Ten Practical Techniques for High Assurance Systems Engineering

Vicraj T. Thomas
Honeywell Technology Center
Minneapolis, Minnesota, 55418

Abstract
This paper presents ten practical techniques that help engineer complex systems that have stringent dependability and real-time requirements. The techniques described are software centric but in most cases they apply to systems as a whole.

1 Introduction
While there is no silver bullet that solves the difficult problem of engineering a high-assurance system there are several techniques that may be used to help tackle this problem. Ten such techniques are listed in this paper. These techniques derive from the author's experiences with designing and implementing process control systems that have high availability and real-time requirements.

2 Techniques for Engineering High-Assurance Systems

2.1 Use tried and tested technology
The choice of technology used to building a system is critical to its success. Unfortunately the choice of technology is often driven by a checklist of technologies that is drawn up by the product marketing department. The more items on the list that can be checked-off for a system the greater the perception that customers are getting more value for their money.

The dangers of using untried technology include the use of inappropriate technology, the lack of expertise in the technology leading to misuse of the technology, the unfamiliarity of the technology distracting the developer from the safety and reliability concerns of the system, and the lack of good tools that support the technology.

An example of a technology checklist item from the mid-1980s is object oriented design and implementation—almost any software system built since then is expected to be object-oriented. A system that claims to be object-oriented is perceived to be inherently good even though it isn’t clear what it means to be object oriented. In most cases this simply means that the system is implemented using a language such as C++. Object-oriented design and implementation techniques, when used properly, can indeed provide numerous benefits. Unfortunately, most programmers, until recently, were not trained in object-oriented methods and were therefore unable to reap the benefits of the technology. Worse, lack of experience with the technology caused them to make bad design choices.

A more recent technology checklist item is Web based technologies. While these technologies are starting to change the way we think about system structuring they are still in their infancy. Not a week goes by without the discovery of some new security or reliability implication of this technology. Moreover, this technology is evolving so rapidly that it is impossible to find the tools necessary for building large and complex systems.

2.2 Know the limits of the design
Document the limits of the design. Every design has its limits, some of which are explicit but many of which are implicit. For example, a fault-tolerant system that uses N-modular redundancy can tolerate no more than N - 1 module failures. It also assumes that the voting mechanism does not fail. Each module may in turn make implicit assumptions such as there being no more than single bit memory errors. Similarly, the voter may depend on the modules being time synchronized. These dependencies are rarely documented; they remain in the minds of the system designer. The problem is even worse with limits built into the software. Examples of such limits include fixed sized tables and an inadequate number of bits to represent a timestamp.

Anticipate failures of design. It is impossible to document all possible design limits of a system module and to guard against all faults that may cause these design limits to be violated. Other modules that interact with this module must therefore anticipate the failure of the module and must be prepared to deal with the failure.

2.3 Reuse with extreme caution
Reuse has been touted as the solution to many of the problems related to building dependable systems. An obvious benefit to reuse is the reduction of development cost and time. Another benefit is the increased reliability of the system—a module that has been tried and tested in another system is likely to have fewer defects than one that has been newly built. Unfortunately, reusability requires a formal description of the behavior of the module and all its design limits. Such a description is impossible for anything but the simplest of modules and software reuse has therefore been limited to class libraries such as the C++ Standard Template Library.

The reuse of modules without a complete description of all its design limits has led to some spec-
tacular failures. For example, the European Space Agency's first Ariane 5 launch vehicle was lost because of the failure of its inertial reference system (IRS). The IRS had been used successfully on many Ariane 4 launch vehicles and hence reused in Ariane 5. Unfortunately, Ariane 5 has a higher initial acceleration and a much higher horizontal velocity build-up than Ariane 4. This caused variables related to the horizontal velocity to overflow and the entire system to fail.

2.4 Minimize software changes over system lifetime

Software, by its very nature, is easy to change. However, software tends to get increasingly brittle as changes are made. This is because it is difficult to make changes without introducing errors. There are two main reasons for new errors being introduced: hidden dependencies between modules that cause a change in one module to cause another module to fail and a violation of some implicit design limit that causes one or more modules to fail. The implication of this observation is that at some point it is far better to start afresh and rewrite the software rather than to make modifications to existing software.

2.5 Foster a "high-assurance engineering" culture

Building a high-assurance system requires that every person on the project be aware of the consequences of the failure of the system. Often the designers of the system are aware of the consequences of the failure and design the system accordingly but the consequences are not communicated to the development team. The development team often has has little knowledge of the eventual application of the system nor does it have any contact with the end users of the system. This can lead to developers making uninformed design decisions. A fatal implementation mistake can be the undoing of the best of designs.

2.6 Institute a formal engineering process

Most software development groups have a formal engineering process in place but the process is rarely tailored to the needs of individual project teams and the types of systems being developed. An project that is developing a critical system will not necessarily benefit from a process that works well for a project developing non-critical systems such as a word processing package.

Management support is essential for any process to work. It is essential that the process be managed by people who are familiar with the system being developed and the technologies being used. Processes need to be able to adapt to the varying needs of the project over its lifetime and only a person who understands the system well can make the right adaptations when needed.

Project management must also be willing to make the necessary investments in people resources and in development time to make a process work. These investments are often not taken into account during program planning and this results in the process being viewed as an undesirable burden on the project rather than as something that helps the project.

2.7 Good language/tool support is essential

The failure of many systems can be traced to faults that could have been avoided if the system designers and developers had good tools and programming languages at their disposal. The current state of practice in programming languages is dismal and unfortunately many good programming languages from the research community have yet to be adopted by commercial software developers. A good programming language must, among other things, encourage good program structuring, support the specification of design limits on modules, detect the failure of modules, and require modules to deal with the failure of modules with which it interacts. If the use of a non-industry standard language is not an option, tools such as good compilers, debuggers, and memory usage monitors are essential.

2.8 Manage complexity

One of the most difficult things about managing a large system development effort is managing complexity. Most systems start off with well defined functionality and a well defined architecture but over the course of the project creeping featurism causes new demands to be placed on the system. The function of the system is no longer clearly defined and the late introduction of new functions may destroy the architectural integrity of the system. Once basic architectural principles are violated, complexity becomes impossible to manage because it is no longer possible to reason about the system.

Creeping featurism may be caused by management that likes it see its pet features supported by the system. It may also be caused by customers who always want systems to do more than what is provided. The most common cause of creeping featurism in software is however the need for the software to overcome problems with the original design. Software is often called upon to compensate for defects in the hardware or in third-party software used in the system.

2.9 Set budgets for each subsystem

It is extremely important that every subsystem of a system with stringent dependability and real-time be assigned a resource budget. This is especially true of embedded systems that have severe resource limitations. Resource budgets include limits on execution times, primary and secondary storage, and network bandwidth. Budget setting must be considered from the early stages of the design and budget allocations revisited as the design is refined.

2.10 Common sense goes a long way

Finally, common sense techniques go a long way in helping develop reliable software. These techniques include validating inputs to modules, testing for boundary conditions, catching exceptions, and the use of tools such as profilers and memory monitors. They are all easy to use, inexpensive, and take nothing more than programmer discipline.