Architecture Reconstruction and Analysis of Medical Device Software

Dharmalingam Ganesan, Mikael Lindvall, Rance Cleaveland
Fraunhofer CESE, College Park, Maryland, USA
{dganesan, mlindvall, rcleaveland}@fc-md.umd.edu

Raoul Jetley, Paul Jones, Yi Zhang
FDA, Silver Spring, Maryland, USA
{raoul.jetley,paul.jones, yi.zhang2} @fda.hhs.gov

Abstract

New research is underway at the FDA to investigate the benefits of integrating architecture analysis into safety evaluations of medical-device software. Due to the complexity in setting up testing environments for such software, the FDA is unable to conduct large-scale safety testing; instead, it must rely on other techniques to build an argument for whether the software is safe or not. The architecture analysis approach, formalized using relational algebra, is based on reconstructing abstract, yet precise, architectural views from source code to help build such arguments about safety. This paper discusses the use of the formal approach to analyze the Computer-Assisted Resuscitation Algorithm (CARA) software, which controls an infusion pump designed to provide automated assistance for transfusing blood. The results suggest that a) architecture analysis offers many insights related to software quality in general and testability (i.e., the ease of testing) and its impact on safety in particular, and b) architectural analysis results can be used to help configure static analysis tools to improve their performance for verifying safety properties.

Keywords: Medical device safety, Testability, Verifiability, Static Analysis, Reverse Architecting.

1 Introduction

Embedded software in medical devices is increasing in content and complexity. For example, state-of-the-art cardiac pacemakers may contain up to 80,000 lines of code (LOC), while infusion pumps may have more than 170,000 LOC [5]. These devices must perform safely and effectively [14], and the US Food and Drug Administration (FDA) has the regulatory responsibility for making determinations about safety and effectiveness in the case for equipment sold in the United States. However, recent studies using the FDA database of medical device failures are pointing to increasing failure rates of medical devices due to software errors [17, 1]. In 1996, 10% of medical devices recalls were caused by software-related issues. In 2006, software errors in medical devices made up 21% of recalls [10]. From 2005 to 2009, more than 10,000 complaints were received annually by the FDA about infusion pumps, including reports of 710 patient deaths linked to problems with these devices [15]. A number of these deaths were attributed to malfunctioning device software.As software in medical devices has become ubiquitous [26, 27], it is not surprising to see a rise in the number of software-related problems. Increased complexity further contributes to the problem. What is disconcerting is the apparent lack of disciplined (safety critical) software engineering practices found during investigations of many of these problems.

The FDA has established a software laboratory within the Center for Devices and Radiological Health (CDRH), the Office of Science and Engineering Laboratories (OSEL), to evaluate software engineering technologies and to support investigations of potential software errors in medical devices. Due to the complexity in setting up test environments for medical device software, the FDA cannot in general test such software for safety and must rely on other techniques to build an argument for whether it is safe or not. As part of their investigations, analysts at the OSEL use state-of-the-art static analysis tools (SATs) to examine source code to understand and identify the root cause of device failure (e.g., [13, 16]). SATs verify the absence of runtime errors, such as division by zero, null pointer dereferencing, and buffer overruns, by evaluating the syntax of the code without executing it.

While sophisticated SATs can detect serious defects in software, they cannot uncover all safety problems in code. In particular, many safety-pertinent issues arise from design problems; an overly complex architecture, especially one that deviates from the documented design [29], for example, can result in inadequate testing and safety problems in the field. SATs are not intended to support these kinds of analysis. In this paper, we propose that safety analysis should include a detailed architecture analysis to help verify software more comprehensively. This analysis can help build an argument about safety based on statements such as a) software that does not have a well-engineered architecture may be unsafe and b) software that has low testability most likely has not been tested enough and therefore may be unsafe.

1.1 Detecting flawed architectures

All medical devices need to comply with quality regulations to ensure that they meet applicable current good manufacturing practices (CGMPs) [24]. In the case of software, this means ensuring that the best software-development techniques are employed during

978-0-7695-4351-2/11 $26.00 © 2011 IEEE
DOI 10.1109/WICSA.2011.33

IEEE computer society

implementation. One way to assess the quality of the software-development process is to reason about the architecture of the implemented source code (in contrast to the intended architecture that may be described in design documentation). For example, if the device has modular blocks, then this suggests that a) individual blocks can be independently unit tested [8], and b) the device can be formally verified for safety properties by focusing testing and verification activities on each modular block [3]. A system that is difficult to test and verify most likely has not been tested enough and therefore is less likely to perform as intended. Such software may be considered flawed, potentially unsafe, and should require more detailed and persuasive arguments about its fitness for it to be approved.

Most SATs cannot easily determine the quality of the software in terms of its architecture; they do not have the ability to tell whether the software being analyzed is well-structured. It falls upon the FDA analysts then to manually peruse software and its documentation to analyze and reason about its architecture. While device manufacturers do submit architectural design diagrams to the FDA, these are typically at a high level of abstraction. Their relationship to source code, testability and verifiability, and safety decisions is often tenuous due to missing details and traceability.

1.2 Approach

The Fraunhofer team has developed a large body of knowledge of architectural analysis of implemented systems (e.g., [6, 7, 8, 9]) in the safety-critical aerospace domain. The FDA believes that analysis methods proven in other safety-critical domains should be evaluated and, if appropriate, adopted in the medical-device domain [5]. New research is underway at the OSEL to perform architecture-level analysis of medical software by reconstructing architectural structures (e.g., [23, 2, 25]) from source code. The goal is to use such abstractions to draw conclusions about software quality properties that contribute to safety (e.g., testability and verifiability).

