Static Analysis Considerations for Medical Device Firmware ESC-423

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

Abstract

Static analysis is an essential tool in the development of safety critical medical device firmware. For some developers, use of commercial off-the-shelf static analysis tools may be an adequate solution. For others, a different solution is necessary. This paper considers an approach that involves the use of customized static analysis tools that focus on domain-centered properties. Furthermore, it proposes a static analysis framework based on design, for which various approaches for usage can be employed. This framework can accommodate methods of execution ranging from manual techniques to automated software tooling. Moreover, due to the prevalence of commercial static analysis tools available for purchase off-the-shelf, this paper considers the practical use of such tools for medical devices in light of potentially more effective and beneficial customized solutions.

Consider: The Keywords

The acronyms, keywords, and expressions below are used several times throughout this paper.

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.

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

> Note: **Custom SAT** is not relegated to software only. The word "tool" is also used to convey a proposed and documented method that facilitates analysis.

Customized COTS SAT

- **(1)** A customized version of **COTS SAT** developed by a vendor for a specific set of requirements to work within a **Custom SAT** framework
- **(2)** Use of configurable or customizable features inherent in a **COTS SAT** version

CodeSonar – A **COTS SAT** from Grammatech, Inc.
QAC – A **COTS SAT** from Programming Research, Inc.

MISRA – Motor Industry Software Reliability Association
JPL – Jet Propulsion Laboratories

CSAP – **Customized COTS SAT (1)** Project between Boston Scientific, Inc. and Grammatech, Inc.

Project Objective - Customize **CodeSonar** to support the following:

- **SDA** – Shared Data Analysis (Usage of global data in pre-emptive multi-threaded design)
- **CTGA** – Call Tree and Graph Analysis (Visual and descriptive model based on unique domain properties)
- **SCCA** – Strongly Connected Components Analysis (Recursion identification)
- **CFUA** – Call Frame Usage Analysis (Maximum stack allocation for specified task threads)
- **PLEA** – Product Line Engineering Analysis (Generation of data to support analysis of building product firmware for multiple configurations)

Complement / Complementary – Serving to fill out or complete (Merriam-Webster)

Consider: The Popular Definition

We will begin our consideration of static analysis by consulting Wikipedia, the source of many popular definitions. A partial definition as of this writing is as follows:

"Static code analysis is the analysis of computer software that is performed without actually executing programs built from that software (analysis performed on executing programs is known as dynamic analysis). In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code. The term is usually applied to the analysis performed by an automated tool, with human analysis being called program understanding or program comprehension.

The sophistication of the analysis performed by tools varies from those that only consider the behavior of individual statements and declarations, to those that include the complete source code of a program in their analysis. Uses of the information obtained from the analysis vary from highlighting possible coding errors (e.g., the lint tool) to formal methods that mathematically prove properties about a given program (e.g., its behavior matches that of its specification).

It can be argued that software metrics and reverse engineering are forms of static analysis.

A growing commercial use of static analysis is in the verification of properties of software used in safety-critical computer systems and locating potentially vulnerable code."

Although one should tread these quickly available, popular, and ever-changing internet resources with a certain amount of trepidation, the words quoted above are technically sound. In addition, they add further points of discussion to the topic rather than just providing a bland definition. However, this subject matter cannot be completely explored and fully appreciated without considering some historical context and discussions of static analysis from classical sources. Two of these sources are books that were published in the previous century, which are discussed in the following section.

Consider: Safety Related Systems

In the 1995 book, *Safeware* [6], Nancy Leveson stated the following about static analysis:

"Static analysis evaluates the software without executing it. Instead, it examines a representation of the software. In some ways, static analysis is more complete than dynamic analysis, since general conclusions can be drawn and not just conclusions limited to the particular test cases that were selected. On the other hand, static analysis necessarily is limited to evaluating a representation of a behavior rather than examining the behavior itself."

In the 1996 book, *Safety-Critical Computer Systems* [8], Neil Storey stated:

"Static testing plays an important role in establishing the characteristics of the system over its entire operating range – a function that cannot be performed during dynamic methods because of the infinite number of tests that would be required."

These comments by Leveson and Storey were written a decade and a half ago at a time when the importance of safety critical software development was just beginning to receive greater attention, before the rise in popularity of **COTS SAT**. The upsurge in the popularity of this topic is now manifesting in recent titles such as *Diagnosing Medical Device Software Defects Using Static Analysis* [5], *Software Forensics Lab: Applying Rocket Science To Device Analysis* [9], *Adopting Static Analysis Tools* [3], and *How to Develop a Coding Standard for an Embedded Project* [1].

The information presented in these articles/papers comes from a variety of sources, including researchers and the FDA (Food and Drug Administration). In *Software Forensics Lab: Applying Rocket Science To Device Analysis* [9], Brian Fitzgerald from the FDA was quoted, saying *"We're hoping that by quietly talking about static analysis tools, by encouraging static tool vendors to contact medical device manufacturers, and by medical device manufacturers staying on top of their technology, that we can introduce this up-to-date vision that we have."*

This statement helps to clarify the intent of driving all of this information to the fore, and explains why many engineers in the medical device industry are being proactively contacted by these '*static tool vendors'* Fitzgerald alluded to in [9]. In addition, it is interesting to note that many of the articles written on static analysis topics are created by representatives of **COTS SAT** vendors, and ultimately by companies who want to sell their product. Having that said, consider the following: While the use of **COTS SAT** may work effectively in some environments, does that mean it is the best choice for static analysis activities in all software development environments?

