, pp. 1-20, 1988

[52] Tip, F.; Jong, D. C.; Field, J.; and Ramlingam, G., “Slicing Class Hierarchies in C++. Conference on Object-Oriented Programming, Systems, Languages & Applications”, pp. 179-197, 1996

[53] Turver, R. J. and Munro, M., "An Early Impact Analysis Technique for Software Maintenance”, *Journal of Software Maintenance and Evolution Research and Practice*, vol. 6, no. 1, pp. 35-52, 1994

[54] Weiderman, N.; Bergery, J.; Smith, D.; and Tilley, S., “Approaches to Legacy Systems Evolution”, Software Engineering Institute, CMU/SEI-97-TR-014, 2002

[55] Wilde, N.; Gomez, J. A.; Gust, T.; and Strasburg, D., “Locating User Functionality in Old Code”, Los Alamitos, CA, USA, IEEE Computer Society Press, Conference on Software Maintenance, pp. 200-205, 1992

[56] Yau, S. and Collofello, J. S., "Some Stability Measurements for Software Maintenance," *IEEE Transactions on Software Engineering*, vol. 6, no. 6, pp. 545-552, 1980

Appendix A. Case Studies

This appendix presents the case studies of the organizations that develop and maintain embedded software that we analyzed in this work. Some of these case studies were published and some were conducted as part of this study. We want to especially give credit to Minna Mäkäräinen for her case studies [Mäkäräinen, 2000] that provided data for this study and allowed us to complete this analysis.

The case studies are organized by product type with lessons learned compiled at the end. The case studies cover the following product areas: Consumer Electronics, DSL-Modems, Field Devices, Mobile Phones, Airplanes, Satellites, Automobiles, and Industrial Machines.

A.1. Consumer Electronics

One of the case studies in [Mäkäräinen, 2000] is a company that develops consumer electronics. Some of their products are used internally; others are sold outside the company and exported. In 1993, the organization had more than 1000 employees.

The products and the software exist in several versions and many units are sold of each version. The products are expected to last a long time and the maintenance phase is, therefore, long. The main goal of the product is to survive under all circumstances. Much of the logic is devoted to this task. The software is developed and maintained on a Sun Sparc station and a simulator on a PC was available. Old products that are still maintained are programmed in Assembler. New products are typically programmed in C and essential parts in Assembler. Hardware components are expected to change during maintenance, which may generate changes in the software.

The software architecture has two layers:

- The Core, in which functions common to all applications are located
- The Application-dependent layer, in which the functions of an individual application are located. The goal is to make the application-dependent layer as small as possible.

New products are developed based on the experience and improvement ideas derived from old products. Ideas for new products and new features of existing products come from the R & D department or customers. Modification of existing products is performed at the request of external clients. The person who makes the implementation decision for minor modifications processes requests from customers abroad. Old products used in the field are usually not updated, but the change is implemented in new products. Change management procedures during the initial development do not exist. Their processing is left to individuals and is ad hoc.

Official procedures exist for reporting error situations in operating devices located in the field, but they are not always followed. Instead, service personnel and external customers who report the errors contact the person responsible for the software, as they often know the person responsible for the software by name. The person responsible for the software files the problem reports instead of the original initiator. As a result, not all the errors are reported. The person reporting the error decides how the reported error situation is further processed.

The hardware components may age, but the product concept itself does not need modification. The software must be adapted to the new hardware environment so that the end user cannot even notice the change in the hardware. Development tools also change during the lifetime of the software. This kind of adaptive maintenance must usually be completed before any other kind of maintenance can be undertaken.

The same people perform both software development and maintenance. No formal quality system or standard is in use. The project manager is responsible for the quality practices used in the project. The main change management effort is completed after the release of the product, when the product software has to be updated and modified. The estimated operational lifetime of software is expected to be 10 years covering some 10,000 machines. The effect is a long maintenance phase.

Some of the problems encountered in maintenance are:

Location of errors is very time-consuming. The most difficult and time-consuming task is locating errors. This is because the person who found the error is typically not the one who reports it.

Impact of change in dependent products is hard to determine. The impact of changes are very difficult to estimate, especially since modifications of the core part of the software also affect other products using the same core.

Incomplete testing causes serious field errors. Testing is done manually without any support. Testing takes too much time, and the coverage of the testing procedures is unknown. Subsequently, the most serious errors are often found when the product is already in the field.

Inconsistent documentation. The development phase does not produce much documentation. The maintainer is responsible for updating the documentation manually if needed. Thus, documentation is often not up-to-date.

Technology changes. New versions of the development tools are typically delivered several times during the maintenance phase of the software. Problems occur if the software has not been changed in any way for a long time and there have been several new compiler releases during the time. It is difficult to use a new version of a tool when the maintenance action needs to be performed quickly.

A.2. Mobile Phones

This case study is based on a telephone interview conducted as part of this work. The interviewed company develops and manages a set of mobile phone products. The mobile phones are developed in multiple versions and distributed in many copies.

Reliability and performance are important, but the requirements are not as strict as, for example, a telephone switch. The customer is, for example, expected to accept rebooting the modem once in a while, while it would be catastrophic if the switch went down.

