**Obsolescence Management of UI**

Rajan Durgadevi

Rama Surendra Babu

Xavier Prasath

Mohan Sreekant

Abstract – UI Obsolescence mitigation is an important aspect of large systems development & maintenance that has often only been considered once obsolescence is impending. For long lifecycle legacy systems, this has become a major concern as the lifecycles of the components/products that are covered within these systems are often far shorter - up to ten times shorter - than the overall system lifecycle. Many systems can be characterized in this manner and therefore require UI Obsolescence mitigation approaches to ensure the continuing ability for the system to perform and evolve. Current UI Obsolescence mitigation practices make recommendations for designing new software development to slow the onset of obsolescence and make the development more flexible when changes for obsolescence is required. However, currently SEMI software development is often ***not designed with this in mind***.

# Introduction

Many systems are targeted for a niche market and are sold in lower quantities compared to other off-the-shelf, mass produced systems. Systems are very expensive, built with contemporary **state-of-the-art technology**, and may be used beyond the **planned End-Of-Life (EOL)**. Design lifetime in SEMI systems are often in the order of 10 - 15 years but many systems are now used for 20 years or more. Examples of such systems include Viscon Micro, Viscon Macro among others.

As these systems evolve through their lifecycle, software & hardware components may be upgraded in order to improve capability or to extend the systems' mission or to compensate for design flaws that only became apparent after extensive use. In addition, software components may be required to be replaced because the system outlives the planned EOL of the component/s, or that over time there is a larger demand than was anticipated for a lifetime buy or the planned level for that component. The likelihood of these types of changes increases in following scenarios:

1. Modification of existing software components.

2. Procurement of a Commercial Off-The-Shelf (COTS) SW/Components like SEMANTIC, Infragistics etc.

3. Procurement of non-developmental items.

4. Development of a new system.

# Types of obsolesce

Following types of obsolescence can be observered:

1. Planned Obsolescence: When obsolescence of components is expected and planned as part of a systems’ lifecycle.
2. Technological Obsolescence: Due to technological innovation, an existing product becomes outdated when a product that performs the function better is introduced.
3. Human Obsolescence: Human workers could be replaced by machine, when the skill sets of experienced workers become outdated due to specialization in technologically obsolete areas like Basic, very old versions of C and C++.
4. Maintenance Obsolescence: When there is a need to enhance current application with increased function. However, the market demands do not support a supplier’s continued production or support of the legacy components/products.

# Reasons for obsolescence

The spectrum of obsolescence effects is quite broad. It encompasses software and components used in legacy systems which are no longer supported, and also software which is no longer compatible and cannot be upgraded to meet current requirements.

**Software Lifecycle Duration**

System lifecycles are an important reason why obsolescence exists. Unlike a cell phone, which is likely to be discarded in 2-3 years, the lifecycle expectations of large, expensive SEMI SW systems are significantly longer than most of the critical components that make up the system itself.

Not only is it a concern that the components within a system will need to be replaced many times over the lifecycle of a system, there is a more complex problem associated with the uneven update schedules. Since these systems are often deployed in various environments, upgrade maintenance time periods are often limited. In addition, if one item requires upgrade before the rest of the system, it can increase the number of configurations (instantiations) of a system. Multiple configurations make design, development, testing and logistics more complex and more costly.

**Changes in Environment**

Changes in environment can occur in two ways for a given system.

Firstly, a system may be used in a fundamentally different physical environment than was anticipated during development. While the specifications may have requirements that cover many environmental conditions, at times there are certain phenomena that only emerge under the new conditions. These changes may cause a system to need modification in order to maintain performance and not become functionally obsolete.

2nd, systems that are fielded for decades must adapt and evolve to the changing threat environment. As a new threat emerges or an existing threat develops mechanisms to try to beat system, that system must adapt and evolve with new countermeasures to prevent functional obsolescence.

