Therac-25:
Will history repeat itself?

Ash Tyndall
20915779

October 22, 2014

CITS5502 Software Processes

Contents

Acknowledgements

Special thanks to Rick Kuhn for attempting to dig up further data relating to his co-authored papers [22, 23] that would have augmented the MAUDE data analysis of Chapter 6. Unfortunately the data in his possession did not have the necessary date information for my purposes, but his effort and quick response time relating to a paper that was published more than a decade ago is appreciated nonetheless.

CHAPTER 1

Introduction

In the early 1970s, two companies, Atomic Energy Canada Limited (AECL) and Compagnie General Radiographique (CGR), collaborated to build updated models of their core radiotheraputic offerings; Medical Linear Accelerators (LINACs). LINACs are an extension of the basic concept of a linear particle accelerator, repackaged for medical applications. They are are designed to create a beam of either electron or x-rays which can be focused on a very specific section of a patients body. These beams can be used to kill cancerous growths without severely damaging surrounding tissue.

Together, AECL and CGR develop two LINACs; the Therac-6 and the Therac-20, both of which were based on previous CGR work, but with mechanical control substituted with control via computer terminal. These machines are designed with accelerators that could produce 6 MeV x-rays and 20 MeV x-rays/electrons respectively. Still in partnership in the mid-1970s, AECL and CGR develop a new Therac that uses a new kind of linear accelerator, a "double-pass" system, which reduces the space requirements and allows cheaper parts to be used. AECL and CGR part ways soon thereafter, citing competitive pressures.

AECL proceeded with the development of this new "double-pass" system, the Therac-25 (so named for its 25 MeV x-ray and electron beam), continuing to develop the software-based control system, and passing the necessary regulatory requirements. The system is announced for public sale in 1983. One of the cutting-edge features of the machine is the removal of hardware-based interlocks and safeguards on the device, AECL instead opting to use a wholly software-based approach to ensure that the appropriate components are rotated in front of the raw electron beam to reduce the dangerous radiation to therapeutic levels.

In 1985, the first reported case of Therac-25 software safeguard failure occurred. Katherine Yarbrough, a breast cancer patient, receives an estimated 15,000–20,000 rads instead of the normal 200 rad range. She suffers severe radiation burns and shoulder and arm paralysis. A month later at a different facility, an unidentified female patient receives four separate overdoses totalling 13,000–17,000 rads within

the space of several minutes, due to a combination of safeguard failure and poorly explained error messages. This patient dies of radiation induced cancer some months later.

It was initially unclear to those who operated the Therac-25 that software errors were the cause of these overdoses. Due to the nature of radiation overdoses, the most serious of symptoms appeared days if not weeks later, causing the resulting deaths and disablements to be attributed to other factors. However, over time, it became clear to different system operators that something was seriously wrong with the Therac-25, sparking an eventual forced U.S. Food and Drug Administration (FDA) recall of the product with a total of six overdoses and two deaths.

Therac-25 was responsible for one of the most published deaths in radiotherapy, a profession that began some 35 years prior, and the confluence of death and computing caused an uproar at the time. Since then, the flames of anger have died down, and there is an opportunity to review the incident objectively. This report will examine the Therac-25 case in detail, trying to determine the answers to several important questions:

1. How does one design safe software, what does it involve, what are the pitfalls and how do we avoid them? (Chapter 3 on page 5)

2. What were the flaws of Therac-25 and how did failures on the part of AECL contribute to the creation of these flaws? (Chapter 4 on page 10)

3. How did standards and regulatory bodies respond to Therac-25 through new guidelines and processes for creation of "safe" medical device software? (Chapter 5 on page 15)

4. Have those guidelines and processes resulted in the creation of safer medical device software? (Chapter 6 on page 18)

5. Are there still areas in the medical landscape where regulation is insufficient? (Chapter 7 on page 21)

6. Will a disaster like Therac-25 happen again? (Chapter 8 on page 23)

CHAPTER 2

Literature Review

Therac-25 was one of the first widely reported software-related disasters, and as a result, a variety of academic work has been performed since the disaster investigating the specifics of the failure of Therac-25, as well as the development of safety critical medical software generally.

Primary work in the area of safety critical software development and system design as been done by Nancy Leveson. Her two books *Safeware* [11] and the more recent *Engineering a Safer World* [12] are highly influential works in the field. They discuss from a variety of disciplines the causes of safety-critical system failure, from poor software testing processes to a failure of system design to adequately inform the user of the system's state. Of particular relevance to our Therac-25 questions is *Safeware* chapter 6, "The Role of Humans in Automated Systems," which describes several models of human-computer interaction (HCI) and the necessary design principles to properly enable them. Additionally, *Engineering a Safer World* chapter 9, "Safety-Guided Design," which discusses several of the Therac-25 design flaws.

Leveson and Turner [13] have also contributed the primary academic report on Therac-25 which discusses the history of the Therac brand, the history of the companies involved, as well as the timeline of the disasters that ensued.

Various work has been done into the area of safety critical software from both the perspective of cause and prevention. Dunn [6] provides us with a helpful definition of safety in terms of "mishap risk," as well as examples of mishap causes.

In terms of cause, in *Failure in Safety-Critical Systems* [9, ch. 3] Johnson discusses in detail the sources of failure, touching upon a broad variety of failures including Regulatory, Managerial, Hardware, Software, Human and Team based failures. Besnard and Baxter [1] discuss two models of system failure, Reason's Swiss-cheese model and Randell's fault-error-failure model, both of which can be used to analyse the Therac-25 disaster.

In terms of prevention, Nolan [16] proposes several strategies to be considered when designing "safe systems of care," as well as methods to reduce "ad-

verse events" which may compromise patient health. Obradovich and Woods [17] perform an investigation of poor Human-Computer Interface design in a medical device, describing how the device is flawed and how both the user and medical supervisor can change their processes to cope with this. Lin, Vicente and Doyle [14] propose a new interface for a specific medical appliance that applies human factors engineering, a key part of the discussed prevention of user error, and provides data demonstrating the effectiveness of such an approach from error minimisation and efficiency perspectives.

Several works discuss the introduction of various international standards and regulations post-Therac-25. Rakitin [18] provides an overview of foundational standards in the risk management space of medical device software. Brown [2] provides an overview of specifically *IEC 61508: Design of electrical / electronic / programmable electronic safety-related systems*. Jordan [10] provides an overview of specifically *IEC 62304: Medical Device Software – Software Lifecycle Processes*.

To allow us to determine if regulation of medical device software has improved the situation, several works which analyse relevant data will be examined. Wallace and Kuhn produced two papers [22, 23] analysing a subset of FDA data relating to "adverse events" and provided statistics on the types of software errors and the medical domain of the devices, as well as information on how to prevent and detect these types of errors. The FDA Manufacturer and User Facility Device Experience Database (MAUDE) dataset [20] is also examined directly by the author to attempt to derive conclusions regarding the proportion of medical device software errors in the broader MAUDE database. Finally, *Failure in Safety-Critical Systems* [9, ch. 5] discusses the under-reporting of adverse events, as well as reporting bias.

