An Analytical Approach to Identifying Software Initialization Issues

Gerald T. Rigdon
Fellow, Software Engineering
Boston Scientific, Inc.
gerald.rigdon@bsci.com

Abstract

Proper initialization is an important component in effective software design. Although commercial static analysis tools offer assistance, they are generally limited in capability. In the absence of analysis tools, design patterns have been established that provide specific interfaces for ensuring explicit initialization. However, while initialization may be complete in the sense that all data is explicitly initialized, it does not necessarily mean that the initialization is proper in a logical context. This paper considers one particular initialization concern related to dependencies in the order of software initialization execution and presents a workable solution in the context of a structured programming language.

Note: The examples in this paper assume a structured programming language and C was chosen. Initialization issues in the context of object oriented languages are out of scope.

COTS – Commercial Off The Shelf

SAT – Static Analysis Tool(s)

> Note: The use of the term **SAT** is for the purpose of this paper only and should not be confused with the acronym associated with static analysis and Boolean Satisfiability.

CodeSonar – A **COTS SAT** from Grammatech, Inc.

Custom SAT – Domain-specific methods and tools used to perform static analysis

Customized COTS SAT – A customized version of **COTS SAT** developed by a vendor for a specific set of requirements to work within a **Custom SAT** framework

False Positive – An identified issue that is not real, a false alarm

IOD – Initialization Order Dependencies

IDA – Initialization Dependencies Analysis

MISRA – Motor Industry Software Reliability Association

Module(s) – A group of related software procedures that support a particular design construct but not necessarily restricted to a single file, and in the context of **IDA**, an abstract assignment construct associated with an initializing procedure.

Baseline – Information or data gathered at the beginning of a period from which variations that subsequently develop are compared (MedicineNet.com)

Introduction

Most software engineers working in safety critical environments would probably agree that the importance of proper software initialization can't be overstated. Moreover, in the NASA Software Safety Guidebook [3], it makes the following comments in the context of software initialization:

"/hi:'zen-buhg/ (From Heisenberg's Uncertainty Principle in quantum physics) A bug that disappears or alters its behaviour when one attempts to probe or isolate it.

In C, nine out of ten heisenbugs result from uninitialized auto variables, fandango on core phenomena (especially lossage related to corruption of the malloc arena) or errors that smash the stack…. Uninitialized variables are the most common error in practically all programming languages.

Initialize global variables in a separate routine. This ensures that variables are properly set at warm reboot."

Furthermore, the "Review Guidelines on Software languages for Use in Nuclear Power Plant Safety Systems", referenced in [3], states the following:

"Initialize variables before use! Using uninitialized variables can cause anomalous behavior. Using uninitialized pointers can lead to exceptions or core dumps."

Software Initialization Concepts

In an attempt to assist software developers, **COTS SAT** invest in rule checkers that can detect when variables are not initialized before use. For those who write software using the C programming language, **MISRA** has established the following rule:

MISRA 9.1 [4] "All automatic variables shall have been assigned a value before being used."

Yet, whether **COTS SAT** enforce **MISRA** or proprietary rules, the analyses are generally limited in scope, i.e. automatic data. For the most part, an assumption is made that global data is initialized (in the case of the C language a default value of zero is assigned at compile time); in other words that software developers are either relying on inherent properties of a given software language or are following recommended design principles. One common design pattern for the initialization of variables with global scope before use is to construct a specific interface (or procedure) that is responsible for "initializing" a **module**. Typically, this type of procedure would be called or executed first, before executing any other code or using data in a given **module**.

At this point let's distinguish between what we will identify as proper initialization versus correct initialization. At a high level, we want to associate proper initialization with completeness (with a caveat to be explored later) rather than whether or not the value assigned at initialization is correct. Look at Figure 1 below. A **COTS SAT**

enforcing **MISRA** 9.1 would generate a warning that the variable "temperature" may not be initialized since its initialization is dependent upon control flow.

Figure 1

```
static void test_temp(void)
{
   int temperature;

   if (temperature_is_valid() == TRUE)
   {
     temperature = get_temperature();
   }

   if (temperature > MAX_TEMP)
   {
     temperature = MAX_TEMP;
   }
}
```

In Figure 2, we made a change. In this implementation, the variable "temperature" is now global in scope, and is no longer automatic data that exists only in the call frame context. Since "temperature" is initialized to zero at compile time (per the rules of the C language), it is considered initialized. However, the value assigned at compile time may not be correct. Determining correctness is beyond the scope of this discussion since it would be difficult for a **SAT** to be useful in this context. Even if an explicit value was assigned to "temperature", a **SAT** would not know whether the value assigned was correct without further input information.

