Chapter 1

Motivation

1.1 Introduction: System Failure

Many system failures result not from component failures but from inadequate component specifications that are correctly implemented but none-the-less lead to unforeseen component interactions [5, 18, 30, 31, 32, 33]. Even if the system failure can be tracked back to a bad decision made by a software component, usually the software component made that decision in accordance with its specification. The system requirements were known and the software obeyed its component specification, but the component specifications were insufficient to enforce the system requirement. For example, a chemical engineer might provide a specification to a software engineer writing code to control an automatic valve, but omit assumptions that all chemical engineers take for granted. The software engineer would then provide a piece of software in accordance with the written spec, but which violates the implicit intentions of the chemical engineer.

These incorrect specifications stem from two sources: a shortcoming on the part of the system engineer to decompose the system requirement into component specifications, and a failure to unambiguously communicate the specification to the software engineers (and the other specialized engineers). As software is increasingly deployed in contexts in which it controls multiple, complex physical devices, this issue is likely to grow in importance. Software does not wear out and fail in the way that a physical

system does, and so the prevailence of software components, systems are harder to validate in traditional methods (e.g. testing, manual inspection, redundancy). The emphasis must shift from preventing component failure to inferring and communicating specifications.

1.1.1 Dependability, Auditability, and Traceability

To be confident that a system meets its requirements, we need something more than skilled engineers and good process. We need an argument that is founded on concrete, reproducible evidence that documents why a system should be trusted.

A *dependability argument* [9] is one that justifies the use of a particular component for a particular role in a particular system. It is not an argument about absolute correctness, and it is not about preventing component failures. Rather, it is about understanding the interaction of components, and inferring sound component specifications. In this work, we are primarily concerned with the dependability of software components.

Building a *correct* argument is not enough; the argument must also be *auditable*. It might be reviewed by a certification authority (such as the FDA, FAA, or NRC [40, 1, 2]), a system engineer deciding if the system is suitable for a slightly different operating context, an engineer wanting to make a change to the system, or even an engineer new to the project. As the system evolves, the dependability argument must be *maintainable*, as reconstructing a thorough dependability argument after each change to the system is impractical.

A key part of making an argument auditable and maintainable is providing *traceability*. Traceability takes two forms: upward and downward. *Downward traceability* answers the question "Which components and what properties of those components ensure that system requirement X is enforced?", and provide confidence that the system operates as desired. *Upward traceability* answers the question "Which system requirements does component X help enforce, and upon what properties of X do those requirements rely?", and allows the system to be more safely modified.

An argument that provides both forms of traceability is termed *end-to-end*; it

connects the high level system concerns down to the low level component properties, based on an explicit description of the structure of the intervening layers.

The research community has approached dependability along four, largely independent routes. Individually, these styles of approach provide insufficient breadth, depth, confidence and/or are not economical on complex systems. Our approach brings together techniques developed in these different contingents to build a composite argument with sufficient breadth, depth, confidence, and at a manageable cost.

Requirements Engineering (RE) focuses on the task of factoring system requirements into component specifications. RE techniques typically considers the interactions of the components, but rarely validate the assumptions made about those components. Roughly speaking, arguments developed in the RE community are *broad* but not *deep*.

Program Analysis (PA) focuses on establishing specifications of individual software components. PA techniques typically do not consider why those specification are important, just whether or not they might be violated. Roughly speaking, arguments developed in the PA communities are *deep* but not *broad*.

Testing can provide the breadth of RE and the depth of PA, but fundamentally cannot provide the *confidence* needed to build a dependability argument. Testing assures that the system operates correctly in the tested scenarios, but provides no guarantees about scenarios not specifically tested.

Formal Methods (FM) provide ample confidence, but are too costly to economically apply to large legacy system. FM have only scaled to large systems when the systems have been built from scratch in a controlled manner by specially trained developers [15, 37]. Applied to an existing complex system, they do not scale adequately to build end-to-end arguments.

Unfortunately, while RE and PA each provide sufficient confidence at acceptable cost, the specifications generated by RE techniques often do not match up with the types of properties that PA techniques can validate. The two halves are typically

connected only informally by a intuition that certain properties about the code (such as the lack of buffer overruns) will correspond to system properties (such as the system being protected from security attacks). There is not a systematic, auditable argument articulated about why the properties checkable by PA are sufficient to ensure the properties called for by RE.

1.1.2 Contributions

This research has been conducted in 4 parallel, but interwoven, tracks:

template We have developed Composite Dependability Argument Diagrams (CDAD), a framework for constructing end-to-end dependability arguments by smoothly integrating a collection of component arguments.

instantiation We have identified techniques for building pieces of a dependability argument. We have applied CDAD to these techniques to produce a composite technique suitable for building dependability arguments for a particular class of software-intensive system properties.

development Where necessary, we have developed techniques to fill the gaps in our instantiation. Most prominently, we developed Requirement Progression, a technique used to connect problem diagrams with code specifications.

case study We have applied that technique to the Burr Proton Therapy Center (BPTC), a medical system currently being used to treat cancer patients as Massachusetts General Hospital (MGH).

1.1.3 Proton Therapy

We will evaluate our methodology by using it to build dependability arguments for aspects of the Burr Proton Therapy Center (BPTC), and reporting the results to the BPTC personnel.

The Burr Proton Therapy Center is a radiation therapy facility associated with the Massachusetts General Hospital (MGH) in Boston. In contrast to other forms of

radiation therapy, proton therapy is more precise and thus more suitable for tumors located in sensitive regions of the body such as eyes and brains, or for treating children.

The machine is considered to be safety critical primarily due to the potential for overdose—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 [47]. The most famous of these accidents are those involving the Therac-25 machine [34, 23], in whose failures faulty software was a primary cause. More recently, software appears to have been the main factor in similar accidents in Panama in 2001 [16].

The BPTC system was developed in the context of a sophisticated safety program including a detailed risk analysis. Unlike the Therac-25, the BPTC system makes extensive use of hardware interlocks, monitors, and redundancies. The software itself was instrumented with abundant runtime checks, heavily tested, and manually reviewed.

1.1.4 Evaluation

The success of this research project depends on the credibility of the dependability analysis performed on the BPTC. We will present our findings to the BPTC personnel, and record their responses. We will note both how important they say that our analysis is, and what changes they actually make to the system in light of our findings. We will produce

- A safety case for the dependability of software system to perform its required role in the system. This will involve a description of the conditions and assumptions under which the software is suitably dependable, and a structured, verifiable, and repeatable argument for why those conditions and assumptions are sufficient.