In an attempt to answer that question, let us once again consider the books on safety critical systems mentioned in the outset. If evaluating a *'representation of a behavior'* [6] and *'establishing the characteristics of the system over its entire operating range'* [8] essentially capture the essence of what static analysis is supposed to be in the truest sense, can that be accomplished by using **COTS SAT**? Perhaps, depending on what such '*behavior'* and '*characteristics'* are understood to mean. However, applying a general-purpose analysis rule set typically comes at the expense of important quantitative factors. Two of these important quantitative factors, Recall and Precision, are described in *Diagnosing Medical Device Software Defects Using Static Analysis* [5], where the equations are given as:

Recall = TP / (TP + FN)
Precision = TP / (TP + FP)

Where,

TP = True Positive (defect found was real)
FP = False Positive (defect found was not real, a false alarm)
FN = False Negative (defect not found)

In the context of **COTS SAT**, these two factors relate to how well the tool can find real defects (AKA Recall), without leading you on an expedition to discover that some of the defects "discovered" are not actually defects at all (AKA Precision). Tools that are not precise are often described in colloquial terms as "noisy."

In a presentation given at a static analysis conference at the University of Minnesota in July of 2009, John Rushby [11] discussed Recall and Precision in the context of Soundness and Completeness where he essentially stated that:

Soundness = a measure of the ability to miss no errors (No False Negatives)
Completeness = a measure of the ability to produce no false alarms (No False Positives)

Therefore,

(TP + FN) is the number of real errors or defects
(TP + FP) is the number of alarms

He also presented the following when discussing the use of tools to automate analyses:

Rice's Theorem says there are inherent limits on what can be accomplished by automated analysis of programs. In other words, you have five choices shown below, but at most you can only choose four of the five.

◦ *Sound (miss no errors)*
◦ *Complete (no false alarms)*
◦ *Automatic*
◦ *Allow arbitrary (unbounded) memory structures*
◦ *Final results*

Even though these quantitative factors, such as Recall and Precision, or Soundness and Completeness, are important and ultimately require compromise to achieve an acceptable balance, they are still limited to the overall static analysis capabilities of the tool itself. Moreover, since not all **COTS SAT** are equal, making sense of such quantitative factors can be challenging. Therefore, considering these inefficiencies and limitations on '*automated analysis of programs,'* is it wise to simply accept that **COTS SAT** is the most effective approach for static analysis?

We certainly cannot speak for Leveson [6] or Storey [8], but it would be a stretch to conclude that they were thinking of automated **COTS SAT** when discussing the topic of static analysis all those years ago in the context of safety critical systems. Furthermore, we are not attempting to stereotype software developers by suggesting that all developers equate static analysis with the use of **COTS SAT**. Finally, we are not dismissing the usefulness of **COTS SAT**, or even the exclusive use of them in some software development environments.

When examining the topic of safety related systems, there are at least three major categories that emerge for consideration:

Category 1. **COTS SAT**
Category 2. **Customized COTS SAT**
Category 3. **Custom SAT**

These categories will be referenced in the following sections.

Consider: The Standards and Rule-Based Approach

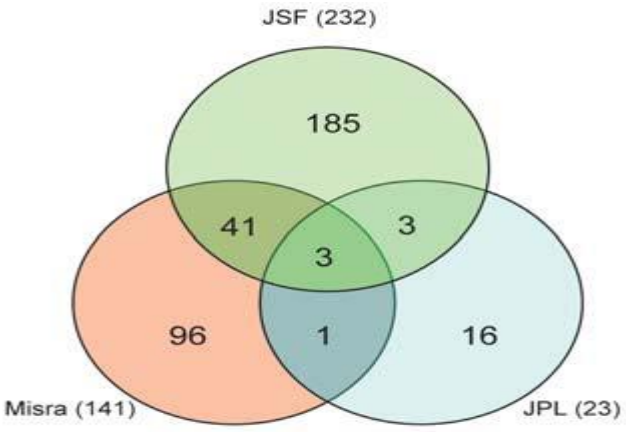
To state categorically that acceptance of **COTS SAT** rule sets should be embraced or that compliance with computer language based coding standards is not debatable would be less than accurate. An honest assessment finds that there are legitimate differing views. A good example of such a rule set is the **MISRA** standard for the development of software based on the C programming language. The value of complete and strict **MISRA** compliance is often subjective and dependent upon internal processes at work within

diverse development environments. However, it is not the purpose or intent of this paper to delve into the **MISRA** debate or the practical use of this standard. Rather, we simply state that some development environments adopt this standard and that some **COTS SAT** are used to enforce it.

As for the evidence backing such decisions on **MISRA** compliance, we can only point to information that facilitates a healthy debate. One such paper, *Assessing the Value of Coding Standards* [2], was a study of the empirical evidence related to the intuition that coding standards prevent the introduction of faults in software. You are encouraged to read the paper and draw your own conclusions, but we will take the liberty of quoting some of the concluding remarks of the study itself, which state that it is '*likely that adherence to the **MISRA** standard as a whole would have made the software less reliable. This observation is consistent with Hatton's earlier assessment of the **MISRA** C 2004 standard [10]'.*

While some **COTS SAT** support **MISRA** as part of their feature set, others take a different approach. An example can be found in *How to develop a coding standard for an embedded project* [1]. In this paper, Paul Anderson at Grammatech discusses the power of their **COTS SAT** by demonstrating how their **CodeSonar** static analysis tool (in addition to their default rule set) also supports rules that are mostly non-intersecting, or **complementary** (represented in Figure1 below as **JPL**) to other standards like **MISRA** for C or JSF (which is a standard for C++ environments).