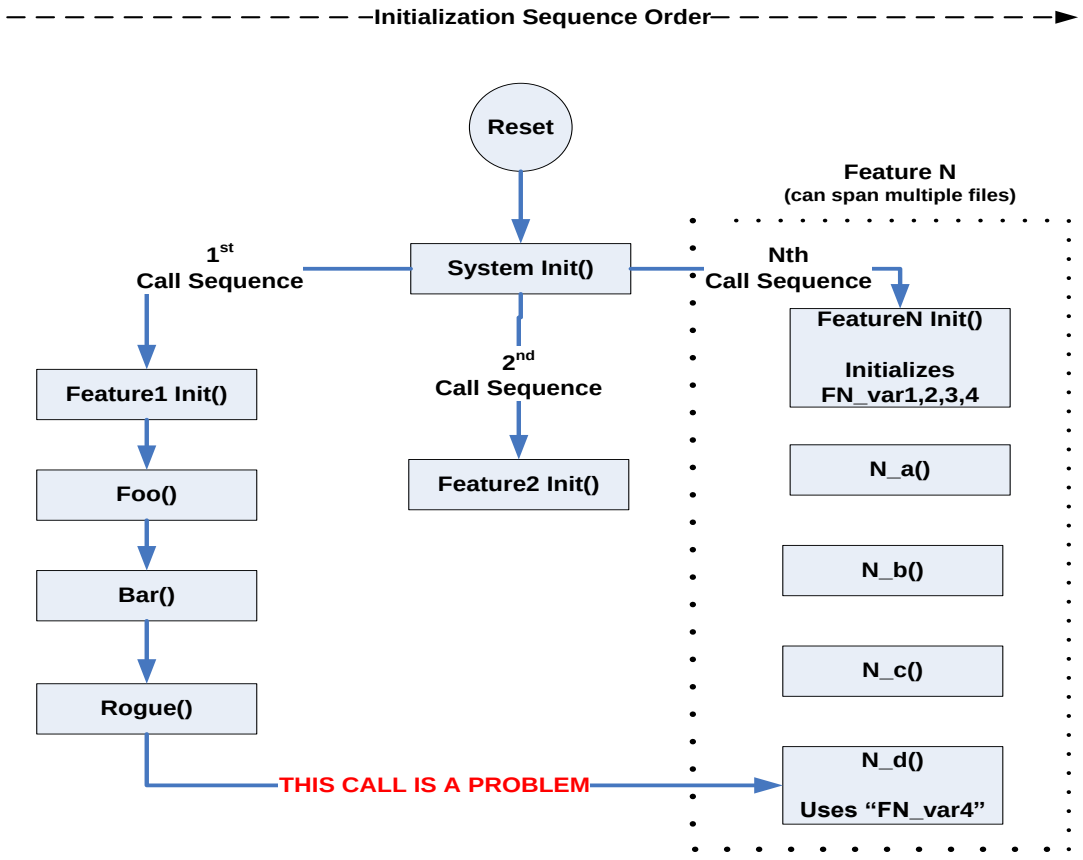
Figure 2

```
int temperature;

static void test_temp(void)
{
   if (temperature_is_valid())
   {
     temperature = get_temperature();
   }

   if (temperature > MAX_TEMP)
   {
     temperature = MAX_TEMP;
   }

}
```

A Deeper Look At Initialization

Now let's return to an earlier point. Does Figure 2 represent proper initialization? Perhaps it does, in this particular example. However, it would be wise to avoid drawing a hasty conclusion about proper initialization. Earlier we associated proper initialization with completeness, but with a caveat. Why? Well, even if one followed the aforementioned NASA recommendations, used the Nuclear Power Safety checklist, employed **COTS SAT** for localized initialization analysis, and addressed global data issues by either compile time or explicit initialization, there remains another initialization related concern that is much broader in scope and often a neglected topic; namely, **IOD** or initialization order dependencies. Completeness in this context does not necessarily mean proper. In the **IOD** model, improper initialization sequences can result in undefined behavior. As shown in Figure 3, let's assume a source code base with N features, where each feature can consist of multiple files and procedures. Assuming each feature is initialized before use as expected (which signifies completeness), how could one analytically determine if the initialization is proper with respect **IOD**? For the sake of simplicity, Figure 3 illustrates the problem in a relatively shallow call stack. In cases where the possible call paths of execution are both deep and wide, the level of analysis difficulty increases exponentially which makes this a good candidate for a **SAT.**

Figure 3



A Real World Solution

In the previous papers, [1] and [2], static analysis customization projects between Boston Scientific, Inc. and Grammatech, Inc. were discussed. The driving objective of those projects was to customize **CodeSonar**, Grammatech's flagship static analysis tool, to meet a very specific set of requirements based on our domain needs. Considering the success of those previous projects, we recently decided to tackle the **IOD** problem described in this paper and once again worked with Grammatech on a new **IDA** project. Whereas in **IOD** we introduced the concept of order dependencies, in **IDA** we identify that the real concerns involve use of data. Thus, calling a procedure "out of sequence", while perhaps not the best design, may not present a problem if there are no data usage concerns. Going back to our example in Figure 3, the Rogue() procedure presents a problem only if procedure N_d() uses data that is initialized by FeatureN_Init(), which may cause unexpected behavior.