- A list of undocumented dependencies, assumptions, and vulnerabilities of the system, and an analysis of their effect on safety. These assumptions will hopefully be added to the official BPTC documentation for the system.

- A list of actual bugs discovered in the system, software or otherwise.
- A record of the value that the BPTC practicioners say that they find in our analysis.
- A description of our experience developing the dependability argument, including analysis of which parts worked well, which need improvement, and at what stage during the process different problems were discovered.

Chapter 2

Technique

2.1 Dependability Arguments

This section introduces Composite Dependability Argument Diagrams (CDAD), a systematic means for classifying the different styles of arguments that can be used to analyze, describe, and document pieces of a system. This classification helps to determine what properties a given argument style is appropriate for. More importantly, it cleanly represents how component arguments can be pieced together to form a coherent end-to-end argument with sufficient breadth, depth, and confidence to constitute a dependability argument.

2.1.1 Granularities

An artifact at one granularity comprises finer grained black boxes plus additional information about the structure of those pieces. For example, an *architecture* is a collection of components plus an organization of the interactions of those components, and each of those components is, in turn, a collection of modules plus an organization of the interaction of those modules.

world context The coarsest granularity regards the system architecture as a black box interacting with the surrounding world and stakeholders. For BPTC, the world contains domains such as investors, doctors, and FDA regulators, as well

as the delivery system itself. The internals of the architecture are hidden from view, but their interactions, communications, and goals are shown. Legal and financial concerns are expressed at this granularity, although our work focuses solely on safety concerns.

architecture The next finer granularity regards the components of the system architecture as black boxes, and examines how those components communicate and interact. Refining our view of the BPTC architecture reveals components such as operators, prescriptions, and the treatment manager. It is at this granularity that we state safety concerns, such as accurate dose delivery, consistent logging, and safe shutdown.

component At the next granularity, we regard modules within a components as black boxes, and examine how those modules interact. In the case of a software components, the modules might correspond to procedures that are connected by function calls and shared data. The BPTC treatment manager component contains modules such as messaging procedures and data structure definitions.

module/procedure At an even finer granularity, blocks within a module are treated as black boxes, but the structure within the module that links together those blocks is exposed. For a software module, the blocks might be linear fragments of code, linked together by conditionals and other non-linear control flow. For example, the "set equipment" procedure includes a block that initializes some variables, the code inside the loop that constructs an array of data, and a block that constructs a message from the array and sends it to the hardware device driver.

block The finest granularity we consider for a software component is the block level: individual statements in the code are considered to be black boxes, and we consider the structure of those statements (according to the the semantics of the programming language).

2.1.2 The Space of Arguments

In system analysis, a claim is often *stated* at one granularity but *established* at a lower granularity. For example, a performance goal might be stated at the world (highest) granularity but established by examining the reliability of interactions at the component (middle) granularity. An *argument* relates a claim at the *stated* level with a collection of claims at the *established* level. An argument justifies the belief that enforcing the finer grained properties will be sufficient to enforce the coarser grained property.

An argument's *breadth* is the granularity of the stated goal, while its *depth* is the granularity into which it recasts that goal. For example, a system refinement argument might state a claim about the architecture as a whole and recast that claim into a set of assumptions about the components of the architecture. As we will see later, a collection of arguments can be strung together to build larger, composite arguments that connect more varied granularities.

Figure 2-1 characterizes a wide array of argument styles that might be used when analyzing or designing a software-intensive architecture. The x-axis position of an argument is its breadth. The narrowest (left-most) arguments deal with goals stated about code blocks, such as assertions and invariants. The broadest arguments deal with goals stated about the context in which the architecture operates, such as safety requirements imposed by regulatory agencies. The y-axis position of an argument is its depth. The shallowest arguments are established at the world granularity, looking at the interactions between the system and its stakeholders, but without considering the architecture of the system. The deepest arguments are established at the code block granularity, looking at the full semantics of the software.

Thus, an $<x, y>$ point on the graph indicates a style of argument with breadth x and depth y. Each point is also labeled with examples of techniques commonly used to construct arguments of that style.

2.1.3 Existing Techniques

A particular technique for establishing an argument also has a cost and provides a certain level of confidence. These characteristics are not shown in Figure 2-1, but should be kept in mind when comparing or selecting techniques. Figure 2-1 does not directly represent the cost (both human and computational) of building the different kinds of arguments. In general, moving deeper (down) and broader (right) raises cost and/or lowers confidence.

The fields of program analysis (PA) and requirements engineering (RE) are represented by clusters of argument types in Figure 2-1.

Program analysis techniques (PA) occupy the lower-left-hand region; the properties are stated and established at a low granularity. PA techniques rarely address properties stated above the architecture granularity, as such properties are too broadly stated to be amenable to automatic analysis. We indicate this obstacle with the vertical *system complexity barrier* along the right-hand side of the PA region in Figure 2-1.

Requirements engineering techniques (RE) occupy the upper-right-hand region; the properties are stated and established at a high granularity. RE techniques rarely establish properties below the architecture granularity, as doing so produces descriptions that are too large and complex to be reasoned about. We indicate this obstacle with the horizontal *component complexity barrier* along the bottom of the RE region in Figure 2-1. In fact, most RE techniques focus on establishing properties at the architecture granularity and stay away from the complexity barrier.

Testing techniques occupy the bottom row of the diagram; they provide deep analysis at various breadths. Testing can provide the breadth of RE and the depth of PA, but fundamentally cannot provide the *confidence* needed to build a dependability argument. Testing assures that the system operates correctly in the tested scenarios, but provides no guarantees about scenarios not specifically tested.

The holy grail of software engineering (SE) is to develop a high-confidence economical technique at the lower-right-hand-most corner – one that states a property at the highest (world) granularity and establishes it at the lowest (block) granularity.

Unfortunately, getting anywhere near the holy grail requires crossing both complexity barriers.

2.1.4 Dependability Arguments

Fortunately, most tasks do not require the holy grail and can make do with more modest approaches. For example, verifying that libraries obey their contracts requires only a <*module, block*> style argument, and can be established sing program analysis techniques such as Forge [13]. Similarly, determining if a given software specification is sufficient to enforce a given system requirement requires only <*architecture, component*> or better, and can thus be satisfied by requirement progression [51]. However, the important class of *dependability* arguments lie outside the ranges of conventional PA and RE techniques, though still short of the holy grail.

