Risk Detection and Mitigation
Metrics and Design Check Lists
for Real Time and Embedded Systems

Lui Sha,
University of Illinois
lrs@cs.uiuc.edu

C. Douglass Locke,
LC System Services Inc.
doug@douglocke.com

April 19, 2009

Acknowledgement

We want to thank Rob Gold for his support in writing this white paper. We also want to thank Jeff Wilcox for his encouragement, and Russell Kegley, Jonathan Preston, Ben Watson, John Ledyard, Jon Claus, Harry Levison, Ted Marz, Jeff Thieret, and Ceci Albert for their discussions and suggestions.

Executive Summary

The causes of the failure or significant cost/schedule overruns of complex embedded software-intensive systems projects almost always involve a combination of management issues and technical issues. This paper focuses on the technical issues of such complex embedded software-intensive systems across multiple domains, including avionics, spacecraft flight systems, and command & control systems.

Having served for many years as senior reviewers and/or consultants on many complex embedded systems, we have witnessed that in many cases major difficulties have often been caused by a relatively small number of complex and difficult hardware and especially software problems. In many cases however, solutions to these problems are known, but only by a relatively small number of experts. This leads to the recurrent phenomenon of many projects encountering serious troubles and subsequently being rescued by "tiger teams".

As observed by the Standish group "*a staggering 31.1% of projects will be canceled before they ever get completed. Further results indicate 52.7% of projects will cost 189% of their original estimates. The cost of these failures and overruns are just the tip of the proverbial iceberg*."[1] In a discussion of the comparison between bridge building and software development, the Standish group also noted that, "*there is another difference between software failures and bridge failures, beside 3,000 years of experience. When a bridge falls down, it is investigated and a report is written on the cause of the failure.*

[1] http://www.cs.nmt.edu/~cs328/reading/Standish.pdf

This is not so in the computer industry where failures are covered up, ignored, and/or rationalized. As a result, we keep making the same mistakes over and over again."

In many cases, the phenomenon of different projects encountering serious troubles caused by similar technical challenges is exacerbated by a lack of effective metrics and associated application procedures in system design reviews for the detection and mitigation of system engineering risks.

The objective of this white paper is to initiate a process to develop system design review and risk mitigation methods that will be effective in reducing the recurrence of high impact technical problems that are common across many programs, in an analogous way to the monitoring of cholesterol levels to help the prediction and therefore the initiation of preventive measures to forestall heart attacks and strokes. In addition, we also include a small sample of useful metrics during program development and early system integration as suggested by some of reviewers.

Clearly, there are many more metrics and design check list items for multiple phases of the software lifecycle beyond the ones described in this white paper. This paper has been created to identify the need to create these metrics and design check lists, building on the experience of many experts who have been involved in the architecture, design, development, and review of embedded software intensive programs. The authors thank many such experts who reviewed this white paper and who may be available to help bring the proposed work to fruition.

1.0 Introduction

In modern complex embedded systems such as avionics system development efforts, significant cost and schedule overruns are very common. For example, after many years of development and lab tests, the F/A-22 flight test program began in late 1997. But it continued to experience serious avionics instability problems as late as 2003 – "*The Air Force told us avionics have failed or shut down during numerous tests of F/A-22 aircraft due to software problems. The shutdowns have occurred when the pilot attempts to use the radar, communication, navigation, identification, and electronic warfare systems concurrently.*"[2,3]

Even in more conservative civilian avionics, during the development of the air Traffic Alert/Collision Avoidance System (TCAS) software, "*version 6.00 was the original software for TCAS II. When using this software, some very interesting problems occurred. False conflict alerts were being triggered by transponders on ships and bridges. Additionally, parallel final approach courses less than 5000 feet apart were causing false alerts. It has even been reported that a pilot's own aircraft can cause a false alarm. In this situation the pilot found himself trying to outmaneuver himself*"[4]

