A Method for Critical Software Event Execution Reliability in High Assurance Systems

Marie-Elena C. Kidd
Sandia National Laboratories
Albuquerque, NM 87185-0535
mckidd@sandia.gov

Abstract

This paper presents a method for Critical Software Event Execution Reliability (Critical SEER). The Critical SEER method is intended for high assurance software that operates in an environment where transient upsets could occur, causing a disturbance of the critical software event execution order, which could cause safety or security hazards. The method has a finite automata based module that watches (hence SEER) and tracks the critical events and ensures they occur in the proper order or else a fail safe state is forced. This method is applied during the analysis, design and implementation phases of software engineering.

1. Introduction

Currently, the focus is on benchmarking and evolving a method for Critical Software Event Execution Reliability (Critical SEER) in environments where transient software or hardware failures or harsh operating environments could lead to critical software driven events being executed out of the expected and safe, secure or reliable order.

The implementation of the Critical SEER method is simple and easy to maintain. This method focuses on adding code to the target software to provide run-time fault detection of the critical software driven event execution order. This is a finite automata based method. The Critical SEER method can also keep a historical log of critical software driven events with time stamps if desired. Extensions are being developed to provide fault correction instead of just fault detection of the critical software driven event order.

This paper discusses the problem that is targeted by the Critical SEER method along with a small sample of the current solutions. The method is then described and its strengths and weaknesses are analyzed.

2. An Important Problem to Solve

2.1 The General Problem

In high assurance applications, critical software driven events must occur in a specific order to maintain safety, security and reliability. If the software is correct and unperturbed by hardware upsets, everything is fine! If, however, transient hardware upsets (e.g. timing or environmental upsets) perturb the software event execution order in such a way as to move the software execution off the intended path to an undesired software location, the result could make the system behave erratically. Furthermore, if this deviation from the expected path occurs and remains undetected, the software execution flow will continue down a newly charted, erroneous path. Figure 1 displays the two scenarios.

Figure 1 - Abstract Examples of Normal and Perturbed Software Execution Paths

When such undesired jumps occur, the results may be undetectable, cause minor problems, corrupt data, or cause major safety, security or reliability problems.

This is not a worry for every application. It depends on
the cost or repercussions of such a failure and the probability of such a failure occurring. Sometimes these failures go completely unnoticed. Other times, the effect can be as devastating as a nuclear reactor reaching an unstable state or a fatal radiation overdose from a common medical treatment [5].

2.2 Transient Program Counter Disruptions

The undesired jumps that cause perturbed software execution paths are generally caused by program counter corruption. The program counter is the register or data location that stores the address of the next instruction to be executed. The existence of the program counter is pronounced and obvious to the software developer in embedded applications where microprocessors or microcontrollers are utilized. However, the program counter is a normal part of any type of device that executes software and presents a single point of failure to perturb software execution flow.

2.3 Causes

The main cause of program counter corruption is electrostatic discharge (ESD) which at high levels will simply make electronic components or chips inoperable. At low levels, as produced by charged device model (CHD) ESD [3], the damage can lay dormant for long periods of time. This can allow a unit to pass testing and yet years later act erratically. Lightning is another type of ESD that can create concerns for some applications. Generally, other environments that will cause similar problems for the program counter are extremes in temperature, magnetic fields, electric fields and radiation.

Whatever the cause, if the problem is transient corruption of the program counter, the Critical SEER method should help. However, if the problem is an enduring fault (e.g. "stuck at" fault in the address and data bus or all memory is erased or greatly corrupted), chances are all of the software will be rendered useless. If, for example, a huge and powerful magnet is run over whatever contains your code, and the component is not shielded for such high doses of magnetism, the software will most likely not function at all.

3. A Sample of the Current methods

Many ad-hoc methods have sprung up as solutions to the transient program disruption problem. Many of them are very creative and can be effective.

3.1 The Encoded Data Fields

The encoded data field family of methods is a fairly common solution. The basic idea is to 1) identify the critical events, 2) bit encode a byte, or some collection of bytes, to identify dynamically, at run-time, a snapshot of which events have occurred or have not occurred, 3) check this bit encoded variable for status before executing a critical software event and 4) update the encoded variable to reflect the completion of a particular critical event by logically changing that bit. Figure 2 shows this scenario.