With the literature available in the field, we can now review what safe software design is, and how to accomplish it.

CHAPTER 3

Safe software design: What and how?

The design of safe software (or as Leveson calls it, "safeware" [11]) is of paramount importance in the medical industry, as in that field software is frequently placed in direct control of equipment that has the potential to cause serious injury or death if operated outside safety margins. In the case of Therac-25, the machine was capable in certain failure states of emitting radiation that was able to cause death and permanent disablement.

It is difficult however to define what exactly "safety" is. Granted, we can say that a system is "safe" if there is absolutely no chance that anything could go wrong with it, however, such systems do not exist in the real world. Dunn [6] touches upon a useful concept to consider safety: A safe system is one with an acceptably low "mishap risk." Dunn uses the US Department of Defence definition of mishap; "an unplanned event or series of events that result in death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment," also suggesting that mishap risk deals with the impact or severity of a given mishap event, and the probability of it occurring. Another common team in the medical field that could be considered a synonym with mishap is "adverse event", which is more specifically and generally defined as "any undesirable experience associated with the use of a medical product in a patient" [21].

Since Therac-25, there has been much discussion regarding strategies to mitigate "adverse events" involving medical device software. In *Safeware*, Leveson argues that safe software design can only be achieved through creation of both a culture of safety, [11, ch. 4] and embedding "safeware" design principles into the software from the beginning [11, ch. 16]. She argues that it is not enough to identify how a system can go wrong, but also institute changes that make those occurrence less likely by design. A strong culture of safety also ensures that complacency regarding safety, safety-apathetic organisational structures and lack of technical skill in implementing and ensuring safety in product are not present, which Leveson argues are key causes of software disaster.

"Safeware" design principles concern themselves primarily with hazard reduc-

tion as a primary goal and consideration when designing software. Leveson notes that in many industries outside of software there exists a body of knowledge (i.e. standards, codes of practice and checklists) which provides a large degree of guidance to the design of physical products with safety considerations. However, such knowledge is present to a lesser extent in software, what knowledge that is present mainly concerns itself with concepts of readability and complexity, rather than "safe" architecture.

Within software, Leveson proposes four broad approaches to reduce accidents through design change. The first of these is hazard elimination, involving preventing hazards through design by making them physically impossible. Some possible hazard elimination methods include;

- **Substitution**: Change components with hazardous effects with others yhat lack those effects.
- **Simplification**: Simplify design such that hazards caused by complex interactions between components are removed.
- **Decoupling**: Reduce adverse effects of unexpected behaviour in a subsystem by making it only loosely coupled with other systems.
- **Elimination of specific human error**: Reduce obvious human error by making system status unambiguous and easy to understand.
- **Reduction of hazardous materials or conditions**: Ensure software only contains code necessary for core functionality, remove unused code.

The second of these is hazard occurrence reduction, whereby the probability of a hazard occurring is reduced. Methods include;

- **Design for controllability**: Provide feedback about system state while critical actions occur, and provide operator time to react to perceived problems with system.
- **Barriers**: Erect physical or logical barriers between safe and unsafe tasks. Unsafe tasks should not be easy to do accidentally.
- **Failure minimisation**: Ensure safety factors and margins, as well as redundancy.

The third of these is hazard control, through which the probability of a hazard leading to an accident can be reduced through a variety of methods;

- **Limiting exposure**: Critical software checks should be as close in execution as the code they protect, critical checks should also not be complimentary.
- **Isolation and containment**: The number of people who could be affected by a hazard should be reduced to as low as practical.
- **Protection systems and Fail-safe design**: Independent watchdog hardware can be introduced to ensure that a failed system is returned back to a safe state.

The final of these is damage reduction. Potential hazards need to be understood, and emergency plans and training to handle these hazards should be provided. Relevant people can then ensure that in the case of an hazard-caused accident, damage mitigating action can occur.

Safeware's proposed accident reduction techniques can be though of in terms of Besnard and Baxter's [1] integration of Reason and Randell's Swiss-cheese and fault-error-failure models, shown in Figure 3.1 on page 9. In this diagram we can see that the first three of Leveson's proposed accident reduction techniques serve as barriers through which adverse events cannot traverse. Though the introduction of faults in these layers, errors are able to arise that eventually become failures that cause holes in these barriers. If these barriers happen to be broken in the same area, it can lead to accidents, harm and loss of life. Damage reduction then serves to reduce the severity or quantity of this harm. Safe software design involves instituting processes that ensure that as little faults are introduced into these barriers as possible.

Lin [14] provides a study of medical device interfaces which we can apply the proposed principles within *Safeware* to investigate their effectiveness. In her study the user interface of a specific medical device, designed to administer specific quantities of a drug, was analysed against a proposed user interface which better applied Human-Computer Interface principles. This interface rearranged the design of the buttons used to input data, as well as changed the graphics displayed when navigating the device to give the user a much better sense of the implications of their actions, as well as the current mode of the machine.

This could be considered a form of hazard elimination due to design; by reducing the confusion between the user's actions and the results, hazards related to erroneously entered parameters can be greatly reduced. When statistically comparing the use of the old interface to the new, Lin noted an average user error reduction of 44%. Lin's changes to the interface were not drastic, but clearly demonstration how a safety orientated approach can be quite effective in preventing error.

In her later book *Engineering a Safer World* Leveson departs from the "broad overview of what is known and practiced in System Safety today" [12, p. xviii] present in *Safeware* to instead describe techniques that she considers new and innovative, approaching the state of the art in System Safety.

> The old safety engineering techniques, which were based on a much simpler, analog world, are diminishing in their effectiveness as the cause of accidents changes [12, p. xviii].

One of the key innovations Leveson champions are the System-Theoretic Accident Model and Processes (STAMP), which she argues shifts the paradigm of safety engineering from considering safety as a matter of preventing failure, but instead as a matter of enforcing the system's safety constraints. She contends that traditional failure models of simple cause and effect are not adequate for the complex computer controlled systems of today, as frequently accidents are caused by complex component interaction or "systemic causal mechanisms" [12, pp. 73–5]. She proposes three key parts of STAMP;

- **Safety constraints**: Events are not key to STAMP, but rather constraints, as events are effects, while a failure to meet constraints is the cause.
- **Hierarchical safety control structures**: Viewing the system in terms of a hierarchy of components enforcing constraints at lower levels helps more readily analyse where there may be insufficient constraints and points of failure.
- **Process models**: Systems should be broken down into process models, which contain the goal (a safety constrain), the action condition (how to accomplish), the observation condition (how to know it is working) and the model condition (how a human controller would understand the process.)