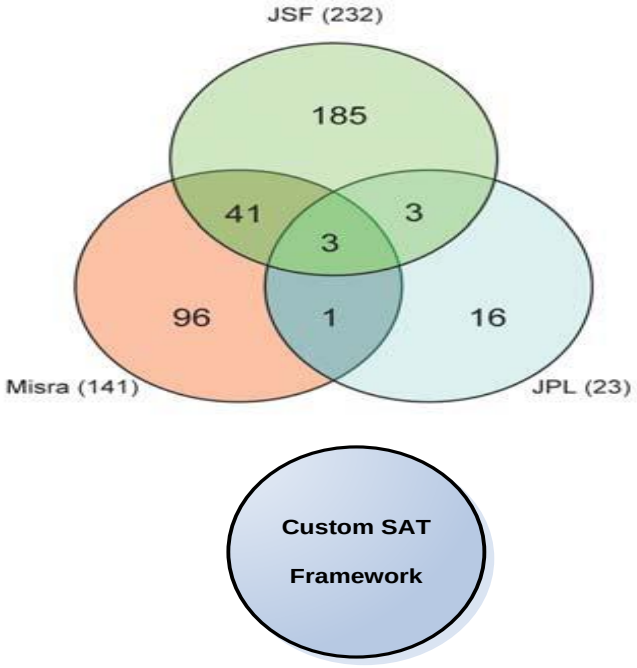
[Figure1] – Courtesy of Anderson [1]



This particular article then presents a balanced conclusion by stating that standards should be used as part of a systematic approach that includes '*testing, design reviews, manual code inspection, or other standard software development techniques'.*

Consistent with the focus of the paper from Anderson [1], we would like to add to his conclusion such that '*other* [non-standard] *software development techniques'* are considered, especially those related to custom static analysis activities. Thus, our adaptation of the diagram from [1] would be as follows:

[Figure 2] Adapted from Anderson



As shown in this adapted drawing, the power of the **Custom SAT** Framework bubble is evident in the fact that it stands alone. While in reality there may be some overlap with existing "standards," it is nonetheless custom and domain-centric. Thus, any potential areas of redundancy would vary from framework to framework since each would be different, possessing unique domain-specific properties. While the standards or rule-based approach can be useful, we propose that a custom framework based on design is more effective. Let us consider that next.

Consider: A Custom Static Analysis Approach Driven by Design

Leveson, in chapter seven of *Safeware* [6], discusses the principles surrounding design constraints. We will not attempt to elaborate on that topic in the context of Leveson's profound work. However, we propose that one can leverage those concepts in building a framework for **Custom SAT**. Within any good safety critical development environment are the standard practices Anderson [1] alluded to earlier, like '*testing, design reviews, code inspection'.* Additional standard practices also include the use of requirements, which start at the system level and trace down into software implementation. While such traceability is important, there are many different approaches one can take when designing a software system that effectively fulfills its requirements. Therefore, at a high-level of design, one should consider not only the requirements, but also high-level design criteria.

As the design becomes more specific, at mid-level (so to speak), is where effective design constraints can be placed. Such design constraints can enforce effective low-level

detailed design and implementation, and further used as a framework for **Custom SAT** to ensure that the detailed design not only meets the requirements, but also does so within the constraints of the system. An example of such a constraint could be as follows:

Design Constraint XX: *"All data allocated to memory section 'data warm' shall be preserved across warm resets."*

Therefore, even though designers may be given some freedom to express their creativity in designs that fulfill the requirements, they must do so given the constraints that are imposed. In this particular case, any design making use of data that needs to survive warm resets would need to be placed in the proper *'data warm'* memory section.

The design constraint can be enforced not only through effective design reviews and code inspections, but additionally through a formal static analysis activity. The design constraint becomes, in effect, the objective for this static analysis activity, which includes a formally documented method to produce the data that in turn must be analyzed in order to determine if the constraint has been violated.

Referencing the components in Figure 2 once again, it is apparent that adopting and then integrating static analysis requires resources, and so comes at a price. However, it would be shallow to think about this only in terms of capital expenditures. Let us now visit the topic of investing in static analysis and see if we can embrace a strategy and a vision.

Consider: The Investments

In *The Business Value of Software Static Analysis* [7], Rotibi states:

"Static analysis at the code level will only get you so far. In practice there are often more errors or defects in earlier stages of the software development lifecycle: i.e. in the software requirements gathering and design stages. As this report has already shown, errors and defects early on in the process will almost certainly have a negative impact on the final quality and the ongoing costs.

The future for the ability to do code reviews and static analysis at many different levels is coming. Understanding whether the design is right or whether it is conceptually flawed is the basis of forensic analysis, which is a practice that is growing in importance and prominence. Static analysis is a precursor to forensic analysis: you cannot conceivably do forensic analysis well without strong static analysis tools and procedures.

As forensic analysis becomes more well-understood and practiced, the most beneficial analysis tools will be those which can adapt to the broadening context of reviews and static analysis: tools which can integrate and interoperate with existing tool investments but enable a consistent approach to applying static analysis and review procedures wherever they may be required. This kind of adaptability and integration capability will enable an environment where knowledge and skills gained once can be applied in multiple places and scenarios."

When one considers the economics of software development, some interesting questions arise. How much does your company presently spend on dynamic testing versus static testing? If your company invests in static analysis tooling, is there an inclination to purchase these tools off-the-shelf instead of investing in the development of **Custom SAT**? Should we assume that the future of effective static analysis is coupled with the use of **COTS SAT**?

These questions are particularly pertinent when one considers the enormous investments often made in dynamic test environments. While **COTS** software tools may facilitate the framework for dynamic testing, the actual test cases are very domain-specific since the purpose of testing is to ensure that the software fulfills its requirements. Any product that includes software as a major component has a very specific set of requirements, and therefore a specific set of test cases. Consequently, another important question emerges. Is there perhaps a widely accepted false assumption that dynamic testing necessitates custom development while static testing tools are better purchased off-the-shelf?

If such an assumption does exist, it would be conceivably difficult to understand its rationale. Given the fact that there is no general-purpose solution for dynamic testing, why would one expect there to be such a solution for static analysis? These expectations can be realized, but only at the expense of constraining static analysis itself to a generic framework which would be more focused on implementation construction rather than design. Considering the complexity of software, making these kinds of assumptions or having these types of expectations would be cause for concern.

In support of Rotibi's [7] remarks above Leveson [6] concludes that in some ways, static analysis is more complete than dynamic analysis, since '*general conclusions can be drawn and not just conclusions limited to the particular dynamic test cases that are selected* '. Furthermore, as Storey [8] said, characterizing one's software system by static analysis is '*a function that cannot be performed during dynamic methods because of the infinite number of tests that would be required'.* Therefore, the business case seems valid in general and is supported by good engineering observations. However, what does that really mean in actual practice? Can such effectiveness and completeness be achieved by simply purchasing **COTS SAT**?