Dependability arguments for software intensive systems should state properties at the architecture granularity (or higher) and establish those properties at the module granularity (or lower). For example, part of the BPTC dependability argument is to establish that patients do not receive more radiation than their prescriptions indicate. Such an argument should be grounded in the code, so that if the requirement is changed (e.g. to say that the patient cannot receive less than their prescription either) or if the system is changed (e.g. to include an additional firing mode), one can determine which parts of the code need updating, if any.

The shaded region in the lower right-hand corner of Figure 2-1 is the space of solutions that are appropriate for building this kind of dependability argument. While we do not need the holy grail to build dependability arguments, we do need something more than we have – neither PA nor RE techniques have sufficient breadth and depth to land in the target region. We can, however, synthesize existing PA and RE techniques, together with some additional work, to create a composite technique that falls within the target region. The challenge of building synthesis techniques is to keep the cost from rising too high without letting the confidence drop too low.

2.1.5 Synthesis

In order to build more powerful arguments that can address questions of system-wide dependability, we will need to build composite arguments out of the small pieces. Under our approach, a composite argument is an alternating sequence of glue and transition arguments; the transition arguments make progress in refining and justifying the top-level claim, and the glue arguments link the transitions together.

Synthesis is more than picking two techniques that, between them, have sufficient breadth and depth. The composed techniques must match up (the obligations generated by one must be discharged by the next), there must be glue to bind them (the output of one technique must be reconfigured to be a suitable input for the next), they must be inexpensive enough to be practical, and they must be thorough enough to provide sufficient confidence.

(a) The techniques need to match up.

We can't reach the bottom right corner (<*world, block*>) with a customer interview (<*world, world*>) and manual reviews of code fragments(<*block, block*>). While they have sufficient breadth and depth, they do not connect to each other: a customer interview produces a claim at the world level, but manual code reviews only establish claims about individual blocks of code. Code reviews simply cannot address the kinds of claims generated by a customer interview.

(b) There needs to be glue between the techniques.

We can't reach <*architecture, module*> using just functional decomposition (<*architecture, component*>) and UML (<*component, module*>). The claims generated by function decomposition are at the right level to be established by a UML analysis, but they may not be in the right form. In order to connect up the two arguments, a glue argument must be provided (at <*component, component*>) to recast the claims generated by functional decomposition with the claims established by the UML analysis.

(c) The composed techniques must provide sufficient confidence at economical cost.

We can reach the bottom right corner (<*world, block*>) using just *deployment testing* (<*world, block*>), but doing so will not provide sufficient confidence. While it has sufficient breadth and depth, testing the entire system on real patients and observing the results does not give us the confidence needed to certify the system as dependable. Testing fundamentally cannot provide the level of confidence needed to certify a complex system.

2.2 Structuring a Dependability Argument

Our general approach to constructing composite arguments can be applied to the BPTC case study by instantiating it with particular component techniques. In Figure 2-2, we show the techniques we combine, as transitions and glue, to form a composite technique for building dependability arguments.

Designations : A list of formal terms, both domains and phenomena, which will be relevant to the argument. Each term is mapped to an informal description, serving to ground our formality in the real world.

This piece of the argument is of the type <*system, system*>.

Problem Diagram : The system requirement is initially expressed with a problem diagram, from the Problem Frames approach [23, 21]. This step recasts the requirement from its original (possibly informal) statement into a form that is be amenable to requirement progression. It identifies the domains relevant to the subsystem under consideration, and the phenomena through which those domains interact, using the formal terms introduced by the designations.

This piece of the argument is of the type <*system, system*>.

Requirement Progression / Argument Diagram : The system requirement is transformed into a software specification using *requirement progression* [51]. The resulting diagram is called an *argument diagram*, which is the problem diagram annotated with a collection domain assumptions (*breadcrumbs*) sufficient

to enforce the original system requirement. Domain assumptions about software components can be used as specifications for those components.

This piece of the argument is of the type <*system, component*>.

Argument Validation : Along with the argument diagram is an Alloy model which mechanically confirms that the breadcrumbs do indeed enforce the desired system property.

This piece of the argument is of the type <*system, component*>.

Breadcrumb Assumption Interpretation : The domain properties inferred by Requirement Progression are interpreted back into the languages of their domains, using the designations, and decomposed into component assumptions about its domain. Each component assumption is classified as software correctness (c), software separability (s), or non-software properties (x). The decomposed assumptions are now amenable to domain specific analysis.

This piece of the argument is of the type <*component, component*>.

Phenomenon Assumption Interpretation : Problem diagrams contain implicit assumptions, such as atomicity and inter-domain consistency. These assumptions are also interpreted, decomposed, and classified as (c), (s), or (x).

This piece of the argument is of the type <*component, component*>.

Substantiation : Each of the component assumptions derived from the breadcrumbs and the phenomena arcs must be verified. In some cases, we discharge this task to domain experts. In the software cases, we check it ourselves. The remaining artifacts in this list pertain to incorporating analyses of software components into the dependability argument.

This piece of the argument is of the type <*component, block*>.

Flow Diagram / Trace Extraction : An informal annotation of the problem diagram, indicating the flow of information through the system. This diagram

guides requirement progression and provides an overall structure for the argument, but does not have a formal semantics. It helps to guide a human in determining the relevant portions of the code base relevant to a particular assumption. The flow diagram is used to label (and thus implicitly order) the domains and the letters assigned to arcs. These labels are purely for the sake of bookkeeping and help us to systematically develop the argument.

This piece of the argument is of the type <*component, module*>.

Translation & Abstraction : The relevant portions of the code base are abstracted and translated into the Forge Intermediate Language. This process is currently performed manually, but full or partial automation is possible.

This piece of the argument is of the type <*module, module*>.

Forge Analysis : The individual pieces are discharged using existing analysis techniques. Separability assumptions are addressed with impact analysis, and correctness properties are addressed using a combination of manual inspection and automatic analysis via the Forge framework.

This piece of the argument is of the type <*module, block*>.

Together, these component techniques provide an argument at <*system, block*>, well within our the zone for Dependability Arguments. The component techniques provide sufficient confidence to allow the overall argument to be used to certify the system.

2.3 Background: Problem Frames

The first step of establishing a system requirement is to articulate it. The Problem Frames approach is a technique for describing and analyzing desired system properties. The Problem Frames approach offers a framework for describing the interactions amongst software and other system components [21, 23]. It helps the developer understand the context in which the software problem resides, and which of its aspects are

relevant to the design of a solution [19, 22, 28]. Once the requirement is articulated as a problem diagram, it becomes amenable to more systematic analysis (Section 2.3.1).