These key components of the broad STAMP accident model are expanded upon significantly within *Engineering a Safer World*, with further discussion on the specific applications, as well as implementation details.

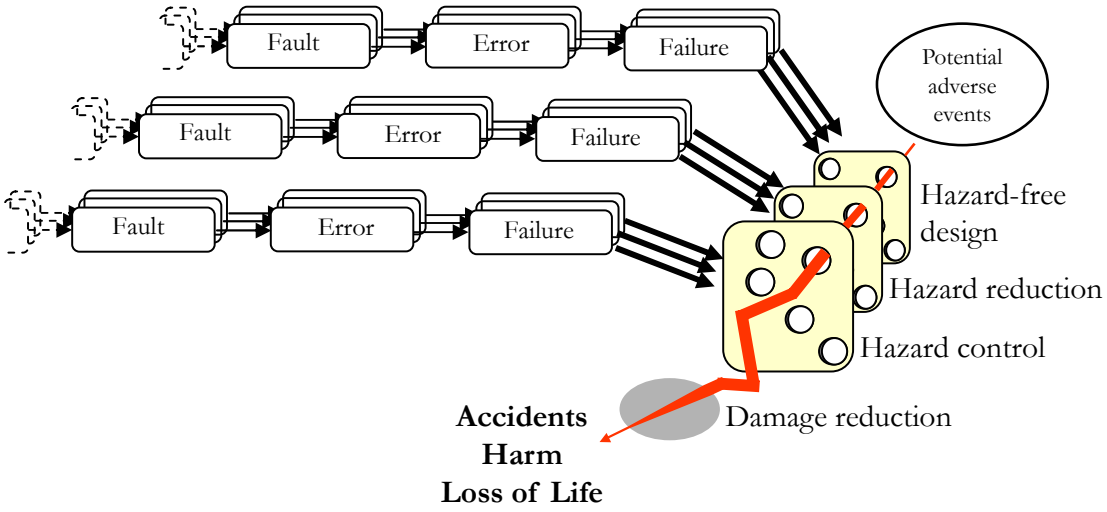

Figure 3.1: Model of *Safeware*'s hazard reduction (adapted from [1, fig. 6])

CHAPTER 4

Flaws and failures: How and why?

The Therac-25 incidents are generally used as a case-study in literature on safety-critical devices, as they serve as an illuminating example of a failure of software processes at multiple levels. As such, extensive investigation on the many facets of those failures exists, as well as a variety of research exploring the incident at multiple levels. Here we'll discuss some of the reasons Therac-25 failed from a management and process perspective, as well as touching upon how some failures of user interface design exacerbated these errors.

The Therac-25 interface was run through a DEC VT100 video terminal, which provided the operator with a rudimentary text interface (see Figure 4.1 on the next page) which they could navigate through the use of the arrow and Enter keys. In this interface they would enter the patient's name, treatment mode, beam type, beam energy, and they would enter the "prescribed" values, which would be compared to the "actual" values which are set manually by the operator within the treatment room itself. If the actual and prescribed values were within a certain tolerance of each other, "VERIFIED" would appear to the right of them. The operator could copy the actual value into the prescribed column (i.e. accept the "actual" value without retyping) by pressing the "Enter" key on the VT100.

The primary software error present in Therac-25 was a race condition. In normal operation the program would wait 8 seconds when the beam mode was changed, as during that time physical elements of the machine were moving into place, leaving the beam improperly covered by either the X-ray or the electron beam dispersal elements for the duration of that time. Because of energy losses in the beam spreading apparatus, the raw power of the beam was set much higher in electron mode to have comparable effects to X-rays. If the operator were to quickly change from X-ray to electron type the 8 second wait would not be enforced. If the beam was activated in less than 8 seconds after a type change, the apparatus would still be rotating, and proper implement would not be in place, thus severely irradiating the patient with the unintended full energy of the beam.

Specifically, the bug was triggered, thus causing an overdose, by the completion

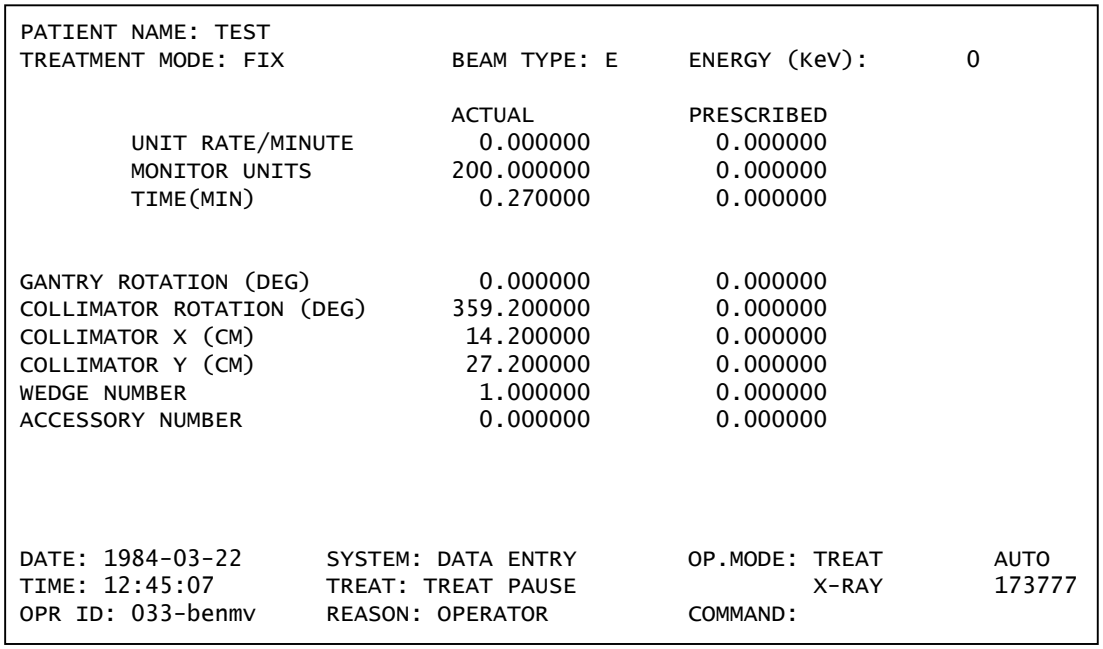

Figure 4.1: Therac-25 interface (adapted from [15])

of steps 3–4 of the below in less than 8 seconds;

1. Press “X” to select X-ray in the Beam Type field.
2. Press the “Up” arrow to re-position the cursor on the Beam Type field.
3. Press “E” to change the selection to Electron mode.
4. Press “Enter” a number of times to submit all other values unchanged and trigger the dispensing of radiation.