# Obsolescence Mitigation

Obsolescence Mitigation and not Obsolescence Prevention, as it is impossible to prevent obsolescence entirely in these large legacy complex systems. In fact, multiple sources indicated that for systems software the obsolescence problem is far worse than it had been in the past because the design and support of those components is dependent on the supplier. In this section, we examine proposed obsolescence be mitigated in various aspects of a product from systems engineering, software engineering, testing.

**Software & Systems Engineering**

Typically, systems engineering is a front-end activity for a product/project that peaks early in the life-cycle and then reduces in activity level as the project/product moves into development. Systems engineering is also important during the integration and test phases and then finally tapers down after system deployment. Obsolescence mitigation can be proactive or reactive, but in either case, to fully leverage a mitigation opportunity, a system engineering approach should be used.

When being proactive in obsolescence management, part of the objectives or criteria may include obsolescence considerations in the initial implementation approach. If obsolescence occurs in a legacy system, alternatives still need to be evaluated, and this overall approach can still be applied.

The key additional consideration, however, is how the obsolete parts of the system interact with the non-obsolete parts of the system. In other words, can obsolescence be addressed as a local problem affecting only single component or does the impact of obsolescence cross interfaces between components or even interfaces between sub-systems or the system and its environment?

Obsolescence as a form of engineering change and indicates that these changes can and should be planned for at the system level and not only the component level.

Obsolescence is bigger than the availability of a component from a supplier, but rather that there are many changes that can make a given system obsolete (e.g. personalization, scalability to number of users, changing threat environments) and for those changes that can be anticipated, a plan should be in place for how to deal with these different types of obsolescence.

This step is typically performed in the basic systems engineering process. However, this process then continues to consider the future problems or requirements of a system and how those might manifest themselves in a solution space. Once those two efforts have been performed, a plan will be developed to evolve the current system into one that will meet the future requirements of the system, thus dealing with functional obsolescence. Adding systems engineering artifacts to the typical systems engineering activities executed for a product: obsolescence management plan, change plan, future systems requirements, future system design, and future system simulations.

**Use of Standards and Specifications**

Common standards help provide structure to an implementation design. In a proactive approach to obsolescence, use of common standards provides some stability in a changing technological environment. If the technology must eventually be swapped out or modified, the new technology will likely be built to the same standard or a newer standard (backward compatible with old one but containing added functionality) for easier integration into the system.

Often obsolescence mitigation activities attempt to minimize impact to interfaces in order to reduce the likelihood of change propagation as changes to interfaces almost ensure that some level of propagation will occur.

The use of common standards supports this goal because the components that are interfaced to the one facing obsolescence will be less likely to be impacted as the standard interface will not need to be changed.

Examples of common standards range from the Ethernet interface standard RJ-45, wireless Ethernet standard IEEE 802.11, USB 2.0, MIL-STD-1553 for a serial data bus for use in applications, to the TCP/ IP protocols.

Clear specifications with traceability from the system level to the component level requirements provide the context for reactive obsolescence mitigation activities. Specifications provide a context and constraints for implementing alternative design and verification planning.

Without clear specifications, it may be hard to determine if a replacement part or component truly meets the capability of the old component. The new component may meet the memory and throughput capabilities of the old requirement, but there may be some aspect of the overall environmental design or another area that may be impacted by the changes that are proposed.

Specifications are areas that can be re-used on a given project or product with limited changes without major impact as they are written at an appropriate level for re-use (unlike lower level software or design choices which are typically application specific or will require change due to technology updates).

**Use of Open Architecture**

The use of open architectures in the initial design of a system is recommended in order to help minimize the impact of future engineering change be it due to evolution of a system or due to obsolescence. Obsolescence mitigation from the very beginning of systems engineering by planning for open architectures that will allow for technology insertion.

Using the technique of “abstraction and introduction of middleware to help minimize the impact of hardware obsolescence on software”. Newly developed components should be backwards compatible with other older technology so that technology refresh can occur at the convenience of typical maintenance.