Using the CARA software [18], we formalize and demonstrate an architecture-reconstruction approach that extracts both static and runtime structures semi-automatically from source code to facilitate analysis of software safety. In order to recognize architectural features in the source code, the approach uses Fraunhofer's knowledge base (KB), including tool support, to analyze the CARA system from different architectural viewpoints, such as modularity, layering, inter-task communication, and built-in support for testing and verification. The CARA software was independently analyzed from two perspectives. First, an FDA analyst ran a SAT on the software. The tool reported that operating system (OS) library files needed to analyze the CARA were missing. Second, Fraunhofer analysts, working in the OSEL software laboratory, used their architecture reconstruction approach to establish architectural views of the CARA that show a) how the CARA architecture supports configuration points for running on different types of OSes, and b) how the source code files are transformed into runtime components. These insights helped the FDA analyst to configure the SAT in order to overcome the missing files issue and perform unit-by-unit software verification for improved performance.

Contributions. The paper contributes the following:

1. An improved understanding of how architectural analysis of software implementations can offer additional insights on software quality, especially testability and verification risks, that are not fully derivable from state-of-the-art SATs.
2. An architecture reconstruction, or "reverse architecting" process and algebraic formalization of our approach for the analysis of static and runtime structures from software implementations.
3. A case study demonstrating the use of extracted architectural knowledge to configure static analysis tools for improved performance.

While other architecture reconstruction techniques exist (e.g., [19, 20, 21, and 22]), to our knowledge, this case study is the first to make use of the extracted static and runtime structures to reason about software testability at the architecture level, and use this information to configure static analysis tools for improved analysis.

2 Reverse Architecting of the CARA

Overview of the CARA: The CARA is a control-loop software system implemented in C/C++ that infuses resuscitation fluids in a patient to maintain the blood pressure. It displays a graph showing the subject's blood pressure and the infusion pump's flow rate. In the autocontrol mode, a graph indicates the setpoint to which the CARA maintains the blood pressure. In the manual mode, the CARA monitors the speed and the volume pumped.

The architecture analysis process used to analyze the CARA has five major steps, see Figure 1. We give an overview of these steps and then exemplify them in Sections 3 and 4.

Step 1 - Clean the codebase: The goal of this step is to identify and filter out noise in the codebase. The codebase often contains duplicate files, which is confusing both to parsers and analysts because of duplicate definition of code elements. Analysts at Fraunhofer have developed an indexing tool that identifies duplicate files based on information retrieval technologies. The Fraunhofer analyst applied the tool on the CARA's codebase and automatically detected duplicate files, e.g., temporary file copies. The analyst excluded all but one copy of each set of duplicate files

because the other copies were redundant. The analyst rebuilt the index after excluding duplicate files because the next steps require a clean index. The output of this step is a cleaned version of the input codebase for next steps.

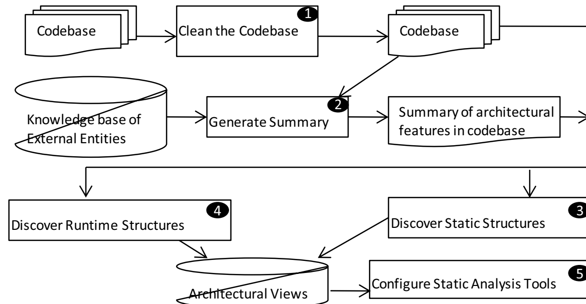

Figure 1: Reverse Architecting Process.

Step 2 - Generate summary: The goal of this step is to identify architecturally significant features present in the source code. Fraunhofer analysts have analyzed more than two-dozen real-world industrial implemented systems [9] for the past ten years or so. These engagements enabled us to build a KB of keywords present in external entities, such as programming language libraries, OS libraries, and commercial off-the-shelf (COTS) software. The KB stores keywords from header file names and function names that play significant roles in the discovery of the implemented architecture. For example, Table 2-1 shows a small snippet of the KB. If a system contains `vxWorks.h` and `taskLib.h`, which handles tasks in the VxWorks OS, then it is reasonable to assume that the system makes use of several tasks and runs on VxWorks.

Table 2-1: Snippet of the knowledge base

OS Type	Header file	Function name	Purpose
VxWorks	taskLib.h	taskSpawn	Spawn a task
VxWorks	msgQLib.h	msgQCreate	Creating a message queue
VxWorks	msqQLib.h	msgQSend	Send a message to a queue
VxWorks	msqQLib.h	msqQReceive	Read a message from a queue
Windows	windows.h	CreateThread	Creates a thread to execute

We have developed a summary generator that takes as inputs the KB and the codebase and outputs a summary of the codebase for further analysis. The analyst applied it to the CARA codebase resulting in a summary similar to Figure 2. The summary provides insights about the implemented architecture, including a) it supports two OSes, b) it uses multiple tasks, and c) it uses message queues for inter-task communication.

Step 3 - Discover static structures from code: The goal of this step is to reverse architect static structures from the source code to facilitate testability analysis. Static structures show the organization of source code files (in modules) on the file system and the inter-dependencies of these modules. Static structures show if the modules are organized into layers and potential inter-dependencies. Static structures also indicate a) support of graphical user interfaces (GUI), b) interactions with external hardware (e.g., blood pressure monitor, infusion pumps), and c) use of OS features.

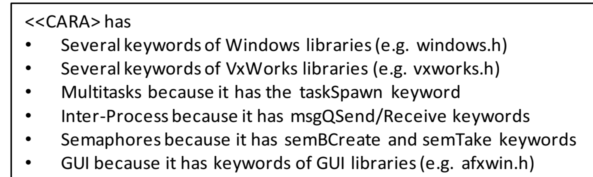
<<CARA> has
- Several keywords of Windows libraries (e.g. windows.h)
- Several keywords of VxWorks libraries (e.g. vxworks.h)
- Multitasks because it has the taskSpawn keyword
- Inter-Process because it has msgQSend/Receive keywords
- Semaphores because it has semBCreate and semTake keywords
- GUI because it has keywords of GUI libraries (e.g. afxwin.h)

Figure 2: Generated summary of the CARA using the KB.