Software is developed on a PC. The development environment has facilities for simulating the target environment, i.e., the phone, so that the software can be run and tested on the PC. This simulation has, however, limited functionality, and all aspects cannot be tested in this way. The software for mobile phones is typically developed in C.

The software architecture is layered. The lowest layers are close to the hardware and the higher layers are increasingly independent of the hardware. The layers that are closer to the hardware are very hard to develop and maintain. Long experience is required for these developers as each single line of code can change the behavior of the phone. Impact analysis is, therefore, extremely important. Currently, an expert group performs that analysis.

Some of the problems encountered in maintenance are:

Behavior in simulation environment differs from target environment. The limitations are typically related to real time aspects that are hard to simulate. One such aspect is that the network requires that a call be answered within a certain time. Another problem is that memory allocations are done in a different way in the simulated environment compared to the target environment. This makes it hard to detect memory problems early.

Debugging on target hardware is hard. It is hard to debug the software once it is downloaded to the cell phone because any changes to the software cause the system to behave in a different way. There exists advanced test equipment that allows such debugging, but this organization has not invested in it yet.

Lack of memory results in code that is hard to understand. Lack of memory is a problem in mobile phones because mobile phones have to be small and are sensitive to any increase in cost, weight, and size. This results in the code being optimized for fitting into a small memory, instead of for readability.

Layers close to hardware are hard to understand, maintain. Code close to hardware is harder to understand than other code because it deals with the underlying hardware.

A.3. Field Devices

This case study is based on an interview conducted by IESE as part of a related study of an IT department of a company that develops field devices, actuators and positioners. The products have 500-1000 end users. Typical software development duration is 2 years and typical maintenance project duration is 2-3 years.

Reliability is seen as the 3rd most important product quality, after functionality and usability.

The target environment is based on a real-time operating system. The user interface is a LCD-based device with 3 to 4 buttons for input. The system interfaces with other systems via Profibus or Fieldbus foundation to PCs, operator stations, and process system controllers. Also, handheld connection capability exists.

The programming languages are C and assembler.

Three groups (product management, R&D management, QA) are involved in the maintenance.

If a software update is needed after the product is delivered, the software is replaced and the embedded software can be diagnosed from a distance.

The Software development process follows an Iterative Incremental model. ISO9000 guides the maintenance process. The following phases of the maintenance process are distinguished:

- Error detection
- Analyses
- Design
- Implementation
- Regression test
- Release

All maintenance related issues are evaluated each quarter. Maintenance projects are managed and tracked by regular working groups and problem reports. One team is responsible for development and maintenance.

Maintenance lifetime is device-dependent (between 3 to 10 years; some 20 years).

A.4. Digital Subscriber Line (DSL) Modems

This case study is based on a telephone interview conducted as part of this study with a company that manages a set of products for telephone and data communication. This description focuses on their Digital Subscriber Line (DSL) modems. This part of the company has about 100 employees.

DSL-modems are relatively small products that are distributed in many copies. The product is expected to remain in operation 2-3 years and then be replaced. Reliability and performance are important, but the requirements are not as strict as, for example, a telephone switch. The customer is, for example, expected to accept rebooting the modem once in a while, while it would be catastrophic if the switch went down.

Software is developed on a PC. The architecture of the product is layered. The goal is to encapsulate the hardware by software so that less hardware-oriented software can be developed. This makes it easier to maintain the software. It also makes reuse for similar products easier.

Development of software for DSL-modems is hardware-oriented, especially the lower layers that are close to the hardware. Adding hardware, such as memory, means increased end-user costs so it is avoided if possible. The small size and weight limitations of the product also put constraints on how the hardware components can be used.

Generally, the same people who develop the software maintain it. A customer support organization receives trouble reports from customers and channels it to the development team who implement them. Many times the bug fixes form part of a larger release, but if the problem is significant, a special bug fix is released. Change requests and new requirements come in from both end users and internally from the development team.

DSL-modems are inexpensive, so each developer can have one on the desk for testing. No simulation environment is necessary. Editing, compiling, and uploading new software to the test modem is typically accomplished in a couple of minutes.

The software is stored in a flash memory so it can be upgraded after the product is delivered. During upgrading, the content of the flash memory is replaced by the new software. This upgrade is done by the end-user and can be somewhat complicated. There is a risk that more memory than supposed is overwritten during upgrading of software. A strategy for reducing the risk of overwriting memory is to have a double, redundant memory. The new software is then uploaded to one of the memory units while the other one serves as backup. If anything goes wrong, the system falls back to the backup.

From the operator’s point of view, it would be desirable to be able to manage all modems from a central location due to the somewhat complicated installation and configuration process. It is also expensive to send out service people.

Some of the problems encountered in maintenance are:

Relatively complex upgrades done by end-users threatens functionality. The upgrading procedure can be complex. If it goes wrong, it threatens the overall functionality of the modem. The most important problem is to protect sensitive memory from being overwritten during upgrading of software.

Cost, size, and weight sensitivity put severe constraints on feasible solutions. Adding more memory (redundant memory, for example) to make sure upgrades do not go wrong adds to the cost of the product. Adding more hardware also adds to the size and weight of the product, which also can be a problem due to such limitations.