Maybe as Rotibi [7] indicates, forensic analysis will some day be well understood and mature. In the meantime, what should we do? Perhaps the hesitancy for some, regarding investment in these types of activities and tools, lie in Rotibi's [7] own remarks that *'the future for the ability to do code reviews and static analysis at many different levels is coming'.* So, given this statement about one possible future, we now add further questions for consideration: Should we continue to invest heavily in dynamic testing at the present time and wait for the future of static analysis to come? Should we go ahead and purchase the **COTS SAT** that are now available and begin the process of integrating these kinds of tools and practices in our development processes? Should we invest in the development of **Custom SAT** that are more commensurate with dynamic testing, and use **COTS SAT** in a **complementary** role?

Actually, that final question is the perfect segue into the recommendation of this paper. Our position is very similar to the conclusion reached in [5] which states: *"Finally, it should be noted that although static analysis offers a number of benefits to medical device developers, it may not address all of the needs or development concerns a manufacturer has regarding code quality. Manufacturers still need to investigate other development tools that can provide a complete tool chain for developers to use at different stages of the software development life cycle. These include architectural analysis, dynamic analysis, and software readiness analysis, among other technologies. Static analysis is most effective when used in combination with such development analysis tools and traditional V&V techniques. It must be viewed as a complement to, rather than a replacement for, traditional methodologies."*

We agree that static analysis is **complementary** with other methodologies. However, this paper actually focuses on the static analysis activity itself and further amplifies this theme as follows:

It is the central theme of this paper that Custom SAT should be given primary consideration within medical device software development, and that COTS SAT should be a secondary consideration with respect to a complementary role in development.

Related to our position, Rotibi [7] also states:

"Tools in general have come a long way in helping us to achieve more reliable software. But purchasing a static analysis tool alone will not guarantee software code quality.

As with any strategy that looks to manage or improve the delivery of software code and applications, the focus should fall on a number of common core areas: people, process, methods, tools and technology."

We propose that focusing on creative solutions within our own custom and unique development environments will not only result in more effective static analysis for our software, but may even help to further the cause of maturity in the static analysis domain itself. Recently, we undertook a project aimed at automating some of our own **Custom SAT** activities for the firmware used in our Cardiac Rhythm Management implantable products. Of the numerous static analyses we presently execute within our software development environment, we focused on automating those that were manually intensive and were the most important with respect to reliability and repeatability.

Therefore, instead of waiting for Rotibi's [7] '*future'* to come, we recommend active participation in making it happen. Moreover, we feel strongly that making the business case within our own work environment can be undertaken by creating a development framework that supports static analysis, and then demonstrating the value. We propose, as did Leveson [6], that in some instances static analysis can be just as or even '*more complete,'* and hence more effective than dynamic testing. When this framework is an integral part of your own unique design, then it becomes easier as Rotibi [7] says to understand *'whether the design is right or whether it is conceptually flawed'*, which should be the foundation of forensic analysis.

Perhaps the discussion of a real-world project can assist in giving some meaning to these words. As we have recommended, our particular firmware development process does indeed include a static analysis framework driven by design constraints. In our particular case, we have many design constraints that ultimately drive numerous formal static analysis activities, which include methods of execution and analysis of results. Each of these analyses is custom. In other words, they cannot be completed by purchasing **COTS SAT**. We do include the latter tools in our process, but they are **complementary** to our primary **Custom SAT**.

Consider: A Real Project

Although there were multiple objectives in the **CSAP** (namely **SDA, CTGA, SCCA, CFUA, and PLEA)**, we are going to focus on the project objective that was the early primary initiative, the **SDA,** which centers on the following design constraint and static analysis activity that enforces this constraint:

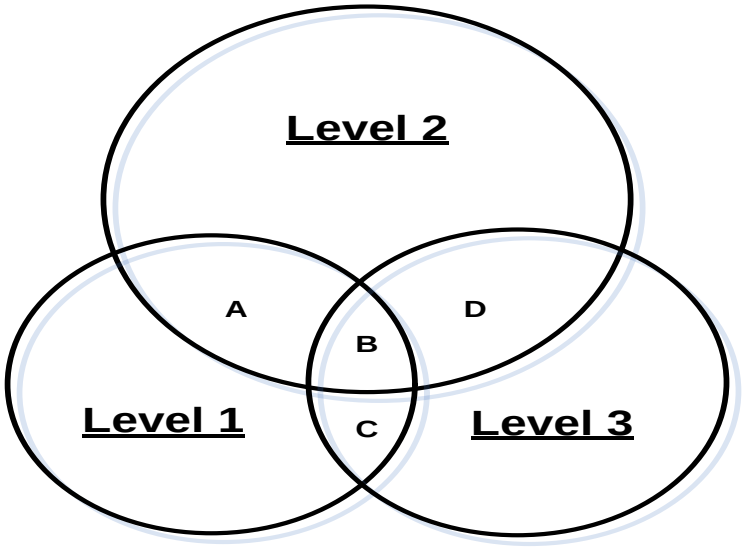
Design Constraint YY: *"All data instances that are accessed from multiple priority levels of threads must be protected with proper guard constructs."*

In practice, enforcement of this constraint required a manual analysis that generally took about a person month to execute. The work on automating this analysis first began in the summer of 2008 with an intern project. This project included the use of a particular **COTS SAT** that claimed to support shared data analysis. However, after a summer of effort, we concluded that the tool was inadequate for the task.

What we learned from the project is that the core static analysis engine of the tool we were using was powerful and that this particular analysis could be effective given some development work and a clear set of tool requirements. Unfortunately, that tool vendor was not interested in making the requested enhancements for us and the project was sidelined until the spring of 2009 when we began a relationship with a different vendor, Grammatech, whose vice president Anderson [1] is referenced earlier.