The summary generator of the previous step reported that the CARA has a GUI and uses Windows as well as the real-time VxWorks operating systems. Using the output of the summary generator, in Section 3, we extracted the static structures of the CARA from the perspectives of GUI, OS, and hardware interaction to facilitate reasoning about testability. We have selected these perspectives because our experience indicates that testability is influenced by how the implementation abstracts certain components. If the implementation has hard bindings to GUI's, hardware elements, and real-time OS features, then it is typically difficult to test [8]. We used the Relation Partition Algebra (RPA) language [4, 21, 9] to discover static structures from the source code. We used the Fraunhofer SAVE-LIGHT tool [9] to visualize static structures. The output of this step is a set of static structures and insights about testability.

Step 4 - Discover runtime structures from code: The goal of this step is to reverse architect runtime structures from the source code. Runtime structures capture how the source code is partitioned into tasks and how the tasks communicate and synchronize with each other [23]. The analyst has to discover runtime structures because there could be a `many-to-many` relation between modules and runtime components (e.g., tasks). The analyst used runtime structures to a) understand how static structures are transformed into tasks and how the tasks communicate and synchronize, and b) reason about testability and verifiability of tasks. Often medical devices are implemented with multitasking capabilities. CARA is one such example, as reported by the summary generator. Based on the summary report, we formalize the extraction of runtime structures using RPA. The formulas presented in Section 4 automate the process of reverse architecting runtime structures using extracted code relations. We used the Prefuse interactive visualization tool [28] for visualizing runtime structures. The output of this step is a collection of runtime structures as well as insights about testability.

Step 5 - Configure and run static analysis tools: The goal of this step is to demonstrate how SATs can be

configured using the extracted static and runtime structures for improved performance, see Section 5.

3 Discovering Static Structures

The goal of this section is to reason about testability by reverse architecting static structures of the CARA.

3.1 Static structure of OS abstraction

The summary generator indicated that the CARA codebase has keywords related to both VxWorks and Windows libraries, so the analyst decided to investigate how the OS variants are handled in the implementation. The analyst analyzed the file `VxWorksSim.cpp` because it uses keywords of both OS types and found that the CARA can simulate VxWorks using Windows libraries. The analyst used a regular expression to extract `include` relations and found that files that include the standard `VxWorks.h` header file also include the `VxWorksSim.h` file, see Figure 3. The simulation capabilities are implemented in the file `VxWorkSim.cpp` and declared in `VxWorkSim.h`.

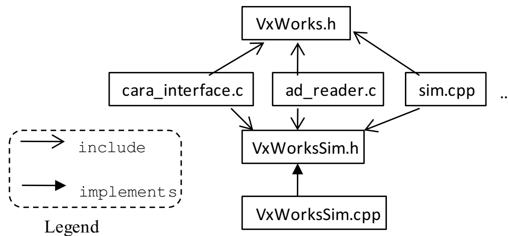

Figure 3: (Snippet) Static structure of OS variants. [1]

Since it makes little sense to include header files of more than one OS type, this caught the analyst's attention. He then analyzed some related files (e.g., `cara_interface.c` in Figure 3), and found that the CARA has a configuration point based on a conditional preprocessor statement: `ifdef WIN32` includes `VxWorksSim.h`, otherwise `vxWorks.h`.

Testability and OS variants. The CARA is a real-time embedded software system. However, this analysis of the OS variants showed us that the CARA can be configured to execute on machines with Windows, without requiring the VxWorks real-time features to be available. This OS dependency view highlights the fact that the CARA has built-in capability to facilitate testing. From FDA's point of view, this view is valuable because it conveys that the developers of the CARA most likely tested the software.

OS variants and an architectural issues. This snippet of the architectural analysis shows that the CARA architecture did not separate OS concerns from other concerns in a clean way. The knowledge of OS variants are scattered in different parts of the system. In some files, there are sixty `#ifdef WIN32` statements, for example. This extra complexity of handling OS variants with conditional preprocessor statements and simulation using Windows could have been avoided if there were an OS abstraction layer (OSAL) [12] built into the architecture, similar to the NASA CFS [6, 8]. The OSAL would facilitate adding and removing new OS types without changing the source code of higher-level layers because the OSAL would offer OS independent interfaces. From FDA's point of view, complexity can be a source of problems. If the source code is complex, then it is difficult to understand and test, and bugs could be lurking in the code. An overly complex architecture might even be considered flawed and unsafe. Developers should strive to minimize complexity by using appropriate architecture design principles, such as an OSAL.

[1] For readability, we manually redrew all figures.

3.2 Parsing the source code of the CARA

Based on the finding that the CARA makes use of mutually exclusive OS variants, the analyst ran the Fraunhofer's C/C++ parser to collect data for further analysis. The analyst configured the parser to choose Windows OS because VxWorks was not installed. The extracted data is listed in Table 3-1 and Table 3-2.

Table 3-1: Definition of extracted sets

Set	Comment
Function	List of functions
GV	List of global variables
Class	List of classes
Files	List of files
PS	List of preprocessor symbols

Table 3-2: Definition of extracted relations

Relation	Domain	Range	Comment
Call	Function	Function	Function f calls function g
FG_{use}	Function	GV	Function f refers to global variable v
Include	File	File	File m includes file g
Inherit	Class	Class	Class C inherits from Class P
Partof	Child	Parent	Function (child) is defined in file (parent)
F_{sl}	Function	Integer	Definiton of Function f starts at line n
F_{el}	Function	Integer	Definition of Function f ends at line n
MP_{use}	File	PS	File m uses a preprocessor symbol p

3.3 Static structure of hardware abstraction

The analyst visualized dependencies from the CARA files to *external files* using the Fraunhofer SAVE-LIGHT Tool. External files are those files which are used by the CARA but do not have a definition. The analyst found that the CARA uses an external header file called `dscud.h`. Because the KB did not contain the `dscud.h`, the analyst searched the Internet and found that it is related to device driver software called the Diamond Systems Corporation Universal Driver

(DSCUD). The DSCUD offers functions for conversions of analog input used in combination with analog input hardware boards that measure blood pressure.