The problem frames approach is an example of *problem oriented software engineering* [27], meaning that it focuses on the context in which a system operates rather than the internal architecture of the system. It emphasizes the distinction between phenomena one wishes to constrain and phenomena the software can directly control. The system requirement is written in terms of the phenomena one wishes to control, but software specifications should be written in terms of controllable phenomena. System analysis is a matter of understanding the indirect links between those two sets of phenomena.

The benefit of this approach is that the analyst formulates the system requirement in terms of whatever phenomena are most appropriate and convenient. As a result, we have a higher confidence that the written requirements accurately reflect the intended requirements. Attempting to directly write the requirement in terms of controllable phenomena can be subtle and error prone. The problem frames approach separates the articulation of the requirement from the transformation of that requirement into a form usable as a specification. In the next section, we describe a technique for helping the designer reformulate requirements and, in doing so, reveal the assumptions they rely upon.

2.3.1 Requirement Progression

Requirement progression [48, 50, 51] is a strategy by which requirement progression can be performed systematically by a series of incremental transformations. For example, given a system requirement that the log accurately reflect the radiation delivered to the patient and a Problem Frame description of how the log and patient (indirectly) interact, an analyst can systematically derive a specification on the Treatment Control System (TCS) that is sufficient to enforce that requirement. In the process of performing that derivation, the analyst leaves behind a trail of *breadcrumbs* in the form of domain assumptions. The analyst is guaranteed that, so long as those assumptions hold, the derived specification for the TCS is sufficient to enforce the

system requirement.

2.4 Patient Identity Argument

Our technique is perhaps best explained by application to a real problem. Consider the *patient identity* subproblem:

> If the therapist instructs the proton beam to fire, then either the patient under the nozzle of the beam will receive the prescription stored in the database for that patient or the therapist will be notified of a failure.

Safe rrror handling, database initialization, and unsafe prescriptions are separate subproblems and will not be addressed here. This argument is focused on the problem of coordinating the therapist's instructions (entered via a GUI), the database values, and the hardware device drivers. As a result, we focus primarily on the treatment manager software at the center of this coordination.

The patient identity subproblem addresses a medium to high severity hazard; it is potentially life threatening to the patient. Delivering one patient the prescription for a different patient could result in minor overdoses (medium severity, since treatment will have to be delayed). Doing so repeatedly could result in a systematic unknown underdose (high severity since the patient's cancer will unknowling remain untreated).

Figure 2-3 shows the argument diagram derived from the application of requirement progression to the problem diagram. The cross cutting dose delivery property (assumption 0 in the diagram) has been decomposed into a collection of "breadcrumb" domain assumptions, each of which only references phenomena from a single domain. These domain assumptions can be handed off to domain experts and independently validated. In our work, we focus on the software-related assumptions made about the Treatment Manager.

We then examine each breadcrumb assumption in turn; we interpet each using the designations relevant to its domain, thus allowing domain-specific tools and experts to be applied to validating them. For example, breadcrumb 4b is interpreted as the following post-condition for the code:

```
pred patient_id_storage [] {
  data.data__msg.mixed_array_index[0]
    == SCR_A1_PATIENT_SELECTION
  data.data__msg.mixed_array_index[1]
    == W_PATIENT_SELECT_BTN
  current_id_patient
     == arg.scrCrtPatientData
         .dbs_patient_type__id_patient
}
```

This property is checked against the code base using the Forge framework [13], along with some accompanying liveness checks to mitigate the chance of overconstraint. In order for perform these analyses, the C sourcecode must be translated into the Forge Intermediate Language. This translation is currently performed manually, but is automatable. In progress is a lightweight lexical translation aid to ease this process.