[2] GAO Testimony to Committee on Armed Service http://www.gao.gov/new.items/d03603t.pdf
[3] The F/A-22 instability problem was finally under control in late 2003 with the assistance of the Avionics Advisory team established by the Office of Secretary of Defense.
[4] http://www.allstar.fiu.edu/aero/TCAS.htm

In a March 2004 GAO report to congress, it was reported that "*in the last 40 years, functionality provided by software for aircraft, for example, has increased from about 10 percent in the early 1960s for the F-4 to 80 percent for theF/A-22*", and that "*DOD's software-intensive weapon system acquisitions remain plagued by cost overruns, schedule delays, and failure to meet performance goals*."[5] It was also reported that in 2003 DoD spent $8 billion (40%) of its software budget in fixing bugs. These problems are not limited to avionics software. There is no indication that the incidence of these problems has decreased in recent years. For example, situations are still occurring in which project milestones are missed with no prior indications from the contractor, indicating either a fundamental lack of understanding of the state of the project or the unwillingness to report known problems.

As reported by IBM, in a typical commercial development organization, "*debugging, testing, and verification activities can easily range from 50 to 75 percent of the total development cost*"[6]. The integration phase is often the most difficult and time consuming phase in large system development. Early in a project, most of the application functional bugs can be localized within individual modules. Unfortunately, challenging subsystem interaction bugs in properties that cut across multiple components such as real time and stability are unlikely to show up early in the project. As integration proceeds, the degree of concurrency and the complexity of interaction increase exponentially. And the larger interaction space allows dormant bugs to turn active, to multiply and to hide. This not only makes debugging a very time consuming and costly activity, but also creates a "sinister" phenomenon in the development of large systems in that a stormy system integration phase is often preceded by a relatively calm unit development and testing phase, which often lowers the guard of the management and review teams. Thus, developing system design review methods and metrics that can detect and help mitigate integration problems lurking beneath the surface are of particular importance. We discuss a small sample of such metrics and design checklists in next section of this paper.

A significant portion of these problems can be traced back to deficiencies in system design such as unexpected interference and delays in resource sharing, inadequate determinism in concurrent interactions, and potential system instability caused by dependency inversion[7]. However, even though these problems generally have their origin in system requirements and design, they very often first become visible during integration. Therefore, many of the metrics and other methods described in this paper are largely oriented for use in guiding problem analysis and recommendations during the integration phase.

In the next section, we give a sample of risk detection and mitigation metrics and design checklists for some of the high impact recurrent problems. We conclude this white paper by describing a proposed process that is needed to develop a more comprehensive set of system evaluation metrics and design checklists.

2.0 Examples of Useful Metrics and Checklists

[5] http://www.gao.gov/new.items/d04393.pdf
[6] http://www.research.ibm.com/journal/sj/411/hailpern.html
[7] Dependency inversion is when critical software (inadvertently) depends on non-critical software

Each set of metrics designed or design check lists for detecting and mitigating a specific type of engineering risk must meet the following requirements:

1. Quantifiably measures, and/or verifiable/testable logical assertion
2. Identifies trends in risk detection and mitigation
3. Predicts domain-significant success/failure
4. Leads to usable information for remediation

Examples include metrics and design check lists to be used during system design review and programmatic ones that can be used during program development

2.1 Determinism

A real time system's behavior is said to be deterministic if, given the same system state and the same inputs, the system makes the same state transitions, produces the same outputs and meets its timing constraints. Otherwise, the system is said to be non-deterministic. Non-deterministic systems are extremely difficult to test, debug, verify, and validate.

Deterministic behaviors of system architecture have profound implications on the quality, cost and schedule of complex systems. For example, the F-35 avionics, while facing many challenges, has a much more stable development record as compared with F-22. This is not an accident, but is rather a result that comes directly from the architecture team's systematic use of constructs with known deterministic behaviors.

