Critical Feature Analysis of a Radiotherapy Machine

Andrew Rae[1], Daniel Jackson[2], Prasad Ramanan[2], Jay Flanz[3], Didier Leyman[4]

[1]Information Technology and Electrical Engineering
University of Queensland, St Lucia, QLD Australia
arae@itee.uq.edu.au

[2]Laboratory for Computer Science
Massachusetts Institute of Technology, Cambridge, MA
dnj@mit.edu

[3]Northeast Proton Therapy Center
Massachusetts General Hospital, Boston, MA
flanz@hadron.mgh.harvard.edu

[4]Ion Beam Applications
Louvain-La-Neuve, Belgium
leyman@iba.be

Abstract The software implementation of the emergency shutdown feature in a major radiotherapy system was analyzed, using a directed form of code review based on module dependences. Dependences between modules are labelled by particular assumptions; this allows one to trace through the code, and identify those fragments responsible for critical features. An 'assumption tree' is constructed in parallel, showing the assumptions which each module makes about others. The root of the assumption tree is the critical feature of interest, and its leaves represent assumptions which, if not valid, might cause the critical feature to fail. The analysis revealed some unexpected assumptions that motivated improvements to the code.

1 Introduction

A key difficulty in the analysis of large software systems is the isolation and evaluation of critical source code. Ideally, safety critical requirements would be implemented by safety critical modules, neatly isolated from the non-critical code. In practice, the safety of a system is tightly bound to its correct operation, and a single safety feature requires the cooperation of many modules.

This paper reports on our experiences analyzing the source code of a radiotherapy machine. We concentrated on a single feature of the software—the emergency stop function. As expected, we found that reasoning about this function required us to make assumptions about the behaviour of other parts of the system. Inspecting the tree of assumptions produced by our analysis exposed some conditions under which the software might not behave as intended.

Our analysis strategy is simple to understand and easy to apply. It is based on a new model of dependences, in which dependences between modules are qualified by specifications. A module A is said to S-use another module B if A relies on B to provide a service described by the specification S. Making specifications explicit in this way, and associating them with dependence arcs rather than modules, is a small elaboration of standard module dependency diagrams. It has major practical ramifications, however, since it allows us to trace dependences in a more fine-grained manner, to identify the code responsible for particular features of the system.

Our paper describes the context of the system (Section 2); an overview of the analysis approach (Section 3), and its application to the case study (Section 4); an evaluation of the results of the analysis (Section 6); and a comparison to related work (Section 9).

2 Context of the Therapy Control System

The Northeast Proton Therapy Center (NPTC) is a new radiation therapy facility associated with the Massachusetts General Hospital in Boston. It is one of only two hospital-based facilities in the United States to offer treatment with protons (rather than electrons or X-rays). Proton beams require much more elaborate and expensive equipment to produce, but can be more tightly conformed, and cause less damage to surrounding tissue. They are thus more suitable for treatments in sensitive areas such as the eye, and for the treatment of tumors in the brains of children, for which collateral damage has more serious long-term consequences. The center occupies a new building adjacent to the hospital, and began treating patients in the fall of 2001.

The Software Design Group in the MIT Lab for Computer Science began a collaboration in April 2002 with NPTC and Ion Beam Applications, the developers of the system, to investigate better methods for the development of safety critical software. The NPTC system would be used as a challenging example of a modern and complex medical device for the purposes of research; in turn, the results of the research would be used where appropriate to improve the safety and reliability of the system.

The NPTC installation has at its core a cyclotron that generates a beam of protons. The beam is multiplexed amongst several treatment rooms, each with its own gantry and nozzle for positioning the beam. Technicians in a master control room supervise the cyclotron and direct the beam to the allocated treatment room. Each treatment room is paired with a treatment control room, in which clinicians enter and execute treatment prescriptions. The patient is placed on a couch which is electromechanically positioned by staff within the treatment room. The beam delivery nozzle is also positioned, and its aim verified by staff using X-rays and lights attached to the emitter. The staff then leave the room, and the treatment is initiated and controlled from the treatment control room. Treatment consists of irradiating a specific location on the patient using a beam of protons with a defined lateral and longitudinal distribution.

The machine is considered safety critical primarily due to the potential for overdose—that is, treating the patient with radiation of excessive strength or duration.

The International Atomic Energy Agency lists 80 separate accidents involving radiation therapy in the United States over the past fifty years [15]. Software was implicated in the failures of the infamous Therac-25 machine [10], and more recently in similar accidents in Panama [2].