A.5. Telecommunication Systems

This case study is based on a description in [Mäkäräinen, 2000]. The company manages a set of product families in the area of telecommunication. The structure of the organization is a matrix organization, where people work in departments and groups according to their domain knowledge. People from different departments form project groups. The company is relatively large.

The products are typical multi-technological products consisting of several software and hardware components. The products are developed in an evolutionary manner. Each development project gets an old baseline, and a set of new requirements and change requests, as input to the project and develops a new version of the product accordingly.

For new development, the hardware is not stable, but changes during the project. Separating ‘development’ and ‘maintenance’ phases is not practical, as software development work is actually change management and seldom consists of building software from scratch. The post-delivery projects only perform emergency corrections and user support; the larger enhancements are made in evolutionary development cycles.

Maintenance projects are usually based on existing hardware and mainly deliver a new release of the software with added, modified or corrected functionalities. The hardware platform is already stable. The hardware of the product is typically designed concurrently with the software in project developing new products.

Software development is heavily based on reusing parts of old software. Maintenance projects typically take a stable enough version of an old product as a baseline for the development and add new features, modify old features and correct defects.

Development is conducted in an iterative fashion. The software development work is actually change management, not building software from scratch.

Change requests are divided into the following categories:

- Changes to correct defects found in testing. Most of the change requests are generated throughout the testing phase.
- Changes to correct defects found in reviews. The projects have extensive review practices. All documents are reviewed before they can be used as the basis for a release and the minutes of the reviews include the defects found in the review meeting. When the software designers analyzed these defects, many defects were found to be so insignificant that they would not have caused problems in later testing phases.
- Changes to add new features or to modify existing features. Requests for new features or feature modifications are the most troublesome group of change requests. Typically, large numbers of requests for new and modified features are submitted. The projects continuously fail in estimating and preparing for the requirement-level changes in their project plans. The most common method to handle requirement-level changes is to eliminate them by postponing or rejecting them, if possible.

In maintenance projects, the hardware environment and other technology parts are very stable and do not generate as many requirement-level changes. The requests for new features or feature modifications mostly originate from customers and marketing.

In projects that develop new products involving new hardware, changes to the requirements occur quite frequently due to the concurrency of the hardware and software development processes.

Some of the problems encountered in maintenance are:

Requirement level changes. Requirements tend to be added and changed frequently, causing many problems in the projects.

Trivial defects consume much time. Reviews reveal trivial, cosmetic defects which would not cause errors or extra work in later phases. Release tests reveal defects, which are not real defects. The problem is that detecting and fixing these errors consume much time.

Code decay due to repetitive reuse. The software is reused repetitively in many subsequent projects, causing the software to decay. It is, however, hard to detect such decay.

Traceability is not maintained. Traceability information between a change request document and the modified document is often missing.

Lack of recorded reasons for modifications. The original motive and effort spent for each modification are not recorded. The modification is usually recorded, but the reason why it was done is not. Project plans are changed according to new requirements, but no statement is included why the change to the plan was done.

Lack of learning based on defect data. No systematic practices for analyzing the change data for learning or improvement purposes exist.

Locating parts to be modified. Locating the parts of the software to be modified and identifying the ripple effects were reported as being the most time consuming tasks. Regression testing is difficult because of the complexity of the system.

A.6. Telephone Switches

This case study involves a company that develops telephone switches for regular and mobile telephone systems. The company employs several thousand people. The case study is based on an interview conducted by telephone as part of this work.

Telephone switches are part of a complex network enabling people to communicate via telephones. The highest priority in development of telephone switches is to minimize the downtime, i.e., maximizing the availability of the switch. The second priority is to maximize capacity. These two properties are what the buyers of the switch (the operator) can charge their customers (the call makers) for. The switch is expected to remain in operation several decades.

Software is developed on Sun workstations. Simulators are available to simulate the software's behavior before it is tested on the target system, i.e., the switch. Even real-time behavior can be simulated in some environments, and the processor speed can be decreased so that the execution can be analyzed. One of the simulation features is a load generator that generates a great number of concurrent phone calls so that the behavior can be analyzed under near real-time conditions.

A proprietary programming language was developed for this purpose. The software of a switch is typically divided into different abstraction layers, which means that the lower layers are dependent on hardware, while the higher layers are not. Probably 10-20% of a telephone switch is related to the core functionality, i.e., to connect telephones, and 80-90% is devoted to support functions that ensure availability and capacity.

The goal of maximizing availability, or uptime, is often achieved by introducing redundancy. For example, systems monitor the hardware and the software in order to determine their status. If the status is not healthy, the execution can be moved to a backup system. Some switches have identical systems that run in parallel. One system has control and the other system monitors. They read the same input and execute the same commands. The only difference is that the output is read only from the system in control. The effect is that they always are in the same state so if something goes wrong with the master system, the backup system can take over at any time.

Another common strategy is to distribute the functionality over many identical components that are run in parallel. The difference is that each of the components contributes to the performance of the switch. If one component fails, the load is distributed over the remaining components that still are operational.

Lack of memory is generally not a problem. If needed, it is possible to add another memory module. The increase in size or cost is not a problem.

Most of the development environment (hardware and software) is non-standard.

