An Approach to Reviewing Software in Medical Devices

Raoul Jetley S. Purushothaman Iyer
Dept of Computer Science
North Carolina State University
Raleigh, NC 27695-8207
EMail: {rpjetley, purush}@csc.ncsu.edu

Paul L. Jones
Center for Devices
and Radiological Health
Food and Drug Administration
Rockville, MD 20857
EMail: pxj@cdrh.fda.gov

Abstract

Formal methods have been long proposed as a technique for producing safety-critical software in spite of the upfront cost. In contrast, however, market forces dictate early and quick delivery of software in medical devices to field. Given these competing forces the US Food and Drug Administration is required, by US Congress, to review applications for introducing new devices within a certain time limit. However, FDA is encouraged to carry out follow-up reviews of approved devices and can recall a device if problems are found.

In this paper we study how *pre-* and *post*-deployment analysis of designs and implementations can be carried out, based on formal-methods based tools, to make the process of reviewing software (in medical devices) easy. We discuss a methodology that is based on using abstractions to relate designs and implementations. We present a case study involving a generic infusion pump, where we show how the various stages can be carried out. Finally, we present a number of experimental results to show our proposed methodology is effective.

1 Introduction

Safety is a primary concern for software used in medical device systems. To ensure safe operation of medical device software, the US Food and Drug Administration (FDA) is charged with enforcing strict standards. In particular, the FDA is charged with carrying an initial review of a product application within a prescribed time limit so that a device manufacturer can bring his product to market in a timely manner. Furthermore, the FDA is allowed to carrying on post-deployment review of a product and monitor its performance. If problems are detected during this stage then FDA could recall the product off the market.

The FDA currently uses process-oriented guidelines and testing methodologies to review and certify medical devices for general use. This approach, though not entirely without merit, is far from comprehensive, and lacks the rigor required for demonstrating that the source code is correct with respect to product specifications. All too often, safety related (software) errors are discovered only after a device is already on the market. When such errors are discovered, it becomes essential to ensure that modifications preserve the overall safety and effectiveness of the device; A noteworthy statistic is that of the 242 recalls conducted by the FDA between 1992 and 1998, 192 (or 79%) were due to software defects that were introduced when changes were made to the software after its initial production and distribution [1].

With the ever increasing growth in the use of personal technologies, such as PDAs, wireless connectivity, and medical devices, a convergence of all three of these technologies is likely to happen in the near future. More importantly, there will a great number of applications from medical device manufacturers, to the FDA, for their products to be reviewed. Since safety of these medical devices is FDA's priority, researchers within FDA have been looking for formal-methods based solutions to the problem of swiftly reviewing applications for manufacturing medical devices, and to carry out reviews once a device has been placed in the field. In this paper, we propose one possible approach for carrying out both pre-deployment (or initial) review and post-deployment review of software in a medical device.

Our proposed approach can be characterized as a model-based approach. In the pre-deployment stage a model can be easily obtained from a design of the medical device. However, for post-deployment review models have to be constructed from implementations. We propose that the technique of abstract interpretation [5] be used to this end. In particular, we propose that abstraction techniques available in software model-checkers such as BLAST [8], CWolf [6] and SLAM [3] be used in this context. Our proposed approach has the advantage of being able to relate designs and implementations from a device manufacturers perspective, thus making it possible for them to relate their implementation to device-specific safety standards published by regulatory agencies. Furthermore, it would allow regulatory agencies, such as FDA, to treat an implementation under review as a white-box; this would be in strong contrast to current practice of treating an implementation as a black-box.

Formal-methods based researchers have produced a number of tools (including Concurrency Workbench, SLAM, SMV etc). But are they usable in our proposed approach? And are they effective enough? These are some of the questions that need to be answered before our proposed approach can be adopted. In general, two sets of issues need to be addressed:

- Is there a general scheme that can be used by FDA and by medical device manufacturers to the mutual benefit of everyone involved? We detail a potential path in Section 2.
- Technical issues relating to the effectiveness of abstraction in establishing meaningful properties of systems using software model-checkers. In Section 4 we present a case study on a generic infusion pump. Our results, presented in Section 5, show that abstractions provided by CWolf [6] are effective enough to construct reasonable models of an implementation.

Road Map. In Section 2 we discuss our methodology, in Section 3

we survey notions and tools needed from work on formal methods, in Section 4 we present the design of a generic infusion pump, as available from FDA, in Section 5 we discuss the results of applying several tools that are currently available to carry out both pre-deployment and post-deployment

2 Methodology

In this section, we detail the proposed approaches to pre-deployment and forensic analyses based on formal methods.

Pre-deployment Analysis

Pre-deployment analysis is based upon the concept of verifying user requirements against the system/design specifications. Figure 1 depicts the general scheme used in our approach to pre-deployment analysis. The specifications, once defined, are used to build a formal state-based model (or automaton) for the system. Traditional model checking techniques can then be used to verify this model, encoding user requirements as temporal properties and verifying them against the automaton.