The ultimate failure of Therac-25 was the lack of safety as a primary consideration in the software development and design process. As described by Leveson and Turner [13] in their investigation of the incident little documentation was kept regarding the specifications of the software running Therac-25, nor on the testing processes performed to ensure that the safety of the system was maintained. Software testing was only considered as part of broader whole-system tests, which would make complex software flaws much more difficult to find and fix. Furthermore, evidence indicates the software was developed by only one individual, leading to important questions regarding the amount of oversight that existed over the Therac-25 software quality and design. It is evident that AECL’s management

either did not posses the will and/or the competence to ensure the thorough testing of the Therac-25 system, and a emphasis safety goals generally, with studies indicating that such an emphasis is one of the strongest indicators of a safe system [12, p. 415].

One of the serous process issues uncovered as part of the Therac-25 investigation was the fact the software used in Therac-25 was partially ported from the prior Therac-6, which while architecturally similar, crucially differed in that it contained electromechanical safeguards in addition to software based ones. With AECL's track record with documentation and adequate software testing in serious question, the Therac-6 software was unlikely to be bug free and developed with a standard safety-critical software development methodology. In that sense, AECL should have considered the Therac-6 code to be Software of Unknown Pedigree (SOUP). The concept of SOUP is a recent addition in medical standards, which discourage software developers from using such software as due to its lack of proper testing, it cannot be assumed to be reliable or trustworthy for a safety-related task, nor can it be trusted not to interfere with software that does perform such a task. It is recommended in such a case that additional testing and verification is performed on such software, which did not appear to occur in AECL's case. It was later determined that one of the primary bugs responsible for the Therac-25 incident was also present in Therac-20, but due to the hardware safeguards it merely caused the machine to shut down, suggesting the bug may have had genesis in this use of SOUP.

Another serious issue with process was the near complete exclusion of software from the core safety analysis of the Therac-25 system. To quote AECL's final report;

> Programming errors have been reduced by extensive testing on a hardware simulator and under field conditions on teletherapy units. Any residual software errors are not included in the analysis [13, p. 4].

In that same report, AECL assigns extraordinarily low probabilities to errors such as "Computer selects wrong mode" (4×10^{-9}). This betrays AECL's attitude towards software safety, and their failure to consider the dangers of using SOUP within their products.

It is also illuminating to analyse the Therac-25 disaster from the perspective of software design as a whole. In *Safeware*, Leveson [11] provides several useful frameworks through which to analyse both a user and a computer's role within a given system. These are the human as monitor, backup and partner models. The most applicable of these to Therac-25 would be "human as monitor;" a system in which the computer itself is responsible for the executive actions in the system,

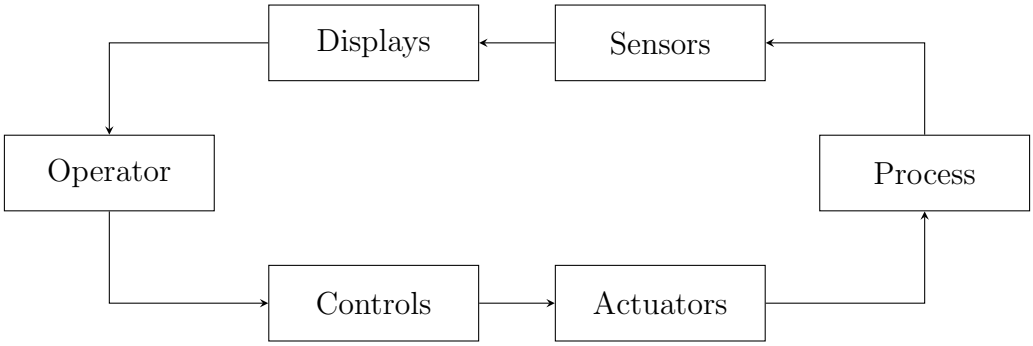

Figure 4.2: Synthesised perception model (adapted from [12, fig. 4.2])

and the human serves a supervisory role ensuring that the process is error free. This may seem at odds when considering the entire "directive" intelligence of the system being still vested in the human, however the actual execution of the actions are the sole domain of the computer system. Leveson goes on to discuss how in such a system, it is important for the human to be sufficiently aware of the system's state to be an effective monitor; "[t]he operator is dependent on the information provided; it is easy to provide too much or too little information or to display it poorly" [11, p. 114]. This is very similar to the "synthesised perception" model (see Figure 4.2) Leveson proposes in *Engineering a Safer World* [12, pp. 77–9], which she argues can cause a failure of safety constraints if the perception model provided does not provide appropriate information in an understandable way. Therac-25 failed to provide appropriate levels of information to the human monitor in several ways;

1. Error messages were vauge with description, with the details contained within unexplained numerical codes, making it impossible for the operator to know specifically what was wrong. E.g. "HTILT 1".

2. Both dangerous and benign failures were represented by the same error code, habituating operators to ignore the messages.

3. Hardware designed to detect level of radiation dispensed had narrow range of detection, and did not report radiation levels exceeding that range to the operator.

Therac-25 is also a useful case study in poor HCI design. The system, for reasons of safety, required that parameters be entered multiple times, and if these entries mismatched, then an error would display and the treatment would not proceed. However, this ended with complaints from operators due to its arduous

nature, thus it was augmented with a system whereby the user could press Enter to confirm that a parameter set was correct. While efficient, this had the net result of habitualising the operators into pressing Enter in quick succession, thus reducing the safety gained from requiring careful consideration of the parameters [12, p. 274].

A similar HCI issue involved the "Treatment Pause" functionality, which allowed an operator to continue with treatment of a patient after receiving an error message up to five times before necessitating a system reboot. Due to the aforementioned poor error messages, this simply lead to the habitualisation of operators resuming treatment in the case of any error, as in the vast majority of circumstances, this would have no ill effect [12, p. 301].

Ultimately it is difficult to pin down who at AECL was responsible for the lack of appropriate safety culture, as subsequent lawsuits and regulatory investigations disincentivise parties with information from speaking out. However, due to the scale of the incident and the public nature of those lawsuits that did occur, enough information does exist to point to broad causes for the failure. The incident provoked intense media interest at the time, which lead to regulators reconsidering how they managed medical device software safety.

CHAPTER 5

Regulatory response: New standards

In September 1987, two years after the first recorded Therac-25 radiation overdose, the FDA announced that they intended to introduce new regulation to prevent the failures of software quality management that had been present in AECL from re-occurring [8]. The following two decades saw the slow introduction of a variety of US national and international standards to deal with the various facets of medical device software safety. These regulations are not necessarily easily unified. To quote one attempt; "with so many different standards, regulatory guidance papers and industry guides on RM [risk management], the task of collating this information into a usable model is itself daunting" [3]. Regulation is highly complex, thus to reasonably restrict scope, only prominent and relevant United States national and international regulation from standards bodies and agencies will be covered.

Before the introduction of the first international standards, it was the sole job of the FDA to regulate medical device hardware within the US. Information from this period is difficult to find, however, Leveson does mention the introduction of mandatory reporting by health-care facilities of adverse events in 1990 [13, p. 13]. This added to the preexisting requirements of manufacturers and importers to report such events.