Upgrading the software in the switch can typically be accomplished without taking down the system. First the backup system is upgraded, then the control is switched to the backup system, and last the master system is upgraded. The switch is available through the network, which means that updating the software can be done remotely.

Some of the problems encountered in maintenance are:

Complexity due to size of the system and many concurrent processes makes it hard to determine the impact of changes. What makes development and maintenance really hard is the pure complexity of a switch. The system in itself might not be hard to understand, but it is very hard to understand the dynamic behavior of the switch when hundreds of thousands of concurrent connections are made. Very few people understand the whole system, and it is very hard to determine the impact of a suggested change.

Specialized equipment requires educated and experienced people. Most of the development environment (hardware and software) is non-standard. This means that little knowledge and experience sharing can occur between the company and the outside world. Likewise, it takes a long time to get new people up to speed due to the fact that they often have no or little experience from similar environments.

Limited access to target system. The equipment is very expensive, which means that each developer cannot have access to the switch, the target environment. Instead, developers have to rely on environments for simulation and testing that do not behave exactly as the target environment. Some tests cannot be conducted in the simulated environment but must be made in full-scale tests with 100,000 concurrent callers.

A.7. Automobiles

This case study is based on an interview conducted by IESE as a part of a related project of a large organization that designs and manufactures automobiles.

Automobiles typically last very long, e.g., some law requires companies to provide maintenance for 15 years. The number of end-users is very large. Maintenance of embedded software over a long period of time bears the risk that hardware components, as well as related development tools, are not available anymore.

Reliability is one of the top quality characteristics for automobiles, e.g., the Motor Industry Software Reliability Association (MISRA[3]) provides assistance to the automotive industry in the application and creation of safe and reliable software within vehicle systems.

They use a model-based development with MathLab Simulink. The programming language used is C.

Software is a functional application, where UI and middleware control software are part of the platform. Physical parts of the embedded software can be actuators, sensors and EPROM.

The development group is responsible for maintenance. Inputs to the new product include problem reports from the field, changes to legal norms, feedback from internal long-running test drives, and changes to related systems. The software can be upgraded after the product is delivered. The law requires that software has to be replaced. It may also be possible to diagnose and re-program software from a distance.

This organization defines maintenance in the following way:

- Corrective: to correct bugs
- Adaptive: to adapt to a new environment or usage scenario
- Perfective: to improve a quality attribute
- Preventive: to avoid problems

The company follows the incremental V-model for both the development and maintenance phases. Maintenance is seen, on the one hand, as the continuation of the (incremental) development process, which is usually performed by the same team with reduced effort. On the other hand, maintenance is not different from development: First, development projects always build upon existing software, and second, corrections and adaptations are mostly also relevant to the software under development. There is no difference in managing and tracking the maintenance project. Inter-group coordination is taken care of by a device responsible for both hardware and software. This organization defines maintenance in the following way:

- Proactive: Releases in new countries, improvements
- Reactive: requirements out of the field, changes to laws or regulations

Like development activities, these activities are synchronized in the overall system context (i.e., the car). Proactive maintenance is seen as important in an embedded context because well-planned maintenance activities support the goal of keeping the structure of the system as clear and understandable as possible, which is key to ensure the required safety of the system during its whole lifetime (i.e., 15 years).

[3] http://www.misra.org.uk/

Some of the problems encountered in maintenance are:

Lack of good documentation. Technically, maintenance requires documentation being used and updated, but it is never good enough and it shows the obvious, not the relevant.

Keeping track of different versions of products and tools. It is not enough to put software under configuration management, but also tools, compilers, and documentation. This can be especially complicated if a contractor did part of the work.

Knowledge disappears regarding tools and languages of older products. Knowledge must be managed so that, if necessary, the organization remembers how to use languages and tools that form part of older products.

Systems are built on several modeling styles and knowledge gets lost when old styles are forgotten. Embedded software is often developed in a model-based style, e.g., Statemate-Simulink or Matrix-X. Modeling tools and approaches change ("trends") - knowledge gets lost. Maintaining systems that were built on several modeling styles is a problem.

Changes cause ripple effects that are hard to determine. In complex systems based on both hardware and software that interact closely, software modules are spread all over. It is hard to determine the ripple effects in this type of environment.

It is very hard to decide whether the hardware or the software caused a failure in practice. There are several reasons for this. Information in failure reports of the software is often not sufficient to allow the cause of a failure to be identified clearly. Additionally, system failures may be difficult to reproduce and it is difficult to judge whether the hardware caused the failure or whether the software was not "intelligent" enough to compensate for existing hardware limitations.

A.8. Space Instruments

This is another case study presented in [Mäkäräinen, 2000]. The company is an aerospace organization that acts as a project subcontractor with expertise in the fields of software, electronics, system analysis, and small satellite technology. It has about 15 employees.

The company develops unique space instruments for a station that is launched into space. The station is completely independent and cannot be controlled from earth. The embedded software cannot be changed after take-off so, in essence, there is no traditional maintenance phase. The development time is long to ensure that the software is free from defects and as reliable as possible.

No information is provided regarding how long the system is expected to last, but this is relatively unimportant due to the fact that the software cannot be updated after launch.