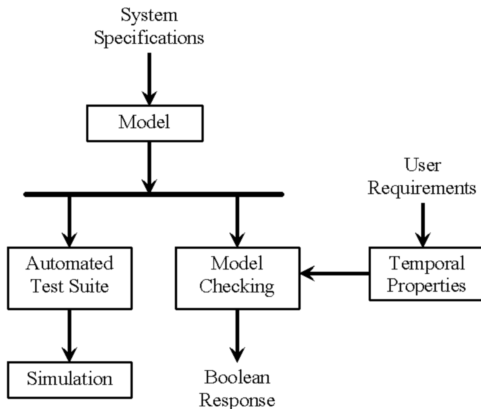

Figure 1. The Pre-deployment analysis process

Another approach for verifying the model is to derive a suite of exhaustive test cases from the model, ensuring that all states are sufficiently covered. Thus, each test case in the suite corresponds to a distinct execution path in the model while the suite itself comprises of the set of all possible execution paths. Guided simulation can then be used to ascertain that all test cases are satisfied, i.e., all paths in the model are correctly executed.

Using pre-deployment analysis to enforce standards:

Pre-deployment analysis is well suited for the early stages of the SDLC, and thus could be quite useful for medical device manufacturers. However, most manufacturers do not perform rigorous formal analysis, attributing it to the high degree of complexity and effort involved in the use of formal methods. Moreover, the success of the analysis depends on the accuracy of the model in capturing the system specifications. Even a slight error in the model could undermine weeks of effort.

As a solution to these problems, and to encourage the use of formal methods, we propose to have pre-deployment analysis incorporated as a part of the FDA safety standards and regulations. To achieve this, a formal model incorporating a common set of properties and safety requirements would be defined and analyzed rigorously using formal methods based techniques. Once this model is ascertained to be complete and correct, it would be included as part of the safety standards and made available publicly to all device manufacturers. The manufacturers could then use this standardized model as their base design and extend it to incorporate additional, more specific features.

The safety standards would also include a set of test cases derived from the standardized model that could be used by the manufacturers to validate their implementations. This suite would include a comprehensive set of test cases that must always be satisfied (corresponding to safety properties), optional test cases that would correspond to features that are desirable but not mandatory (best practices), and test cases that must never hold true (error conditions). Using these test cases for validation would ensure that (at least) the minimum safety guidelines would be satisfactorily met.

The advantages that such a scheme of pre-deployment analysis in conjugation with safety guidelines would provide the FDA are manifold. They include the following:

1. It would ensure that the minimum safety standards are satisfied.
2. It would save effort and minimize errors associated with the model building process.
3. It would facilitate and aid the post-deployment/forensic analysis process, as the results and safety properties established during pre-deployment analysis could be used to evaluate the models extracted from the implementation as well.

Post-deployment Analysis

Post-deployment or forensic analysis is the process of verifying the software implementation (i.e., the source code) against user requirements. The verification is performed, as in pre-deployment analysis, against a formal model of the software. The model used in forensic analysis, however, is abstracted from the source code rather than derived from system specifications. Figure 2 depicts the general scheme for post-deployment analysis.

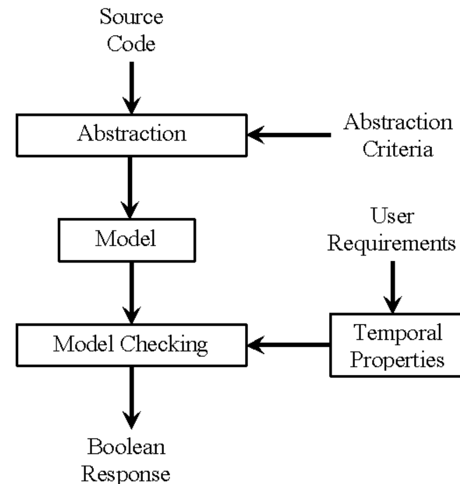

Figure 2. The post-deployment analysis approach

The critical part in this scheme is the extraction of an abstract

model from the source code. It is important to note however, that the abstracted model is not the same as the specification-based model used in pre-deployment analysis. The abstract model here is obtained by mapping statements in the concrete program domain to states in the abstract model domain. The semantics for this mapping are defined using the abstraction criteria, as illustrated by Figure 2.

Two popular strategies for extracting abstract models from source code are data abstraction and predicate abstraction. The basic idea of data abstraction is to evaluate data variables in the program over a smaller abstract domain, while still preserving the existential behaviors of the program execution. Predicate abstraction [14] on the other hand, encodes concrete states using the truth evaluation of a set of predicates present therein. Thus, any concrete states sharing the same assignment to all of the predicates are abstracted into the same representation in the abstract state.

Potential Problem with Proposed Methodology

A major problem with abstraction is that the extracted model may not accurately depict the behavior of the software. The abstraction may be too coarse or too fine; resulting in a model that either produces spurious errors, or is not detailed enough to detect all the errors in the software. A precise abstraction should be able to determine the slightest change in the source code and produce a model that can be distinguished from the model generated before modifying the code.

We assess the precision of the abstraction in our approach by comparing models extracted from original code to those extracted after an error is fixed. The Concurrency Workbench of the New Century (CWB-NC) [13] is used to detect whether the two models are semantically equivalent. If the CWB-NC is unable to distinguish between the two models, it would mean that the abstraction used is too coarse. This comparison between models also helps assess the impact of the error correction and ensures that the changes made during error correction do not have any negative impact on the software.

3 Background

Formal methods are based on a mathematical representation of software and are used to verify system specifications through an exhaustive search across the system domain. We used concepts from model checking and abstraction for the analysis of the generic infusion pump in our case study. In this section we give a brief introduction to these concepts and the tools used to implement these functionalities.

