A Formal Methods Approach to Medical Device Review

Article *in* Computer · May 2006

DOI: 10.1109/MC.2006.113 · Source: IEEE Xplore

Raoul Jetley
North Carolina State University
52 PUBLICATIONS **713** CITATIONS
SEE PROFILE

Paul L Jones
U.S. Department of Health and Human Services
22 PUBLICATIONS **592** CITATIONS
SEE PROFILE

A Formal Methods Approach to Medical Device Review

Raoul Jetley and S. Purushothaman Iyer
North Carolina State University

Paul L. Jones
US Food and Drug Administration, Center for Devices and Radiological Health

As device software becomes more complex, regulators need rigorous evaluation tools and methods to assure safe operation of this software. The research presented here applies formal methods-based techniques to this problem space.

With software playing an increasingly important role in medical devices, regulatory agencies such as the US Food and Drug Administration need effective means for assuring that this software is safe and reliable. The FDA has been striving for a more rigorous engineering-based review strategy to provide this assurance.

The use of mathematics-based techniques in the development of software might help accomplish this.[1] However, the lack of standard architectures for medical device software and integrated engineering-tool support for software analysis make a science-based software-review process more difficult.

Regulation of medical device software encompasses reviews of device designs (*premarket review*) and device performance (*postmarket surveillance*). The FDA's Center for Devices and Radiological Health performs the premarket review on a device to evaluate its safety and effectiveness. As part of this process, the agency reviews software development life-cycle artifacts for appropriate quality-assurance attributes, which tend to reveal little about the device software integrity.

Once a device has been placed on the market, CDRH postmarket surveillance processes monitor its performance. If CDRH receives a report of a device (or software) failure or malfunction resulting in actual or potential serious injury or death, it will typically perform a health-hazard evaluation. If warranted, the Center authorizes an investigation to determine the failure or malfunction's cause—a process termed *forensic analysis*—which can result in required corrective actions or device recalls.

Although these regulatory processes work reasonably well for device production processes, they're insufficient for assessing software. From a premarket perspective, subtle errors and latent bugs might exist in the software that only formal model-checking techniques or exhaustive white-box testing can detect. Such techniques require a model-based software design against which investigators can check the device's temporal properties or test cases.

Few device manufacturers use model-based designs. Rather, they build the software based on their understanding of the product requirements. This can produce usage inconsistencies among the various devices, which can result in harmful situations.

Establishing the cause of software faults during forensic analysis isn't an easy task either. Often, the malfunctions under review are system dependent or nondeterministic, making reproducing them virtually impossible. In such cases, the only way to trace the failure's exact cause has historically been to manually review the source code itself. Given the complexity of modern medical device software, this is a nearly impossible task for a third-party investigator with no prior knowledge of the software.

To address these problems, researchers at the FDA's Center for Devices and Radiological Health/Office of