The development of a comprehensive and concise set of metrics and design checklists to gauge the determinism of an architecture is of great importance. However, this effort is beyond the scope of this white paper. Instead, we will give some examples that are important indicators of non-determinism. For example, race conditions, especially distributed race conditions, and concurrent fault handling are leading causes of non-deterministic behavior. In this paper, we specifically discuss distributed race conditions.

2.1.1 Distributed Race Conditions

In a distributed system, a global computation is a function of distributed system states and hence all the machines participating in the global computation must have consistent views of the state. For example, in a distributed fault tolerant application, machines that are still working must agree on which have failed in order to carry out a successful reconfiguration. Reaching consistent views takes an elapsed time of at least the one end-to-end computation and network delay. Thus a global computation runs at a slower rate than that of a local computation. A local computation frequently requires the collaboration of other machines, for example, when adjusting the speed of a locally controlled engine.

Networked real time distributed systems such as avionics are often called globally asynchronous locally synchronous (GALS) systems, because clock skews between local clocks can only be bounded but not eliminated. When global computation between nodes is directly driven by local clocks or driven by local events without first being mediated by

a proven correct clock synchronization protocol, the resulting interactions become inherently asynchronous. Given an asynchronous design, the correctness of system interactions becomes a function of relative clock skews and computation and communication delays. From a system engineering perspective, we say that the system design has distributed race conditions. Figure 1 gives a simple illustrative example.

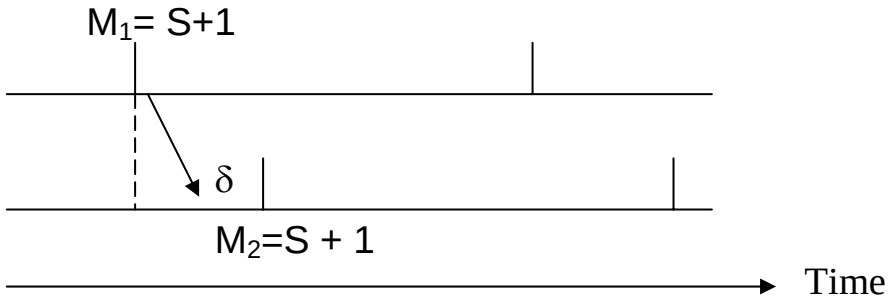

Figure 1: Sending a message to one's logical past

Let M_1 and M_2 be replicated applications on two different nodes for fault tolerance. Both M_1 and M_2 are encountering state (S+1) in their respective local nodes. Suppose machine M_1 enters state (*S+1*) first, while M_2 lags slightly behind. It will enter state (*S+1*) δ time units later as illustrated in Figure 1.

Let the communication delay between the two machines be $\varepsilon < \delta$. When machine M_1 at state (S+1) sends an event (e.g., a message) to replicated machine M_2, the event could be received by M_2 while it is still in state *S* as illustrated in Figure 1. Since these are replicated machines, this is logically equivalent to sending a message to one's own past, resulting in a violation of causality! It is also easy to see that if the pilot (or a high level controller) sends a command to these two replicated machines, it is possible that M_1 receives the event at state (*S+1)*, while M_2 receives the same event at state *S*, leading to divergence in their state transitions and a globally inconsistent state. Distributed race conditions are a known cause that leads to the failure of fault tolerant systems as well as unexpected startups, and restarts. Such failures will be observed only during system integration and/or deployment, and are highly intermittent, so they will be extremely difficult (and expensive) to replicate and find/fix.

In summary, when interactions are driven by asynchronous events, distributed race conditions may occur. When there are distributed race conditions, the verification space of interaction correctness faces combinatorial explosion, since we need to check all the resulting states for all the possible combinations of relative delays at the resolution of each clock tick! This directly results in huge problems in integration testing, debugging, verification and validation.