Model Checking

Model checking [9] is based on the abstract-check-refine paradigm: build an abstract model, then check the desired property, and if the check fails, refine the model and start over. Central to this process is the concept of a software model. The model can be either functional, object-oriented or automata-based and can be derived either from the system specifications or automatically abstracted from the implementation (code). This model can then be used to check system requirements (as temporal properties), generate automated test cases to ensure coverage, or produce counter-examples that can be used to prove undesired behavior.

Abstraction

Model abstraction is based on the idea of viewing the analysis of a program as an abstraction of the program's behavior. The concept for abstraction derives from model checking and can be defined as the process of formally extracting the semantics of a program from the source code. Figure 3 shows the general scheme for model abstraction. The abstraction process, as shown in the figure, uses a set of predefined abstract interpretations (AIs) to convert a given program to an abstract model, or a Kripke structure, based on the (abstraction) criteria defined. Abstract interpretation is a framework for executing a program, using values from an abstract domain in place of the original concrete values [5]. The model generated by the abstraction process, usually represented as a labeled transition system (LTS), can be used to verify a temporal property ϕ and ascertain the validity of the program, or generate an error trace as a counter-example.

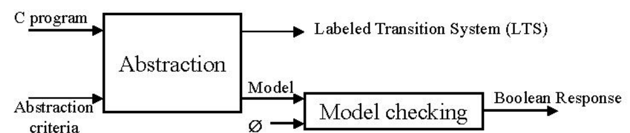

Figure 3. The model abstraction process

Formal Analysis Tools

A number of different software tools were used to assist in the analysis process. A brief introduction to these tools is given here.

Matlab [11] is a commercial software package developed by Mathworks for numeric computation, technical graphics and visualization, and an intuitive programming language for applications in engineering and science. Matlab provides a family of add-on application-specific solutions called toolboxes that extend the MATLAB environment to solve particular classes of problems. **Simulink** is one such toolbox that provides an interactive system for the nonlinear simulation of dynamic systems. Simulink is a graphical program that allows systems to be modeled as real-time or discrete systems. It is often used in conjunction with **Stateflow**, another toolbox used to visually model and simulate complex reactive systems based on finite state machine theory. Stateflow is an interactive design and simulation tool for event-driven systems that provides language elements to describe complex logic in state transition and flow diagram notations.

Reactis [2] is an embedded-software design automation (ESDA) tool suite developed by Reactive Systems. It is based on the Simulink/Stateflow modeling paradigm, and can be used to test, simulate and validate Matlab models.

Reactis consists of three main components: a Tester, a Simulator, and a Validator. The Reactis Tester automatically generates test suites from Simulink/Stateflow models. The test suites provide comprehensive coverage of different test-quality criteria, and while at the same time minimizing redundancy in tests. The Reactis Simulator enables users to visualize model execution in a manner similar to those of traditional debuggers. The Reactis Validator performs automated searches of models for violations of user-specified requirements, returning a test case as a counter-example in case a violation is detected.

Uppaal [4] is a tool suite for modeling and verification of real-time

systems, based on constraint-solving, and on-the-fly techniques, developed jointly by the Design and Analysis of Real-Time Systems group at Uppsala University, Sweden and Basic Research in Computer Science at Aalborg University, Denmark. UPPAAL provides a graphical interface to define system descriptions as networks of timed automata and uses Linear Temporal Logic (LTL) formulae to verify the specified model.

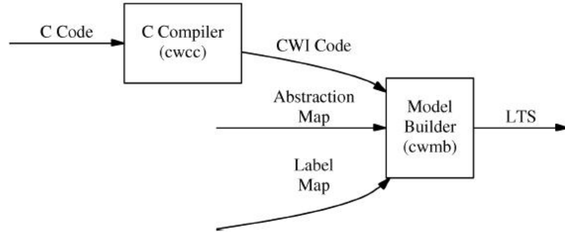

Figure 4. Architecture of CWolf

CWolf [7] is a tool developed at NC State University for automatically generating data-abstracted finite models of C programs. The front end for CWolf consists of a compiler (cwcc), which reduces a C program to an intermediate representation, known as the CWolf Intermediate (CWI) format. The back-end is a model generator component (cwmb), which is built around an abstract virtual machine used to execute abstracted C programs. The model generator operates on three input files. The first is the CWI file containing a compiled C program. The second is an abstraction map file, which describes abstractions for each variable in the input program. The third is a label map file, a text file describing the labels for the transitions of the generated model. The output of the model generator is either a graph with labeled states and edges for viewing by the user in a format readable by the CWB-NC. A simplified system architecture reflecting this view of CWolf is shown in Figure 4.

The **Concurrency Workbench of the New Century (CWB-NC)** is a verification tool that provides the user with a facility to verify automata-based models using model checking and semantic equivalence techniques. The model can be defined in a number of system-design notations, including CCS, CSP and LOTOS, and verified against μ-calculus or GCTL* properties [13]. Similarly, different equivalences can be used to assess the behavior of the model, including pre-order checking and bisimulation equivalences [10].

4 Case Study: Infusion Pump.

Our case study was based on requirement specifications and an implementation of a Generic Patient Controlled Analgesic (GPCA) infusion pump. The GPCA infusion pump is a standardized medical device system being developed at the Center for Devices and Radiological Health (CDRH), FDA. The specifications for the GPCA infusion pump software are defined with respect to a set of possible situations that the pump could be in and a corresponding set of input events.