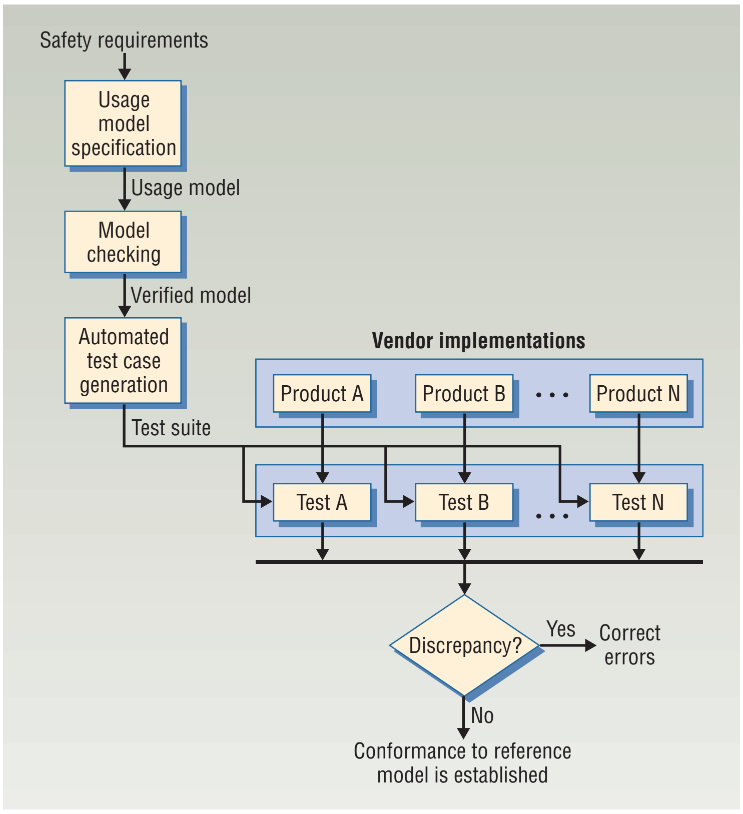

Figure 1. Schematic showing how a usage model is used to enhance CDRH's premarket review process. The scheme as depicted involves deriving a test-case suite from a verified usage model—essentially a one-time effort—and using this test suite to assess individual device software.

Science and Engineering Laboratories (CDRH/OSEL) have been collaborating with university researchers to explore ways to use formal modeling methods and static analysis techniques to improve the review process. These techniques include developing usage models to aid premarket review and using abstraction-driven slicing techniques to facilitate postmarket forensic analysis.

USAGE MODELS IN PREMARKET REVIEW

A *usage model* is a formal software representation aggregating a particular device's (or class of devices') common characteristics and safety features. In building a usage model, CDRH/OSEL aims to assure that all device implementations meet minimum safety conditions. It achieves this by deriving a set of test-case sequences through exhaustive path exploration on the model. Manufacturers can then use these test cases to verify safety characteristics of devices submitted for premarket review. The usage model can also be made available publicly and distributed among manufacturers as a reference for developing their software specifications, thus contributing to standardization of safe designs.

For example, the usage model for cardiac defibrillators would include safety features that all vendor implementations of such defibrillators must include. All cardiac defibrillators submitted to CDRH for premarket review would therefore be evaluated against test cases derived from this usage model, and the CDRH would consider for market approval only those that pass all tests.

Model schematic

The schematic in Figure 1 shows how a usage model can enhance CDRH's premarket review process.

The usage model input is a set of safety requirements and domain specifications relevant to the device being modeled. CDRH defines the model using formal-methods-based specification techniques, typically representing it as a state machine or finite-state automaton. The model is rigorously verified by CDRH using model-checking techniques to ensure its correctness and adherence to the safety requirements.

After verifying the model, CDRH uses the model to automatically derive test-case sequences. Each test case is essentially an execution path in the usage model and corresponds to a set of user inputs or system events. CDRH records all such distinct paths (test sequences) and their corresponding results and maintains them as a test suite for products in the device family. This test suite is then made available publicly to all device manufacturers. While developing a device, the manufacturer checks it against the test suite for conformance. Any discrepancy between the expected results (as obtained through the usage model) and observed outputs indicates a failure of compliance for the product. If the results of the tests are in agreement, however, the product is deemed compliant with the established safety criteria. Such evidence can then be used when making a safety case for the device in a premarket submission.

Case study: Infusion pump usage model

To support this premarket review scheme, we've developed a usage model for infusion pump software. An infusion pump is a typical safety-critical medical device that uses software to control the underlying hardware and events. We categorize infusion pumps into five major types: large volume, patient-controlled analgesia (PCA), elastomeric, implanted, and ambulatory.

These pumps are similar in that they're all used to administer fluids into a patient's circulatory system. However, significant differences exist across the different categories in terms of features such as infusion mode, volume injected, and patient interaction. Similarly, the software for controlling these pumps also shares certain common functionalities, but differs when it comes to pump-specific features.

Our usage model exploits the relation between the various pump categories. We represent the model as a two-tiered architecture:

- a generic core model that captures common features, and
- extensions to the core (or *wrappers*) to model the different categories' specific aspects.

Figure 2 gives an overview of the usage model for infusion pump software. In addition to the core and wrappers, the model includes a user interface, a pump module, and a simulation of the patient undergoing treatment.

We model the core as a finite automaton, with each state in the automaton representing a particular pump configuration at a given instant of time. For example, the state "On/Cold/Alarming" indicates that the pump is switched on, is currently dormant (cold), and the alarm is on because of some malfunction. Transitions in the automaton represent inputs provided by a human user (such as pressing a button) or the system environment (for example, time-out or malfunction). In total, the core model consists of 292 states and about 3,500 transitions.