System Evaluation Metric: Distributed race conditions are best prevented by auditing the design; these type of problems cannot be found through the normal "black box" testing. Once we find that a proven synchronization protocol for interactions is missing, we can easily design a test to demonstrate its existence. However, relying on black box

testing to find distributed race conditions will be worse than looking for a needle in a haystack.

1. Identify every instance of global computation in a distributed system.
2. Check whether a proven correct interactional synchronization protocol is used for each instance.

Recommendation:

1. Interaction synchronization should be a service embedded in middleware, or as library functions. Letting individual application programs reinvent the interaction synchronization solution will lead to excessive design, implementation, integration and maintenance cost.

2. If a system experiences intermittent lockups or anomalies across nodes, audit the design to see if an interaction synchronization protocol is missing or implemented incorrectly.

3. Consider use of a current model-checking tool, because model-checking tools can enumerate all possible interleavings and can therefore detect faults that are undetectable through normal testing.

2.1.2 Fault Reproducibility and Traceability

The ability to reproduce a fault is a prerequisite to successful debugging. Race conditions, especially distributed race conditions are a leading cause of difficulty in reproducing detected faults and failures. Next on the list is event driven design. In an event driven design, a system reacts to each event immediately. Under an event driven architecture, when a fault leads to incorrect state transitions in nodes, the correlation between the chain of events and the time at which a fault has been detected is weak, making a fault much more difficult to reproduce as compared with a time triggered architecture in which events are buffered first and are acted on only at predetermined instants of time. Good fault reproducibility and traceability pays for itself.

System Evaluation Metric: To quantify fault reproducibility and traceability, we can

1. Define the families of common fault types found in similar programs,
2. Conduct randomized fault injection drawn from the selected fault types,
3. Compute the percentage of faults that can be produced and traced.

Recommendation:

- Use a time triggered architecture whenever it is applicable
- Use targeted instrumentation based on risk analysis

2.2 Real Time Performance Metrics

Most embedded systems, including avionics and spacecraft flight systems contain some hard real time tasks. Using an analytic approach such as Rate Monotonic Analysis (RMA) to guide the development and integration of real time tasks is a key to meeting stringent timing requirements. The theory of RMA provides a sound theoretic foundation

and is currently widely considered to be a best practice. In this section, we assume that RMA is used. If not, our recommendation is to immediately train all project systems engineers on this topic.

The prediction provided by RMA will hold only if bounds on worst case computation time and bounds on priority inversion exist and are accurately determined. In this section, we review three important issues impacting real time performance for efficient and predictable real time performance: (1) bounds on priority inversion; (2) bounds on cross partition interference under Integrated Modular Avionics (IMA); and (3) rate group schedulability margin. Although we use avionics as an example, the techniques discussed here are also generally applicable to other real time embedded systems

2.2.1 Bounds on Priority Inversion

Most embedded real-time systems are likely to use static priority scheduling based on rate monotonic analysis (RMA). When a high priority task is delayed by one or more lower priority tasks, priority inversion is said to occur. Bounds on priority inversions, if not calculated correctly, will render an RMA analysis invalid, leading to unexpected timing failures during integration or deployment. Bounds on priority inversion must be computed for each type of shared resource, especially:

1. Bounds on the duration of priority inversion on CPU sharing
2. Bounds on the duration of priority inversion on I/O interfaces
3. Bounds on the duration of priority inversion on each communication switch.

The basic concept of priority inversion is now well known. However, a significant percentage of engineers focus only on priority inversion in the CPU and fail to analyze priority inversion in complex I/O interfaces such as a PCI bus or a network switch. As a result, many real time performance failures during system integration and/or deployment are found in systems with a heavy I/O or communication load.

The experimental investigation of priority inversion bounds must be guided by the actual system architecture and specific configuration model. Otherwise, the number is meaningless. For example, a PCI bus has many different physical configuration and bus transaction types. However, the bound on priority inversion is specific to these physical configuration and selected bus transaction types. As another example, priority inversion for application tasks depends on the specific real time operating system and the real time synchronization protocol that it implements.