The NPTC system was developed in the context of a sophisticated safety program. Unlike the Therac-25, the NPTC system makes extensive use of hardware interlocks, including a hardware relay system and a redundant PLC-based system which handles safety critical functions, both of which run in parallel with the software control system. Video cameras inside the control room allow the technicians to view internal mechanisms, including a beam stop that can be inserted to isolate the treatment room from the cyclotron. The software itself is instrumented with abundant runtime checks, including a software 'heartbeat monitor' to ensure continued operation of critical processes. A detailed system-level risk analysis was performed. The software implementation was heavily tested, and manually reviewed against rigorous coding standards.

2.1 Therapy Control System

The software of the system, called the *Therapy Control System* (TCS), is written primarily in C, and is installed on commodity workstations running Unix, a commercial messaging system (Talarian's SmartSockets), a centralized Oracle database, and Motif X-windows. Low-level control is implemented in assembler on a VME crate running VXWorks. About 250,000 lines of C code is organized into a few hundred modules.

The TCS handles the storage and retrieval of patient data; entry and editing of prescriptions; scheduling of treatments and maintenance; patient positioning; and beam delivery. In concert with the hardware and PLC safety systems, it is designed to help manage three main hazards: a physical collision between moving parts of the system and a patient or staff member; accidental irradiation of a patient or staff member; and inaccurate radiation of a patient (including overdoses and underdoses). In its most critical features, such as the emergency stop feature analyzed here, the software is backed up by redundant hardware.

NPTC's acquisition strategy follows an evolutionary model. The goal is to have in place a safe working system, and then to expand the functionality and enable the addition of new modes of treatment. The analysis in this paper is based on Version 1 of the software, which has been in use since 2001. Version 2 is currently undergoing planning and design.

2.2 System architecture

A view of the system architecture is shown in Figure 1. The *Human/Computer Interface Layer* consists of a graphical user interface, implemented as a collection of definition files, listeners, and so forth. The *Application Layer* is the core of the system, and contains most of the code. It is divided somewhat arbitrarily into four modules: *System Management*, which controls user sessions, operational modes, and event reporting; *Beam Management*, which handles allocation and operation of the proton beam; *Treatment Management*, which handles the patient treatment sequence

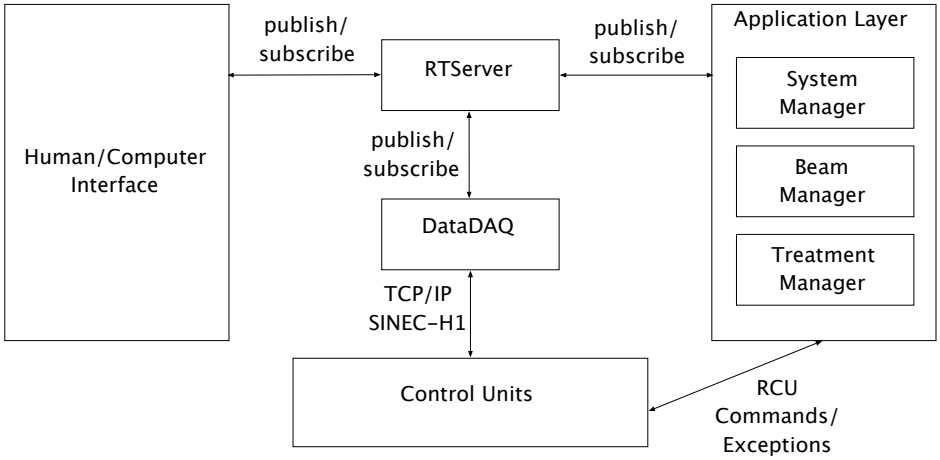

Figure 1: Architecture of Therapy Control System

from prescription to irradiation; and *Database Management*, which provides functions to allow the other modules to access the TCS database. The *Control Unit Layer* contains the drivers for the physical devices; these are implemented in a table-driven fashion as low level state machines.

From a communications perspective, the core of the system is a commercial messaging system (Talarian's *SmartSockets* [18]). The control units communicate with application-level processes using various protocols, including RPC, TCP/IP and SINEC-H1, depending on the type of hardware involved and the nature of the message. The other modules communicate through the server using a publish/subscribe mechanism that it provides: modules subscribe by registering callback procedures against particular events, and when an event occurs, the server calls all registered procedures.

3 Dependence Analysis

Our approach is based on a simple dependence model, which we outline here, but which is described more fully elsewhere [7,8]. The analysis involves a traversal through the dependence graph of the code, which generates, as a byproduct, a tree of assumptions. Examination of the tree may reveal flaws in the system, in which critical features are found to depend on unwarranted assumptions.

3.1 The Dependence Graph

The dependence structure is represented as a graph. An example is shown in Figure 2. Modules are represented as nodes. The module name is shown as the upper-most label inside the node; the other annotations are explained below. An edge labelled S from module A to module B says that A has a dependence on B mediated by the specification S. In other words, to fulfill its specification, A relies on B, but only to the extent that it satisfies S.