We characterize models for the different categories (or pump types) by the types of inputs and events they allow. Each category model therefore corresponds to a set of wrappers that maps real-world values to abstract Boolean inputs for the core. Abstracting the events this way not only facilitates modeling pump categories, but also helps reduce the core model's state space (and thus its complexity). Moreover, abstracting event-handling details away from the core improves the overall usage model's adaptability, making it much easier to incorporate changes.

The usage model user interface is simple but comprehensive. It provides all possible buttons (both hard and soft) and screens available to the care provider, but lets the user disable them if necessary. For example, screens corresponding to the point-of-care unit (PCU) can be provided for large-volume pump models, but would be disabled for models corresponding to ambulatory or implantable infusion pumps.

The pump module represents the physical pump along with peripherals such as the syringe, boluses, and vials. It simulates the physical pump and provides system-generated events such as interrupts and time-outs.

The patient module is mostly passive, except in the case of the PCA pump model, in which the user can trigger an event by requesting an analgesic dose. Ideally, this module would be more comprehensive and would simulate the patient's reactions to the medication being administered (for example, by monitoring the patient's heart rate or blood pressure). However, defining such a simulation model is a nontrivial task and an area of active research on its own.

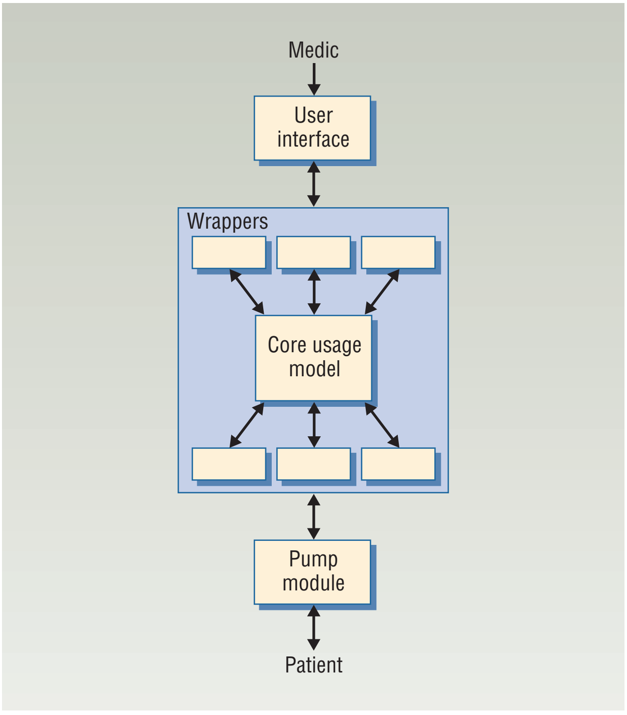

Figure 2. Usage model for infusion pump software. The core model is defined as a state chart machine in Matlab. Wrappers provide mappings between the abstract states in the core model to corresponding functionalities in actual software implementations.

Although we haven't yet applied the usage model to verify a real-world device, it has performed well in laboratory simulations of the software. We used the infusion pump usage model to test prototype PCA pump software developed at OSEL. The test suite derived from the model consisted of approximately 39,000 test-case sequences, which we used to perform guided simulations for the prototype, and which yielded numerous previously undetected error conditions. The testing uncovered a total of 44 errors, of which five were identified as serious, safety-critical errors.

POSTMARKET FORENSIC ANALYSIS

With extensive premarket analysis, the need for postmarket analysis should be minimal. However, there might be situations not covered by the core usage model that could result in harm. Bug fixes and adaptive maintenance can also introduce errors in software, resulting in code that is different from the original, reviewed software. A safety-related error could necessitate a postmarket review by CDRH to determine the error's cause and initiate corrective action.

Although manual source-code scanning remains infeasible for complex software, static analysis tools and techniques could aid the Center's forensic-analysis process. *Program slicing*[2] is the most popular of these methods. Debuggers and maintenance engineers commonly use this