A Bit-Encoded Byte:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

0 = event has not occurred
1 = event has occurred

Figure 2 - An Example of an Encoded Data Field

Many variations of this scheme exist. For example, if single bit errors are a worry, duplicate the bit mapping on to two bytes, instead of one byte, and possibly change the order of encoding in each.

When checking to see what events have occurred, simply look at the byte value to determine the answer. This shows a snapshot history of the event completions at the given point in time.

This method does give limited information on the order of events. It does not give repetition counts for events (i.e. loops). If two events are mutually exclusive, that must be built into the check point code with a logical equation (i.e. masking of selected bits). This method also takes care of sequential events (e.g. event c followed by event d) and selection (e.g. either event a or event b, but not both).

A difficulty with this method is maintaining such a scheme over time. Each time a critical event is added, moved or removed, the entire logic of the bit encoding scheme must be revisited and reworked. Also, the check points which do the logical arithmetic are implemented as code that is usually spread throughout the target code, so one must hunt down many different modules to make a change to the scheme.

3.2 The real-time Addition/Subtraction Data Field

Another method that is employed is based on very simple
DISCLAIMER

 Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
mathematics, mainly addition and subtraction. The basic idea is to 1) identify the critical events, 2) determine all possible valid paths between the events and 3) assign a simple equation to add or subtract a unique, predetermined integer to an event keeper variable. The event keeper variable is initialized to some value and then during execution is dynamically updated with each identified path segment visited. Refer to Figure 3 for an example.

Figure 3 - An Example of the Addition / Subtraction Scheme

This method does take care of loops if the exact number of repetitions is unimportant, sequential events (e.g. event c followed by event d) and selection (e.g. either event a or event b, but not both). The designer must be very creative and careful to ensure that unique values for each path and mode are chosen.

Again, the difficulty with this method is maintaining such a scheme over time. Each time a critical event is added, moved or removed, the entire equation scheme must be revisited and reworked with great care. Also, the equations are implemented as code that is usually spread throughout the target code, so one must hunt down many different modules to make a change to the scheme.

3.3 The Need for Something More

The problem with these creative methods is that maintenance of the resulting code is often difficult. The first problem with maintenance of such code is understanding the method itself. The second difficulty is figuring out how to alter the scheme design to move, add or delete critical software event execution items from the scheme. The third problem is finding all of the pieces of code for the scheme to make the implementation changes. Of course, if the same designer/coder is around forever, then that same person can modify the code. Unfortunately, most of us live in a world where personnel changeover is common and the creative genius that created our code is long gone by the time it needs to be updated.

Additionally, any software scheme to tackle the critical software event execution reliability problem is going to add complexity to the software system and could, at least theoretically, add bugs to the system.

Because of the problems with ad-hoc methods, it is believed that a formalized, repeatable, easy to maintain, easy to change (move, add or delete critical events from the scheme) and solid method is needed that adds a minimal amount of complexity.

4. Our Proposed Method, Critical SEER

4.1 General Description

Initially, one starts with target code which has several critical software execution events identified in it as depicted in Figure 4. The critical events should be identified by safety, security or reliability requirements on the system and possibly by a hazards analysis.

Figure 4 - A Look at Normal Code Segments With Various Critical Events

Figure 4 is an abstract view that shows pieces of code, which could be functions, modules, objects or just fragments of code. At this level of abstraction, it really doesn’t matter. This method works equally well with Structured methods and Object Oriented methods. The method is not tied to any particular implementation language either.

The Critical Software Event Execution Reliability (Critical SEER) method has three parts which are added to target code 1) check points (as a function call), 2) update points (as a function call) and 3) a finite automaton module which encapsulates the functionality into one module or object.

4.2 The Check Points

The check point is the point at which the software
execution will check to determine if continuing normal execution is acceptable or not. Continuing normal execution means that all preconditions have been met and the execution of the following critical event is enabled. A check point (implemented as a function call) is placed as close as possible proceeding a critical event which is in most cases the proceeding line of code (or in assembly could be several instructions prior). Figure 5 shows where the check points are added with respect to the critical events.

![Figure 5](image)

**Figure 5 - A Look at Adding Check Points Before Critical Events**

The exact syntax will vary based on the implementation language used. The pseudo code format is as follows:

```pseudo
call protection_modulename (mode, critical_event_name)
```

or

```pseudo
check_point (critical_event_name)
```

The `protection_modulename` is whatever the finite automaton module is called. The `mode` is either initialize, check point or update point. The `critical_event_name` is the identifier for the current critical event being pre-checked. This identifier could be an integer that is an index into a finite automaton table.