From a testability point of view, the natural question is whether the CARA is vendor locked or whether it uses abstract interactions with hardware boards so that testers can test the CARA software on their regular computers without access to analog signals and hardware sensors. The analyst wrote a RPA query that helped extract a view of how the CARA interacts with the DSCUD. This revealed that the CARA uses eight C functions and four data types of DSCUD, accessed through a wrapper only. The wrapper is declared in `cara_da.h` and implemented in `cara_da.c`, which includes the COTS software header file `dscud.h`, see Figure 4. Thus, the implemented architecture is not vendor locked because of the wrapper and all dependencies to the external COTS software are only through the wrapper (`cara_da.c`).

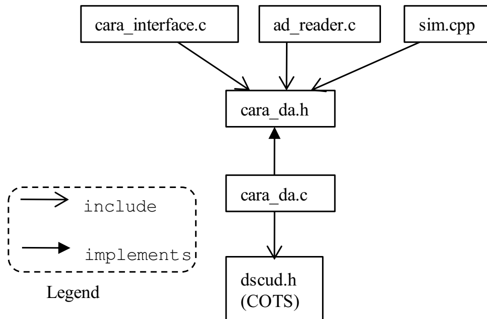

Figure 4: Wrapper view of DSCUD COTS abstraction.

The analyst queried the `call` relation to identify functions that use the wrapper functions of the DSCUD COTS, see Figure 5. The purpose of this query was to understand if it is possible to redirect the control flow to stub functions in order to avoid accessing the functions of the unavailable COTS software during unit testing.

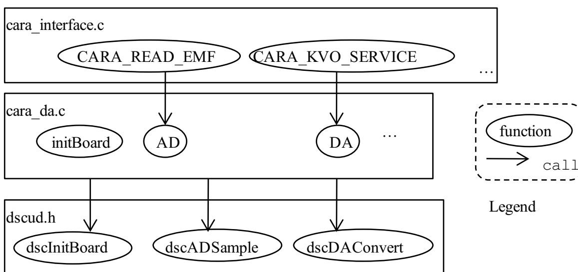

Figure 5: (snippet) Call graph of COTS interaction.

The code relations, as revealed by the query, do not contain stub or mock definitions of the wrapper functions of `cara_da.h`. Thus, the analyst analyzed higher-level layer functions (i.e., functions defined in `cara_interface.c`) that call the wrapper. Within `cara_interface.c` there is a conditional preprocessor statement called `#ifndef TEST` that decides whether or not to use the wrapper functions. If the `TEST` switch is disabled, the CARA will interact with the hardware board through the wrappers, otherwise a constant will be returned. For example, in Figure 6, the `AD` wrapper function of the hardware will only be used if the TEST is disabled.

```
double CARA_READ_EMF(void)
{
   #ifndef TEST
   /* Read the EMF value from the A/D board. */
   float Actual_Value;
   Actual_Value = AD(EMF_CHANNEL);
   ...
   return (double) Actual_Value;
   #else
   return CARA_EMF_VALUE;
   #endif
}
```

Figure 6: Avoiding hardware interaction for testing.

Using the extracted preprocessor symbol usage relation MP_{use}, the analyst confirmed that the `#ifndef TEST` statement is used in all places that interact with the hardware board using the intermediate wrappers (defined in `cara_da.h`). In general, over usage of IFDEFs can cause confusion and increase complexity.

3.4 Static structure of GUI abstraction

The source code files of the CARA were delivered as two folders. Using the KB, it became clear that the files in one folder deal with the GUI concern while the other folder deals with non-GUI concerns. By visualizing dependencies to external GUI libraries, the analyst concluded that the CARA implementation separates the GUI concern from core logic parts, which is a good architectural principle. This structure indicates that the CARA can be tested without the GUI, because testing in the presence of a GUI is difficult, if not impossible.

3.5 Insights of extracted static structures

The first insight is the indication that the developers of the CARA indeed tried to test the software without depending on necessary hardware-in-the-loop, see Figure 6. The second insight is that the testing feature was not designed into the software; it was inserted into the code afterwards. We can say this because of the extensive use of `#ifdef` to choose between the production and the test code. This could have been avoided by defining a stub implementation of wrapper functions. At link-time, developers/testers can then choose whether to bind to the real wrapper implementation or to the stub implementation, similar to the NASA CFS [8].

4 Discovering Runtime Structures

The goal of this section is to reason about testability by reverse architecting runtime structures of the CARA.

4.1 Data extraction for runtime structure

The summary generator reported that the CARA system uses keywords related to OS libraries that deal with multitasking and inter-task communication using message queues. The analyst extracted associated data to reverse architect runtime structures by developing regular expression based pattern matchers of function signatures of standard OS libraries. These pattern matchers emit data defined in Table 4-1 and Table 4-2. For example, in order to extract which function writes to which queues (i.e. the FQ_{write} relation), our pattern matchers emit a table (called `Send_Table`) that contains file names, queue identifiers, and the line numbers that match the syntax shown in Figure 7. The first argument to `msgQSend` is the queue identifier. The C/C++ parser emitted starting and ending line numbers of each function definition (see F_{sl} and F_{el} in Table 3-2). The analyst performed a join operation using the relations F_{sl} and F_{el} with the table `Send_Table` and derived the FQ_{write} relation.

```
msgQSend (msgQId, buffer, nBytes, timeout, priority)
```

Figure 7: VxWorks syntax to send messages on a queue.

Table 4-1: Definition of extracted sets

Set	Comment
Task	List of task names
Queue	List of queue identifers
SemId	List of semaphore identifers

In general, simple pattern matching would not be sufficient to extract this type of data because queue identifiers can be passed as arguments among different functions or be dynamically generated, where the actual value of queue identifiers cannot be retrieved until a comprehensive data-flow-based evaluation of the regular expression is conducted. However, in the CARA, all queue identifiers and semaphore identifiers are global constants.