A situation represents the configuration of the pump and its peripherals at a given instant of time. For example, the situation 'On/Cold/Alarming' indicates that the pump is switched on, is currently dormant (cold), and has the alarm set off due to some malfunction. Events are used to model inputs provided by a human user or the system environment. For example, a human input could be pressing a button or administering a dose, while environment events could include timeouts at intervals of 1 hour and 4 hours respectively. Table 1 gives a list of all possible events for the GPCA infusion pump.

Each event is associated with a probability value, indicating the probability of occurrence for that event with regard to the current situation. The probability of occurrence for each event can range from a minimum value of 0 (indicating an impossible scenario) to a maximum of 1 (indicating that the event is always possible). At each situation, an event can be chosen non-deterministically (with a higher probability event having a better chance of being selected). Upon execution of the chosen event, the pump transitions to a new situation, determined by a transition relation δ defined as

$$\delta : Q \times E \times [0, 1] \times Q$$

where, Q is the set of all situations that the GPCA pump could be in, E is the set of events for the system (as given in Table 1), [0, 1] is the range of reals between 0 and 1 defining the probability of occurrence for each event at a given situation.

The specifications for the GPCA pump were collected by aggregating the behavior of various real-world PCA infusion pumps and were consolidated using the Jumbl model builder developed at the University of Tennessee [12]. The specifications were provided to us in the form of tables and spreadsheets listing the observations for each situation.

The code for the GPCA pump was in the C language, and implemented the various situations and events defined. Supplemental code relating to specific pump functionality was added to facilitate source compilation and to ensure a more complete implementation. In all, the code comprised of 23 files, consisting of approximately 20,000 lines of code (20 KLOC).

To verify the correctness and completeness for the GPCA software, it was not enough to merely detect errors and anomalies in the generated models. Instead we had to ensure the correct operation of the pump for all possible combinations of situations and events defined by the specifications. To achieve this, a number of global properties were defined that when completely satisfied would ensure conformance for each and every situation and event in the GPCA pump system. This set of global properties consisted of the following:

1. All possible situations must be reachable.
2. All valid events must be executable (follows trivially from 1).
3. There must be no deadlocks in the system.
4. There must be no livelocks of length $|Q|$ in the system, where Q is the set of all situations.
5. The combined probability for all events for a given situation must always be equal to 1.
6. If the probability for an event is zero for a given situation, then that event must never be executed.
7. If the probability for an event is non-zero for a given situation, then it must be possible to execute that event.
8. The event 'malfunction' must always trigger the alarm (i.e., it must always result in a situation labeled 'Alarming').

Event #	Event	Source
1	Switch pump on	User Input
2	Switch pump off	User Input
3	Start delivery	User Input
4	Stop delivery	User Input
5	Set correct dose	User Input
6	Set incorrect dose	User Input
7	Set correct mode	User Input
8	Set incorrect mode	User Input
9	Insert correct vial medication / concentration	User Input
10	Insert incorrect vial medication / concentration	User Input
11	Correct connection of a correct administration set	User Input
12	Incorrect connection of a correct administration set	User Input
13	Timeout - one hour	System Environment
14	Timeout - four hours	System Environment
15	Malfunction	System Environment
16	Correct review and error correction	User Input
17	Incorrect dose review	User Input
18	Incorrect mode review	User Input
19	Incorrect vial review	User Input
20	Incorrect administration set review	User Input
21	End use (includes terminate power supply, disconnect patient, and remove pump)	User Input

Table 1. Input Events for the GPCA Software

Modeling the GPCA Pump

The GPCA model was defined using the Stateflow toolbox of Mathworks Simulink, which is a part of the Matlab tool suite. Matlab was chosen as a modeling tool owing to its support for state transition notation and numerical computations.

Figure 5 shows the front-end interface for the GPCA infusion pump software designed in Simulink. The pump is modeled as a closed loop system with the input events being sent to the model at fixed step discrete time intervals of 0.5 seconds.

The model can be viewed as comprised of three subsystems, labeled INPUT, STATECHART and FEEDBACK respectively. The INPUT subsystem is used to model events to the GPCA state machine. The input to this module is simply an integer value between 1 and 21, representing one of the events described in Figure 5. In all, the Stateflow chart consists of 292 states and approximately 3500 transitions. A customized parsing tool was used to automatically extract the model from the specifications provided in order to improve the efficiency and minimize errors during the model building process. The Stateflow API was used to code the states and transition behavior.

The 'GPCA state machine' block has 21 inputs corresponding to each of the input events possible, and three outputs - current state, an integer indicating the current configuration the state machine is in; alarm, a boolean flag set when the system is in an alarming state; and prob, an array listing the probability for all possible transitions for the current state. The function call generator f() is used to trigger a new transition in the state machine every 0.5 seconds.

The FEEDBACK subsystem is used primarily to display the status of GPCA state machine and to select a next valid transition as input to complete the feedback loop in the model. The block ProbabilisticChooser uses a Matlab function to non-deterministically select a transition using the values in the array prob (a higher probability value translates into an increased chance of that transition being selected). The chosen transition index, an integer between 1 and 21, is then re-entered to the INPUT subsystem at the next 0.5 second discrete interval, and the loop continues over the next cycle.

Pre-deployment Analysis

To verify the model described above, the Reactis and Uppaal tools were used. The tools were used to ensure that the global properties listed earlier in this section were covered satisfactorily.

Verification using Reactis.

The Reactis tool uses program flow analysis to automatically generate exhaustive test case suites for a Matlab model. These test cases can then be used to verify the behavior of the model. This verification is usually performed against a set of assertions and user-defined targets. Any undesirable property that is satisfied during model execution results in an assertion violation. On the other hand, a user-defined target must be reachable to ensure that a desired property is satisfied.