One of the first international standards that was discussed as a potential contender for the field is *IEC 61508: Functional Safety of Electrical / Electronic /*

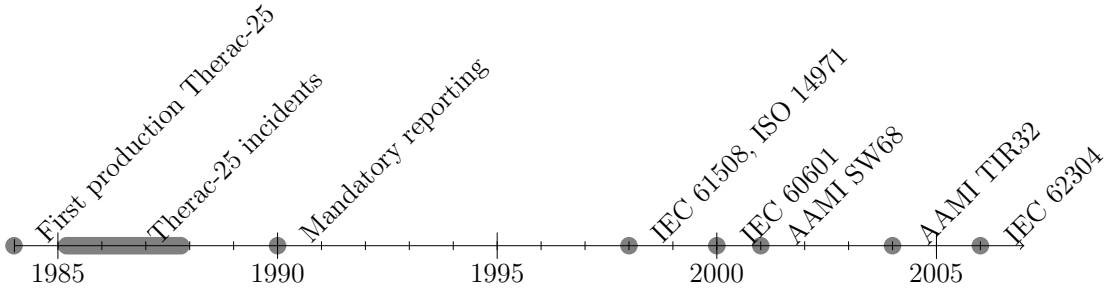

Figure 5.1: Regulation timeline

Programmable Electronic Safety-related Systems, the first portions of which were published by the International Electrotechnical Commission (IEC) in 1998. This standard provides a general set of standards creating a safety life-cycle that could be adapted for specific industries. Notably, unlike AECL, it recognised that software risk cannot be reduced to zero, it can only be reduced, and as Leveson suggests, safety considerations must be present at every stage of the development lifecycle [2]. Jordan [10] suggests however that the application of this standard to the medical area is difficult, as the area inherently has high risk, and the standard is most effective when dealing with low risk environments.

In the same year, the International Standards Organisation (ISO) released their own standard specifically relating to medical device risk management; *ISO 14971: Medical devices – Application of risk management to medical device*. One of the key features of this standard was the requirement of evidence to exist in documentation that mitigation of risk in software was present, one of the key failings of AECL [18].

In 2000, a revision to parts 1–4 of *IEC 60601: Medical Electrical Equipment* was released, which assisted in defining risk management more clearly, and adopted a definition of "mishap risk" that recognised both the severity and the probability of an issue [18].

In 2001, the Association for the Advancement of Medical Instrumentation (AAMI) in association with the American National Standards Institute released *AAMI SW68: Medical device software – Software life cycle processes* which extended the risk management attributes of ISO 14971 into a whole software development life cycle [18]. Burton, Mc Caffery and Richardson [3] suggest that both ISO 14971 and AAMI SW68 were deficient in that they required risk management processes to be in place, but did not provide any specifics as to the form such a process would take.

In 2004, *AAMI TIR32: Medical Device Software Risk Management* was published, which aimed to address the aforementioned deficiencies of ISO 14971 and AAMI SW68 by clarifying the risk management processes [18]. In particular, it provided clear definitions of reliability and safety, and importantly how they differ;

> Reliability is the ability of a system to perform its required functions under stated conditions for a specified period of time. Safety is the probability that conditions (hazards) that can lead to a mishap do not occur, whether or not the intended function is performed. Reliability is interested in all possible software errors, while safety is concerned only with those errors that cause actual system hazards [19].

Finally, in 2006 the IEC published the most recent standard in the area, *IEC*

62304: Medical device software – Software life cycle processes, which specifies those processes, activities and tasks that are necessary for the creation of dependable and reliable medical device software. Huhn and Zechner note however that no specific models nor methods are prescribed to accomplish these tasks, the manufacturer must instead argue that their own processes allow for the creation of such software that fits the standard's criteria [7]. Jordan suggests that in the industry, such processes, activities and tasks are already core to reputable manufacturers, thus the lack of prescription is welcome to them [10].

CHAPTER 6

Data analysis: Are we safer?

While the discussion of regulation is useful, it is hard to say to what extent that regulation has resolved the safety issues in medical device software that Therac-25 demonstrated. Have regulators appropriately determined what is necessary to ensure these bugs are minimal? Is it possible to create software that has sufficiently low "mishap risk" in this realm?

One data set that could potentially provide answers to these questions is the FDA's MAUDE, a massive database of adverse events reported to the FDA available for download on their website. This dataset extends from the year 2000 to the current day.

MAUDE data consists of information relating to each adverse event, including the manufacturer involved, the device involved, the category of adverse event and a free-form text field which contains further description from the manufacturer about the nature and specifics of the adverse event.

Using MAUDE, it may be possible to measure the proportion of issues manufacturers have reported as being related to medical device software in some form. From those statistics, it may be possible to derive trends, such as if device software is becoming more or less safe over time. The most useful data attribute for such a purpose is clearly the category; the FDA provides an extensive (nearly 1,000) list of categories, to one of which each adverse event can be assigned.

Because of the sheer size of MAUDE and time constraints, manual analysis of data is not feasible, thus automated analysis techniques will be investigated. The analysis technique on MAUDE chosen involved determining a list of categories considered relevant to medical device software adverse events, and then querying the database to determine what proportion of adverse events each year were contained within those categories. The categories chosen for that purpose are shown in Table 6.1 on page 20.

The database was analysed, with the proportion of software events never becoming greater than 0.4%. Figure 6.1 on the following page shows the proportion of incidents of the above categories within the larger database. Upon analysing