Table 4-2: Definition of extracted relations from code

Relation	Domain	Range	Comment
FT_{create}	Function	Task	Function f creates task t
FT_{delete}	Function	Task	Function f deletes task t
$FT_{restart}$	Function	Task	Function f restarts task t
TF_{enter}	Task	Function	Function f is the entry point for task t
FQ_{create}	Function	Queue	Function f creates queue q
FQ_{delete}	Function	Queue	Function f deletes queue q
FQ_{read}	Function	Queue	Function f reads from queue q
FQ_{write}	Function	Queue	Function f writes to queue q
FS_{create}	Function	SemId	Function f creates semaphore s
FS_{delete}	Function	SemId	Function f deletes semaphore s
FS_{take}	Function	SemId	Function f takes semaphore s
FS_{give}	Function	SemId	Function f gives semaphore s

The analyst also found that there is no indirection of the function pointer (`entryFunPtr` in Figure 8) passed as an argument to the `taskSpawn` function for creating an OS task. In fact, the `taskName` and `entryFunPtr` are hard-coded strings in the CARA codebase. Therefore, our regular expressions based pattern matcher could extract the necessary data.

```
taskSpawn (taskName, priority, options, stackSize, entryFunPtr, …)
```

Figure 8: VxWorks syntax (snippet) to create a task.

4.2 Discovery of runtime structures

The analyst used the extracted relational data in Table 4-2 and derived architecturally significant relations by using RPA queries. These derived relations exposed the latent runtime structures of the CARA. Several runtime structures were immediately derived using RPA's relational and set operators, including the powerful transitive closure operator for reachability analysis. Using RPA queries, the analyst discovered several runtime structures including the following examples:

1. Which code elements are shared among tasks?
2. Which code elements are unique to tasks?
3. Which tasks create/delete/restart other tasks?
4. Which tasks read from/write to queues?
5. Which tasks create/take/give semaphores?

The automatically derived relations are listed in Table 4-3. Each derived relation refers to one concern of the runtime structures. Each derived relation can be visualized separately to get concern-specific insights. These relations were automatically extracted by running the corresponding queries formulated in Figure 9. The algebraic notations are explained in Figure 10.

Table 4-3: Automatically derived runtime structures

Relation	Domain	Range	Comment
TF_{use}	Task	Function	Task t uses function f
TG_{use}	Task	Variable	Task t uses variable g
TM_{enter}	Task	File	Task t entry point is file m
TM_{use}	Task	File	Task t uses file m
TT_{create}	Task	Task	Task t creates task u
TT_{delete}	Task	Task	Task t deletes task u
$TT_{restart}$	Task	Task	Task t restarts task u
TQ_{create}	Task	Queue	Task t creates queue q
TQ_{delete}	Task	Queue	Task t deletes queue q
TQ_{read}	Task	Queue	Task t reads from queue q
TQ_{write}	Task	Queue	Task t writes to queue q
TS_{create}	Task	SemId	Task t creates semaphore s
TS_{delete}	Task	SemId	Task t deletes semaphore s
TS_{take}	Task	SemId	Task t takes semaphore s
TS_{give}	Task	SemId	Task t gives semaphore s

For example, in order to extract which set of functions are needed to run a task one can run the query shown in equation (1) of Figure 9. The query works as

follows: The first part computes the transitive closure (*) after computing the union of extracted relations TF_{enter} and `Call`. The second part of the query restricts those tuples whose domain is in the `Task` set.

$$TF_{use} = (TF_{enter} \cup Call)^{*} \upharpoonleft_{domR} Task \quad (1)$$
$$TG_{use} = TF_{use} \circ FG_{use} \quad (2)$$
$$TM_{enter} = TF_{enter} \circ Partof \quad (3)$$
$$TM_{use} = (TF_{use} \cup TG_{use}) \circ Partof \quad (4)$$
$$TT_{create} = TF_{use} \circ FT_{create} \quad (5)$$
$$TT_{delete} = TF_{use} \circ FT_{delete} \quad (6)$$
$$TT_{restart} = TF_{use} \circ FT_{restart} \quad (7)$$
$$TQ_{create} = TF_{use} \circ FQ_{create} \quad (8)$$
$$TQ_{delete} = TF_{use} \circ FQ_{delete} \quad (9)$$
$$TQ_{read} = TF_{use} \circ FQ_{read} \quad (10)$$
$$TQ_{write} = TF_{use} \circ FQ_{write} \quad (11)$$
$$TS_{create} = TF_{use} \circ FS_{create} \quad (12)$$
$$TS_{delete} = TF_{use} \circ FS_{delete} \quad (13)$$
$$TS_{take} = TF_{use} \circ FS_{take} \quad (14)$$
$$TS_{give} = TF_{use} \circ FS_{give} \quad (15)$$

Figure 9: RPA queries for derived relations of Table 4-3 (please see Table 3-1and Table 3-2 for notations).

Notation	Definition
A^{*}	– Transitive closure of the relation A
$A \upharpoonleft_{domR} S$	– Restrict the domain of the relation A to the set S
$A \circ B$	– Composition of relations A and B
$A \cup B$	– Union of two relations/sets

Figure 10: Definition of RPA notations.

Structure of task creation. Storing the runtime structures as relations enabled the analyst to visualize them interactively. For example, the analyst loaded the TT_{create} relation into the Prefuse visualization tool and found that all tasks are created by the `CARA_Main` task. By querying the TT_{create} relation, the analyst found that the CARA has eleven tasks or runtime components. See Figure 11[2] for a snippet of the task creation structure.

Structure of common code and task-specific code. The analyst queried and visualized the derived relations TF_{use} and TG_{use} in order to identify code elements that are shared among tasks or used only by one task. Similarly, the analyst used the TM_{use} relation and visualized how source code files are shared among tasks or used only be one task. A snippet of how files are partitioned among tasks is shown in Figure 12. Such partition structures show which files are task specific and which ones are shared among tasks. These structures gave remarkable insights regarding how the static source code elements are transformed into runtime components.