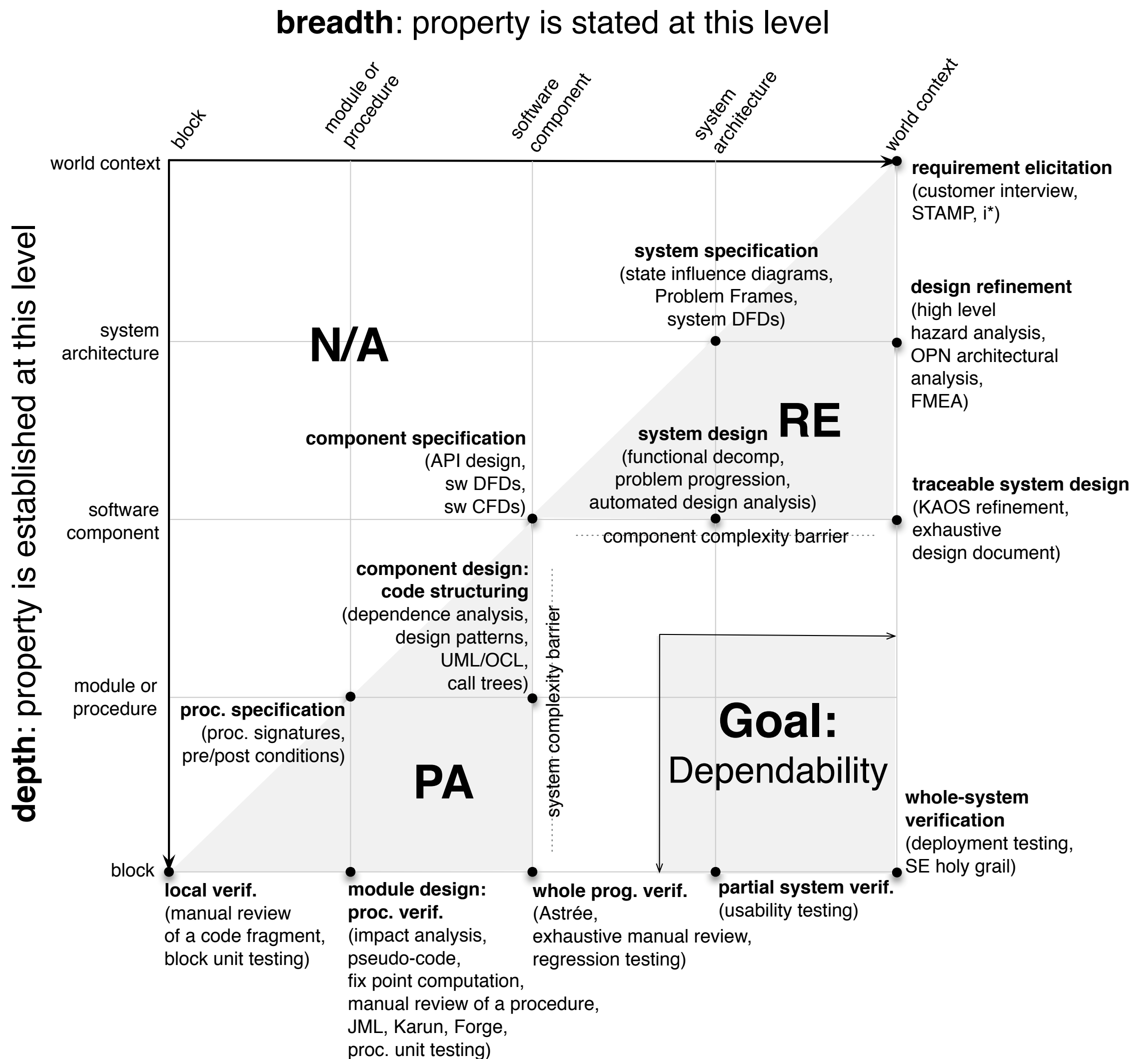

Figure 2-1: A Composite Dependability Argument Diagram (CDAD) showing the space of arguments that can be constructed about a software-intensive system and the traceability each such argument provides. Each point represents a style of argument in which a property is stated at one level (x-axis, breadth) and established at another (y-axis, depth). Each point is labeled with sample techniques often used to constructed arguments of that style. Dependability arguments (Goal) are not addressed by conventional requirements engineering (RE) or program anlaysis (PA) techniques. Such arguments must address system level properties (or higher) and establish those properties at the module level (or deeper).

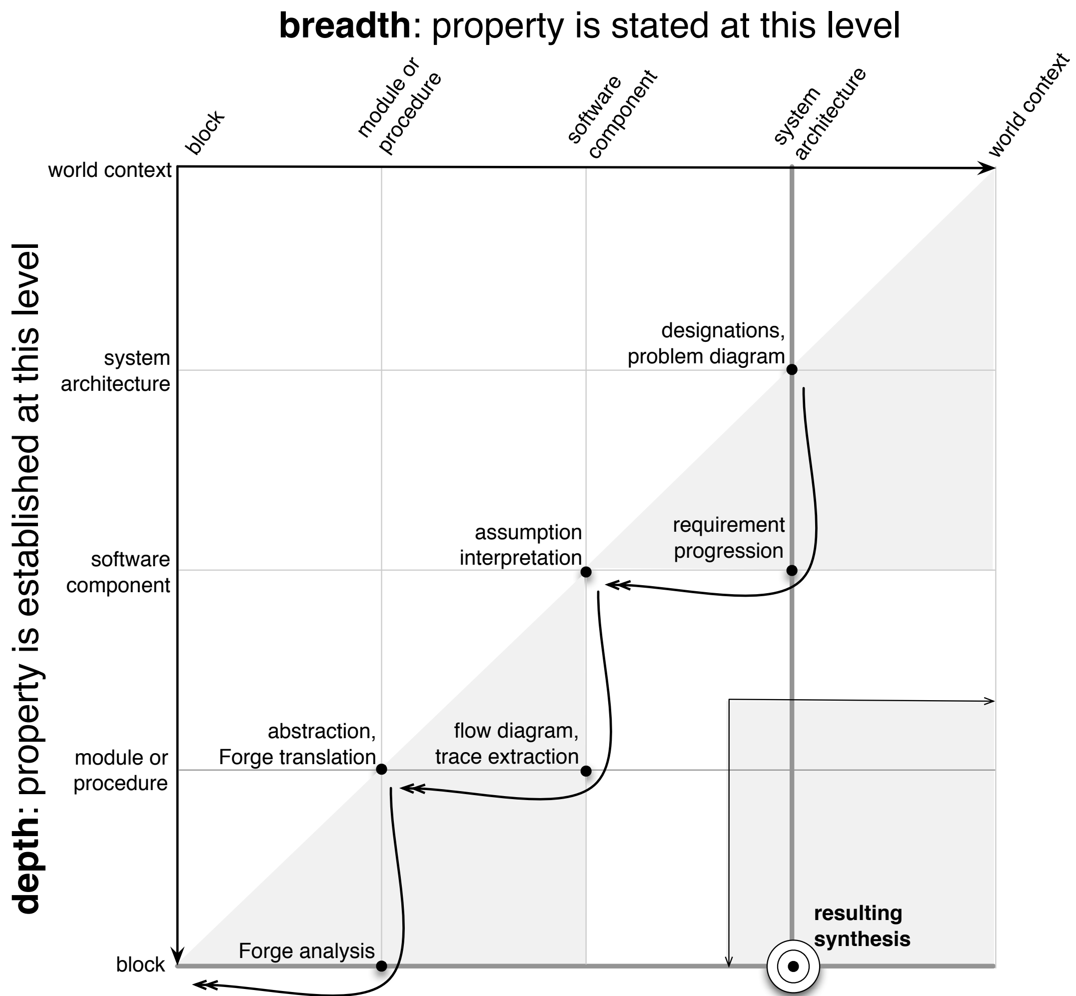

Figure 2-2: The components we synthesize to build a mid-cost, end-to-end dependability arguments.