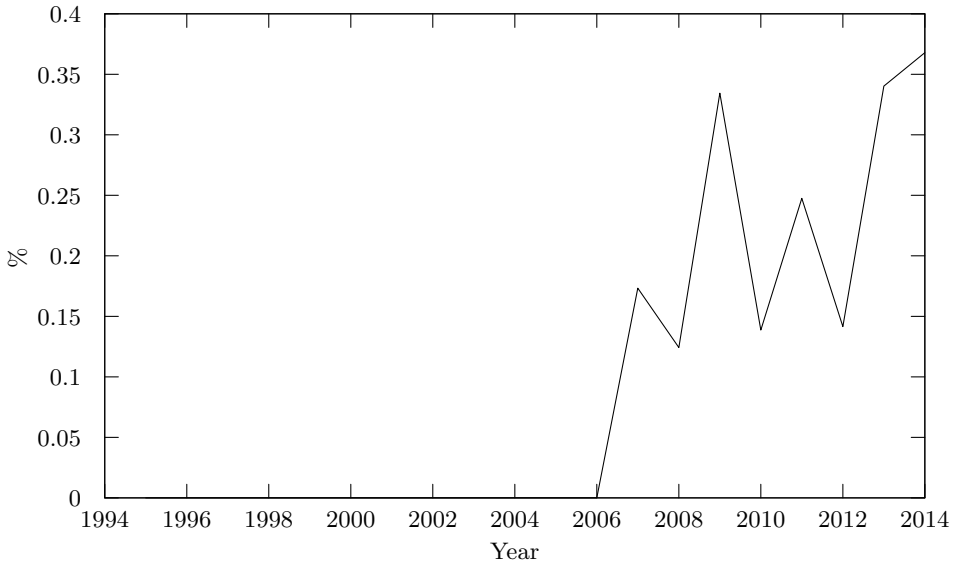

Figure 6.1: Proportion of incidents in "software related" categories over time

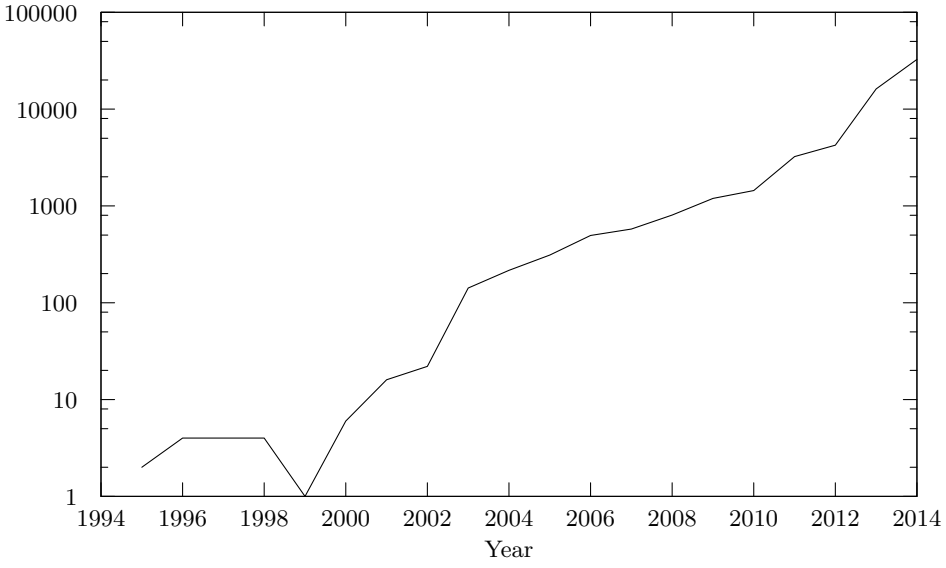

Figure 6.2: Number of reports in MAUDE over time (log scale)

- Computer failure
- Computer hardware error
- Computer software issue
- Incorrect display
- Error or warning message, failure to produce
- Power calculation error due to software problem
- Incorrect software programming calculations
- Algorithms, inconsistent
- Semiautomatic code, failure to override
- Year 2000 (Y2K) related problem
- Date-related software issue
- Application network issue
- Application program issue
- Application program version or upgrade problem
- Application security issue
- Computer operating system issue
- Computer system security issue
- Data back-up problem
- Loss of Data
- Operating system becomes non-functional
- Operating system version or upgrade problem
- Problem with software installation
- Programming issue

Table 6.1: "Software related" categories in MAUDE

the data in this database, it is difficult to come to justifiable complete conclusions about these proportions.

Before 2007 there are no adverse events corresponding to the aforementioned categories in MAUDE, suggesting that either those categories were either non-existent, or the introduction of categories entirely was introduced in that year. From 2007–12, there are less than 10 reports matching those medical device software categories, suggesting that those categories are not readily used. 2013–14 shows significantly more items in these categories, but those years do not alone provide enough data points to draw any conclusions.

Wallace and Kuhn [23] performed a previous study of this area with a manually classified non-public set of similar FDA data. They found that from 1983–91 approximately 6% of all incidents could be considered to be software related, with this increasing in the years 1994–96, to 11%, 10% and 9% respectively. This is massively at odds with the data collected from the public MAUDE database, suggesting that the categorisation technique used is insufficient. Furthermore, it appears that the publicly accessible data records within MAUDE is only a recent phenomenon, with Figure 6.2 on the previous page showing how the public data present per year increases from nearly zero to over 30,000 in the period covered.

CHAPTER 7

Regulatory gaps: What's next?

As discussed in the prior section, it is easily seen that regulation does not address problems with emerging technologies with necessary speed. Crumpler and Rudolf [4] discuss a draft policy released by the FDA in 1989 designed to clarify and deal with the introduction of medical device software as components of sold products, or as accessories of those products: Such software would have the same regulations applied to it as a "parent" product unless it is specifically and separately classified. The FDA defines accessories as software which mediates data input/output between the user and the medical device.

An interesting and relevant type of software in this area is radiation therapy treatment planning software, which allows medical professionals to import various scan data from a patient, and constructs a "virtual patient" thorough which they can review the patient's situation and construct a high level plan to treat their illness. This high level plan can then be deconstructed by the software into a series of less complex radiation exposure specifics. Such software has risen in prominence as computational power has increased to allow such complex simulations to be feasible.

Such software is of particular interest as in some cases such software is developed in-house, so that it may handle the specific needs of a medical centre. While under the 1989 policy the FDA would still require such software to be regulated, as it does meet the definition of an accessory of a radiotherapy device, it becomes much more difficult for the FDA to do so effectively within its framework. The FDA's main information gathering is contained within the "pre-market notification" requirements, in which software creators are required to notify the FDA before selling a product; this may never occur in the case of in-house solutions, making it difficult for the FDA to know about such development in the first place. Similarly, the FDA's main power exists in being able to take defective products "off the market" as it were, a measure that is not necessarily relevant or effective in the the case of in-house software.

It is difficult to measure how many accidents occur due to in-house software,

however, some studies do exist on such accidents. Within the last decade France made changes to legal reporting obligations surrounding ionising radiation that included medical applications, Derreumaux et al. [5] describe several accidents that were results of in-house radiation treatment planning medical device software error that can be analysed.

In the first case described, the software provided with the medical centre's LINAC was unable to handle situations involving a particular method of altering the beam. In-house software to simulate that circumstance was used, with the results of that simulation exported to the LINAC's Record & Verify system. During the export process, the R&V system was not properly configured to accept the data input format provided, thus misinterpreting the data and setting a critical parameter in that mode to zero. This default value resulted in a radiation overdose of $> 20\%$ over the course of the treatment, which lead to the patient's effected issue undergoing necrosis.

In the second less severe case, a new type of LINAC procedure was introduced which was encoded into a centre's in-house radiation therapy treatment planning software. It used a reference dose rate calculation that did not correctly take into account the differences in dosage caused by beam divergence in the new procedure. Due to these error, it is estimated hat around 4,000 people were exposed to an overdose ranging from 3%–7%.

Both of these cases were a result of software bugs that stringent testing procedures should have found and mitigated. To quote Derreumaux et al.;

> The first accident throws light on the problem of in-house software: many radiotherapy centres use such handcrafted software, which are not standardised and may not be thoroughly checked before clinical use [5].