### 4.3 The Update Points

The update point is the point in the software execution where it will be recorded that a particular critical event occurred and is now complete. The update point (implemented as a function call) is placed as close as possible after the critical event which is in most cases the following line of code (or in assembly could be several instructions after). Figure 6 shows where the update points are added with respect to the critical events to our evolving model.

![Figure 6](image)

**Figure 6 - A Look at Adding Check Points Before Critical Events and Update Points After Critical Events**

The exact syntax will vary based on the implementation language used. Here are two possible pseudo code formats:

```pseudo
call protection_modulename (mode, critical_event_name)
```

or

```pseudo
update_point (critical_event_name)
```

The `protection_modulename` is whatever the finite automaton module is called. The `mode` is either initialize, check point or update point. The `critical_event_name` is the identifier for the critical event just completed successfully.

### 4.4 The Finite Automaton Module

The finite Automaton (FA) module is the core of the Critical SEER method. It contains a mathematically based map of all acceptable critical software event order configurations. During execution, it keeps track of the critical events with respect to that map. It can be thought of as a run time policing module for all identified critical events. In other words, no critical event is executed without the FA module’s permission. This is ensured by the fact that the check points are actually function calls to the FA module asking for permission to proceed and the update points are function calls to the FA module reporting successful completion of a critical event. Figure 7 depicts this scenario.
The Critical SEER method covers the following operations: selection, sequence and repetition of critical events. Refer to Appendix A for a brief review of basic Finite Automata. The existing initialization module in the target code is a natural place to make sure the FA module is initialized appropriately at system startup.

Actual implementation of the FA matrix, the transition table, can be more efficient by taking advantage of sparse matrices. This can greatly reduce the data memory depending on the complexity of the FA.

The FA should also be reduced to a unique, minimum, deterministic state. For example, if the Regular Expression (RE) of the critical event execution order is "and + at", a minimized (by one element) RE is "a (nd + t)". Note, appendix A gives a brief review of Regular expressions as they relate to Finite Automata.

When the FA module detects a breach of order, it forces a fail safe, secure or reliable path as dictated by the requirements. Carefully consider the correct way to fail because a plain shutdown is not always safe, secure and reliable.

4.5 Where in the Software Engineering Life Cycle is the FA Model Derived and Used?

Ideally, the critical event FA model should be derived from models of the requirements. The requirements should include safety, security and reliability requirements as appropriate. A hazards analysis is another excellent source for such requirements. The FA model should then be migrated to design models and finally implemented with the target code. Figure 8 depicts this progression.

Figure 7- A look at Adding Check Points, Update Points and the FA Module

Figure 8 - A Very Abstract View of the Interaction with the Software Engineering Life Cycle

Many types of software models exist. Some are better suited for deriving the critical software event FA model for the FA module than others. Basically, any model that clearly depicts order is useful as long as it is the order that corresponds to the critical events.

State transition diagrams are a very clean way to model the critical event FA. Data flow diagrams may be used if the critical events are process based. Flowgraphs [1] or flowcharts work well, but are more design based. Any other method that models order relevant to the types of critical events in the system will work.

Examples of models that are not good for modeling the critical event FA are hierarchy charts, structure charts and data structure charts because these do not depict an order with respect to critical events.

4.6 The History Log Extension

One extension that fits very cleanly in the FA module structure is the addition of a history log of critical events. This is a natural extension because the FA module must keep track of which events have occurred and what the last critical event was. The history log just records that data in memory. This memory will be available at the "end" of the software execution (e.g. possibly nonvolatile
memory) or, if the software operates continuously and has no “end”, the history log can be downloaded. The history log is analogous to an airplane data flight recorder (the black box that is found at an airline crash site and tells the investigators what events took place before the disaster). Obviously, to use the history log for post analysis, the memory must survive the software execution (e.g. be in nonvolatile memory and must not be blown up).

The cost of this extension is memory. So, depending upon the application, this may be highly desirable or it may be impractical due to limited memory.

4.7 The Time Stamp Extension

As long as a history log is being kept, it is a very natural extension to retrieve and save the time stamp of each event for later analysis. Once again the cost of this extension is memory. This is especially true based on the size and format of your time stamp field.