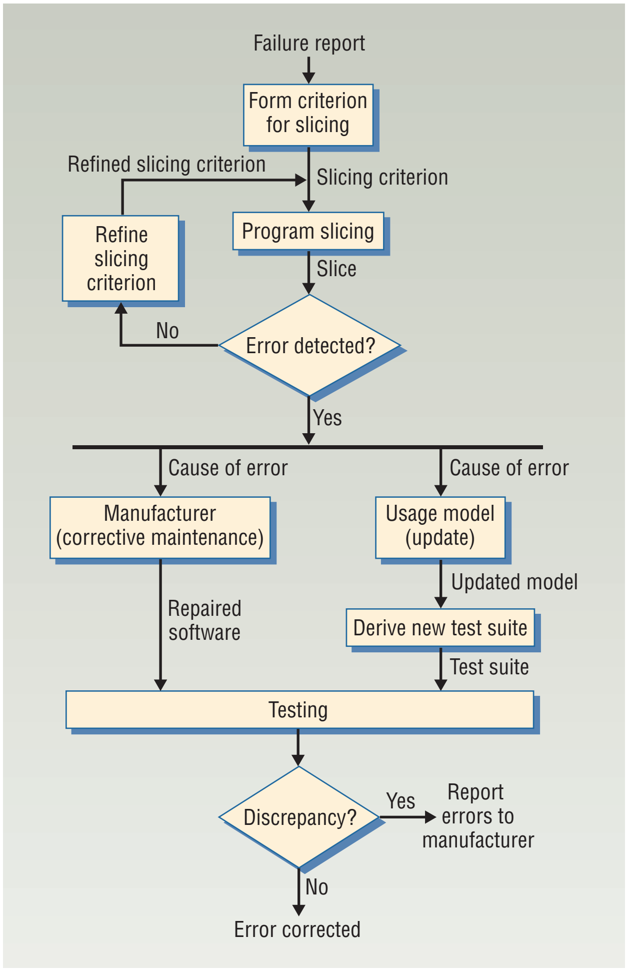

Figure 3. A slicing-based approach to postmarket review. Criteria for slicing are refined iteratively until the error is successfully resolved. Corrected software is tested against test cases derived from the usage model before it's released again to the market.

technique to facilitate understanding and debugging of programs by focusing on select semantic aspects. Slicing aids analysis by deleting those parts of the program with no effect on the semantics of interest, and focuses attention on the code segment that might contain a fault.

Formally, we define a (static) slice for a program p with respect to a tuple (p, x, V), where x is a point in the program (typically a node in p's control flow graph), and V is a set of variables belonging to p. We can remove a statement from p to form a static slice, s, if it doesn't affect the value of any variable in V when the next statement to be executed is at point x.

Figure 3 depicts a scheme for using program slicing in the postmarket review process.

To trace software errors to source code, analysts (or forensic investigators) use a failure or malfunction report to form the initial slicing criterion. They then use this criterion to obtain a program slice that helps them identify the code responsible for the failure. The slicing is carried out in an iterative manner, with the slicing criterion refined over each iteration, producing smaller, more concise slices.

Once analysts have successfully traced an error, they issue a report to the manufacturer, who can then perform corrective maintenance to remove the bug from the software. An error traced to the product's design might indicate a problem with the usage model as well (assuming that the device passed all tests during premarket review). In this case, the analysts update the usage model to correct the fault and subsequently to derive a new test suite.

CDRH can then reevaluate the updated software (from the manufacturer) against the new test suite to ascertain that the error has been fixed and that new errors haven't been introduced in the code.

Integrated approach

Slicing-based analysis is popular because of its simplicity and ease of use. However, it's not always the most efficient, especially when used for large, highly cohesive programs. The slices obtained for these programs are usually too large and require several refinements and iterations before analysts can successfully trace the error to its source. Moreover, executing or dynamically debugging the code during postmarket review is rarely possible, rendering infeasible the use of runtime-analysis techniques such as dynamic slicing and execution backtracking.[3] Consequently, the analyst is constrained by the limitations of static slicing and spends considerable effort during the postmortem analysis process.

One way to overcome this problem and improve the efficiency of forensic analysis is to combine static slicing with model abstraction.[4] As the "Genesis of Model Abstraction" sidebar describes, model abstraction is based on the idea of viewing a program's analysis as an abstraction of the program's behavior. Using abstraction together with slicing gives analysts an abstract semantic representation of the program that both improves their understanding of the program and helps them define better slicing criteria to derive more concise slices. The slices in turn contribute to deriving more refined abstract models for the program. Using the two in tandem reduces the number of iterations and slices required during analysis and thus the effort required for postmortem analysis.

The rationale behind combining slicing and abstraction is to provide the analyst with a better understanding of the code—in other words, to improve the level of program comprehension. The "Program Comprehension Strategies" sidebar discusses some popular approaches to program comprehension and explains how using abstraction and slicing exploits these strategies.