A visual aid to understanding the **SDA** problem statement is shown below:

[Figure 3]



For the purpose of understanding Figure 3, we will assume three levels of priority in a pre-emptive multi-tasking system, although theoretically this model can be extended to a Level N task priority model. In such a system, a task scheduler can interrupt and suspend a currently running task in order to run another task associated with a higher level of priority. The priority levels are defined as follows:

Level 1 – Low priority level tasks
Level 2 – Medium priority level tasks
Level 3 – High priority level tasks

For the purpose of this discussion, we will introduce the terms below.

Shared Data - Data that is accessed in different priority level tasks

Protected Data – ***Shared Data*** that is protected from access in different priority level tasks due to explicit use of ***Protection Mechanisms***

Unprotected Data – ***Shared Data*** that is not protected from access in different priority level tasks due to either absence or de-activation of ***Protection Mechanisms***

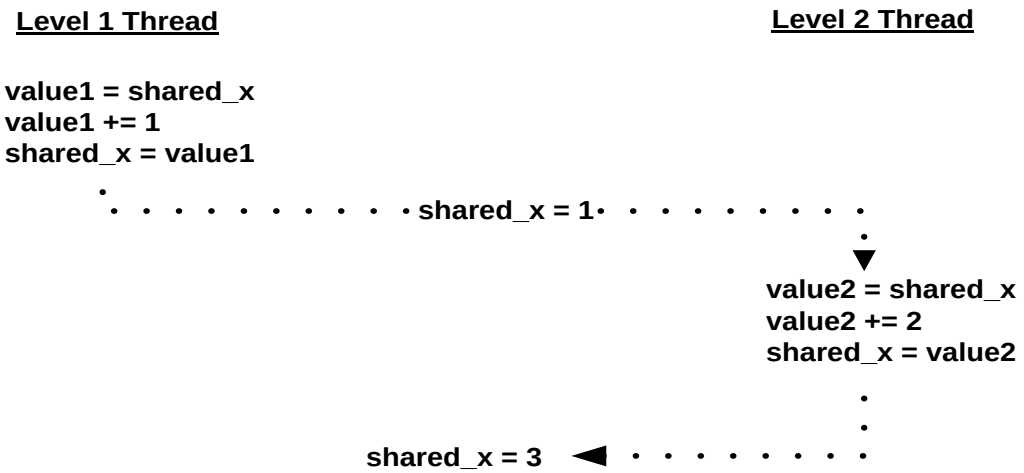
Protection Mechanism – A software implementation *'guard construct'* or design that protects data accessed in a lower priority level task from access in a higher priority level task due to task priority pre-emption

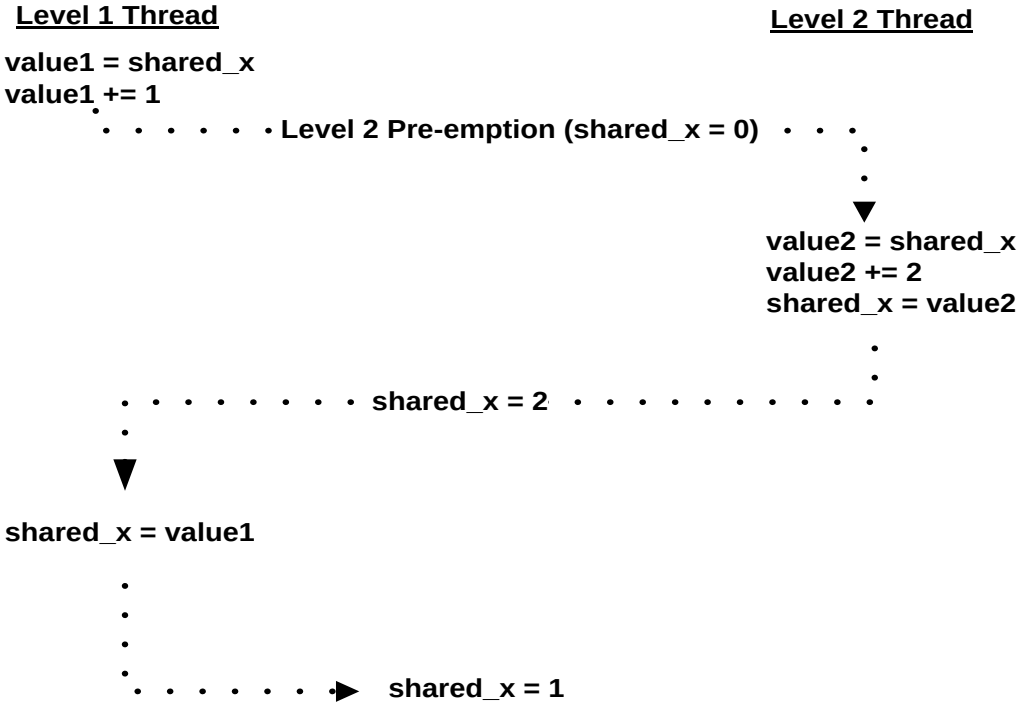
Now, referencing Figure 3 again, we provide the following analysis of the letters representing regions of intersection in each of the Priority Level Bubbles.

Region "A" – Data in this region is ***Shared Data*** between Level 1 and Level 2 tasks.
Region "B" – Data in this region is ***Shared Data*** between all Level 1, 2, and 3 tasks.

Region “C” – Data in this region is ***Shared Data*** between Level 1 and Level 3 tasks.
Region “D” – Data in this region is ***Shared Data*** between Level 2 and Level 3 tasks.

One way of looking at this problem is illustrated below. In this example, assume the variable “shared_x” is initialized to zero for each case.





The above example is deliberately simplistic and contrived in order to demonstrate how the final value of “shared_x” can be affected by pre-emption. Thus, in the context of a

pre-emptive, multi-tasking software design with thousands of variables, it does not take long to appreciate the value of Design Constraint YY referenced earlier, namely:

"All data instances that are accessed from multiple priority levels of threads must be protected with proper guard constructs."