In order to verify the GPCA infusion pump model using Reactis, a couple of changes needed to be made to the model:

1. Reactis does not support a closed-loop model; thus the feedback loop was replaced by an equivalent open-loop input module. The inputs were chosen at random with an equal probability distribution, and configured randomly to generate an integer with a value between 1 and 21. The Matlab block 'ProbabilisticChooser' was replaced by a state machine emulating a human user.
2. Assertion blocks and user-defined targets were added to verify properties for the model.

Figure 6 shows a snapshot during verification in Reactis. The (top) left panel in the figure shows an assertion block with inputs defined as the current state and probability values for each event. The input probabilities are compared against the transitions active at

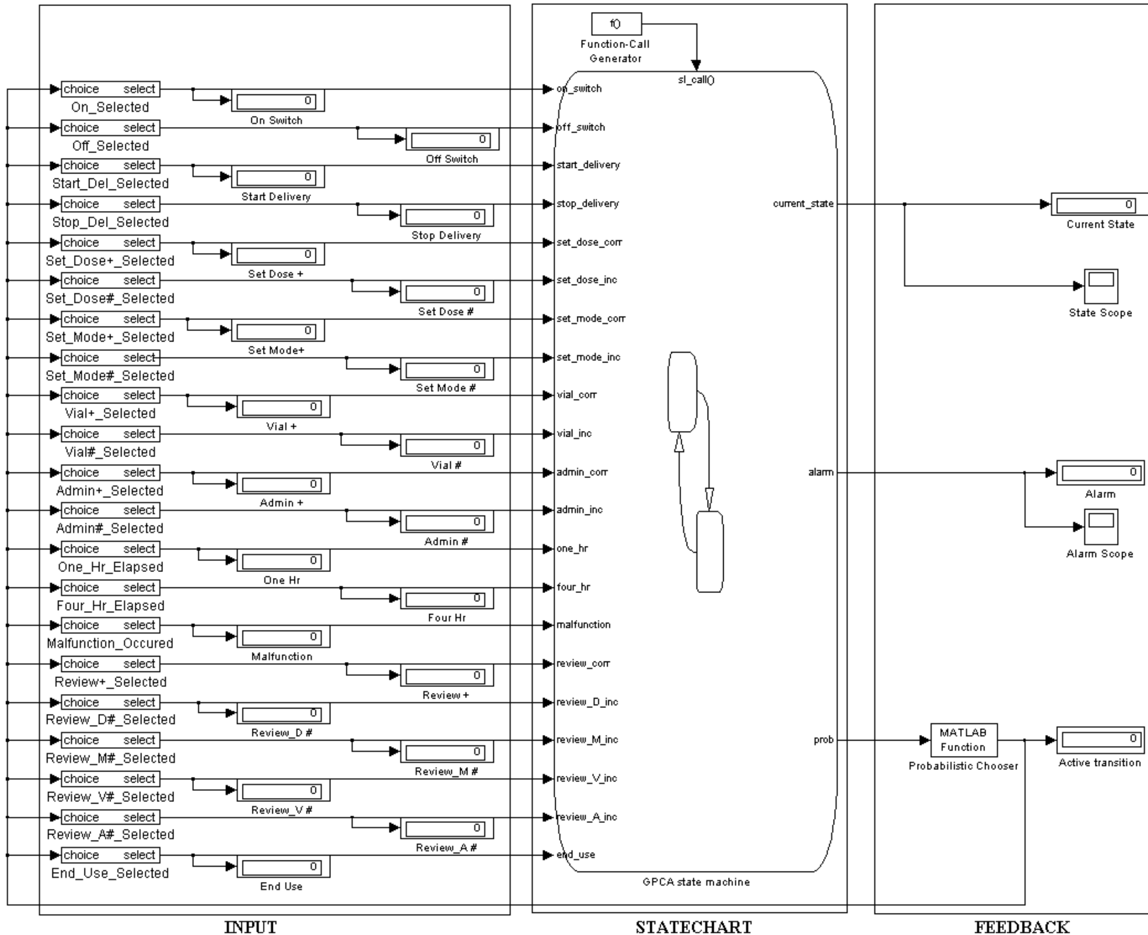

Figure 5. The GPCA model in Simulink

the current state. Any discrepancy between the two (e.g., a valid transition with a null probability value) is detected by the Relational Operator block and an assertion violation signaled.

The bottom panel in the figure shows the trace for a particular state (state 61). The valid transitions active at the state are listed in row 2 (each valid transition is represented by a 1 in the 21 element array). The corresponding probability values for the state are listed in row 3, and rounded off to the highest integer in row 4. The Relational Operator block compares the values in row 2 and row 4. The output of this block is listed in row 5. All 'false' entries in the list represent an inconsistency between transitions and event probabilities, and therefore correspond to an error in the system specifications.

Verification using Uppaal.

Reactis was very useful in detecting anomalies based on probability values. However, the test suites generated did not cover all states in the model. The best coverage achieved by a test suite generated by Reactis was only about 82%. A number of test cases needed to be input manually to ensure complete (100%) coverage. Moreover, being an open-loop model, Reactis could not correctly ensure reachability for all states in a closed loop mode. To compensate for these limitations, Uppaal was used to verify reachability and coverage properties for the GPCA infusion pump model.