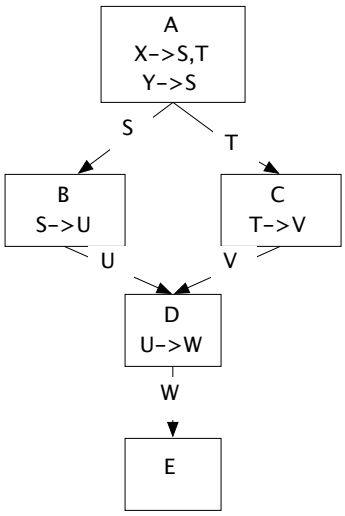

Figure 2: Sample dependency diagram

A *module* is just a syntactic unit of executable code; in a C program, for example, it might correspond to a file. Source code units that are not executable—such as header files in C and interfaces in Java—are not regarded as modules.

A *specification* is, more generally, a description of an assumption that a module may make about its environment that is discharged by another module. In the simplest case, it corresponds to a distinct service, associated with some subset of the procedures or methods of a module. But it may also correspond to assumptions that are not typically viewed as specifications, such as that some procedure within the module is called with a certain frequency, or that a global variable of the module is set in a certain way. A specification does not necessarily call for actions to occur; it may say that certain actions do *not* occur. We shall talk of the provision of services rather than the discharging of assumptions, since the words flow more easily, but the reader should bear in mind that the more general notion is always implied.

A module may use any number of other modules. One module may use another module under multiple specifications; it may use several other modules under the same specification (if, for example, the service it requires is discharged at different times by different modules); and a module may itself be used under multiple specifications by different modules.

In general, then, a module offers many specifications (its exports), and makes use of other modules via many specifications (its imports). A given export will usually not require all the imports. To express this, we can record internal dependences for a module between its imported and exported specifications, writing

```
A: X -> S, T
```

for example, to say that module A's provision of X relies on A's use of modules that provide both S and T. On the other hand, we may have that

```
A: Y -> S
```

so that A's provision of Y relies on A's use of a module that provides S. In the diagram, these internal dependences are shown inside the relevant nodes.

The internal dependences of a module make a more fine-grained dependence analysis possible. In this case, we can see that in providing the service Y the module A does not actually rely on the module C. And likewise, since the internal dependences of module D show that it provides V directly, without further demands on other modules, we can see that module C depends only on module D and not module E, even though there is path in the graph from C to E.

3.2 The Assumption Tree

Given a dependence graph, and particular service offered by a particular module, we can construct a tree showing all the services that contribute directly and indirectly to this service. These services represent the assumptions underlying the correct working of the system in providing this service.

At the root lies the critical feature; at the leaves are assumptions that cannot be decomposed further and must be evaluated on their own merits: they are either assumptions about the operational environment, or assumptions that are discharged locally, in their entirety, by modules. Note that the modules whose assumptions appear as leaves need not themselves be leaves in the dependence graph: a module may discharge some assumptions directly, but delegate other assumptions to further modules.

Suppose the service of interest is X at module A. We label the root of the tree A:X. Using the internal dependences of A, as described above, we find the services on which X depends. Here, these services are S and T. Now from the dependence graph we determine that these are provide by modules B and C respectively, say. This gives us two new nodes of the tree, which we label B:S and C:T.

This process is continued until the leaves of the tree represent modules that provide the required services with no further dependences (or, in the case of a circular dependence, until the same service is encountered a second time). For the diagram of Figure 2, we obtain:

```
1       A:X
1.1       B:S
1.1.1       D:U
1.1.1.1       E:W
1.2       C:T
1.2.1       D:V
```

A module may depend on an assumption about the environment; by modelling the environment as a module of sorts, we ensure that the tree never has any dangling edges. A node labelled ENV:Z thus indicates that the environment discharges the assumption Z.

For constructing the assumption tree, regarding dependences as mediated by specifications is crucial. The successful provision of a service by some module often relies on other modules meeting only very partial specifications. Analysis of the emergency stop function, for example, reveals many cases in which a module re-

lies on a procedure call to another module, but does not demand that it meet its full specification. Instead, it depends only on the procedure call terminating, or not returning an error value. In contrast, an analysis of full correctness would require that a called procedure satisfies its full specification (which could therefore be left implicit), resulting in a much larger assumption tree.

3.3 Analysis of the Assumption Tree

The value of a dependence analysis lies not just in identifying and classifying assumptions, but in assessing whether those assumptions are reasonable. In evaluating assumptions for critical functions, the following criteria are applied:

- critical functions should not depend on the performance of non-critical functions;
- critical functions should be contained within limited and well-defined sections of the software;
- fail-safe functions should only be conditional if performing the function may be more dangerous than not performing the function; and
- where critical functions depend on reused or COTS modules, the fitness of those modules should be evaluated with respect to the critical functions.