In this sense, it appears that in-house software may be the next "wild-west" of medical software, having far less regulation ensuring that appropriate software development and risk management practices are observed. It appears likely that regulation in this area will lag behind for some time, but eventually effective processes to prevent incidents such as this will be enacted first on national, then international levels.

CHAPTER 8

Conclusion: Will history repeat?

The science of safe software has developed significantly since the Therac-25 incident, with a broad understanding now present as to the pitfalls of software development that can lead to safety critical software products that lack the proper emphasis on safety in their development cycle. Leveson's *Safeware* discusses in detail the various aspects of this, and here we touch upon those related to process that were particularly relevant in the case of Therac-25. We can view Leveson's proposed accident reduction techniques through the useful adaptation of a unified failure model, as proposed by Besnard and Baxter, as well as viewing her more recent and innovative STAMP as another more modern lens through which to analyse the complex causal safety relationships in safety-critical software projects today.

The exact causes of Therac-25 have been thoroughly studied by the safe software development community, with the root cause of Therac-25's most deadly bug being a race condition that triggered under very specific circumstances. Why that bug existed in the first place was a failure of AECL to properly contemplate and integrate safe software creation principles into their development efforts. AECL reused software from previous efforts, but did not adequately and separately test the software in the changed circumstance of no hardware interlocks; they instead showed little regard for the role software bugs could play in safety critical software. The user interface of Therac-25 also failed by habituating a variety of negative operator behaviours, such as disregarding error messages and continuing treatment after receiving error messages. Further research regarding how interfaces on more recent LINACs differ and avoid Therac-25's HCI pitfalls, as well as those that still persist in industry, would assist in determining what improvements can or have been made in this area.

Regulation in the medical device software area was partially driven to the forefront due to the publicised nature of the Therac-25 incident, however despite this clear and present danger of software, national and international regulation has only trickled in slowly over the subsequent decades. These standards are also complex, interwoven and difficult to easily follow, with a variety of standards

bodies publishing different standards addressing different aspects of the process at different times. The most recent standard in the area appears to be *IEC 62304: Medical device software – Software life cycle processes* which was released for the first time in 2006.

Determining if these regulations have helped curb adverse events relating to software is a difficult task. One way to do this is to analyse the freely available data present in the FDA MAUDE database and consider the proportion of categorised data that was software related. While this avenue was initially promising, it appears that the dataset is poorly categorised for such a purpose, and manual categorisation would be outside of the scope of this report. The quantity of public data available each prior year also decreases significantly, making it difficult to draw statistically significant conclusions more than a few years in the past. Further study with a fully categorised and larger dataset would yield more informative results, particularly in conjunction with a broader analysis of the interaction between aforementioned regulations, as well as a further history of FDA regulation in the area.

One area that regulation may not adequately cover is that of in-house software accessories to LINACs and similar devices. While FDA recommendations as early as 1989 indicating that this software should be subject to regulation, the FDA is not well placed to perform such regulation, as the FDA's main methods of control relate to the restriction of sale of products, which is not relevant in the case of in-house software. Instances of in-house software bugs, potentially resulting from poorly regulated software development, are discussed, with overdoses similar to those caused by Therac-25 resulting. Further research into how regulation is beginning to handle this area, and what is necessary to ensure safety within it is, will become increasingly important as in-house software grows in influence and use.

When analysing regulation pre and post-Therac-25, one can clearly say that the regulatory environment has improved, and that such regulation should help reduce those instances of adverse events relating to medical device software. However, the snails pace at which these regulations were introduced is very worrying, as the technological landscape can change far more quickly than regulators can respond. The rise of in-house software is one instance where regulation on this issue is not necessarily sufficient to ensure patient safety.

Therac-25 was an tragic disaster that helped highlight the necessity of safe software design processes. From Therac-25, many changes were instituted to try and prevent similar incidents from occurring. Thus the question remains; will history repeat itself? When viewing the state of in-house software today in conjunction with insufficient regulation in the area, the landscape is eerily reminiscent of that

which preceded the Therac-25 disaster, indeed, we have already seen accidents in this area that were a result of poor practices. The question then becomes not *will* history repeat itself, but rather *has* history already done so? Unfortunately, it appears the answer is "Yes."

APPENDIX A

MAUDE data analysis

Data retrieved from [20] on 20 September 2014.

A.1 Raw results

Year	Software Reports	Total Reports	Percentage
1977[1]	0	1	0
1988[1]	0	1	0
1991[1]	0	1	0
1995	0	2	0
1996	0	4	0
1998	0	4	0
1999	0	1	0
2000	0	6	0
2001	0	16	0
2002	0	22	0
2003	0	142	0
2004	0	216	0
2005	0	311	0
2006	0	495	0
2007	1	577	0.173310225
2008	1	805	0.124223602
2009	4	1196	0.334448161
2010	2	1442	0.138696255
2011	8	3232	0.247524752
2012	6	4239	0.141542817
2013	55	16163	0.340283363
2014	120	32621	0.367861194

[1] Excluded due to lack of continuity.