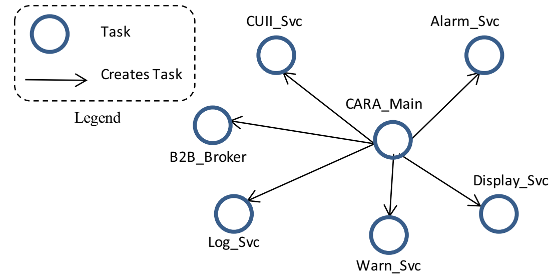

Figure 11: Snippet of task creation structure (i.e., TT_{create})

The analyst visualized the TM_{enter} relation and found that the entry functions of seven of the ten tasks that are started by the main task are defined in `cara_interface.c`. The entry function of the main task is defined in `cara_main.c`, and the entry functions for the remaining three tasks are defined in three different files, see Figure 13.

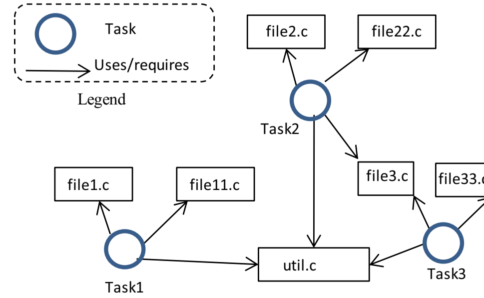

Figure 12: (Snippet) Partition of files among tasks.

Testability and partition of files among tasks. One consequence of defining seven entry points of seven tasks in one file (i.e. `cara_interface.c`, Figure 13) is that changes to code elements of this file require recompilation and retesting of all tasks using the newly compiled binary code. This time could be reduced if the entry functions as well as functions uniquely used by the entry functions are moved to separate files. Only the tasks that are affected due to recompilation need to be unit tested again.

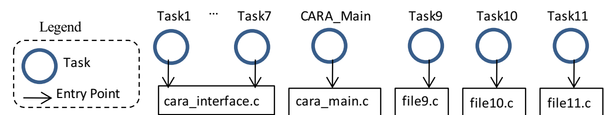

Figure 13: Tasks and their entry files of execution.

Because the current partition structure does not allow independent compilation of task-specific code from other tasks, it is impossible to produce a separate executable for each task to facilitate unit testing. This partition structure as such is not amenable to task-by-

[2] Names of the CARA artifacts are partly sanitized.

task verification of safety properties and difficult, if not impossible, to unit test and verify.

Structure of inter-task communication. After understanding how the source code elements are partitioned among tasks, the next step was to discover how the tasks communicate with each other. The analyst visualized the inter-task communication structure using the derived relations $\mathtt{TQ}_{\mathtt{read}}$ and $\mathtt{TQ}_{\mathtt{write}}$ that show which tasks read from queues and which tasks write to queues. The inter-task communication structure offered several insights including a) the CARA has five queues, b) some of the tasks, for example, `Alarm_Svc` in Figure 14, do not use queues to communicate with other tasks but instead use shared global variables, c) some of the tasks only read messages from queues and do not write messages to queues (e.g., `Log_Svc`), d) there is one central queue (i.e., `CARA_MSGQ`) used by several tasks, and e) there is no connector abstraction, i.e., all tasks are responsible for reading from / writing to queues using standard VxWorks APIs as well as handling low-level errors, such as unable to write to or read from a queue. Using queues to communicate between tasks results in a complex network, because a queue is a binary connector between two tasks. For the same message (e.g., shutdown) to be sent from a task to all other tasks would require a one-to-many queue connection. In the NASA CFS and GMSEC cases, a software bus (SB) is used to exchange messages between tasks by applying the publish-subscribe paradigm [6, 7]. The SB takes care of low-level communication and synchronization concerns. Thus, tasks can focus on computation instead of coordination.

Creation and deletion of Queues. The analyst visualized the derived relation $\mathtt{TQ}_{\mathtt{create}}$ and found that all queues are created by the `CARA_Main` task. All queue identifiers are global variables and therefore all tasks are able to use queues. The $\mathtt{TQ}_{\mathrm{delete}}$ relation showed that all tasks are deleted by the `CARA_Main` task.

Semaphores. By visualizing the derived relation $\mathtt{TS}_{\mathtt{create}}$ the analyst found that all semaphore identifiers are created by the `CARA_Main` task as global variables. By visualizing the $\mathtt{TS}_{\mathtt{take}}$ and $\mathtt{TS}_{\mathtt{give}}$ relations, the analyst found that the semaphore `give` and `take` concepts are scattered across all tasks. Such concepts that are needed for all tasks could be abstracted into a common set of services. Ideally, each task is not aware of the fact that semaphores are used for synchronization.

Testability and inter-task communication structure. The analyst used the discovered inter-task communication structure to assess the difficulty of testing these tasks. It is a fact that the higher the number of queues the more difficult it is to test the system. For example, if a task reads input messages from only one queue (e.g., `Log_Svc` of Figure 14), then a small test program can be developed to write messages on the other end of the queue. On the other hand, a task similar to the `CARA_Main` reads messages that are written by several tasks into the input queue. Thus, testing the `CARA_Main` is relatively difficult because several different types of messages have to be written into the input queue in order for the test to be realistic. Because the semaphore and the queue concepts are not abstracted, unit testing of each task would require initializing the semaphores and queues as well, even if only one task is running at a time during unit testing. In addition, some of the CARA tasks share global variables. Such communication structures require significant testing in order to make sure the state space of the global variables is properly understood by all tasks using them. Based on this analysis, the analyst concludes that the implemented architecture is not unit-test friendly, if not impossible to unit test.