4 Applying the Analysis: Emergency Stop

Our case study investigates one particular function of the Therapy Control Software, namely the *Emergency Stop* function. When a button, known as the *Crash Button*, is pressed in one of the control rooms, the system should insert a set of 'beam stops'. These block the beam from entering any of the treatment rooms. The system should also freeze motion of the equipment. Whilst hardware interlocks provide an alternate path for the Emergency Stop function, the control software is required to provide a redundant mechanism for the function.

Figures 3 and 4 show part of the dependence diagram and assumption tree for the emergency stop feature of the Therapy Control System. The full expansion of the tree of Figure 4 is shown in the appendix. The critical feature at the root node is 'Emergency Stop Works'. For emergency stop to work, it is assumed that the RTServer, the beam manager and the PCU will all behave in certain ways. These translate into services which must be provided by these modules. These modules in turn make assumptions about other modules which appear lower in the assumption tree.

4.1 Generating the Assumption Tree

The starting point for the analysis of the Emergency Stop function was to identify the boundary between the software function and its external environment.

Selecting the boundary of an analysis such as this is necessarily somewhat arbitrary. The boundary determines which modules are included in the dependence diagrams. All assumptions relating to modules included within the boundary should appear in the assumption tree.

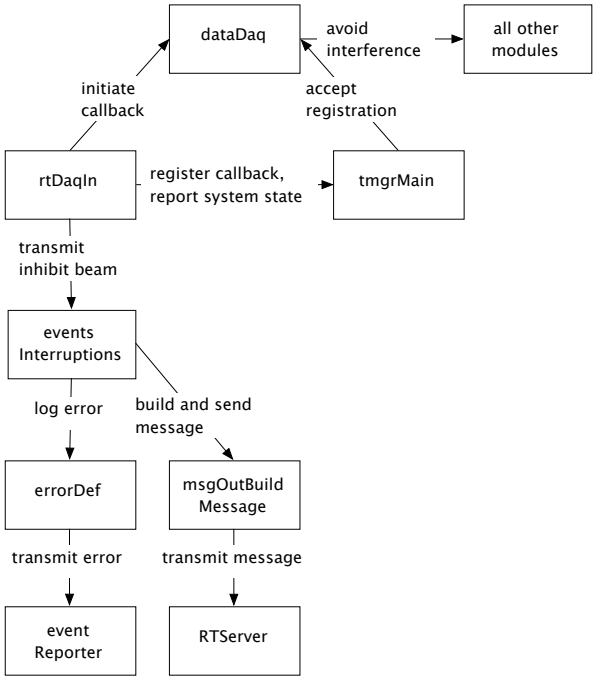

Figure 3: Dependency diagram for part of Therapy Control System

2 Entire program: Emergency Stop Works

2.1 rtDaqIn: Transmit INIHIBIT_BEAM. If RTServer receives a TMGR_GLOBAL_CRASH_PRESSED message, the Beam Manager will receive an INHIBIT_BEAM message.

2.1.1 tmgrMain: Register Callback. The Treatment Manager registers the rtDaqInputcallback procedure with rtdaq.

2.1.1.1 rtDaqIn: Accept Registration. rtdaq registers the rtDaqInputcallback.

2.1.2 tmgrMain: Report System State. tmgrMain reports the system state as anything but INITIALISATION.

2.1.3 rtdaq: Initiate Callback. rtdaq calls rtDaqInputcallback when a TMGR_GLOBAL_CRASH_PRESSED message is received.

2.1.3.1 Entire Program: Avoid Interference. No other module removes or changes the rtDaqInputcallback registration.

2.1.4 eventsInterruptions: Transmit INHIBIT_BEAM. If eventsInterruptions is called by rtDaqIn, eventsInterruptions transmits an INHIBIT_BEAM message.

2.1.4.1 errorDef: Log Error. If errorDef is called to log an error, it returns a value of TRUE.

2.1.4.1.1 eventreporter: Transmit ERROR. If eventReporter is called to transmit an error message, the message is transmitted, stored and displayed to the HCI correctly.

2.1.4.2 msgOutBuildMessage: Build And Send Message. If msgOutBuildMessage is requested to build and send an INHIBIT_BEAM message, it does so correctly.

2.1.4.2.1 RTServer: Transmit Message. SmartSockets transmits messages correctly.

Figure 4: Part of the assumption tree for the Therapy Control System

When the Crash Button is pressed, a signal is sent from the Control Unit (CU) to the dataDaq module (see Figure 1). The dataDaq module responds to the signal by transmitting a message labelled TMGR_GLOBAL_CRASH_PRESSED via the RTServer. The correct response to this message is for a set of commands to be sent to the control units stopping movement of the equipment and inserting the beam stops. For the purpose of the analysis, the function begins after a crash button message is received by dataDaq, and ends when commands are sent to the control units.