The model defined in Uppaal was in essence similar to the 'GPCA state machine' block in the Matlab model and captured the behavior of the GPCA machine using states and transitions in a similar manner. Uppaal does not support real numbers however, therefore a simplification was made while assigning the probabilities for transitions. Each valid transition (i.e., a transition with a probability value greater than 0) was assumed equally likely. This assumption did not affect the reachability of the states, since any transition with a non-zero probability value must occur eventually.

The Uppaal model was ported directly from the Stateflow block and consisted of the same number of states and transitions. The coverage for the model was verified using Linear Temporal Logic (LTL) formulae that were evaluated against the state machine defined in the model. Figure 7 lists some typical LTL formulae that were verified against the GPCA model.

Forensic Analysis

The post-deployment analysis of the GPCA pump software was carried out as explained in Section 2. Errors in the implementation were detected by extracting abstract models using CWolf and verifying these models against temporal formulae in the CWB-NC. Labeled Transition Systems (LTSs) were generated to visualize these models in a human readable form, and to iteratively refine the criteria for abstraction.

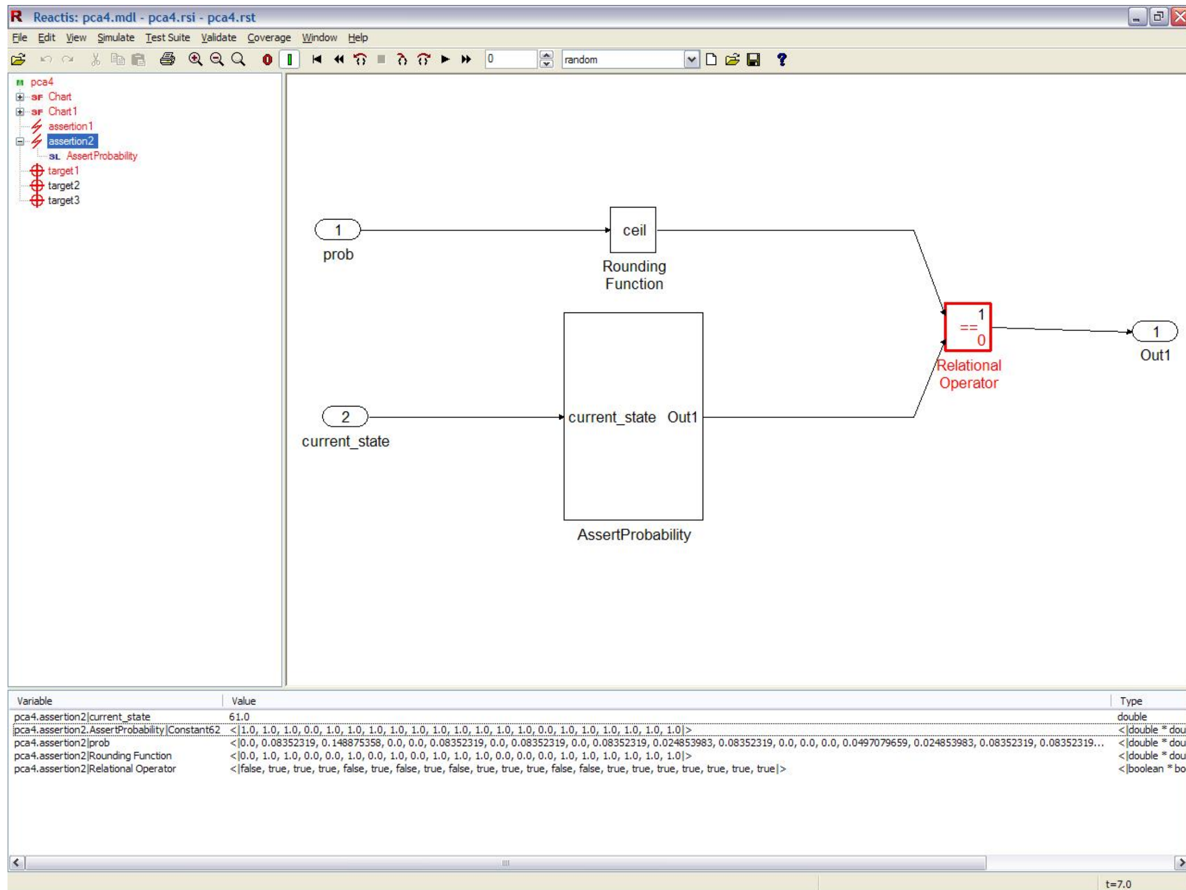

Figure 6. Verifying the GPCA model in Reactis

EF (pcaPump.State245)
State 245 in the PCA pump model can be reached.
EF (pcaPump.Trans1925 == true)
Transition 1925 must eventually be executed.
EG ($\forall$s State(s) $\wedge$ (pcaPump.s $\Rightarrow$ pcaPump.State1))
It is possible to visit the initial state in all possible paths.
AG ($\neg$ deadlock)
The system is deadlock free.
AF (pcaPump.State1)
At least one path has State 1 true all the time.

Figure 7. Sample LTL Formulae used in Uppaal

A major problem during this process was the large size of the LTSs generated. Since the LTS represents the control flow for the entire program, the number of states generated even for a moderately sized program could be very large (often in the region of several thousand states). Trying to comprehend and analyze such large systems would have been quite cumbersome and time-consuming. To overcome this problem, we made use of several optimizing algorithms provided by CWolf to generate manageable, succinct models. These algorithms included mainly variations of simulation and peephole refinements [DuV02]. Further reductions in the size of the LTS were achieved by using program slicing [15] on the source code to eliminate statements that did not impact the specified abstraction criteria.