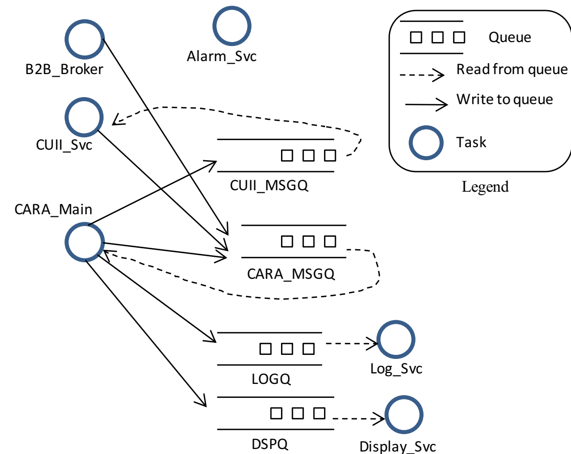

Figure 14: (Snippet) Inter-task communication structure.

5 Verification using Extracted Structures

The findings from reverse architecting of the CARA system were used to aid in static analysis of the source code. We used a commercial grade SAT to analyze the CARA, first using the default configuration, and then with annotations garnered through reverse architecting. In this section, we discuss how the discovered implemented architecture was used for: a) configuring the SAT, b) adapting the source code of the CARA for improved performance in terms of code coverage, and c) interpreting the warnings generated by the SAT.

Configuring the SAT using the extracted structures. An analyst at the FDA's OSEL laboratory independently ran the SAT on the CARA. The analyst had no prior knowledge of the implemented architecture of the CARA. The result was that the SAT reported that it cannot run due to missing VxWorks OS files. At this point, the results of the architectural analysis came to light. The CARA has a configuration point that allows it to compile and run under Windows. We used that

extracted architectural knowledge to configure the SAT and resolved the problem of missing files because the missing files were no longer needed under the selected configuration. Thus, the SAT was able to analyze the CARA using Windows as the OS configuration when compiling the source code. Work is also underway to setup the VxWorks environment for further analysis.

Using the extracted structures for adapting the source code. The SAT reported that the CARA has several dead functions that cannot be verified for safety properties. By visualizing the extracted runtime structures, it became clear that those dead functions are entry functions to different tasks. The entry functions are intended to be indirectly called by the OS's task spawning function. Analysis of the stub code of the task spawning function generated by the SAT revealed a surprising fact that the stub did not "understand" the meaning of the task spawning function and failed to activate all entry functions to tasks. As a result, entry functions were reported wrongly as dead code. In order to overcome this problem, the analyst defined his own stub implementation of the task spawning function that activated the entry function.

Because the CARA uses queues for communication among tasks, the analyst checked how the SAT handles functions related to queues. The SAT did not "understand" the meaning of OS functions used for sending messages to queues and receiving messages from queues. Thus, the SAT was not able to handle the indirection involved in sending/receiving messages. The analyst had to adapt the generated stub functions, making them closer to the reality.

Interpreting the warnings generated by the SAT using the extracted structures. The SAT reported that the CARA has a redundant predicate `if(!exiting)`. The implemented runtime structure, stored in the TG_{use} relation, showed that `exiting` is a global variable shared by all the eleven tasks, see Figure 15. This structure helped the analyst to refute the warning because it turned out that the SAT was not able to handle concurrency aspects well. Also, even if `exiting` was never non-zero, it is prudent to have redundant checks for safety reasons to ensure that critical operations are performed only if the system is not exiting. Thus, if the analyst does not know the implemented architecture of the CARA, then there is a risk that the SAT results lead the analyst to the wrong conclusions about the safety of the medical device.

Figure 15: (Snippet) Tasks and global variables usage.

6 Discussion

Recommendations to device manufacturers. This study has shown that architecture analysis can shed light on quality issues, especially testability and verifiability properties that can translate into safety issues. Based on this research, testability and verifiability properties should be addressed in the construction of a device (safety) assurance case. Arguments of the assurance case might refer to static and runtime structures, including how the implemented software handles OS variants, COTS software interactions, partitioning of the source code elements into runtime tasks, and how tasks communicate and synchronize with each other. Approaches such as those discussed can provide the evidence needed to justify the arguments.

Recommendations to SAT users. This study identified several issues related to using a SAT in the absence of knowledge about the implemented architecture of the system under verification. In order to avoid misleading results, SAT users must pay significant attention to understanding how the SAT handles multitasking as well as OS and library calls. Using the extracted runtime structures we learned the importance of mock implementations of analog/digital converter APIs' to facilitate verification of safety properties of tasks that interact with hardware boards. Otherwise, there is a high risk for incorrect conclusions about the software safety properties believed to be verified by the SAT. Thus, in order to leverage the full power of the SAT, the user needs to be aware of structures of the software under analysis so that the SAT can be configured appropriately.

Preparing for modular verification. Verifying safety properties using SATs takes time. Many SATs allow running verification as a collection of parallel jobs for improved performance. However, SAT users must identify modules of the system under verification. Using the runtime structure of the CARA, the OSEL analyst identified how to run the SAT in a modular fashion. The OSEL analyst defined a verification job for code unique to each task. The runtime structure helped the analyst plan for modular verification. Work is underway to measure response time of verifying several jobs in parallel against verifying the CARA as one job.

Generalizability of the approach. In order to replicate this case study on other systems, we believe the reader can benefit from the step-by-step description of how we did reverse architecting of software structures including the static and runtime structures. Using the CARA, the paper discussed a set of abstract, yet precise, structures that can be used to systematically reason about software testability. Medical devices often handle OS variants, hardware interaction, GUI, COTS software interactions, and multitasking concerns. Thus, the analysis outlined in this paper are repeatable in other

devices. The formulas for statically extracting runtime structures defined in Figure 9 are independent of the CARA, and therefore are reusable for other medical devices that are based on task-oriented architectures. A KB of external libraries can be of significant value to first get a high-level summary of architectural features in addition to a parser that extracts code relations. In this case we did not have the need for performing data flow analysis. For other systems, such analysis might be needed. A query language similar to the RPA [4] facilitates reverse architecting significantly because questions related to software architectural properties can be quickly answered by using a suite of relational queries. A graph visualization tool [28] is also important to visualize software structures and recognize patterns visually.