Once again, returning to Leveson [6] and Storey [8], this would be an intractable problem of enforcement with only dynamic testing at one's disposal. Thus, this provides an excellent example where static analysis is obviously the best choice to enforce the design constraint since *'general conclusions can be drawn'* and *'characteristics of the system over its entire operating range'* can be established. However, how does one proceed? Would not a manual static analysis method to find such instances of data access violations be difficult or perhaps just as intractable as a dynamic testing model?

The answer to these questions depends much on the design and complexity of the software. However, given a complex system and the very nature of such a manually intensive analysis, one could expect Recall to be susceptible to False Negatives. In addition, such an analysis could also be susceptible to variability depending upon the skill of the engineer performing the analysis. Granted, a tedious method of execution could be created to improve reliability and repeatability, but not without a price in the form of the amount effort needed. This is exactly where we were when our project began. Static analysis was a better choice than dynamic testing and a time-consuming, manually intensive analysis was the only viable alternative, since we were unable to find **COTS SAT** that could be used effectively prior to **CSAP**.

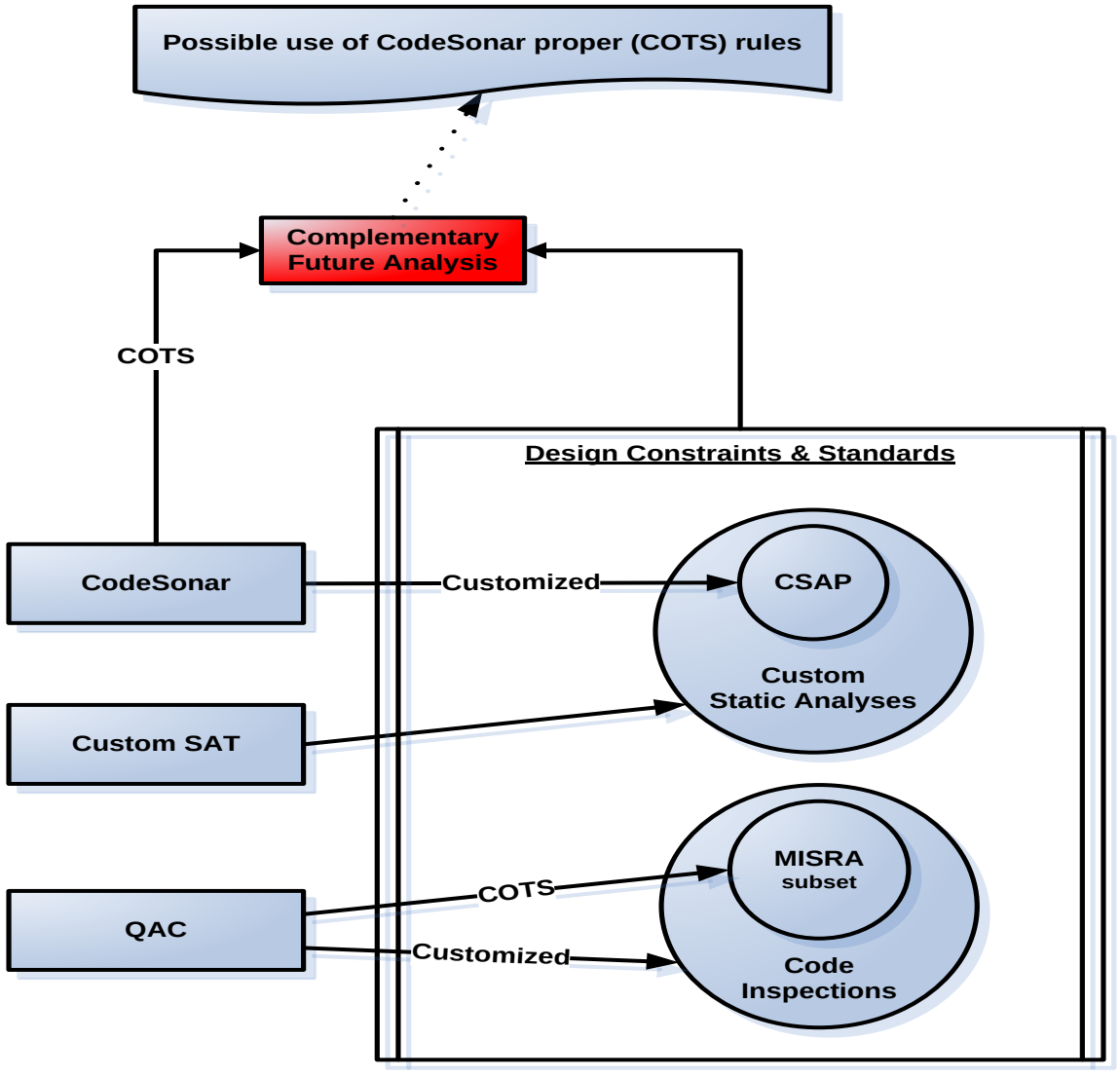
As this project progressed, we were very impressed by Grammatech's ability to deliver a **SAT** that met our particular needs. This tool is actually a customized extension to **CodeSonar.** In essence, it is a **Customized COTS SAT (1)**. The custom design revolves around the ability to create a modeling file, which prevents the need for modifications to the firmware. More specifically, special C language preprocessor definitions are used to override our existing implementation of '*proper guard constructs'* (AKA ***Protection Mechanisms***) per our design constraint, which redirects our task priority scheduler mechanism through the "modeled" extended file, allowing **CodeSonar** to track task thread execution with respect to Level N task priority information. The resultant output is a large flat file indicating ***Unprotected Data,*** with multiple attributes like task name, priority, effective priority, shared access indication, accessing file:func:line, shared variable name:type, etc. This approach to output allows us to import the information into a database and execute queries in order to manage the results more effectively.

Now, let us further consider a real static analysis framework and get a better understanding of how **CSAP** was integrated.