Our starting point for analysis is the module rtDaqIn, part of the RTServer of Figure 1. For rtDaqIn to be notified when a TMGR_GLOBAL_CRASH_PRESSED message to be generated, it depends on trmgrmain (the Treatment Manager) to initialise a callback, and dataDaq to notify the callback at the appropriate time. In turn, trmgrmain relies on dataDaq to register the callback. The dataDaq module relies on all other modules, since in order to guarantee that it will provide the service expected of it by rtDaqIn, it depends on all other modules not to delete or alter the callback registered by trmgrmain.

To proceed further, rtDaqIn relies on trmgrmain to report the system state. If this is INITIALISATION, the Emergency Stop function is disabled.

Once the callback has been called, rtDaqIn depends on eventsInterruptions to actually send the message to the Beam Manager. The first action eventsInterruptions takes is to log an error message. In order to complete the message send, it is not directly necessary for the logging to be done. However, since eventsInterruptions will not proceed until logging is complete, it depends on a procedure errorDef to terminate with a value of TRUE. To achieve this, errorDef depends on eventsReporter to correctly log the error message.

The eventsInterruptions module relies on msgOutBuildMessage to send the INHIBIT_BEAM message to the Beam Manager. The msgOutBuildMessage module in turn relies on the *SmartSocket* services provided by RTServer.

5 Practical Details

The first stage of the analysis involved the construction of meaningful dependency diagrams. To start with, cross references were generated through the use of a syntactic analysis, RedHat's Source Navigator [17]. These were processed with a combination of scripts and manual editing to produce raw dependency diagrams. The diagrams showed a link where-ever one module referred to a function or data structure contained in another module.

How easy such an analysis is to do depends heavily on how well lexical features of the code (such as the names of identifiers) capture semantic properties. In some cases, for example, two different constants were used for the same message type; this makes it very hard to locate code that reads messages of that type. This confirms Griswold's argument for the importance of transparency in lexical structure [3].

It was necessary to evolve the raw diagrams into our enhanced dependency graphs with specifications. This required extraction of specifications from the code of the modules. Extraction of the right set of specifications is key to the effectiveness of our method, but it is difficult to quantify the merit of a particular specification. A good specification is one which meaningfully describes a discreet service provided by the

module to other modules. For our analysis, specifications were written manually and in plain English after inspection of the code.

The second stage of the analysis required transformation of the dependency diagrams into assumption trees. This step was performed as described in Section 3.2, again using visual inspection to unfold the dependences to form the trees.

The third and final stage of the analysis was evaluation. Firstly, the assumption trees were inspected for reasonableness, as outlined in Section 3.3. Secondly, the assumptions were compared to the code to ensure that they were in fact satisfied.

6 Evaluation

Software safety analysis does not take place in a vacuum, but rather as an integral part of a system safety program which includes both software and hardware. The designers of the Proton Therapy System included an elaborate hardware backup, implemented using PLC technology. The emergency stop function is handled directly by the hardware, so any flaws in the software revealed by our analysis are likely to have no impact on the behaviour of the system. Nevertheless, the developers of the system take a prudent attitude, and are concerned to preserve the redundancy in order to maximize safety.

Through our analysis, four issues were revealed in the implementation of the Emergency Stop function, discussed in the following sections. These were presented to the developers. The first highlights the need for further analysis; the remaining three were determined to be previously unidentified issues requiring rectification. None of the issues found turned out to be artifacts of the documentation or the analysis.

6.1 Implementation of Callbacks and Message Queueing

The first observation made was that there were numerous dependences on the off-the-shelf RTworks modules. In most cases, such as in assumption 2.1.3, registration and activation of a callback procedure, these dependences reflected standard, well-tested use of the COTS modules.

However, as noted in assumption 2.1.3.1, RTworks does not prevent other modules from altering callback registrations, resulting in an open-ended dependency. The RTServer, as used in the Proton Therapy System, also does not provide a mechanism for prioritising messages. Thus, assumption 2.1.4.2, that eventsInterruptions transmits the INHIBIT_BEAM message, cannot be verified without identifying all other messages that might be queued for transmission, and demonstrating that they will not cause failure or excessive delays.

This problem can be classified as a result of violation of encapsulation for the third-party software. The dependency argument identifies and documents the behaviour expected from the RTworks modules. By comparing this with the actual behaviour of the modules, shortcomings can be noticed and addressed.

6.2 Logging before Acting

Correct operation of the Emergency Stop function is dependent on correct logging of an error message (assumption 3.3.3 in the appendix). This violates the principle that critical functions should not be dependent on non-critical functions.