System Designs Check List:

1. Check whether the operating system, middleware, and communication support real time synchronization protocols.
2. Validate the bounds on priority inversion for the CPU, IO, and communication switches
3. Establish an engineering process to ensure that if the architecture model changes, bounds are updated accordingly.

2.2.2 Bound on CPU Stall Induced Worst Case Execution Time (SWCET) Inflation

RMA uses each task's worst case execution time as part of the required inputs for schedulability analysis. Many developers assume that the worst case execution time of a task remains the same when it runs alone or runs together with other tasks. Unfortunately, this is NOT true.

As illustrated in Figure 2, modern "smart" I/O devices can be independent bus masters. If a task's cache was first invalidated by prior tasks, the current task will try to reload the cache with instructions across the front side bus. If there is an ongoing bus transaction on behalf of other tasks, the filling of the cache can be significantly slowed because a typical bus master uses a round-robin sequence for competing bus transactions. This causes the CPU to stall and results in a significant slowdown of the task execution. Using a PCI bus, the execution time of the task has been found to increase as much as 37% in laboratory experiments. Indeed, this a key reason for frame overruns that frequently occur when there is heavy I/O.

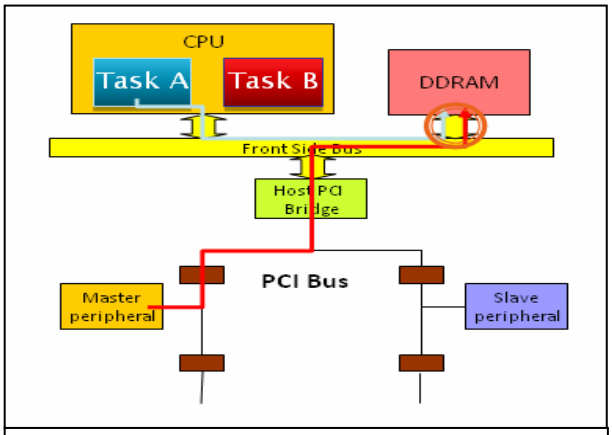

Figure 2: Interactions between CPU, cache & bus

CPU stall induced worst case execution time (SWCET) has ominous implications for modern IMA architectures because many users of IMA architectures mistakenly believe that the CPU cycles allocated to each ARINC 653 partition are isolated from those in other partitions. When tasks in the one partition invalidate the cache of a later partition, tasks in the later partition must reload their cache via the front side bus. The bus is subjected to the interference of direct memory access (DMA) from other partitions. Thus, a partition dedicated to a safety critical real time application can be adversely affected by I/O for non-safety critical applications in other partitions. The solution for this problem is to have an integrated CPU and I/O real time architecture, which is, however, outside the scope of this white paper. What we are concerned with here is to calculate the bound on SWCET, especially in the context of IMA because most avionics systems have widely adopted ARINC 653.

To measure the SWCET for IMA systems, we

1. Flush the cache before the application in next partition starts
2. Conduct heavy DMA on the front side bus as permitted by the existing design
3. Measure the increase of Worst Case Execution Time (WCET) as compared with the case in which there is no DMA on the front side bus.
4. Add SWCET to WCET in the RMA analysis for all of the hard real time tasks

2.2.3 Capacity Margin and Peak Load

In hard real time applications, it is important to monitor peak load and spare capacity for future growth. However, when a task runs, it always uses 100% of CPU. When there is no task running, it idles at 0% CPU load. So what does peak load mean? Furthermore, in a real time system using fixed priority scheduling or rate group scheduling, a task can miss its deadline with a workload considerably less than 100%. How do we usefully measure spare capacity and peak load?

For rate group scheduled tasks, the correct metric is the rate group schedulability margin. The rate group schedulability margin can be estimated by using exact schedulability analysis to compute the worst case margin for each rate group ((Margin = deadline – worst case completion time). The system peak load corresponds to the minimal margin over all of the rate groups. However, margin and peak load computation is correct only if the estimation of worst case execution time, the bound on priority inversion, and bound on SWCET are all valid.