The algorithm in Figure 4 gives an overview of this integrated approach to forensic analysis. The key to the

Genesis of Model Abstraction

Software model checking[1] is based on the abstract-check-refine paradigm: build an abstract model, check the desired property, and, if the check fails, refine the model and start over. The most critical (and most time-consuming) task in this approach is defining a model that sufficiently and correctly captures the features of the system under consideration. Because human analysts develop the model, it can be erroneous and incomplete. Such errors could be hard to detect and could easily undermine the analysis effort.

To avoid (or in any case, reduce) such errors, and to expedite the model-building process, model abstraction techniques are used to automatically extract the model from the code. We can define model abstraction as the process of formally extracting a program's semantics from the source code. This process uses a set of predefined abstract interpretations[2] to convert a given program to an abstract model, or *Kripke structure*,[3] based on the abstraction criterion defined. We can represent the model that the abstraction generates as a labeled transition system (LTS), giving an abstract representation of the program's semantics, or used to verify a temporal property ∅ and ascertain the program's validity.

References

1. S. Merz, "Model Checking: A Tutorial Overview," *Modeling and Verification of Parallel Processes*, F. Cassez et al., eds., LNCS 2067, Springer-Verlag, 2001, pp. 3-38.
2. P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *Proc. Symp. Principles of Programming Language*, ACM Press, 1977, pp. 238-252.
3. E.M. Clarke and E.A. Emerson, "Synthesis of Synchronization Skeletons for Branching Time Temporal Logic," *Logic of Programs: Workshop,* LNCS 131, Springer-Verlag, 1981.

Program Comprehension Strategies

The rationale behind combining slicing and abstraction is to provide the analyst with a better understanding of the code—in other words, to improve program comprehension. Several cognition models attempt to explain how program comprehension works. Two of the most popular of these are the top-down and bottom-up models.[1,2]

The top-down model of program comprehension is typically invoked when the code under consideration is familiar, and a description of the application domain's conceptual components and how they interact is available.

Analysts understand code by exploiting this knowledge to formulate hypotheses about the meaning of the program segments they are analyzing. They confirm their hypotheses by scanning the code for *beacons*—pieces of code implementing typical program constructs—and iteratively refine these hypotheses, producing new subgoals to be verified again by scanning code. The process ends when analysts have identified each component in the code.

The bottom-up model of program comprehension is invoked when the code under consideration is completely new to the analyst. The first mental representation of the code an analyst builds is a *program model,* a control-flow abstraction.

The analyst creates the program model by chunking microstructures into macrostructures and via cross-referencing. Starting from the program model, analysts can abstract a further model that maps the control-flow knowledge about code to real-world domain knowledge. The process continues recursively as analysts formulate new models to aggregate these plans into higher-order plans.

Anneliese von Mayrhauser and A. Marie Vans suggest a more holistic, integrated model.[3] They base their model on the premise that the comprehension process doesn't proceed exclusively in either the top-down or the bottom-up direction, but instead switches continuously between the two approaches. Our forensic analysis approach—that is, using slicing and abstraction in tandem—is based on such an integrated model of program comprehension. Traditional static slicing realizes top-down comprehension, whereas the program models generated via abstraction provide bottom-up comprehension.

References

1. R. Brooks, "Towards a Theory of the Comprehension of Computer Programs," *Int'l J. Man-Machine Studies*, vol. 18, no. 6, 1983, pp. 543-554.
2. N. Pennington, "Comprehension Strategies in Programming," *Empirical Studies of Programmers: Second Workshop*, G.M. Olson, S. Sheppard, and E. Soloway, eds., Ablex Publishing, 1987, pp. 100-113.
3. A. von Mayrhauser and A. Marie Vans, "Program Comprehension during Software Maintenance and Evolution," *Computer*, Aug. 1995, pp. 44-55.

algorithm lies in its iterative nature. The algorithm starts by producing a large, top-level slice, S, for the user-defined slicing criterion SC (in the worst case, S could be as large as the entire program). Based on the variables in SC, analysts derive an abstraction criterion, AC, through a symbolic execution of the slice S. They then use AC to

```
Input: A program P, a failure report F, and a slicing criterion SC = (x, V), where
x is a program point and V is a set of variables.
while F not successfully traced to code do
        Obtain a slice S for the criterion SC.
        Derive an abstraction criterion AC using S and SC.
        Obtain a labeled transition system (LTS) L for the slice S using AC.
        if F is narrowed to a specific part of LTS L
                Choose a refined slicing criterion SC = (x, V) and continue.
        else
                Change slicing criterion SC = (x, V) and generate a new slice S.
        end if
end while
```

Figure 4. An algorithm describing the integrated approach to forensic analysis by combining program slicing with abstraction. The slice S *and LTS* L *become more refined in each iteration of the algorithm. During each iteration, the abstraction criterion is derived using* S *and the slicing criterion obtained from* L.