Whilst the functional requirements for Emergency Stop and logging have been correctly identified, priority and precedence has not been assigned to the requirements. That is to say, there is no ordering of the importance of the requirements, or of the order in which required actions should occur. As a result, in some parts of the code an action is performed and then logged, whilst in other parts of the code an action is logged and then performed. There are some instances where this is quite appropriate—for example, in order to avoid overdose, it seems reasonable to avoid radiating the patient unless the radiation can be logged. For Emergency Stop, however, it is more important to stop the system than to record the reason why it was stopped.

Even if both functions were equally important, there is a more immediate temporal requirement to stop the system.

6.3 Unnecessary Checks

Analysis of the Beam Manager also revealed some interesting dependences. Most notably, a series of tests must be passed before the beam can be inhibited. These are described in assumptions 3.3.4 through 3.3.5.1 in the appendix. These checks violate the principle that fail-safe functions should only be conditional if performing the function is more dangerous than not performing the function.

It seems unlikely that the developers of the system were unaware of this principle. More likely, it was not articulated explicitly, and was thus not one of the criteria applied in code review. During normal operation, it is good defensive programming to check the consistency of data at regular intervals. These same checks should not, however, be applied to Emergency Stop. The fact that the crash button has been pressed implies that something has gone wrong, and that something may well result in, or be the result of, incorrect software behaviour. Emergency procedures should not depend on the system behaving normally.

6.4 The Special Case of the Extra Treatment Room

Assumption 3.3.4 (in the appendix) is potentially a serious design error that prevents Emergency Stop from operating if the beam is assigned to any treatment room other than Room 1 or Room 2. When we discussed this error with the software engineers responsible for the system, it became evident that the error had arisen through evolution of the code during development. The original code was only designed to handle two treatment rooms. This code was insufficiently generic, and so extension of the code resulted in the other treatment rooms being treated as special cases, through the `ExtraTreatmentRoom` modules. Creation of this special case required many subtle changes in the software, not all of which were made consistently.

7 Broader Considerations

The use of C as a programming language not only makes the dependence analysis harder – because of the lack of explicit interfaces, but also potentially compromises its results. Whereas in a safe language one can assume that the module dependences are those that are evident in the naming of resources, in a language such as C any violation of the memory discipline (such as an array bounds error) can cause one module to corrupt the computation of another. This vulnerability is addressed in the development of the Therapy Control System by extra vigilance in code review and testing. In the long term, a more productive and dependable approach might be to use a safe language. Praxis Critical Systems, for example, has made extensive use of a subset of Ada in its developments, which not only rules out certain errors ab initio, but also permits powerful static analysis to be performed that can expose subtle errors missed by a compiler [4].

The approach described in this paper is essentially a form of directed code inspection, driven by dependence information obtained from analysis tools. Code inspection is not of course a sufficient technique for establishing the safety of software. For critical features, it would be desirable to follow up the dependence analysis with an analysis of the correctness of the feature, reasoning over the code fragments identified by the analysis. Our analysis should therefore be viewed as the first, and not the last, step in a the verification of safety features.

8 Challenges and Benefits

The greatest challenge in applying our technique is establishing familiarity with the system under investigation. In the case of the Emergency Stop analysis, the evaluation team worked independently of those responsible for writing and maintaining the code. This independence is not essential, and the advantages gained are outweighed by the extra time and effort required to learn how the code works, with a resultant understanding which is at best imperfect.

With well-understood code and appropriate tool support, the analysis itself need not be particularly time consuming. The most difficult stage is extracting module specifications, but these are themselves a useful by-product of the analysis.

As well as the exposure of design issues, our technique facilitates deeper understanding of the mechanics of the design. It provides a documented record of how and by whom each module is used, which is a valuable resource for code maintenance.

9 Related Work

Our assumption tree has something in common with a fault tree. Unlike a fault tree, however, its nodes represent the successful provision of a service or function, rather than a failure. In a conventional fault tree analysis, the structure of the fault tree is not easily obtained from the system. Our assumption tree, in contrast, is directly extracted from the dependence diagram.

The assumption tree has no *and/or* structure, since a node of the tree represents all possible behaviours associated with a given assumption, rather than a behaviour in a particular state. This dramatically reduces the size of the tree; more direct applications of fault tree analysis to code which account for control-flow structure (such as [9]) seem unlikely to scale.

Software Failure Modes and Effects Analysis (SFMEA), as applied by Lutz [11], involves considering what can go wrong with each module, and the effects that this will have on each of the dependent modules. It can be viewed as the inverse of fault tree analysis, working bottom-up rather than top-down. Hazard and Operability Studies (HAZOP) [1] obtains similar results by listing the connections between modules, and classifying them with key-phrases such as 'too fast', 'too large', and 'incorrect order'. In this way it systematically explores the effects which each module can have on other modules. Each of these techniques can be considered to ask the question 'What might go wrong?', whereas our technique asks 'What must go right?'.