Figure 8 shows an example LTS generated by CWolf optimized using the aggressive peephole refinement strategy. The LTS shown is abstracted with respect to the data variable *current_state*, as shown by the abstraction criterion in the figure. The original (non-optimized) LTS consisted of 47 states, which were refined to 8 states with 11 transitions.

5 Results and Observations

Errors Detected

A number of errors were uncovered through the pre-deployment and forensic analyses of the GPCA pump system. Most of these errors dealt with anomalies in the system specifications that were propagated to the code as well. In all, 46 errors were detected in the GPCA pump system. Five of these were found to be safety-critical errors that could cause potential hazard situations. The other 41 were less severe, but could still adversely affect the performance of the pump. Importantly, we were able to ensure that this list of errors was comprehensive, and that the model was correct for all other cases. The errors detected were classified in the following four categories, with category 1 errors being the most severe and category 4 errors the least severe:

1. **Null probabilities.** Four errors were detected due to null probabilities assigned to valid transitions. This meant that these events would never be executed, despite there being an active transition between the respective source and destination states.

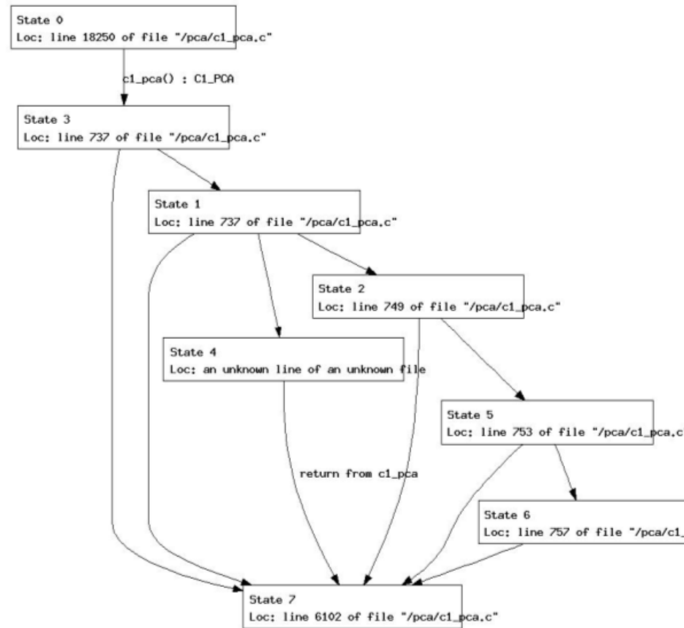

Figure 8. An LTS generated by CWolf

2. **Missing transitions.** There was one error due to a missing transition for a defined probability value. This could lead to a potential deadlock whenever the event with the associated probability would be executed.
3. **Missing self-transition probabilities.** There were 40 errors due to missing self-transition probabilities. These errors were more subtle, since they did not result in invalid transitions or unreachable states. They did, however, translate to exaggerated probabilities for other transitions originating at the given state and could lead to incorrect system behavior in the case of internally generated events such as timeouts at 1 hour and 4 hour intervals.
4. **Unreachable states.** One state was identified as unreachable. This was essentially a superfluous state, with no transitions leading to or from it to any other states.

All the errors uncovered in the system specifications during pre-deployment analysis could also be traced to the source code with the help of forensic analysis. This demonstrated that the two analyses techniques were commensurate and that formal methods based analyses could be used as an effective means for conformance checking of medical device software.

Evaluating the abstraction

The usefulness of an abstraction can be measured by its ability to produce distinguishable abstract models for subtle variations in the source (i.e., in the concrete domain). We evaluated the abstraction mappings used during forensic analysis of the GPCA software by comparing the models used to detect bugs in the source code with similar models generated from rectified code. The same abstraction mappings were used to generate both sets of models.

The corrections made to the software to mitigate the errors either introduced new events for existing structures, or modified existing properties for these structure. Using the notion of bisimulation equivalence available in the CWB-NC, we were able to distinguish the corrected models from the original models in 39 of the 45, or 86%, of the corrections made. The changes that could not be traced to the new models involved redundant transitions and states that were never executed or which lay outside the scope of the abstract interpretation defined (e.g., modification of existing floating point values).

Effort Comparison

The major bottleneck in conventional model checking (and correspondingly pre-deployment analysis) is the model building process. Most of the effort spent during pre-deployment analysis can be attributed to understanding the system specifications and building a formal model based on them.

Forensic analysis, by virtue of automatically extracting abstract models from the implementation does not suffer this bottleneck. Deriving abstract models from software is much more efficient in terms of effort spent per model. However, analyzing these abstract models using formal methods is a far more intricate process and requires considerable effort as opposed to model checking of pre-deployment specification models.

It should be noted that the forensic analysis approach is an iterative process, with abstract models being refined continuously with the help of LTSs and abstraction maps. The larger the program being analyzed, the higher the number of iterations would be. Using slicing and optimization helps reduce the cost of analysis, but the effort involved is still much greater than that for pre-deployment analysis. For the GPCA pump system, an average of 6.2 iterations

were required to trace each bug to the software.