Test coverage and reliability of the corrections have to be very high because corrections and modifications are impossible to install when the station has been launched into space.

The software is developed on a PC. The programming languages are normally Ada for flight software and C/C++ for other software. A mix of C and Assembler is also common.

It runs separate processes for each instrument. Every hardware component has a driver. Some of the components are connected directly to the processor, some through a bus. A simple multitasking operating system has been developed, but can probably not be reused due the fact that it is specific to hardware and software.

Each project is unique, so instruments are always built from scratch. During the initial phases of the project, the changes are mostly requirement-level changes. The specifications get more detailed as knowledge accumulates and optimal implementation solutions are found. The environment typically becomes more stable as the project proceeds. In the testing phase, the nature of the changes shifts to corrective work.

Maintenance of the software is not possible after delivery, and thus change management is relevant only during the development of the product.

Changes (corrections) occur at two levels:

- **Contractor level:** These changes are recorded using problem report forms and are classified. The change control board, drawn from both the contractor and subcontractors, manages decisions regarding these changes.
- **Subcontractor level:** These changes are made informally. These include errors found in the testing phases and modifications on the internal interfaces of the device constructed by the subcontractor.

Software is developed in a PC environment. No automated test generators, configuration management tools, or metrics tools are used during the development. The RCS configuration management tool is used.

Integration with the other components of the station has been made using testing sessions to test the interfaces of the components. Components have been tested separately during the development phase, and all of them were combined after the first delivery of the software.

Some of the problems encountered in maintenance are:

Requirements change. Requirements kept changing during the project and, thus, changes made to the original requirements (which have not been communicated effectively to all partners) caused problems in the integration phase.

Lack of maintenance of documentation results in inconsistencies. The specifications and requirements for the software have changed frequently. Most changes have been made directly to the source code. Sometimes document updates have been omitted because of frequent changes that resulted in inconsistent documentation.

Version control of the code has been difficult. No automated configuration management tools have been used during the prototyping phase. Generally, one person has completed the programming of each module and when someone else has made changes in the code, version control of the code has been very difficult.

Hardware-oriented code is hard to understand. As the software was very hardware-oriented, the source code was complicated and hard to understand. No tools for analyzing, restructuring or simplifying the code have been used.

High reliability requirements and unsystematic methods make testing time-consuming. Because of the requirement for high reliability of the software, a lot of time is spent in testing. No systematic method to ensure that the coverage of the case software test is complete and sufficient has been used.

Concurrent development delays system integration. Because the product parts are designed and built simultaneously by separate subcontractors, the integration of the devices cannot be completed until each partner has achieved a stable version of their parts.

A.9. Airplanes

This case study is based on a telephone interview conducted as part of this work with a large company that develops and manufactures airplanes.

Operators distributed worldwide use the airplanes mainly to transport people. The most popular plane was developed to meet regional airline operators' requirements for the highest possible standard of passenger comfort at minimum cost. The plane entered service in the mid 1980s following an intensive four-year development and certification program. The product has been continually developed and improved since then, driven by engineering refinements, responses to customer needs and airworthiness requirements.

The software has high reliability requirements due to the fact that failure can cause loss of human life.

Development and maintenance of software is done in a PC environment. Several different simulation systems are available so that the software can be tested in environments that are closer and closer to the target environment, the airplane itself.

The system is really a system of systems involving many different processors, many of which run at different speeds. The hardware is non-standard, developed specifically for this purpose.

Quality is of highest importance. The code is, for example, inspected often and thoroughly, and a high price is paid in that productivity is relatively low.

Some of the problems encountered in maintenance are:

Real-time aspects cannot be simulated. Real-time aspects do not show in the PC environment. Some of these aspects can be spotted in more advanced flight simulators, but the most intricate ones do not manifest until operation on the airplane.

Complexity due to the fact that many different computer systems are involved. A major difficulty for software developers in this environment is the complexity of different computer systems that communicate with each other, and part of this problem is the fact that different systems run at different processor speeds. One system might run at one speed, another at half the speed, and yet another one at a quarter of the speed. Data is moved back and forth between these systems and it can easily seem as though the data was transferred correctly when actually, in real time, not enough time was devoted for the memory cells to stabilize. This makes it very easy to make mistakes.

Specialized systems require specially trained and experienced people. Most of the systems, including the hardware, are developed specifically for one purpose, which creates problems with staffing. A long training process is, for example, necessary for new people not used to the environment.

Changes in the technical configuration cause software changes. Changes in the technical configuration are a big problem because they cause changes in the software. Therefore, the strategy is to stick to a given technology as long as possible. The selected technology has to be available for a long time.

A.10. Satellites

This case study is based on a description in [Kea, et. al., 1999]. The case study company designs and manufactures satellites to be launched into space.

All software on board the satellite is classified as embedded systems and typically has the characteristics of being real-time and dealing with communication or numerical computation.

The satellite is not safety critical, but the system needs to be reliable because it is not easy to reboot, and software failure is likely to cause loss of system or mission.

The major programming languages for development of embedded systems were reported to be C (90 %) and Assembly (10%).

Specialty hardware is an issue, but a significant subset use standards or common products such as a 1553 or 1773 bus, VxWorks RTOS, X86 chips, etc.