The idea of module dependences goes back to Parnas's seminal work [14]. Our dependence diagram differs crucially in the labelling of edges, and in the internal dependences that link a module's incoming and outgoing dependences, thus making it possible to trace the dependences associated with a particular subfunction. Most work on slicing [19] has been at the statement level (see, eg, [5]), although there have been efforts to apply slicing at the procedure [6] and component [16] levels. Wong discusses the phenomenon of the 'long thin slice' of source code related to any given hazard [20].

With appropriate tool support, the extraction of the tree could be made largely automatic. A static analysis that can account for aliasing and procedures-as-values is required; a type based analyzer such as Lackwit [13] may suffice, in combination with a means of lifting statement-level results to the module level (such as the reflexion model tool [12]).

Conclusion

The analysis we have presented is simple but effective. It can be conducted without extensive knowledge of the code, since the analysis itself highlights those parts of the code that are relevant. It is feasible without tool support, although tool support would make it less burdensome, and would make mistakes less likely. Even on a well-tested system, it exposed interesting issues. An analysis of this form should perhaps be included in standard code reviews for safety critical code.

Acknowledgments

This research was performed when the first author was a post-doctoral fellow in the Software Design Group at MIT. It was funded by grant 0086154 from the ITR program of the National Science Foundation, and by the High Dependability Computing Program from NASA Ames, cooperative agreement NCC-2-1298.

References

1. P. Fenelon and B. Hebbron. Applying HAZOP to software engineering models. *Risk Management And Critical Protective Systems: Proceedings of SARSS*. Altrincham, England. The Safety And Reliability Society. Oct. 1994. pp. 11–116.
2. Food and Drug Admininstration. *FDA Statement on Radiation Overexposures in Panama*. Available at http://www.fda.gov/cdrh/ocd/panamaradexp.html.
3. William G. Griswold. *Coping With Software Change Using Information Transparency*. Technical Report CS98-585, Department of Computer Science and Engineering, University of California, San Diego, April 1998 (revised August 1998).
4. Anthony Hall and Roderick Chapman. Correctness by Construction: Building a Commercial Secure System. *IEEE Software*, January/February 2002.
5. S. Horwitz, T. Reps and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*. Volume 12. 1990. pp. 26–60.
6. Daniel Jackson and Eugene J. Rollins. A New Model of Program Dependences for Reverse Engineering. *Proc. SIGSOFT Conf. on Foundations of Software Engineering*, New Orleans, December 1994.
7. Daniel Jackson. Module dependences in software design. *Monterey Workshop on Radical Innovations of Software and Systems Engineering in the Future*, Venice, Italy, October 2002.
8. Daniel Jackson. *Dependences and decoupling*. Lecture notes, 6170: Laboratory in Software Engineering. Dept. of Electrical Engineering and Computer Science, MIT, Sept. 2002, Available at: http://6170.lcs.mit.edu/www-archive/Old-2002-Fall/lectures/lecture-09.pdf.
9. Nancy G. Leveson, Stephen S. Cha, and Timothy J. Shimeall. Safety Verification of Ada Programs Using Software Fault Trees. *IEEE Software*. Vol. 8, No. 4. July/August 1991, pp. 48–59.
10. Nancy G. Leveson and C. Turner. An investigation of the therac-25 accidents. *IEEE Computer*. Vol. 7, No. 26, 1993, pp. 18–41.
11. Robyn R. Lutz and Robert M. Woodhouse. *Experience Report: Contributions of SFMEA to Requirements Analysis*. pp. 44-51. Available at http://citeseer.nj.nec.com/article/lutz96experience.html.
12. Gail C. Murphy, David Notkin, and Kevin Sullivan. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995, pp. 18–28.
13. Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. *Proceedings of the 1997 International Conference on Software Engineering (ICSE'96)*, Boston, MA, May 1997.
14. David Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, Vol. SE-5, No 2, 1979.
15. Robert C. Ricks, Mary Ellen Berger, Elizabeth C. Holloway and Ronald E. Goans. *REACTS Radiation Accident Registry: Update of Accidents in the United States*. International Radiation Protection Association, 2000.
16. Judith A. Stafford and Debra J. Richardson and Alexander L. Wolf. Architecture-Level Dependence Analysis for Software Systems. *International Journal of Software Engineering and Knowledge Engineering*. Volume 11, No. 4, 2001. pp. 431–451.
17. Red Hat, Inc. *The Source Navigator IDE*. Available at: http://sourcenav.sourceforge.net.
18. Talarian, Inc. *SmartSockets*. http://www.talarian.com/rtworks.html.
19. Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*. Vol. SE-10, No. 4, 1984, pp. 352-357.