7 Closing Remarks

We investigated the benefits of incorporating architectural analysis of implemented systems for safety analysis of a medical device. We reconstructed a catalog of static and runtime structures from the system's source code. We then did a rigorous and detailed analysis by extracting one structure at a time, ignoring irrelevant details with respect to the selected structure. Evidence that architectural analysis can offer useful insights on testability and verifiability, from which we can form an assurance case for safety was collected. We showed how architectural analysis can be integrated with SATs to verify safety properties. We have not extracted and analyzed all structures related to testability and safety. For example, our runtime structures did not yet address interrupt handling and real-time watchdog behaviors of the CARA. Specific analysis questions are a) how easy is it to trigger interrupts and test their handling capability? b) how easy is it to test real-time features such as the behavior of the watchdog, if some functions do not reply within expected time constraints? Work is underway to extract those runtime structures. Our vision is to collect a catalog of structures to reason about testability and safety of medical devices at the architecture level.

References

1. P. A. Laplante, C. J. Neill, and R. S. Sangwan. Healthcare Professionals' Perceptions of Medical Software and What to Do About it. IEEE Computer, 39(4), 26-32, 2006.
2. P. Kruchte. Architectural Blueprints — The "4+1" View Model of Software Architecture. IEEE Software, 12(6), 42-50, 1995.
3. A. Betin-Can, T. Bultan, M. Lindvall, and B. Lux. Application of design for verification with concurrency controllers to air traffic control software. ASE, 2005.
4. L. Feijs, R. Krikhaar, and R. Van Ommering. A Relational Approach to Support Software Architecture Analysis. Software Practice and Experience, 28(4), 1998.
5. B. Fitzgerald. CDRH Software Forensics Lab: Applying Rocket Science to Device Analysis. In Medical Devices Today magazine, October 15, 2007.
6. D. Ganesan, M. Lindvall, D. McComas, M. Bartholomew. Verifying Architectural Design Rules of the Flight Software Product Line. In SPLC, 2009.
7. D. Ganesan, M. Lindvall, L. Ruley, R. Wiegand, V. Ly, and T. Tsui. Architectural Analysis of Systems based on the Publisher-Subscriber Style. In WCRE, 2010.
8. D. Ganesan, M. Lindvall, D. McComas, M. Bartholomew, S. Slegel, and B. Medina. Architecture-based Unit Testing of the Flight Software Product Line. In SPLC, 2010.
9. D. Ganesan, C. Verhoef, R. Krikhaar, and M. Lindvall. The ADAM Project, www.few.vu.nl/~rkrikhaa/adam/
10. T. Gee. FDA Raises Bar on Medical Device Software Testing. In Medical Connectivity magazine, October 24, 2007.
11. Universal Driver. http://docs.diamondsystems.com/
12. OSAL. http://opensource.gsfc.nasa.gov
13. R. Jetley, P. Jones, and P. Anderson. Static analysis of medical software using CodeSonar. Workshop on Static analysis, 2008.
14. N. Leveson and C. Turner. An Investigation of the Therac-25 Accidents. IEEE Computer, 26 (7), 18-41, 1993.
15. B. Meier. FDA Steps Up Oversight of Infusion Pumps. In the New York Times, April 23, 2010.
16. P. Jones, R. Jetley, and J. Abraham. A Formal Methods-based verification approach to medical device software analysis. Embedded Systems Design, April 2010.
17. D. R. Wallace and D. R. Kuhn. Lessons from 342 Medical Device Failures. IEEE International Symposium on High-Assurance Systems Engineering, 1999.
18. S. Purushothaman, D. Hislop, P.L. Jones, J. Lee, F. Pearce, and S. van Albert. 2004. Introductory paper of the CARA. Int. J. Softw. Tools Technol. Transf. 5, 4 299-300, 2004.
19. D.R. Harris, H.B. Reubenstein, and A.S. Yeh. Reverse Engineering to the Architectural Level. ICSE. 186-195, 1995.
20. R. Kazman and S. J. Carrière. Playing Detective: Reconstructing Software Architecture from Available Evidence. ASE. 6(2), 107-138, 1999.
21. R. Krikhaar. Software Architecture Reconstruction. PhD Thesis, University of Amsterdam, 1999.
22. R. Schwanke. An intelligent tool for re-engineering software modularity. Proceedings of ICSE, 83-92, 1991.
23. H. Gomma. Software Design Methods for Concurrent and Real-Time Systems. Addison-Wesley, ISBN 0-201-52577-1.
24. CFR - Code of Federal Regulations Title 21, FDA.
25. C. Hofmeister, P. Kruchten, R.L. Nord, H. Obbink, A. Ran, P. America. A General Model of Software Architecture Design Derived from Five Industrial Approaches. Journal of Systems and Software, Vol. 30, Issue 1, pp. 106-126, January 2007.
26. R. Krikhaar et al. Enabling system evolution through configuration management on the hardware-software boundary. Journal of Systems Engineering, Vol 12, 13, 2009.
27. P. Van der Spek and C. Verhoef. Balancing time-to-market and quality in embedded systems. Views on Evolvability of Embedded systems, Embedded Systems, 2011, Springer.
28. Prefuse visualization tool. http://prefuse.org/
29. M. Lindvall and D. Muthig. Bridging the Software Architecture Gap. IEEE Computer 41(6), 98-101, 2008.
30. B. Schmerl, B, J. Aldrich, D. Garlan, R. Kazman, and Y. Hong. Discovering Architectures from Running Systems. In IEEE TSE, 32(7), 454-466, 2006.

Acknowledgements. Dr. William Hartmann, Executive Vice President, Fraunhofer USA for encouraging the research collaboration with the FDA. Steve van Albert, Dr. Fred Pearce, and Jaime Lee from Walter Reed Army Institute of Research (WRAIR) for making the CARA software available to us. John Martin, Small Bear Technology, developed and implemented the CARA software. Dr. Rene Krikhaar and Dr. Chris Verhoef from VU University Amsterdam for great feedback.