The different software systems were classified into five different categories:

- Embedded Systems (ES)
- Mission Ground Systems
- Information Management Support
- Science Processing
- Advanced Technology

The selection of processes and standards to be followed for software maintenance varies significantly from project to project. Testing methods used by teams working on embedded systems were Functional testing, Regression testing, Software simulators, Unit testing, Build testing, and Acceptance testing.

The current understanding of embedded software maintenance is that it is similar to regular software maintenance, only more so. That is, no specific tools or techniques that you would use for embedded vs. regular maintenance have been found yet. The main difference is that many of the normal constraints are a lot tighter.

One of the problems encountered in maintenance is:

Remote, disconnected system. It is not easy to reboot and software failure is more likely to cause loss of system or mission.

A.11. Space Shuttles

This case study is based on a description in [Rus and Zeikowitz, 2000].

The company is an independent unit of a Space Agency and provides independent verification and validation services for it. The company strives to improve software safety, reliability, and quality of Space Agency programs and missions through effective application of Systems and Software Independent Verification and Validation (IV&V) methods, practices, and techniques. The company provides tailored technical, program management, and financial analyses for programs, industry, and other agencies, by applying software engineering "best practices" to evaluate the correctness and quality of critical and complex software systems throughout the system development life cycle. It has 24 employees.

The Space Shuttle software is onboard flight software. The Space Shuttle program is comprised of four orbiters: Endeavor, Atlantis, Columbia and Discovery. Software builds, or operational increments (OI), are used for repeated missions on all four orbiters. Shuttle software development has constraints unique to the four space orbiters. However, mission safety and reliability are utmost criteria for all missions and each new software release. Hardware must also be reliable and withstand the rigors of space travel. Because of this, software is constrained to executing on legacy hardware with limited memory.

Failures of Space Shuttle software can both cause loss of human life and economically large losses.

The target environment is composed of general purpose computers (GPCs), AP-101S from IBM. GPCs consist of a central processor unit and an input/output processor in one avionics box instead of the two separate CPU and IOP avionics boxes of the old GPCs.

The Space Shuttle software is written in High-order Assembly Language (HAL/S).

Embedded software consists of Primary Avionics Software System (PASS) and the Backup Flight Software (BFS) flight software subsystems for the space shuttle. Critical subsystems are ascent GN&C, entry GN&C, on-orbit GN&C, sequencing, data processing system, and main engine controller.

The hardware is non-standard, developed specifically for this purpose.

The customer takes care of maintenance. Input to the new product comes from the needs of coming missions and experiences from past missions. The software is upgraded after the product is delivered. Core functionality is reused and enhanced by extensions that differ from mission to mission.

A development cycle takes approximately 16 months.

In the *definition phase,* flight software change needs for the new mission, and also from Operational Increments (OI) operation, evaluation, and review (discrepancy reports - DRs), are identified. The flight software change proposals are analyzed, and if they are approved, they are changed into change requests (CRs). Each CR is assigned a champion (in a development organization) to analyze the CR impact, risk, resources needed and to recommend a development plan. Results of analyses and recommendations are forwarded to be included in a formal requirements inspection process. Issues that identify risk associated with hazards, integration, and implementation are documented as problem reports. The formal requirements inspection process that follows includes the developer stakeholders. This facilitates developer understanding of the change requirements, the proposed approach relative to reuse, and verification and validation requirements.

The next phase is *Development*, which consists of design, coding and unit/module testing of the approved CRs. The design and code are inspected. These activities are performed by developers, and the result is a *build* that has the complete required functionality.

After a build is completed, the developers perform levels 1,2, and 3 testing; L1 – *Unit testing* (equations, paths, range of values), L2 – *Module testing* (user interface, user commands, functional interface), and L3 – *Functional system level testing* (multiple functions, timing, system interfaces).

The developers hand the product over to the Developer V&V (which is a separate group in the development organization) and to the IV&V. Developer V&V performs *Developer verification testing* (*Functional verification*, and *Performance verification*). This ends at CI (*Configuration Inspection*). CI is a formal review milestone, after which the CI loads are released to customer.

The *Mission Preparation* begins with the release of the PASS (Primary Avionics Software) and BFS (Backup Flight Software) OI loads from the development contractors. The mission preparation phase requires approximately 9 months from the delivery of the OI loads until the first STS mission is flown using the new OI. This phase has two cycles: Engineering Cycle (or initial reconfiguration) and Flight Cycle. Besides load configuration, *Developer Validation Testing* (final reconfiguration load and flight equivalent GPC – not real time simulation), *Integrated Avionics Testing* (flight software, orbiter avionics hardware, support systems interface) and *Operational testing* (mission plan/procedure, mission training – real time simulation) are performed. Three weeks prior to flight, the Software Readiness Review (SRR) is held. Two more reviews (FSS) are held two weeks and respectively two days before flight. Five days prior to launch, the orbiter MMUs (mass storage – magnetic tapes) are dumped and compared by the developer to the mission baseline load.

Maintenance of the flight software during the mission is, in principle, possible but, in practice, not done for security reasons.