Note that this computation for peak load is very different from the peak load measurement generally reported for most computer systems. Most peak load values are generated by measuring the utilization of a background task, or by summing the measured loads of each rate group. These methods for computing peak load lead to highly optimistic views of the worst-case system performance

System Design Check List: During system integration, an experimental measure of rate group schedulability margin and peak load should be conducted to guard against inadvertent mistakes in parameter estimation.

1. For tasks not yet written, replace them with dummy tasks using busy loops and dummy I/O.
2. Run all the tasks under stress scenarios, including lower priority tasks. Running low priority tasks is important to check for priority inversions that may have been overlooked in analysis.
3. Permitted by the design, engineer test runs with heavy I/O workload and heavy application CPU workload concurrently.
4. Program the logic analyzer and capture the minimum schedulability margin for each rate group task (Margin = deadline - completion time)
5. Plot the schedulability margin for each task.

The system peak load is (1.0 – the minimum of the task schedulability margins).

Recommendation IMA is relatively new and many engineers and even system architects are not aware that if nothing is done, inter-partition interference can be as high as 30-40%. It is not wise to roll the dice.

1. Establish a small but competent real time performance engineering team and architect.
2. Partition structures with low priority inversion and low inter-partition interference.
3. Measure, validate and track bounds of priority inversion and inter-partition interference.
4. Develop a schedulability model.

5. Estimate, measure and track schedulability margin and peak load.

2.3 Defect Rate and Defect Closing Rate and Related Indicators

The profiles of the difference between defect reporting rate and defect closing rate are simple and useful high level indicators of difficulty encountered by the program development. These are a commonly used "lagging" indicators of problems found to be resistant to identification, isolation, and remediation, but are frequently missed as indicators that a software technical review should be conducted immediately to prevent significant risk of cost and schedule overrun.

During unit testing, the typical profile is an initial spike of defect rates, followed by a fast defect closing rate, and then followed by a slower closing rate for a small number of more serious defects that frequently take weeks or even longer to close, typically due to the need for a redesign. If a unit has a significant number of defects that require redesign, the system engineering team should immediately step in and provide support and oversight. Simply pushing the schedule and pushing the problematic unit into integration is a recipe for generating serious system integration problems.

During system integration, the typical profile begins with a continuing rise of defect reports. Unfortunately, this rise is often not followed by a rapidly declining closing rate, due to the difficulty of tracking down and fixing interaction bugs. Here are two example warning signs of potentially serious system integration problems:

Cascaded Interface Changes: Cascaded interface changes are a serious warning of architecture instability due to either poor design and/or requirement instability.

> **System Evaluation Metric:** Cascade interface changes are measured by counting
> 1) The number of groups encounter inter-related interface changes
> 2) The number of interfaces change in each group
>
> **Recommendation:** Prime contractor senior management must immediately step in and establish/strengthen the system architecture team to ensure that requirements are stable and to adopt an architecture model that is known to support the requirement in question. If requirements instability comes from customer requests, then the prime contractor must get together with the customer's senior management, identify the sources of instability, and stabilize the requirements *at the contractual level*.

Time to Closure/Time Open: A persistently poor defect closing rate at the system integration time usually indicates that there are serious defects in the system design and/or implementation. Some of the common problems are listed next subsections. When we see a persistently poor closing rate during system integration, it is like a doctor encountering a patient who has radiating chest pains.

System Evaluation Metric: Count the number of open defects that persist over time intervals such as 1, 2, 3, > 4 months.

Recommendation: This calls for an immediate system architecture design audit. Strengthening the system architecture team must become a top management priority. A much better program management process is to utilize an experienced independent review team to audit the architecture and track the follow-on designs to make sure that the designs are free from common pitfalls such as the ones discussed in these subsections.