Use of open architecture and standard interfaces as these change far less frequently than the hardware and software on which they are implemented. A plan must be in place for future technology insertion and backwards compatibility will help minimize the impact of that insertion. These approaches are typically established at the beginning of a project such that the system is designed with modularity such integration of a new component or part for obsolescence mitigation can be performed smoothly.

While open architecture may be recommended for managing obsolescence among other reasons, proprietary solutions are often selected to gain a different type of competitive advantage. For a system, the designer of the proprietary system has the required knowledge to more easily make changes for these systems, thus positioning them for future maintenance and upgrade contracts. Open architectures make those future contracts more competitive as more companies possess the required knowledge. These competing concerns must be evaluated early in the product design such that whatever approach is chosen, the appropriate obsolescence strategy can be developed.

Developing modular domain specific software architectures to mitigate obsolescence. By doing this, libraries can be developed which would promote reuse and therefore reduce the qualification and test activities required. Encapsulation and Abstraction to help distance the software from potential changes in the hardware.

**Model Based Architecture**

Model based architecture development using the UML syntax, object-oriented design and automatic code generation to limit the impact of platform obsolescence to the code that needs to be developed. The model would only require an interface change to the new platform and then the entire code base could be automatically generated off the same existing model. Use of a model that ties out to the requirements will help bridge the communications gap between the systems engineering and software engineering development groups. re-use and design of a model with units that can be easily changed when functionality upgrades are required. In recent years this approach is also being extended to Systems Engineering in the Model Based Systems Engineering domain using various languages

**Change Propagation Analysis**

Change Propagation Analysis is a technique that can be used to help understand or predict how change in one part of a system can affect other parts of the system. Few definitions surrounding change propagation for a given component as listed below:

1. Constants: Not affected by change.
2. Absorbers: Absorb more change than they cause.
3. Carriers: Absorb a similar number of changes to that they cause.
4. Multipliers: Generate more change than they absorb.
5. Buffers: Minimize the effect a change.
6. Resistors: Components for which there is an engineering or business reason to minimize or eliminate change

Differing design strategies impact the likelihood of change propagation in a system. A highly optimized system leaves little margin in order to drive down cost or increase performance thus increasing the overall change that a change will propagate throughout the system.

However, systems built with added margin have included the option for future flexibility by introducing buffers into the system. Multiplier components would be good candidates to examine to determine if added margin can be developed into the initial design to reduce the impact of change to the overall system. In summary:

"successful change management needs to be informed by all of the following aspects of change and their relationships to each other”:

* The source of the change and the underlying causes.
* The interdependencies between components and systems.
* The types of propagation behavior and their dependence on tolerance margins.
* The consequences of change on product quality, cost, and time to market.
* The state of tolerance margins on key parameters.

**Testing**

Obsolescence can affect test in multiple ways. New test methodologies may need to be developed to test the design changes that result from obsolescence. In addition, the fielded product may not be the only area experiencing obsolescence. The test harnesses and testing SW may also experience obsolescence and should be considered early in any activity where obsolescence is being managed.

The COTS components costs are amortized over the many users; however, the testing is also only focused on the areas of the system most likely to be exercised by most users. If the new projects application of a COTS component diverges from that of the masses or from the context of previously tested projects/products, specialized testing is still needed in order to reduce the likelihood that the difference in application of the COTS component could cause problems during use.

"Even with no explicitly defined operational distribution, testers usually have some information or intuition about how the software will be used and therefore emphasize, at least informally, testing of what they believe to be its central or critical portions. These priorities will likely change, however, if it is decided to incorporate the component into a different software system. The original system may commonly execute portions that the other never will, which makes much of the testing irrelevant to the new user in the new setting.