4.8 The Fault Recovery Extension

One of the areas being investigated is fault recovery at the critical event level of granularity. Since the FA module keeps track of all critical events that have occurred, upon a failure of that order, the FA module knows which critical event was last to be successfully completed. Therefore, it seems like a natural extension to be able to look at the FA model and determine where to restart executing software.

The short coming to this approach is that the level of granularity is so high when just dealing with critical events, that a lot of information would be unknown at a lower level of granularity.

5. The Advantages and Disadvantages of the Critical SEER Method

The Critical SEER method is proposed as a method to ensure critical software event execution reliability. Many unique software usage scenarios and applications exist and each application has its own unique requirements and environments. Therefore, the use of this method depends upon how well the method fits the application’s specific requirements and operating environments.

Here are some general advantages and disadvantages of the Critical SEER method.

5.1 Advantages

In general, it is easy to maintain and modify the three pieces of software (check point and update point function calls and the FA module). The logic is very easy to follow and implement.

The FA module encapsulates all of the logic into one module. Therefore, the programmer does not have to hunt down each piece of code to modify the FA model. The parts that are not encapsulated are the update point and check point function calls which have a very specific and readable format.

Since the FA module is based on already developed FA theory and the underlying algebraic expressions, this method is considered to be math based. That basis may make qualification easier if that is a concern.

5.2 Disadvantages

The Critical SEER method implementation does take memory, both code and data memory. However, the other methods take up memory space as well, but they take up more code than data memory. Depending on the actual operating environment, it may be possible to put the FA tables into code memory rather than data memory if data memory is sparse.

This method, when implemented, does take CPU cycles during execution. So, if you are implementing for a microcontroller and have nanosecond timing constraints, you may want to opt for the other methods and cut your execution time to the bare minimum or use a minimal implementation of the Critical SEER method.

There is a small learning curve for basic FA theory. However, most high assurance software developers come from a computer science background (where they will have had automata theory), an electrical engineering background (where they will have learned about state machines) or a mathematical background (where the algebraic nature will be comfortable).

The granularity level is at the critical event execution level. That means that program upsets causing perturbations that occur between two critical events will not be caught until (and if) they actually perturb the critical event order.

5.3 What Types of Applications are Best Suited for the Critical SEER Method?

The best applications for the Critical SEER method have the following attributes.

- The system has allocated safety, security or reliability critical functions to the software.
- The application has sufficient computing speed and free data and code memory available for the implementation of the Critical SEER method.

Owners of such applications should also take these additional issues under consideration.
• Is there a reasonable chance that the software will be changed due to changing requirements or maintenance in the future. If so, the encapsulation of the Critical SEER method will be helpful.

• Since the FA module is based on FA theory, the design will be based on algebraic expressions that describe the critical software event execution order. This may help in the software qualification efforts.

• If there is some code and data memory available for implementation of the Critical SEER method, but not very much, consider taking out the extensions. For example, the history log and time stamp extensions can take up a lot of data memory.

5.4 Types of Applications that are not Well Suited to the Critical SEER Method

The worst applications for the Critical SEER method have either of the following attributes:

• The application has insufficient computing speed to accept the overhead.

• The application has insufficient free data or code memory for the implementation of the Critical SEER method.

6. Preliminary Benchmarks of the Critical SEER Method

Preliminary benchmarking has been done for the Critical SEER method. The results were not at all unexpected taking into consideration the nature of FA models.

The first benchmark was done in the C programming language and was cross-compiled from C to 8051 assembly language for a real-time, embedded application. For this benchmark, an existing application with an existing ad-hoc method to ensure critical software event execution order was used. The application had 9 critical events.

The code memory used was minimal. The following measurements are for the Critical SEER-related code memory usage only.

• Critical SEER method with the History log: 399 bytes (281 for the FA module and 118 for function calls)

• Critical SEER method without the History log and a minimized check point/update point scheme: 235 bytes (173 for the FA module and 62 for the function calls)

The following measurements are for the Critical SEER-related data memory usage only

• Critical SEER method with the History log: 32 bytes

• Critical SEER method without the History log and a minimized check point/update point scheme: 13 bytes

The ad-hoc method, an application from the encoded data field family as discussed in section 3.1, took 109 bytes of code memory and 1 byte of data.

The addition/subtraction method example we have access to took up more than twice as much code memory as the encoded data field example. This method is described in section 3.2.