The infrastructure needed to support **IDA,** as introduced in [2], includes an input Procedure Specifier file, which is a comprehensive list (CSV file) of all file:procedure pairs in the software tagged with various attributes. The new attribute of interest to support **IDA** is "Initializes Module", which designates the procedure within a file that is responsible for initializing a particular **module**. Additionally, a Variable Module Assignment List (CSV file) input is created to associate data items with **modules** and ultimately with the procedures responsible for their initialization. The choice to use a **module** name is deliberate in order to add a layer of abstraction and flexibility for assignments and associations if the need arises. The intent of **IDA** per our example in Figure 3, is to identify that procedure N_d() uses what we will now identify as global variable "FN_var4" (see Figures 5, 6). Hence, the call from procedure Rogue() is a violation because this global variable is dependent upon procedure FeatureN_Init() for initialization.

To further support **IDA** in the context of the example presented, the procedure System_Init() per Figure 3 is specified as the main initialization entry point or initialization root node. Starting there, **CodeSonar (**a **Customized COTS SAT** in this context**)** traverses the software initialization call sequences and call stacks tracking data uses. The legality of those data uses is dependent upon the order of initialization per the specified initialization entry point and the associations defined in the input CSV files (see Figures 4, 5, 6). In other words, are variable uses reachable from the initialization entry point without first traversing their corresponding initializers? The technical approach is an inter-procedural dataflow analysis, where the abstract state is the set of **modules** that have been initialized. **IDA** output is the set of procedures that occur on any call stack where a variable is used without being initialized. The asymmetry between "initializes" and "uses" is deliberate. **CodeSonar** automatically determines variable uses but the variable initialization associations are an input since this requires detailed design and implementation knowledge. (See Figure 7).