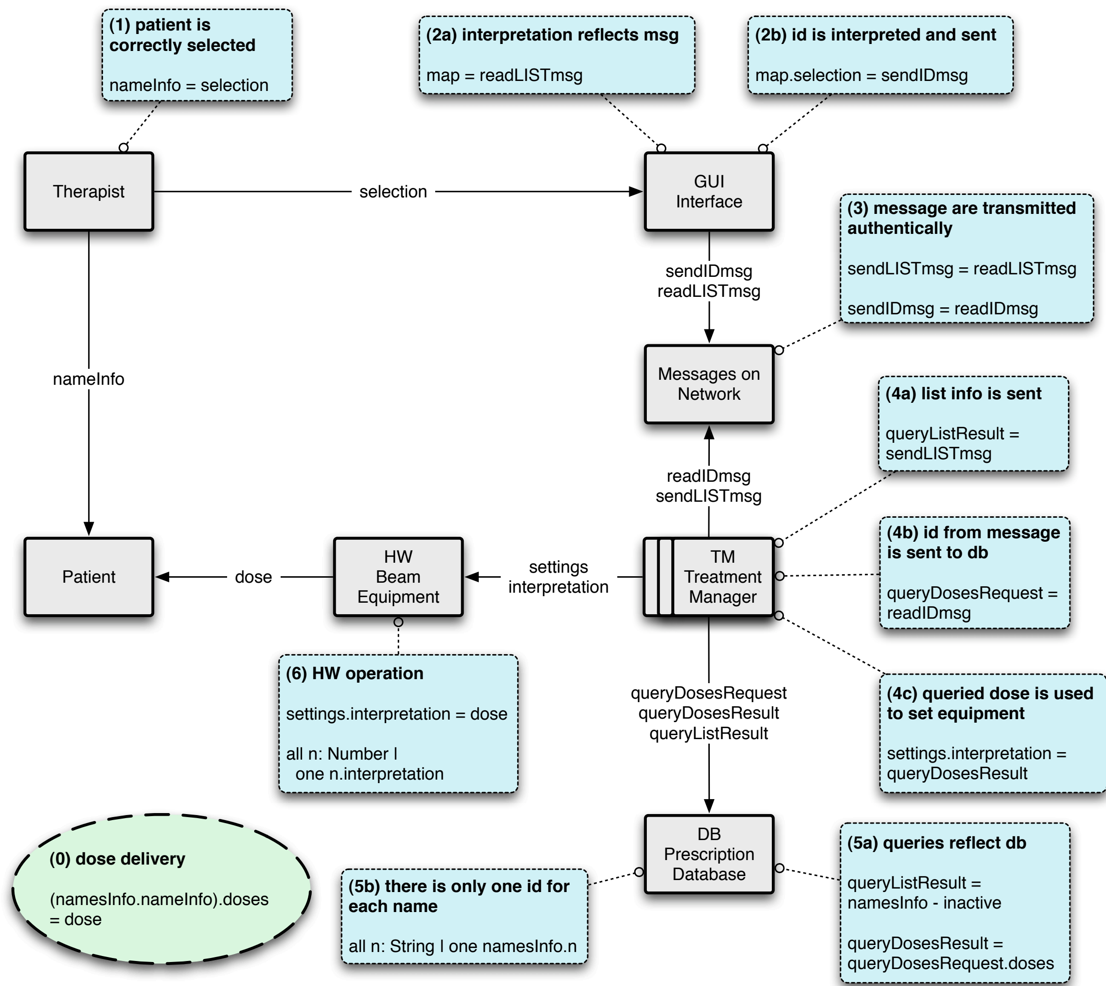

Figure 2-3: Argument diagram for the patient identity subproblem.

Chapter 3

Feasibility

3.1 Related Work

3.1.1 Requirement Decomposition

Like our Requirement Progression technique, many approaches to system analysis involve some kind of decomposition of end-to-end requirements into subconstraints, often recursively.

Assurance and Safety Cases

Assurance and safety cases [3, 30], for example, decompose a critical safety property. They tend to operate at a larger granularity than problem frames, in which the elements represent arguments or large groupings of evidence, rather than constraints. Another class of analyses focus on failures rather than requirements (such as HAZOP [41]), in which decomposition is used to identify the root causes of failures. Our work, like that of assurance cases, provides confidence that a given requirement will hold, rather than establishing that a particular type of error will not occur.

Leveson's STAMP approach involves decomposing design constriants, with a focus on managerial control over the operation of a system [32, 33].

I*, Tropos, KAOS

More similar to our approach are frameworks, such as `i*` [54] and KAOS [11, 12, 8, 6], that decompose system-level properties by assigning properties to agents that work together to achieve the goal. For KAOS, patterns have been developed for refining a requirement into subgoals [12]. In our approach, we have not given a constructive method for obtaining the new constraint systematically, and the refinement strategies of KAOS may fill this gap.

Similar to i*, Tropos [7, 17, 43] is based on actors with different goals for the system and different measures of success. It is focused on early design stages, and is mostly for human-human communication plus some simulation/evaluation support for making sense of larger models.

KAOS refinements has been applied to agent-oriented policy decomposition and applied to Systems of Systems (SoS) [20]. It is used as a means for combatting emergent behaviors that result from independently designed systems combined into a single system.

Four Variable Model

The four-variable model [42, 53] makes a distinction, like Problem Frames, between the requirements, the specification, and domain assumptions. However, in Problem Frame terminology, it assumes that a particular frame always applies, in which there is a machine, an input device domain, an output device domain, and a domain of controlled and monitored phenomena.

Requirement Elicitation

Letier and Lamsweerde show how a goal (requirement) produced from requirement elicitation can be transformed into a specification that is formal and precise enough to guide implementation [29]. That approach is centered around producing operational specifications from requirements expressed in temporal logic, and focuses on proving the correctness of a set of inference patterns. Such inference patterns are correct

regardless of context, in contrast to our approach in which transformations are only justified by context-specific domain assumptions.

Refinement

Johnson made an early use of the phrase "deriving specifications from requirements" in 1988 when he showed how requirements written in the relational logic language *Gist* can be transformed into specifications through iterative refinement [26]. Each refinement step places limits on what domains may know and on their ability to control the world, and exceptions are added to global constraints. A specification is not guaranteed to logically imply the requirement it grew out of, and the two descriptions may even be logically inconsistent with each other. In contrast, as we refine (transform) a requirement, the breadcrumbs we add expand our assumptions about the domains rather than restricting them, and a specification will always be consistent with the requirement it enforces.

3.1.2 Problem Frames

Problem Progression

Michael Jackson sketches out a notion of *problem progression* in the Problem Frames book [23]. A problem progression is a sequence of Problem Frame descriptions, beginning with the full description (including the original requirement) and ending with a description containing only the machine and its specification. In each successive description, the domains connected to the requirement are eliminated and the requirement is reconnected and altered as needed. He does not work out the details of how one would derive the successive descriptions, but it seems that he had a similar vision to our own. However, rather than eliminating elements (domains) from the diagram at each step, our approach adds elements (domain assumptions), providing a trace of the analyst's reasoning in a single diagram.

Jackson and Zave use a coin-operated turnstyle to demonstrate how to turn a requirement into a specification by adding appropriate environmental properties (do-

main assumptions) [24]. Their approach is quite similar to our own, and uses a logical constraint language to express domain assumptions. Our work strives to generalize the process to be applicable in broader and more complex circumstances, and to help guide the analyst through the process with the visual notion of pushing the requirement towards the machine.

Problem Reduction

Rapanotti, Hall, and Li recently introduced *problem reduction*, a technique that uses causal logic to formalize problem progression in Problem Frames [46]. Like our own work, they seek to formalize and generalize problem progression in a way that provides traceability as well as a guarantee of sufficiency. Problem reduction follows the style of problem progression described in the Problem Frames book [23], in which the requirement is moved closer to the machine by eliminating intervening domains.