A.2 Processing code

```
# Written for Python 3.4
import csv, pprint, datetime, random

incidents = {}
incident_problem = {}
incident_text = {}
key_resolve = {}
code_problem = {}
problem_code = {}

computer_related = [
  'Computer failure',
  'Computer hardware error',
  'Computer software issue',
  'Incorrect display',
  'Error or warning message, failure to produce',
  'Power calculation error due to software problem',
  'Incorrect software programming calculations',
  'Algorithms, inconsistent',
  'Semiautomatic code, failure to override',
  'Year 2000 (Y2K) related problem',
  'Date-related software issue',
  'Application network issue',
  'Application program issue',
  'Application program version or upgrade problem',
  'Application security issue',
  'Computer operating system issue',
  'Computer system security issue',
  'Data back-up problem',
  'Loss of Data',
  'Operating system becomes non-functional',
  'Operating system version or upgrade problem',
  'Problem with software installation',
  'Programming issue',
]

date_resolve = [
  'DATE_OF_EVENT',
  'DATE_FACILITY_AWARE',
  'DATE_MANUFACTURER_RECEIVED',
  'DATE_REPORT_TO_MANUFACTURER',
  'DATE_REPORT_TO_FDA',
  'DATE_REPORT',
  'REPORT_DATE',
  'DATE_RECEIVED',
]
```

```
print('Reading in data...')
with open('deviceproblemcodes.txt', newline='') as csvfile:
  mdrreader = csv.reader(csvfile, delimiter='|', quoting=csv.QUOTE_NONE)

  for code, name in mdrreader:
    code_problem[int(code)] = name
    problem_code[name] = int(code)

with open('foidevproblem.txt', newline='') as csvfile:
  mdrreader = csv.reader(csvfile, delimiter='|', quoting=csv.QUOTE_NONE)

  for report, code in mdrreader:
    if code.isdigit():
      incident_problem[int(report)] = int(code)

with open('foitext.txt', newline='') as csvfile:
  mdrreader = csv.reader(csvfile, delimiter='|', quoting=csv.QUOTE_NONE)

  for report, _, _, _, _, text in mdrreader:
    if not (len(text) < 10 and ("(B)(4)" in text or "(B)(6)" in text)):
      incident_text[int(report)] = text

with open('mdrfoi.txt', newline='') as csvfile:
  mdrreader = csv.reader(csvfile, delimiter='|', quoting=csv.QUOTE_NONE)

  skip_first = True
  for row in mdrreader:
    if skip_first:
      key_resolve = {y : x for (x, y) in enumerate(row)}
      skip_first = False
    else:
      report_key = int(row[key_resolve['MDR_REPORT_KEY']])
      adverse_event = row[key_resolve['ADVERSE_EVENT_FLAG']] == 'Y'
      product_problem = row[key_resolve['PRODUCT_PROBLEM_FLAG']] == 'Y'

      if report_key not in incident_problem:
        continue

      err_code = incident_problem[report_key]

      # The FDA data has dates of varying quality, find the best one
      year = 0
      for r in date_resolve:
        date_str = row[key_resolve[r]]
        if date_str == '':
          continue

        date = datetime.datetime.strptime(date_str, '%m/%d/%Y')
```

```
          year = date.year

          if year > datetime.datetime.now().year or year < 1965:
            continue

          break
        else:
          print('Could not determine date format!')
          raise Exception

        if year not in incidents:
          incidents[year] = {}

        if err_code not in incidents[year]:
          incidents[year][err_code] = []

        incidents[year][err_code].append(row)

sincidents = sorted(incidents.items())

comp_text = set()
non_comp_text = set()

print("Processing...")
with open('stats2.csv', 'w', newline='') as resfile:
  csvw = csv.writer(resfile)

  for year, edict in sincidents:
    comp_count = 0
    all_count = 0
    computer_related_ids = [problem_code[x] for x in computer_related]

    for ecode, records in edict.items():
      if ecode in computer_related_ids:
        comp_count += len(records)

      for r in records:
        key = int(r[key_resolve['MDR_REPORT_KEY']])
        if key in incident_text:
          if ecode in computer_related_ids:
            comp_text.add(incident_text[key])
          else:
            non_comp_text.add(incident_text[key])

      all_count += len(records)

    csvw.writerow([year, comp_count, all_count])
```

Bibliography

[1] Besnard, D., Baxter, G., et al. Human compensations for undependable systems. Tech. Rep. CS-TR-819, University of Newcastle upon Tyne, 2003. Accessed: 2014-09-20, URL: `http://www.dirc.org.uk/publications/techreports/papers/12.pdf`.

[2] Brown, S. Overview of IEC 61508. Design of electrical/electronic/programmable electronic safety-related systems. *Computing & Control Engineering Journal 11*, 1 (2000), 6–12.

[3] Burton, J., McCaffery, F., and Richardson, I. A risk management capability model for use in medical device companies. In *Proceedings of the 2006 international workshop on Software quality* (2006), ACM, pp. 3–8.

[4] Crumpler, E. S., and Rudolph, H. FDA software policy and regulation of medical device software. *Food & Drug LJ 52* (1997), 511.

[5] Derreumaux, S., Etard, C., Huet, C., Trompier, F., Clairand, I., Bottollier-Depois, J.-F., Aubert, B., and Gourmelon, P. Lessons from recent accidents in radiation therapy in france. *Radiation protection dosimetry* (2008).

[6] Dunn, W. R. Designing safety-critical computer systems. *Computer 36*, 11 (2003), 40–46.

[7] Huhn, M., and Zechner, A. Arguing for software quality in an IEC 62304 compliant development process. In *Leveraging Applications of Formal Methods, Verification, and Validation*. Springer, 2010, pp. 296–311.

[8] Jacky, J. Programmed for disaster. *The Sciences 29*, 5 (1989), 22–27.

[9] Johnson, C. *Failure in Safety-Critical Systems: A Handbook of Accident and Incident Reporting*. University of Glasgow Press, 2003.

[10] Jordan, P. Standard IEC 62304-medical device software-software lifecycle processes. In *Software for Medical Devices, 2006. The Institution of Engineering and Technology Seminar on* (2006), IET, pp. 41–47.

[11] Leveson, N. *SafeWare: System Safety and Computers*. Computer Science and Electrical Engineering Series. Addison-Wesley, 1995.

[12] Leveson, N. *Engineering a Safer World: Systems Thinking Applied to Safety*. Engineering systems. MIT Press, 2011.

[13] Leveson, N. G., and Turner, C. S. An investigation of the Therac-25 accidents. *Computer 26*, 7 (1993), 18–41.

[14] Lin, L., Vicente, K. J., and Doyle, D. J. Patient safety, potential adverse drug events, and medical device design: a human factors engineering approach. *Journal of biomedical informatics 34*, 4 (2001), 274–284.

[15] MIT Department of Electrical Engineering and Computer Sciences. Therac-25 Hands-On Assignment, 6.033 - Computer System Engineering, 2007. Accessed: 2014-10-20, URL: `http://web.mit.edu/6.033/2007/wwwdocs/assignments/handson-therac.html`.

[16] Nolan, T. W. System changes to improve patient safety. *BMJ: British Medical Journal 320*, 7237 (2000), 771.

[17] Obradovich, J. H., and Woods, D. D. Users as designers: How people cope with poor HCI design in computer-based medical devices. *Human Factors: The Journal of the Human Factors and Ergonomics Society 38*, 4 (1996), 574–592.

[18] Rakitin, R. Coping with defective software in medical devices. *Computer 39*, 4 (2006), 40–45.

[19] Software Risk Management Task Group, AAMI Medical Device Software Committee. Medical device software risk management. Tech. Rep. TIR32, Association for the Advancement of Medical Instrumentation, 2004.

[20] U.S. Food and Drug Administration. Manufacturer and User Facility Device Experience Database (MAUDE), 2014. Accessed: 2014-09-20, URL: `http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/PostmarketRequirements/ReportingAdverseEvents/ucm127891.htm`.

[21] U.S. Food and Drug Administration. What is a Serious Adverse Event?, 2014. Accessed: 2014-09-20, URL: `http://www.fda.gov/safety/medwatch/howtoreport/ucm053087.htm`.

[22] Wallace, D. R., and Kuhn, D. R. Lessons from 342 medical device failures. In *High-Assurance Systems Engineering, 1999. Proceedings. 4th IEEE International Symposium on* (1999), IEEE, pp. 123–131.

[23] Wallace, D. R., and Kuhn, D. R. Failure modes in medical device software: an analysis of 15 years of recall data. *International Journal of Reliability, Quality and Safety Engineering 8*, 04 (2001), 351–371.