20. Ken Wong. Looking at Code With Your Safety Goggles On. *Reliable Software Technologies: 1998 Ada-Europe International Conference on Reliable Software Technologies*. Lars Asplund, ed. Uppsala, Sweden,1998. Lecture Notes in Computer Science, Vol. 1411. Springer, 1998.

Appendix: Assumption Tree

3 Entire program: Emergency Stop Works
3.1 rtDaqIn: Transmit INIHIBIT_BEAM. If RTServer receives a TMGR_GLOBAL_CRASH_PRESSED message, the Beam Manager will receive an INHIBIT_BEAM message.
3.1.1 tmgrMain: Register Callback. The Treatment Manager registers the rtDaqInputcallback procedure with rtdaq.
3.1.1.1 rtDaqIn: Accept Registration. rtdaq registers the rtDaqInputcallback.
3.1.2 tmgrMain: Report System State. tmgrMain reports the system state as anything but INITIALISATION.
3.1.3 rtdaq: Initiate Callback. rtdaq calls rtDaqInputcallback when a TMGR_GLOBAL_CRASH_PRESSED message is received.
3.1.3.1 Entire Program: Avoid Interference. No other module removes or changes the rtDaqInputcallback registration.
3.1.4 eventsInterruptions: Transmit INHIBIT_BEAM. If eventsInterruptions is called by rtDaqIn, eventsInterruptions transmits an INHIBIT_BEAM message.
3.1.4.1 errorDef: Log Error. If errorDef is called to log an error, it returns a value of TRUE.
3.1.4.1.1 eventreporter: Transmit ERROR. If eventReporter is called to transmit an error message, the message is transmitted, stored and displayed to the HCI correctly.
3.1.4.2 msgOutBuildMessage: Build And Send Message. If msgOutBuildMessage is requested to build and send an INHIBIT_BEAM message, it does so correctly.
3.1.4.2.1 RTServer: Transmit Message. SmartSockets transmits messages correctly.
3.2 rtDaqin: Transmit pcuCrashStop. If RTServer receives a TMGR_GLOBAL_CRASH_PRESSED message, the PCU will receive a pcuCrashStop call.
3.2.1 eventsInterruptions: Transmit pcuCrashStop. If the eventsSafetyEvent procedure of eventsInterruptions is called, eventsInterruptions calls pcuCrashStop of the PCU.
3.3 beamMgr: Insert Beam Stops. If beamMgr receives an INHIBIT_BEAM message, the beam stops are inserted.
3.3.1 rtdaq: Accept Registration. rtdaq registers the beamConnMsgBeamActionCb.
3.3.2 rtdaq: Initiate Callback. rtdaq calls beamConnMsgBeamActionCb when an INHIBIT_BEAM message is received.
3.3.3 beamMgrErrorLib: Log Error. beamMgrErrorLib terminates and returns.
3.3.3.1 eventReporter: Transmit ERROR. If eventReporter is called to transmit an error message, eventReporter terminates and returns.
3.3.4 extraRoom: Report Beam Allocation. extraRoom reports that the beam is allocated to room 1 or room 2.
3.3.5 beamMgrTools: Report Beam Allocation. beamMgrTools reports that the beam is allocated to the room referred to by the INHIBIT_BEAM message.
3.3.5.1 Entire Program: Avoid Interference. No module changes the beam allocation between when the INHIBIT_BEAM message is constructed and when the allocation is checked by beamMgrTools.

3.3.6 beamControl: Insert Beam Stops. beamControl inserts the beam stops.

3.3.6.1 bsIoLib: Insert Beam Stops. bsIoLib inserts the beam stops.

3.3.6.1.1 ecubctu: Return. ecubctu does not generate an exception.

3.3.6.1.2 vxWorks: Transmit STOP. vxWorks transmits to the beam stop hardware.

3.4 PCU: Immobilise Gantry. If the PCU receives a pcuCrashStop call, the commands to immobilise the gantry will be issued.

3.4.1 pcuStateMgr: Immobilise All Movement. If pcuCrashStop is called, pcuStateMgr halts all gantry movement.

3.4.1.1 jogLib: Stop Jog Movement. If jogStop is called, jogLib stops jog movement.

3.4.1.2 pathLib: Stop Path Movement. If pathStop is called, pathLib stops path movement.

3.4.1.3 axisLib: Stop All Axes. If axisAllStop is called, axisLib stops movement on all axes.

3.4.1.3.1 macLib: Stop Mac Movement. If macStop is called, macLib stops mac movement.

3.4.1.3.2 steuLib: Stop STEU Movement. If steuStop is called, steuLib stops gantry movement.

3.4.1.3.3 sreuLib: Stop SREU Movement. If sreuStop is called, sreuLib stops snout movement.