Appendix B. Resources

Readers interested in more information on embedded software maintenance, either in order to learn or to improve their embedded maintenance software processes, can find useful information available on the web. We have listed some the resources we found during this study, but many more are available. The description is taken, in many cases, directly from the Web. The fact that a resource is listed here does not indicate any evaluation, judgment, or endorsement from Fraunhofer Center Maryland or DACS.

B.1. Magazines, Journals, and Conferences

Embedded Systems Programming (ESP): http://www.embedded.com/

> This web site brings together various resources related to embedded software. *Embedded Systems Programming* is, for example, a monthly magazine for engineers, programmers, and project leaders who build micro controller and embedded microprocessor-based systems.

PC/104 Embedded Solutions Magazine: http://www.pc104-embedded-solns.com/

> *The PC/104 Embedded Solutions Magazine* is targeted to developers of embedded systems that use PC/104 technologies and non-backplane modular embedded solutions. The purpose of the magazine is to bring PC/104-related business, technology, applications, and product news to engineers who are developing embedded systems.

Embedded Linux Journal: http://embedded.linuxjournal.com/

> This magazine focuses on the use of Linux in embedded systems.

Embedded System Engineering: http://www.esemagazine.co.uk/

> This is an online resource for embedded systems based in the UK.

Embedded System Conferences: http://www.esconline.com/

> These conferences are being held each year in San Francisco, Chicago, Boston, and many European and Asian countries. Some of the features are tutorials, paper presentations, panel discussions, exhibitions, and lectures, etc.

List of conferences: http://www.cera2.com/WebID/embedded/conf/blank/conf/1-a-e.htm

B.2. Notification Services and Tools

eCLIPS is an email alert service from eg3.com. The user selects which keywords to receive alerts for. Keywords can be, for example, “embedded” and “software maintenance”. Whenever eg3.com finds new seminars, white papers, Web sites, free software, etc., for the particular keywords, an email alert is issued.

There is a list of topics related to Embedded Software, Embedded Systems, Embedded Chips, and Embedded Networking.

For more information: http://www.eg3.com/eCLIPS/

B.3. Tools

TASKING products are tools for embedded software development that bring together the advanced software design technology needed to compete in the embedded communications era. The TASKING integrated development environment, compiler, debugger, embedded Internet and RTOS offerings support a wide range of Digital Signal Processors (DSPs) and 8-, 16- and 32-bit microprocessors and micro controllers for all areas of embedded development.

Altium's TASKING products include:

- Development tools for a wide range of DSPs and 8-, 16-, and 32-bit devices
- Comprehensive Embedded Development Environment
- Target-specific, highly optimized compilers
- Powerful and easy-to-use debug tools
- Integration with RTOS and TCP/IP solutions
- Embedded Internet software solutions

For more information: http://www.tasking.com/
and http://www.cera2.com/navi/embed.htm

B.4. Consulting and Training

B.4.1. Embedded Research Solutions

“Embedded Research Solutions’ (ERS) vision is to provide contracting, consulting, training and research services for customers who want to transition from their current design practices to using component-based technology. ERS is further committed to developing and delivering the real-time component technology that enables system-on-a-chip applications and small embedded systems to use a building-block approach for creating complex software applications.”

ERS provides expert consulting on several topics, including real-time systems design and analysis, embedded systems architecting, domain-specific component-based software (i.e., software building block) solutions, re-engineering legacy code, using and modifying real-time operating systems, multi-rate cyclic executives, and device driver design.

For more information: http://www.embedded-zone.com/

B.5. The Embedded Community

The Embedded Community is a forum for service providers to interact directly with individuals and companies in the embedded marketplace. It provides help to:

- Source a project
- Find consultants
- Educate about Embedded Systems

It also provides a search engine with embedded hardware and software categories. The user can narrow down the search using options like products and support, industry and marketing. The search results are conferences, journals, papers, magazines, etc., that are related to the query.

For more information: http://www.embeddedcommunity.com/

B.6. Applied Research: VTT Electronics

VTT Electronics' main mission is to create new technologies and visions to assist the clients in developing new products and technology solutions.

VTT, Technical Research Centre of Finland, is the largest contractual R&D organization in the Nordic countries. VTT is an impartial expert organization that carries out technical and techno-economic research and development work. VTT develops technologies both to improve the competitiveness of companies and the basic infrastructure of society and to foster the creation of new businesses.

Within VTT Electronics, experience regarding telecommunication solutions and applications and embedded systems has been gained from numerous research and industrial projects. Current research is focusing on the development of product line architectures and services architectures for future telecommunication systems, as well as on the design, analysis and testing methods needed in the application development. VTT Electronics' research groups in the embedded software research area comprise expertise on software product quality, software process improvement and measurement, component-based software engineering, development of product line architectures and middleware services for networked embedded systems, as well as the design and analysis methods needed in the application development.

For more information: http://www.vtt.fi/

B.7. University Research

B.7.1. Software Engineering for Real-Time Systems (SERTS)

"The Software Engineering for Real-Time Systems (SERTS) Laboratory is leading University of Maryland's efforts in advancing software technology for embedded systems."