Figure 4: Initialization Entry Point Configuration

Entry Point FileName	Entry Point ProcedureName
Main.c	System_Init

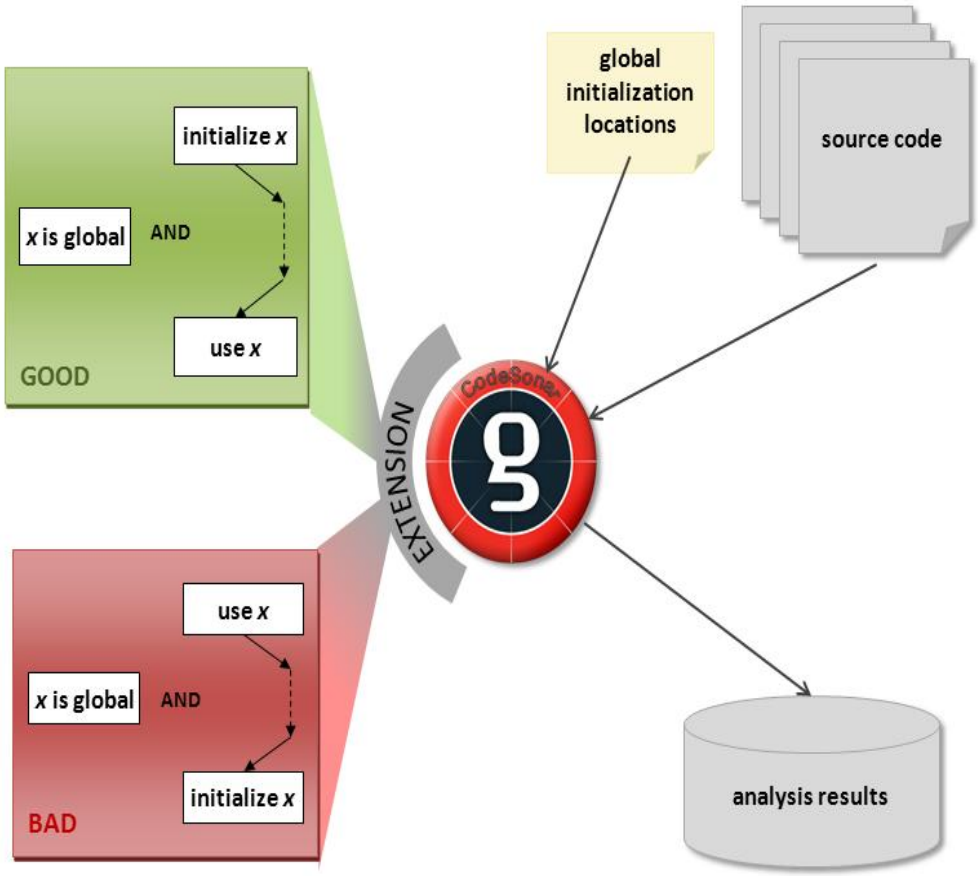
Figure 5: Procedure Specifier

FileName	ProcedureName	Initializes Module
Feature1.c	Feature1_Init	Feature1
Feature1.c	Foo	
Feature1.c	Bar	
Feature1.c	Rogue	
Feature2.c	Feature2_Init	Feature2
FeatureN.c	N_a	
FeatureN.c	N_b	
FeatureN.c	N_c	
FeatureN.c	N_d	
FeatureN.c	FeatureN_Init	FeatureN

Figure 6: Variable Module Assignment List

VariableName	ModuleName
F1_var1	Feature1
F1_var2	Feature1
F1_var3	Feature1
F1_var4	Feature1
F1_var5	Feature1
F2_var1	Feature2
F2_var2	Feature2
FN_var1	FeatureN
FN_var2	FeatureN
FN_var3	FeatureN
FN_var4	FeatureN

Figure 7 (Courtesy of Grammatech, Inc.)



It should be noted that the example presented in Figures 4, 5, 6 is deliberately simplistic. In practice, the tool infrastructure is flexible and accommodates multiple scenarios such as:

- Associating multiple "Initializes Module" references to a variable. In the following example, "Variable_X" is considered initialized only after File1_Init() and File2_Init() procedures have been executed.

FileName	ProcedureName	Initializes Module
File1.c	File1_Init	Measure
File2.c	File2_Init	Math

VariableName	ModuleName
Variable_X	Measure;Math

- Allowing one initialization interface to initialize more than one **module**. In the following example, "Variable_Y" and "Variable_Z" are both considered initialized after the File3_Init() procedure has been executed.

FileName	ProcedureName	Initializes Module
File3.c	File3_Init	Sensor;Actuator

VariableName	ModuleName
Variable_Y	Sensor
Variable_Z	Actuator

Other Tooling Considerations

As discussed in a previous paper [1], **COTS SAT** don't always provide analysis for the kinds of real world problems that we as engineers desire to solve. Therefore, for many, customized solutions become the only viable option. Customizing tools though is not a trivial pursuit and forces engineering tradeoffs with respect to **SAT** sophistication. Once again referencing Figure 3, let's assume the following C language implementation in the procedure Rogue(), designated as the source of our initialization problem.

Figure 8

```
Rogue (int Y)
{
  if (Y != 0)
  {
    N_d();
  }

  else
  {
    setup_data();
  }
}
```

Examining the implementation in Figure 8, if one knew for a certainty that Y==0 during the initialization sequence, then procedure N_d() would not be called. Hence, whether or not the **SAT** enunciates a warning is dependent upon tool capability. In order to simplify the analysis, one might only require the tool to identify all potential issues. However, this approach can generate false alarms**.** On the other hand, a more sophisticated tool might attempt to analyze data dependencies, thus potentially eliminating these types of alarms**.** As shown in our example code fragment above, identifying the Rogue() procedure call would be a **False Positive** if the Y==0 property was true during initialization. Ultimately, **SAT** complexity and the management of tool output is a design choice tradeoff that is often dependent upon realistic expectations of tool capability and what is considered acceptable for a given domain or environment. For instance, managing **False Positives** may be less of an issue if one has the capability of baselining existing warnings and the capability of identifying only subsequent warnings post-**Baseline**. In any case, embarking on the development of a **Custom SAT** to solve a problem such as **IOD** should not be taken lightly, especially if tool development is not your expertise. In the absence of **COTS SAT** capability what are the alternatives? One possibility is to work with a **SAT** vendor to develop a **Customized COTS SAT.**

Conclusion

In this paper we presented an analytical approach to identifying a specific class of software initialization issues related to dependencies in the order of software execution. As discussed, a customized solution was chosen primarily due to a lack of capability in **COTS SAT**. As expressed in previous papers [1] and [2], it is our desire that **COTS SAT** evolve to address real world software problems that are otherwise neglected or left to tedious and often less-effective manual approaches. In the interim, we hope to continue sharing our experiences from the medical device industry in order to expand the knowledge base and further the discussion of **SAT** capabilities.

References

[1] Gerald Rigdon. Static Analysis Considerations for Medical Device Firmware. July, 2010.

[2] Gerald Rigdon, Hiten Doshi, Xin Zheng. Static Analysis Considerations for Stack Usage. July, 2010.

[3] NASA. Software Safety Guidebook. NASA-GB-8719.13. March 31, 2004

[4] MISRA. Guidelines for the use of the C language in critical systems. MISRA C-2004