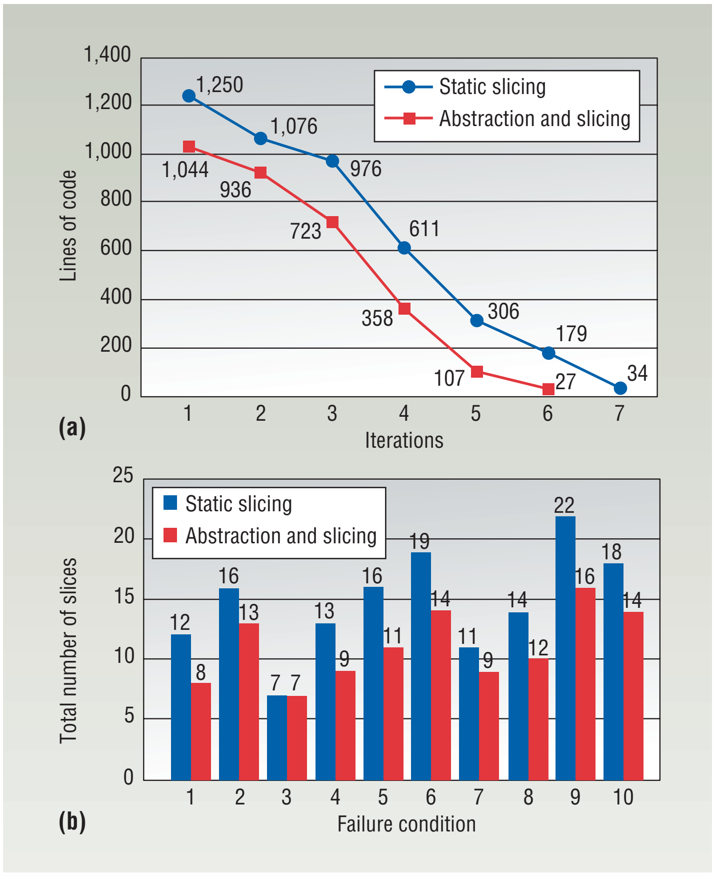

Figure 5. Forensic analysis of the XiO software: static slicing vs. the integrated approach. (a) Comparison of the two approaches with respect to lines of code per slice. (b) The number of iterations required per error for both approaches.

obtain a labeled transition system, L, corresponding to S. The LTS gives the analysts an abstract representation of the slice and helps them obtain a newer, smaller slice by refining SC. Analysts use this slice to produce a newer, more accurate LTS, and the mutual refinement process continues.

Each successive iteration of the algorithm further refines S and thus reduces its size. Similarly, the abstract model, or the LTS, L, becomes more specific and detailed in each iteration. The successive iterations return more refined slices and specific models until the problem is successfully resolved or the slice is shown not to contain the error under consideration. In the latter case, a new slicing criterion is defined, and the entire process repeated.

Case study: XiO system

To illustrate the use of our proposed forensic analysis approach, we use it to trace errors in the XiO system software. XiO is a radiation therapy planning system used to develop treatment plans for cancer patients. It features a graphical environment with a combination of drop-down file menus and icons to provide access to planning functions. Users have reported several failures for the system. We've logged the symptoms in corresponding problem reports and have performed a forensic analysis of the software based on the reports.

For the purpose of the case study, we limited ourselves to the Port menu interface for the teletherapy module and the errors detected therein. The module consisted of 40,000 lines of C code. Because the module was part of a larger system, however, the code was incomplete by itself. It contained several references to missing global variables and external function calls. To deal with these, we had to make a few assumptions and approximations before beginning the analysis.

Our analysis used the Codesurfer code navigator[5] and the C Wolf model-extraction tool[6] to provide static slicing and abstraction, respectively. To assess our approach's impact, we performed the analysis in two independent phases:

- In the first phase, we used only static slicing to trace the errors.
- In the second phase, we combined model abstraction with slicing.