"Industry has a significant demand for engineers experienced in developing embedded real-time software. In fact, the growth of the embedded systems market has been so substantial that some experts cite a 25% growth rate each year, in an annual market of more than $10 billion. The training offered by universities in embedded systems, however, is growing by only a fraction of that amount. As a result, many of the people creating embedded software are engineers trained in other areas - most notably hardware design, mechanical engineering, controls, or communications. This lack of software engineering expertise leads to lower quality software and higher development costs.

There are two fundamental methods of improving this serious manpower problem:

- Educate more engineers in the field of embedded real-time software
- Reduce the complexity of embedded software, so that less expertise is needed to build the systems

The first method is addressed by the educational goals in SERTS, while the second method is addressed by the research objectives."

For more information: http://www.ece.umd.edu/serts

B.7.2. Institute for Complex Engineered Systems (ICES)

"The Institute for Complex Engineered Systems (ICES) is a strategic initiative for pursuing multidisciplinary research on Complex Systems, both within the College of Engineering and across colleges at Carnegie Mellon. The ICES vision is "to develop enabling technologies and systems that seamlessly connect people with their physical and information environments."

For more information: http://www.ices.cmu.edu/frames.html

B.7.3. Embedded and Reliable Information Systems Laboratory (ERISL)

"In the Embedded and Reliable Information Systems Laboratory, our vision is to develop systems and tools that fundamentally improve the effectiveness of embedded computer and networking technologies. To accomplish this requires an extremely broad view of systems, and an appreciation for the complete life cycle of products and processes. Connections among industrial sponsors, class projects, and university researchers are emphasized in order to bring together real-world issues and cutting edge solutions for training tomorrow's engineers."

For more information: http://www.ices.cmu.edu/frames.html

B.7.4. Embedded Systems Laboratory (ESLAB)

The Embedded Systems Laboratory, Department of Computer and Information Science, Linköping University, Sweden (ESLAB) conducts research on the design and test of embedded systems. Some of the relevant research areas are:

- Testability Support in a Co-design Environment – COTEST

  The main objective of the project is to assess whether new high-level test techniques (whose application has, up to now, been limited to RT-level descriptions) can be suitably extended to provide the designer with some support concerning test issues when a behavioral-level description of the system is available.

- Self-Test in Embedded Systems

  The main goal of the project is to develop efficient self-test methodologies and tools to act as an enabling technology for the design of competitive embedded systems, which will be of great importance for a wide spectrum of Swedish industries.

- Hardware/Software Co-Design

  The main objective of this research is to develop methods and tools for the description, evaluation and partitioning of application-specific computer systems which consist of both hardware and software components.

- Design of Heterogeneous Embedded Systems with Distributed Hardware/Software Components

  "The main focus of the research is to develop hardware/software co-design techniques for real-time applications implemented as distributed embedded systems. Our long-term objective is to develop techniques and tools to allow the designers to quickly explore the different design alternatives and find a cost-effective solution for a mixed hardware/software implementation of a given real-time system."

For more information: http://www.ida.liu.se/~eslab/

B.7.5. RISE

"The Research Institute for Software Evolution (RISE), formerly the Center for Software Maintenance (CSM), was established in April 1987, at the University of Durham, England. It is the first such center worldwide to concentrate its research on software evolution. Informally, software evolution refers to all those activities that take place after a software product has been delivered to the customer, but the more formal definition used by the RISE is:

> Software evolution is the set of activities, both technical and managerial, that ensures that software continues to meet organizational and business objectives in a cost effective way."

The institute publishes a journal named "Journal of Software Maintenance" and holds seminar series and workshops in relevant fields.

For more information: http://www.dur.ac.uk/CSM/

B.8. Research projects: DARPA

B.8.1. Model-Based Integration of Embedded Software (MoBIES)

The program will create a new generation of system/software co-design technology, which is highly customizable and composable according to the specific needs of different application domains. Composability means that system level properties can be sufficiently and verifiably predicted from subsystem properties.

For more information: http://www.darpa.mil/ito/research/mobies/projlist.html

B.8.2. Program Composition for Embedded Systems (PCES)

"Program Composition for Embedded Systems (PCES) is developing new technology for programming embedded systems with greatly reduced programming effort and reduced brittleness of the resulting code. Programs for real-time embedded weapons systems are highly tailored to ensure cross-cutting properties, such as synchronization of concurrent operations; processor fault isolation; sensor input and actuator output timing constraints; and safe and efficient cache, register, and memory management. This project is developing technology for programming these cross-cutting aspect properties and for introducing them into the core codes that implement functional requirements of the system. The goal is a set of reusable software for aspect suites, supported by software analysis and composition tools that enable (a) reasoning about the complex interactions and tradeoffs among crosscutting aspects and (b) safe code manipulation.

For more information: http://www.darpa.mil/ito/research/pces/projlist.html

B.8.3. Networked Embedded Software Technology (NEST)

“The Networked Embedded Systems Technology (NEST) program will enable ‘fine-grain’ fusion of physical and information processes. The quantitative target is to build dependable, real-time, distributed, embedded applications comprising 10^2-10^5 simple computing nodes. The nodes include physical and information system components coupled by sensors and actuators.

The program will create code bases, tools and reference implementations for the emerging generation of networked embedded systems, which are based on MEMS sensor and actuation technology, and on the rapid progress in photonics and communication.

For more information: http://www.darpa.mil/ito/research/nest/vision.html