Consider: A Real Framework

Before introducing our static analysis framework represented in Figure 4 below, we would like to emphasize that this has been an evolutionary process. This framework is not the result of a master architect with a plan. Rather, it was forged over time and emerged from a relatively mature medical device software development environment. As discussed previously, **CSAP** is the latest added component. Since this framework for **SAT** is built upon design constraints, which are domain-centric, future use of **COTS SAT** in our software development environment would mostly likely be integrated like **CSAP**, with a focus on customization.

[Figure 4]



As shown in Figure 4 above, our medical device software process is built around our custom framework as follows:

- **Custom SAT** (used to enforce our domain-centered design constraints)
  - Software tools
  - Analysis methods

- **CodeSonar** (**CSAP** to enforce a subset of domain-centered design constraints)
  - A **Customized COTS SAT (1)** for **SDA, CTGA SCCA, CFUA, PLEA**

- **QAC** (used for code inspections)
  - **COTS SAT** to enforce a subset of **MISRA**
  - **Customized COTS SAT (2)** used to enforce internal coding standards

Thus, within our development processes, we actively employ **COTS SAT** to **complement** our design constraints framework. In our case, we enforce a subset of the **MISRA** rules by using **QAC**. Earlier, we referenced the empirical study of **MISRA** in [2], which concluded that enforcement of the entire standard as a whole would have made the software less reliable. However, the paper also states, "*selection of rules that are most likely to contribute to an increase in reliability maximizes the benefit of adherence while decreasing the necessary effort."*

Again, consistent with the position of this paper, we chose to use **MISRA** in a **complementary** way by selecting a subset of the standard. Thus, our choice in selecting **QAC** was made to enable the analysis of this subset, as well as the automatic analysis of our own coding conventions. Therefore, we have customized our usage of this tool to fit our environment and our static analysis process.

What about the use of **CodeSonar** proper, the default off-the-shelf rule set? Based on our positive experience from the **CSAP** with Grammatech and use of **CodeSonar**, we would not hesitate in making a positive recommendation. For less mature product development environments, **COTS SAT** like **CodeSonar** would most likely be a good fit and provide valuable analysis feedback. In our case, the benefits of general static analysis by using the off-the-shelf rule set in a **complementary** way still needs to be explored.

Furthermore, if one were to reconsider Figure 4 above in the context of **CodeSonar** usage in the conventional way of using the off-the-shelf rule set, there would be some overlap (we have not analyzed with any detail the full extent at this point in time) given the analyses we already perform within the static analysis framework of our design constraints. A good example of such overlap is the Division by Zero analysis. This is specifically discussed in [5] and is typically a useful function of **COTS SAT** like **CodeSonar**. However, we have a very specific targeted **Custom SAT** that can do a much more thorough job presently. In our custom Division by Zero analysis, we focus on achieving superior Recall, since we are able to parse the assembly language that is the result of compiling firmware source code and determine every case where our unique divide module is executed. The existence of this module allows us to isolate any potential Division by Zero cases in order to initiate the correct system safety response. In addition, this also allows us to analyze, within the context of static analysis, all functions that use

the divide module, by inspection of all input ranges of possible values. Furthermore, this range of values is documented within our interface specifications, which is not considered by **COTS SAT**, since they only analyze source code. Granted, it may be possible to input the data range information into **COTS SAT** for a more effective analysis, but such an endeavor would not be trivial and would reflect **Customized COTS SAT (2).** Ultimately, the analysis would be limited by the capabilities of the tool.

A preliminary analysis of **CodeSonar's** Division by Zero warnings generated against our code base indicates that the tool keeps False Positives to a minimum. It only generates warnings in cases where Division by Zero appears probable. Actually, one of the warnings it generated for our firmware was valid. However, since our custom static analysis has superior Recall, this particular Division by Zero case was already analyzed and well documented in the firmware. Moreover, this was a case where allowing the violation to remain was the best choice, given our overriding system safety response. Therefore, our design constraints and systems context trump the implementation rule. If one were to view this scenario in light of **JPL** rule 10, which we will elaborate on in the next section, this would be an example where a warning generated by a source code analyzer is well understood, but not eliminated. In summary, **CodeSonar** proves to be a useful implementation rule checker for this class of warnings, but does not provide a comprehensive Division by Zero analysis. In our design, this is important because an external device is allowed to program values into the embedded system, and some of these values are used in computations that can ultimately become denominators in arithmetic operations.

Use of **COTS SAT** can prove to be adequate in a **complementary** way, and we are looking forward to identifying the **CodeSonar** off-the-shelf features that we may potentially employ in such a manner in the future. However, we remain convinced that there is no substitute at present for our domain-centered design constraints enforced by **Custom SAT.** Although, as we have demonstrated by the **CSAP**, given the proper circumstances it is possible to work with a tool vendor to leverage their technology in order to help solve specific problems.

Consider: Some Recommendations and Ideas

Making recommendations should never be taken lightly, and so we will limit our recommendations to those that are particularly worthy of consideration. However, before doing so, we are going to revisit [2]. In this study, the conclusions about **MISRA** were reached because the study found a negative correlation between some **MISRA** rule violations and observed faults. Thus, inputting this into Adams' observation that all modifications have a non-zero probability of introducing a fault [12], this led to the conclusion in [2] that adherence to the **MISRA** standard, as a whole would have made the software less reliable.

Although we concede to be taking liberties by broadly applying the findings of that study to other areas of software static analysis, we propose that there is a lesson here. Making a

large number of changes to software based on the results of **COTS SAT** rule sets is probably not a good design pattern, especially if the goal is simply to achieve zero warnings. Additionally, it would not be an overstatement to say that many engineers who have years of experience with complex designs (such as medical devices) would generally conclude that most of the real defects in fielded products are related to either missing/incomplete requirements or unexpected behaviors from interactions between components in the software system.

In light of these findings, given a mature software development process with a more customized approach to static analysis, it would be a concern if such a process is required to respond to the results generated from **COTS SAT** by always changing the source code. Consistent with the position of this paper, the process should drive the selection and proper usage of tooling, not vice versa. Furthermore, in a mature software development environment, the number of warnings or alarms generated by **COTS SAT** is not necessarily indicative of the quality of the software. Thus, in the context of a **Custom SAT** framework, **COTS SAT** should be used to fill in the gaps or used in places where practical, in ways that are **complementary**.