In all, we analyzed 10 failure conditions, which we successfully traced to their source in the XiO software.

Figure 5a shows the slice refinements over successive iterations with respect to both static slicing and the integrated approach presented in Figure 4. As the figure shows, the average size of successive slices decreases in both approaches as we refine the slicing criteria. Also, the slices used in the integrated approach are consistently smaller. Figure 5b shows the number of slices required to trace an error for the two approaches.

The chart indicates the total number of slices, including the top-level slices and the refined slices obtained through successive iterations. As the figure shows, the integrated approach requires fewer slices and is thus more efficient.

Both approaches traced the errors successfully, but combining slicing with abstraction required 39 percent less effort than static slicing alone. The total effort required during forensic analysis using static slicing was 37.5 person-hours, as compared to only 22.6 person-hours for the integrated approach using slicing and abstraction, further strengthening the case for the integrated approach.

Experimental evidence shows that our approach to premarket review and forensic analysis is effective. However, a general acceptance of the proposed methodology by both the FDA and device manufacturers would depend on further, large-scale studies that show the advantage of using these formal-methods-based tools.

The effectiveness of model-based premarket reviews depends on how comprehensive the usage model is. A detailed, well-defined model can ensure greater conformance to safety-critical properties, and consequently safer device software. Similarly, the forensic analysis approach can be further improved by automating the integration of slicing and model abstraction. We're currently implementing such a tool at North Carolina State University.[7]

Although the two research efforts presented here have different focuses, they're complementary and share common theories. Applying these shared techniques can strengthen the analysis process and resulting rigor of regulatory processes.

Acknowledgments

Raoul Jetley performed part of this research as an ORISE fellow under the US Food and Drug Administration's Oak Ridge Institute of Science and Education program. We thank Oak Ridge National Laboratories for providing the grant. We also thank CMS Inc. for providing the software and error reports for the XiO radiation therapy planning software.

References.

1. J. Rushby, *Formal Methods and the Certification of Critical Systems*, tech. report, Computer Science Laboratory, SRI Int'l, Dec. 1993.
2. M. Weiser, "Program Slicing," *Proc. 5th Int'l Conf. Software Eng.*, IEEE CS Press, 1981, pp. 439-449.
3. H. Agrawal, R. DeMillo, and E. Spafford, *Efficient Debugging with Slicing and Backtracking*, tech. report, Purdue Univ., 1990.
4. P. Cousot and R. Cousot, "Systematic Design of Program Analysis Frameworks," *Proc. Symp. Principles of Programming Language*, ACM Press, 1979, pp. 269-282.
5. P. Anderson et al., "Tool Support for Fine-Grained Software Inspection," *IEEE Software*, vol. 20, no. 4, 2003, pp. 42-50.
6. D.C. DuVarney and S.P. Iyer, "C Wolf—A Toolset for Extracting Models from C Programs," *Proc. 22nd IFIP WG 6.1 Int'l Conf. Houston Formal Techniques for Networked and Distributed Systems* (FORTE), Springer-Verlag, 2002, pp. 260-275.
7. R. Jetley, Y. Zhang, and S.P. Iyer, "Using Abstraction Driven Slicing for Postmortem Analysis of Software," to be published in *Proc. 14th Int'l Conf. Program Comprehension*, 2006.

***Raoul Jetley** is a doctoral candidate at North Carolina State University. His research interests include static analysis and verification of medical device software. Jetley received an MS in computer science from NCSU. Contact him at rpjetley@alumni.ncsu.edu.*

***S. Purushothaman Iyer** is a professor in the Computer Science Department at North Carolina State University. His research interests include programming and specification languages, software model checking, probabilistic models of concurrency, and probabilistic model checking. Iyer received a PhD in computer science from the University of Utah. Contact him at purush@csc.ncsu.edu.*

***Paul L. Jones** is a senior systems/software engineer at the US Food and Drug Administration, Center for Devices and Radiological Health/Office of Science and Engineering Laboratories. He divides his time between transitioning high-confidence software and systems research to the medical device industry regulatory process and internal consulting to the FDA on medical device software system safety. Jones received an MS in computer engineering from Loyola College, Maryland. Contact him at PaulL.Jones@ fda.hhs.gov.*