Calculus of Requirements Engineering

Hall, Rapanotti, Li, and M. Jackson are developing a calculus of requirements engineering based on the Problem Frames approach [27, 35, 36, 45]. They examine how problems and solutions can be restructured to fit known patterns. Part of their technique involves transformation rules for problem progression, in which a requirement (expressed in CSP) is replaced by an equivalent requirement in an alternate form. In contrast, our technique is a form of requirement progression, in which the transformations only change the constraints, not the underlying domain structure. Furthermore, our transformations are not semantics-preserving; they are justified by a set of explicit assumptions rather than proofs of equivalence.

3.1.3 Analysis of the BPTC

Jackson and Jackson have examined the *gantry creep* in the BPTC, in which the angle of delivery slowly shifts over the course of many treatments of the same patient [10].

Rae et al have used lightweight code analysis to determine conditions under which

the BPTC emergency stop button would not operate correctly [44].

Dennis et al have shown how commutitivity analysis can be used to detect race conditions between operators of a system, even when that system uses atomic single threaded operations. They apply the technique to the automatic beam scheduler currently employed in the BPTC [14]

In earlier work, we have used the BPTC to motivate the development of a technique for performing requirement progression [48, 49, 50, 51].

3.2 Research Schedule

2008 IAP

- select thesis committee
- submit proposal

2008 Spring

- get BPTC feedback on patient id problem
- polish patient id preservation argument
- revisit monitor units & logging case studies
- lightweight C-to-Forge lexical translation, use Toshiba's implementation?

2008 Summer

- read related work in more detail, write up comparison
- case study expansion and polishing

2008 Fall

- show work to Jay Flanz at BPTC and get final feedback
- defend thesis
- finalize thesis document

3.2.1 Flexibility

opportunities to cut

- reduce the extent of monitor units case study (in light of Doug Miller's exit, the prior and only real expert on the BPTC code). That case study is a compelling one, so I hope to push it through despite his absence.
- reduce the extent of the automated support for C-to-Forge translation. Automating this to some extend is important for being able to handle larger case studies (such as monitor units), but the tool does not need to be as powerful as it can be; this thesis is about the technique, not the particular tool support for that technique.
- focus less on program analysis and more on the requirements engineering. The contributions of this work are twofold (a) synthesis of techniques into an end-to-end argument, and (b) the development of requirement progression in order to enable that synthesis. Since (b) is complete, one option is to focus more on those aspects of (a).

potential bottlenecks

- time delay in getting feedback from BPTC
- sorting out the complexities of the monitor units example, given that Doug Miller (an expert on the system) is no longer working at the BPTC
- automating the translation of C to Forge, either with a lightweight syntatic translation that I write myself or by integrating Toshiba's implementation

Acknowledgments

This work is part of an ongoing collaboration between the Software Design Group at MIT and the Burr Proton Therapy Center (BPTC) of the Massachusetts General Hospital. We appreciate the assistance of Jay Flanz and Doug Miller at the BPTC.

This research was supported, in part, by grants *0086154* ('Design Conformant Software') and *6895566* ('Safety Mechanisms for Medical Software') from the ITR program of the National Science Foundation.

	end-to-end integration	Problem Frames & Progression	Code Analysis (Forge)	STAMP integration
technique	√	√	√	√
patient id end-to-end example	√	√	~ automatic safety check	~ context for requirements
theory and sideproofs	(X)	~ completeness, patterns	(X)	~ 'bad smells' in STAMP model
tool support	√ alloy synthesis model	(X) layout, check, suggestion	X lightweight syntactic trans.	(X)
related work & alternatives	~	√	X	~

Current Work;
to be included in Thesis

Future Work;
not to be included in Thesis

To the extent that it will be included in the thesis, this task is...

√ done
~ partly done
X not done but will be
(X) won't be included

Figure 3-1: Thesis research progress.

Bibliography

[1] United States Federal Aviation Administration. FAA: Federal aviation administration. website, 2008. http://www.faa.gov/.

[2] United States Nuclear Regulatory Agency. U.S. NRC: Protecting people and the environment. website, 2008. http://www.nrc.gov/.

[3] Air Force, Space Division. System safety handbook for the acquisition manager, January 1987. SDP 127-1.

[4] Issa Bass. Failure mode and effects analysis - FMEA. website, 2007. http://www.sixsigmafirst.com/FMEA.htm.

[5] T. E. Bell and T. A. Thayer. Software requirements: are they really a problem? In *Proceedings of the 2nd International Conference on Software Engineering (ICSE'67)*, pages 61–68. IEEE Society Press, 1967.

[6] P. Bertrand, Robert Darimont, E. Delor, Philippe Massonet, and Axel van Lamsweerde. Grail/kaos: an environment for goal driven requirements engineering. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998.

[7] Jaelson Castro, Paolo Giorgini, Stefanie Kethers, and John Mylopoulos. A requirements-driven methodology for agent-oriented software. In Brian Henderson-Sellers and Paoli Giorgini, editors, *Agent-Oriented Methodologies* . Idea Group Pub, NY, USA, 2005.

[8] Christophe Damas, Bernard Lambeau, P. Dupont, and Axel van Lamsweerde. Generating annotated behavior models from end-user scenarios. In *IEEE Transactions on Software Engineering, Special Issue on Interaction and State-based Modeling*, volume 31, pages 1056–1073, 2005.

[9] Lynette I. Millett Daniel Jackson, Martyn Thomas. *Software for Dependable Systems: Sufficient Evidence?* National Academies, Washington, DC, May 2007.

[10] Michael Jackson Daniel Jackson. *Separating Concerns in Requirements Analysis: An Example.* Springer-Verlag, 2006.

[11] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[12] Robert Darimont and Axel van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Proceedings of the 4th International Symposium on the Foundations of Software Engineering (FSE'96)*, pages 179–190, San Francisco, Oct 1996.

[13] Greg Dennis. Forge: Bounded program verification. website, 2008. http://sdg.csail.mit.edu/forge/.

[14] Greg Dennis, Robert Seater, Derek Rayside, and Daniel Jackson. Automating commutativity analysis at the design level. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'04)*, July 2004. Boston, MA, USA.

[15] Praxis Engineering. Praxis engineering. website, 2008. http://www.praxiseng.com/.

[16] Food and Drug Administration. FDA statement on radiation overexposures in panama. www.fda.gov/cdrh/ocd/panamaradexp.html.

[17] Paolo Giorgini, John Mylopoulos, and Roberto Sebastiani. Goal-oriented requirements analysis and reasoning in the Tropos methodology. In *Engineering Applications of Artificial Intelligence*, volume 18/2, march 2005.