So, the Critical SEER method does take code and data memory for implementation, but the gains in maintainability, repeatability and possibly qualifies in are in most cases worth it, assuming the application is suited to the Critical SEER method.

7. Summary

This paper proposes a Critical Software Event Execution Reliability method for use on high assurance software driven systems where transient hardware upsets or faults could lead to transient program upsets which could, in turn, disrupt the order of the critical software events.

This method focuses on adding code to provide run-time fault detection at the critical software event execution order level of granularity. An extension may be added that keeps a history and time stamp of each critical event occurrence. Fault correction is also considered a natural extension at this level of granularity.

This method has three parts that are implemented in the target code. The first part consists of check points, implemented as function calls, that proceed each critical event. The second part consists of update points, also implemented as function calls, that follow each critical event. The third part is a finite automaton based module that contains a mathematically based map of all acceptable critical event orders that tracks and polices actual run-time critical event order.

This method is easy to maintain over the life of the software due to the simplicity of the method and the encapsulation of the logic in one module.

The value provided by this method is the avoidance of possible disasters caused by the software executing critical events in an erroneous order. An example is, don’t let the wheels of an aircraft retract before “take off” is complete.
Appendix A - Brief Review of Basic Finite Automata

The review information in this section is derived from [2] and [4].

A Finite Automaton (FA) is defined as a quintuple involving states and input values.

\[ FA = (Q, \Sigma, \delta, q_0, F) \]

- \( Q \) is the finite set of states.
- \( \Sigma \) is the finite input alphabet.
- \( \delta \) is the transition function mapping \( Q \times \Sigma \) to \( Q \) such that the signature of the transition function is \( \delta: Q \times \Sigma \rightarrow Q \). Using function notation, this is \( \delta(q_i, a) = q_f \). This means, when in state \( q_i \), which is an element of \( Q \), with input \( a \), which is an element of \( \Sigma \), the resulting state, \( q_f \), is given by the transition function, \( \delta \). Another way to describe this is that the transition function takes each possible state and input pair and defines the resulting state.
- \( q_0 \) is the start state (also known as the initial state). And, \( q_0 \in Q \), which means \( q_0 \) is an element of the set of states, \( Q \).
- \( F \) is the finite set of final states. And, \( F \subseteq Q \), which means the final states, \( F \), are a subset of the set of states, \( Q \).

The two standard representations for finite automata are transition diagrams represented as directed graphs, as depicted in Figure A1, and transition tables, as in Table A1.

The two standard representations for finite automata are transition diagrams represented as directed graphs, as depicted in Figure A1, and transition tables, as in Table A1.

Figure A1 - Example of a Finite Automaton Represented as a Directed Graph

Table A1 - Example of a Transition Table

<table>
<thead>
<tr>
<th>states in Q</th>
<th>inputs in ( \Sigma )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td>( q_0 )</td>
<td>q_1</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>q_2</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>q_f</td>
</tr>
<tr>
<td>( q_f )</td>
<td>q_f</td>
</tr>
</tbody>
</table>

Regular Expressions (RE) are simple algebraic expressions describing languages that are accepted by an associated finite automaton. For example, the previous section gave a finite automaton that accepts the set of input strings of the form 'a' followed by 'nd' or 'a' followed by 't'. This is a long way of describing a very simple expression. Regular expressions give us a simple and compact way to describe such expressions. Table A2 gives the basic syntax of regular expressions. A and B are sets of input symbols.

Table A2 - Regular Expression Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A + B</td>
<td>This is sequence or concatenation. It means A followed by B.</td>
</tr>
<tr>
<td>A*</td>
<td>This is called Kleene Star or Kleene closure. It means 0 or more occurrences of A which is repeated concatenation.</td>
</tr>
</tbody>
</table>

This is a very basic review of finite automata. There are more complex and advanced areas within automata theory, but they are not necessary for an understanding of this paper.
References


*Marie-Elena C. Kidd* is a computer scientist and Senior Member of the Technical Staff at Sandia National Laboratories. During her ten years at Sandia, she has worked as a software engineer on embedded, real-time software systems for such applications as robotics, nuclear weapon components, and control systems. She has also worked on lab-wide information sharing software systems and software engineering initiatives. She is currently working on high integrity software research, development and processes as a member of the High Integrity Software Systems Engineering Department. She has a B.S. in Computing and Information Sciences from Trinity University in San Antonio, Texas and an M.S. in Computer Science from Purdue University in West Lafayette, Indiana.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.