Let us expand upon this topic by considering another rule set, the **JPL** Power of Ten [4]. Our highly skilled **CSAP** partners at Grammatech are fond of the Power of Ten, which is a feature set supported by **CodeSonar**. **JPL** rule 10 is often cited, stating:

"All code must be compiled, from the first day of development, with all compiler warnings enabled at the compiler's most pedantic setting. All code must compile with these setting without any warnings. All code must be checked daily with at least one, but preferably more than one, state-of-the-art static source code analyzer and should pass the analyses with zero warnings."

While we agree with the spirit of that rule, we propose that a variant is worthy of consideration. This proposed variant would embrace a larger context that is consistent with the major theme of this paper. Rule 10 is consonant with the other **JPL** rules, which are focused primarily on language driven implementation. Examples of such rules are: never use goto statements, impose upper bounds on loops, check return values of functions, use pointers carefully, keep sizes of functions small, use the C preprocessor carefully, etc. While these are good rules to follow, they should not be confused with domain-centered design constraints. Instead, consider this proposed variant of the second part of **JPL** rule 10:

All code must be analyzed on a regular basis by static analysis tools and methods that enforce design constraints and language driven implementation rules. All rule violations resulting from static code analyzers must be either eliminated or completely understood and documented.

The purpose of this proposition is not to create a new rule to be followed, but rather, to present another way of looking at static analysis. In this case, the proposed variant is meant to capture the original intent and expand it in order to embrace and emphasize

domain-centered design constraints. Instead of emphasizing only source code analyzers which are most often associated with **COTS SAT** and programming language driven rules**,** the emphasis is shifted to static analysis methods and tools in a broader context. Furthermore, the language of the proposed variant is less restrictive and rigid, which allows for more flexibility when static code analyzers are used.

If **COTS SAT** was part of the development process from the beginning, then a more Draconian enforcement policy (like '*zero warnings'* being mandatory) would be a more reasonable objective. However, this is usually not the case. Often, tools (like source code analyzers) are introduced to an existing mature code base and this likely real world situation should be accommodated, especially considering the earlier point that making changes to software based on **COTS SAT** warnings should not be automatically associated with improved quality.

Consider: Some Concluding Thoughts

The enforcement of rules by **COTS SAT**, whether driven by standards such as **MISRA**, **JPL**, or others, can be effective. However, each design and each software domain consists of unique properties that have to be analyzed. For domains that are safety critical (such as medical devices), we offer the following for consideration from *Safeware* [6]:

"Many of the software design features suggested in various standards are aimed at reliability, maintainability, readability and so on….When these qualities coincide with safety requirements in a particular system, they may make the software safer; when they conflict with safety or have little to do with the particular system hazards, they may have little effect on safety and may increase risk."

Furthermore, the epilogue in *Safeware* [6] is entitled "*The Way Forward."* In this section, Leveson highlights the major themes of the book with a series of bullet points, two of which we feel are appropriate to enforce our final comments.

- *Complacency is perhaps the most important risk factor in a system and a safety culture must be established that minimizes it.*

- *The safety of software can only be evaluated in the context of the system within which it operates. The safety of a piece of software cannot be evaluated by looking at the software alone.*

Each software design is different, and so are the problem sets. In the context of complex software design, static analysis is often the superior choice for some problem sets, given dynamic testing as the alternative. When this understanding is fully appreciated, the investment in static analysis needs to be realized. Such investment may include purchasing **COTS SAT**, which is most effective when used to **complement** an existing design focused static analysis framework that is customized to fit the needs of one's particular software domain.

Going forward, it may be possible to leverage the learning acquired from customization efforts in order to help build a bridge to Rotibi's [7] '*future'*. Perhaps these activities will ultimately lead to the integration of more powerful static analysis techniques into **COTS SAT**. At the very least, they demonstrate good engineering practices that emphasize focus on the most important design and domain-centered problems that can be discovered by static analysis tools.

Grammatech Acknowledgements:

- Curtis Bragdon for answering my telephone request in the affirmative and carrying the torch
- Dave Vitek for accepting the technical challenge and delivering an effective tool

Boston Scientific Acknowledgements:

- Byron Schneider for management support
- Xin Zheng for editing contributions
- Hiten Doshi as a fellow design collaborator and for helping build the original design framework that gave me something to write about

References

[1] Paul Anderson. How to develop a coding standard for an embedded project. Grammatech. June, 2008.

[2] Cathal Boogerd and Leon Moonen. Assessing the Value of Coding Standards: An Empirical Study. Report TUD-SERG-2008-017.

[3] Andrew Dallas. Adopting Static Analysis Tools. Full Spectrum Software. Published in MD&DI. August, 2008

[4] Gerard J. Holzmann. The Power of Ten – Rules for Developing Safety Critical Code. NASA/JPL Laboratory for Reliable Software.

[5] Raoul Jetley and Ben Chelf. Diagnosing Medical Device Software Defects Using Static Analysis. Coverity. Published in MD&DI. May, 2009

[6] Nancy Leveson. Safeware. Addison-Wesley, 1995

[7] Bola Rotibi. The Business Value of Software Static Analysis. Macehiter Ward-Dutton Limited. August, 2008.

[8] Neil Storey. Safety-Critical Computer Systems. Addison-Wesley, 1996

[9] Chloe Taft. CDRH Software Forensics Lab: Applying Rocket Science To Device Analysis. Published in Medical Devices Today. October 15, 2007.

[10] L. Hatton. Language subsetting in an industrial context: A comparison of MISRA C 1998 and MISRA C 2004. Information & Software Technology, 49(5):475–482, 2007.

[11] John Rushby. Introduction to Static Analysis for Assurance. Computer Science Laboratory, SRI International, 2009.

[12] E. N. Adams. Optimizing Preventive Service of Software Products. IBM J. of Research and Development, 28(1):2–14, 1984.