As a result of this elaborate analysis process, the effort expended during forensic analysis is invariably much greater than that for pre-deployment analysis. Figure 9 shows the effort distribution for the GPCA case study spread over the entire analysis process. As can be seen from the figure, more than 60% of the effort was spent on modeling the GPCA system and refining of the abstract models. The total effort spent on forensic analysis was 209.4 person months, 47% more than that for pre-deployment analysis.

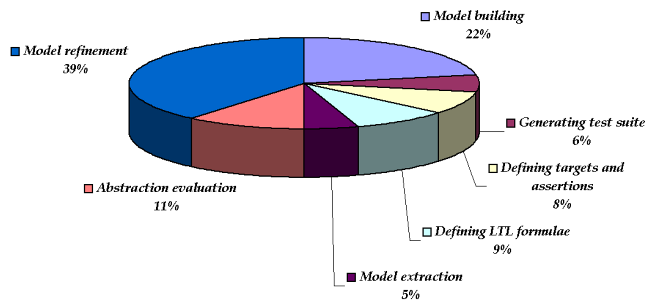

Figure 9. Effort distribution for the analysis of the GPCA system

It should be noted that the code used during forensic analysis for the GPCA pump was only a prototype. The effort required for real-world commercial software is expected to be even higher. However, this would still be a significant improvement over ad hoc reverse engineering and test-case analysis methods currently in use.

6 Conclusion and Future Work

We have proposed a methodology for carrying pre- and post-deployment analysis on medical device applications. We believe that a principled approach, based on construction of models from designs and implementations, could lead to a faster review of medical device applications. We have also provided experimental evidence to show the viability of our approach. Most importantly, our experimental evidence shows the technical approach of using abstractions is effective for the task on hand. However, a general acceptance of the proposed methodology by both FDA and by device manufactures would depend upon a number of political considerations, and great many studies that show the advantage of using formal-methods based tools. Thus, we, the co-authors of this paper, look upon this study as the first step in a long process of convincing everyone in the Software Engineering community, the FDA and device manufacturers of the potential of using formal-methods based techniques. Some of the problems, in our view, that need to be studied include the following (which is clearly not an all-inclusive list):

Extending the GPCA model. One of the aims of developing the GPCA pump model was to include it as part of a safety standard, to be used as a reference for conformance checking by all infusion pump manufacturers. Keeping this in mind, the GPCA infusion pump specifications were derived from various commercial infusion pump software. However, in order to make the specifications as generic as possible, several events and situations were abstracted and fit into the framework of 292 situations and 21 input events. The resulting model thus was too abstract to be of much use to the manufacturers as anything apart from a high-level design prototype.

To have a formal model as part of a more stringent standard, we need to elaborate the system specifications and build a more comprehensive model using these specifications as a basis. For example, the current model abstracts all dose inputs as 'correct dose' and 'incorrect dose'. In a real-world situation however, the dose would be a floating-point value set by the medic. The GPCA pump model would need to be modified to reflect this and would define different ranges for possible dose inputs. One way to incorporate this with the current model would be to add an external module representing the medic to non-deterministically generate these dose ranges and interface this module with the existing model. Similar modules could be added to reflect vial concentrations, bolus settings, etc. Another example would be to decompose all 'Alarm' states into 'Level 1' and 'Level 2' alarm states with different silence times associated with each.

Forensic analysis for commercial infusion pumps. The code derived from the GPCA infusion pump model served to provide a proof of concept for post-deployment analysis using abstraction. Though the analysis was reasonably successful, to truly assess the usefulness of the process, we need to perform similar (forensic) analyses using production software for real-world commercial infusion pumps. Error traces collected by end-users and logs recorded during in-house testing could be used to trace malfunctions to their source in the software. Moreover, these analyses would need to be performed on various software implemented in several different programming languages.

Automating the forensic analysis process. A major bottleneck in the post-deployment analysis process is the iterative process of manually inspecting the LTS and the refining the abstraction map. Though the manual intervention cannot be completely eliminated, the efficiency of the process could be greatly enhanced if this refinement iteration were to be automated. One way to achieve this automation would be to use automated program slicing techniques to refine the LTS. A mapping could be provided to derive criteria used for slicing from the abstraction criteria, while the slice itself could be constructed from abstract models. An interactive GUI could be provided to aid visualization and navigation through the abstracted models, further expediting the analysis process.

7 References

[1] General principles of software validation; final guidance for industry and fda staff, Jan 2002.

[2] Model-based testing and validation of control software with reactis. Technical report, Nov 2003.

[3] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[4] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[6] Daniel C. DuVarney. *Abstraction-Based Generation of Finite State Models from C Programs*. PhD thesis, 2002.

[7] Daniel C. DuVarney and S. Purushothaman Iyer. C wolf - a toolset for extracting models from c programs. In *FORTE '02: Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, pages 260–275. Springer-Verlag, 2002.

[8] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, pages 235–239. Lecture Notes in Computer Science 2648, Springer-Verlag, 2003.

[9] Stephan Merz. Model checking: A tutorial overview. In F. Cassez et al., editor, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, Heidelberg, 2001.

[10] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[11] Cleve B. Moler. MATLAB user's guide. Technical report, University of New Mexico. Dept. of Computer Science, November 1980. This describes use of Classic Matlab, the prototype for the very-much expanded professional Matlab from The MathWorks. Classic Matlab is no longer available.

[12] S. J. Prowell. Jumbl: A tool for model-based statistical testing. In *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9*, page 337.3, Washington, DC, USA, 2003. IEEE Computer Society.

[13] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 394–397, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[14] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[15] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.