Likewise, the new user may use parts of the component that correspond to extremely unlikely scenarios in the original system's behavior, and these may have been untested or only lightly tested when the software was developed." Minimum unit testing, integration testing and systems testing are essential for a system and further types of testing such as feature testing, performance testing, load testing, stability testing, stress testing, and reliability testing may also be required. And maintaining the software specification whenever new functionality is added and maintaining traceability between the specification and detailed test cases that have been used to test the software.

|  |  |
| --- | --- |
| OBSOLESCENCE MANAGEMENT CRITICALITY METRIC | |
| Approach | Criticality |
| Replacement available, same footprint | Low |
| Replacement available, different footprint (new layout is required) | Medium |
| No direct replacement available, Different functionality  1) Design modification required  2) New layout Software changes could be Required | High |
| No direct replacement available, process/ technology obsolete  1) New component design  2) Module redesign New layout Software | Critical |

Fig 1. Obsolescence Criticality Metrics

The framework containing common components can be used across different inspection projects improving developing effort and decreasing time to market. It can be reused across different projects without any code change (plug n play) and can also be extended based on the project needs. ​The common framework is main goal is to speed up the project development with good quality and at the same time improve the quality, reliability, scalability, modularity and robustness of new software. In case of handling complex Legacy SW, the need to use automated third-party tools becomes necessary which can help in managing the critical components of the software by giving new avenues of sustenance.

# Risk of Obsolescence

The overall risk of obsolescence can be calculated from the impact of obsolescence and the probability of its occurrence, using a risk matrix (Fig 2).



Fig 2. Obsolescence Risk Matrix

Risk analysis can be done in multiple passes where low risks are disregarded and medium and high risks are analyzed in more detail.  This helps in prioritizing the software components and develop strategies, working through the top priorities first.

# Conclusion

The proactive obsolescence mitigation techniques that could be applied early in the program development include - Open Architectures, use of Interface Standards, Model-

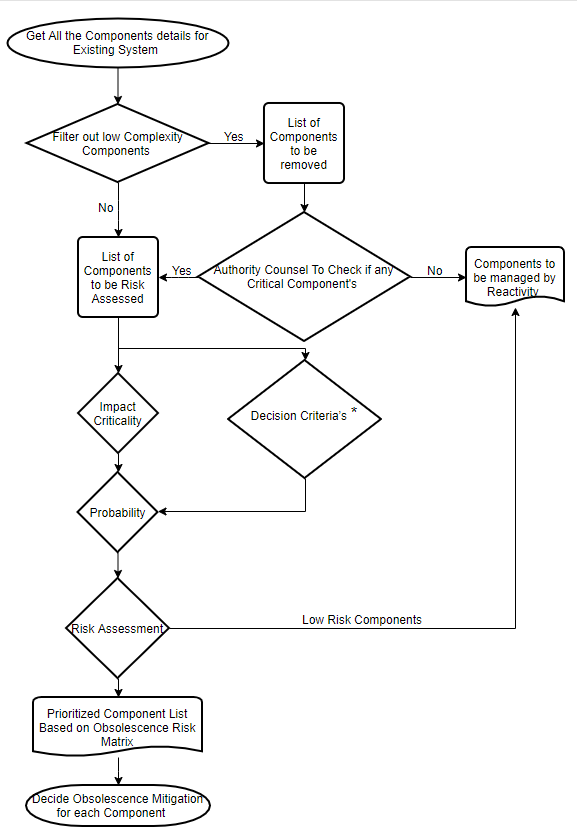


Fig 3. Obsolescence Analysis Flowchart

based Architecture and anticipating future system requirements. A new system can be developed by including obsolescence analysis in the design phase, and then deploy an obsolescence plan throughout the product lifecycle. However, a legacy system does not have this luxury. Legacy systems were designed without obsolescence considerations. Even though Obsolescence mitigation solutions can be applied in legacy system, the system is likely to struggle.

The reactive obsolescence mitigation tends to focus on obsolescence identification through change propagation analysis (Fig 3) to legacy system in order to classify the system components as multipliers, carriers, absorbers or constants. These classifications can be used to identify which components are "critical" and their impact of the entire system.

\* Decision Criteria’s – Following sheet has been considered for various parameters, software assessment with Ranking