2.4 System Stability

Software faults are caused by defects triggered by a combination of system states and external inputs. Hence, the reliability of a computing system is a function of its application profiles. Stability is a measure of the resilience of a system's critical services under heavy work load and under interferences from the erroneous behaviors of failed less critical services. That is, stability reflects the reliability of critical/essential services under stress conditions. Mathematically, it is the mean time to failure of critical services under profiles of overload conditions, out-of-sequence behaviors, and faulty behaviors from non-critical services.

The leading causes of low stability are:

1. Poor isolation in the sharing of physical and/or logical resources across components with different levels of criticality.
2. Dependency inversion in design or implementation: a critical service (inadvertently) depends on the service of less critical components. A critical service may use but not depend on the service of less critical components. For example, a critical service must never wait for a less critical service without a time limit.

Recommendation: for systems with multi-level criticality, it is advisable to measure system stability during early stages of system integration as follows.

1. Separate services into different criticality levels.
2. Define the levels of workloads that exceed requirements but may occur.
3. Define the families of common fault types in non-critical services.
4. Conduct randomized fault injection tests drawn from the fault types into non-essential components. The purpose of injecting faults into non-critical services is to test the quality of isolation mechanisms which include memory protection, temporal isolation in CPU, I/O and network, and input error checking.
5. Compute the mean time to failure of the essential services under excessive workload and injected faults.

3.0 Conclusions

Developing complex software-intensive embedded systems, generally with real-time constraints such as advanced avionics systems, is certainly risky as evidenced by the epidemic of cost and schedule overruns. It is risky because we constantly push the envelope of existing knowledge and technology.

Changing technology landscapes generate changing risk profiles. For example, recent widespread adoption of IMA by the avionics industry has created a new form of system integration risk that has not been seen in previous federated system architectures.

In this white paper, we have provided a small sample of useful programmatic metrics and system design check lists that can be used in program reviews to detect some recurring system integration problems. We recommend the development of a full report which systematically addresses technical challenges during design time, implementation time and integration time. A major goal of such a report would be to generate significant progress toward reducing the number of projects encountering cost and schedule overruns at integration time due to errors in requirements, design, and implementation decisions.

Some of the design check lists described here are applicable to early system architecture designs and others are applicable during detailed design. Some programmatic metrics are useful during system development and some are useful during system integration. Thus, it is important to keep at least the core of the system architecture team in place for the entire system development cycle.

The principles that govern the development of risk detection and mitigation metrics are:

1. **Before the System Development:** The development of risk detection and mitigation metrics must be based on the analysis of actual recurring high impact problems, analogous to what NTSB does for accidents.
2. **During the System Development:** Metrics and design checklists must be selected according to the type of system in question, and adjusted for each major stage of development. Currently, a significant fraction of the metrics used in system review is time consuming to collect and ineffective in practice. Developers often ignore them before and after program reviews.
3. **After the System Development:** There must be a feedback process to examine the effectiveness of the metrics. Ineffective metrics and design checklists must be eliminated, weak ones must be strengthened, and missing ones must be added. Similar to the NTSB after an accident, the root cause(s) of the observed failures should be identified and mitigation processes should be defined to prevent their recurrence in other systems.

The Bottom Line: We recommend the Office of the Secretary of Defense establish a DoD or National Systems Architecture and Design Agency (SADA) with in-house and external experts to execute independent design reviews, drawing experts from senior architects, leading researchers, and technical leaders in "tiger teams" to:

1. Identify and agree on the high impact recurrent problems, the contexts, and the early warning signs.
2. Develop effective metrics, standardized design checklists, and supporting engineering management processes for detecting, tracking, and mitigating high impact recurrent problems.
3. Capture the best organizational and technical practices and provide recommended practices.
4. Develop a plan for making this information readily known to project management and technical personnel across the system development community, both in the government and the contractor communities.