[18] Sol Greenspan, John Mylopoulos, and Alex Borgida. On formal requirements modeling languages: RML revisited. In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*, pages 135–147. IEEE Computer Society Press, 1994.

[19] Charles B. Haley, Robin C. Laney, and Bashar Nuseibeh. Using Problem Frames and projections to analyze requirements for distributed systems. In *Proceedings of the 10th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'04)*, volume 9, pages 203–217. Essener Informatik Beiträge, 2004. Editors: B. Regnell, E. Kamsties, and V. Gervasi.

[20] Martin Hall-May and Tim Kelly. Defining and decomposing safety policy for systems of systems. In *24th international conference on computer safety, reliability, and security (SAFECOMP'05)*, volume 3688, Fredrikstad, Norway, September 2005. ISBN 3-540-29200-4.

[21] Michael Jackson. *Software Requirements and Specifications: a lexicon of practice, principles and prejudice*. Addison-Wesley, 1995.

[22] Michael Jackson. Problem analysis using small Problem Frames. *South African Computer Journal*, 22:47–60, March 1999.

[23] Michael Jackson. *Problem Frames: analyzing and structuring software development problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[24] Michael Jackson and Pamela Zave. Deriving specifications from requirements: an example. In *Proceedings of the 17th International Conference on Software Engineering (ICSE'95)*, pages 15–24, New York, NY, USA, 1995. ACM Press.

[25] Chris W. Johnson. *Failure in Safety-Critical Systems: A Handbook of Incident and Accident Reporting*. Glasbow University Press, October 2003.

[26] W. Lewis Johnson. Deriving specifications from requirements. In *Proceedings of the 10th International Conference on Software Engineering (ICSE'88)*, pages 428–438. IEEE Computer Society, 1988.

[27] Michael A. Jackson Jon G. Hall, Lucia Rapanotti. Problem oriented software engineering. Technical Report 2006/10, Department of Computing, The Open University, 2006.

[28] Robin C. Laney, Leonor Barroca, Michael Jackson, and Bashar Nuseibeh. Composing requirements using Problem Frames. In *Proceedings of the 12th IEEE International Requirements Engineering Conference (RE'04)*, pages 121–131. IEEE Computer Science Press, 2004.

[29] Emmanuel Letier and Axel van Lamsweerde. Deriving operational software specifications from system goals. In *Proceedings of the 10th International Symposium on Foundations of Software Engineering (FSE'02)*, pages 119–128, 2002.

[30] Nancy G. Leveson. *Safeware: system safety and computers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[31] Nancy G. Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE Transactions on Software Engineering*, 26(1):15–35, January 2000.

[32] Nancy G. Leveson. A new approach to hazard analysis for complex systems. In *International Conference of the System Safety Society*, August 2003.

[33] Nancy G. Leveson. A systems-theoretic approach to safety in software-intensive systems. 1:66–86, 2004.

[34] Nancy G. Leveson and C. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 7(26):18–41, 1993.

[35] Zhi Li, Jon G. Hall, and Lucia Rapanotti. A constructive approach to Problem Frame semantics. Technical Report 2004/26, Department of Computing, The Open University, 2005.

[36] Zhi Li, Jon G. Hall, and Lucia Rapanotti. From requirements to specifications: a formal approach. In *Proceedings of the 2nd International Workshop on Applications and Advances in Problem Frames (IWAAPF'06), co-located with the 28th International Conference on Software Engineering (ICSE'06)*, page 65, Shanghai, China, May 2006. ACM Press.

[37] S. Liu and R. Adams. Limitations of formal methods and an approach to improvement. In *APSEC'95: Proceedings of the Second Asia Pacific Software Engineering Conference*, page 498, Washington, DC, USA, 1995. IEEE Computer Society.

[38] R. R. Mohr. Failure modes and effect analysis. presentation slides, January 1994. 8th edition, Sverdrup.

[39] Donald A. Norman. Design rules based on analyses of human error. *Commun. ACM*, 26(4):254–258, 1983.

[40] United States Department of Health and Human Services. FDA: U.s. food and drug administration. website, 2008. http://www.fda.gov/.

[41] Henry Ozog. Hazard identification, analysis, and control. *Hazard Prevention*, pages 11–17, May-June 1985.

[42] David L. Parnas and Jan Madey. Functional documentation for computer systems engineering, vol. 2. Technical Report Technical Report CRL 237, McMaster University, Hamilton, Ontario, Sept 1991.

[43] Tropos Project. Tropos: requirements-driven development for agent software. website, 2006. http://www.troposproject.org/.

[44] Andrew Rae, Prasad Ramanan, Daniel Jackson, and Jay Flanz. Critical feature analysis of a radiotherapy machine. In *International Conference of Computer Safety, Reliability and Security (SAFECOMP 2003)*, Edinburgh, September 2003. http://sdg.lcs.mit.edu.

[45] Lucia Rapanotti, Jon G. Hall, and Zhi Li. Deriving specifications from requirements through problem reduction. In *IEE Proceedings – Software*, volume 153: Issue 5, pages 183–198, October 2006. ISSN: 1462-5970.

[46] Lucia Rapanotti, Jon G. Hall, and Zhi Li. Problem reduction: a systematic technique for deriving specifications from requirements. Technical Report 2006/02, Department of Computing, The Open University, Feb 2006. ISSN 1744-1986.

[47] 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.

[48] Robert Seater and Daniel Jackson. Problem Frame transformations: Deriving specifications from requirements. In *Proceedings of the 2nd International Workshop on Applications and Advances in Problem Frames (IWAAPF'06), co-located with the 28th International Conference on Software Engineering (ICSE'06)*, pages 65–70, Shanghai, China, May 2006. ACM Press.

[49] Robert Seater and Daniel Jackson. Problem Frame transformations in the context of a proton therapy system. Unpublished manuscript. Unpublished manuscript, 2006.

[50] Robert Seater and Daniel Jackson. Requirement progression in problem frames applied to a proton therapy system. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, Minneapolis, MN, September 2006.

[51] Robert Seater, Daniel Jackson, and Rohit Gheyi. Requirement progression in problem frames: Deriving specifications from requirements. 2007.

[52] Elizabeth A. Strunk and John C. Knight. The essential synthesis of problem frames and assurance cases. In *Proceedings of the 2nd International Workshop on Applications and Advances in Problem Frames (IWAAPF'06), co-located with the 28th International Conference on Software Engineering (ICSE'06)*, pages 81–86, Shanghai, China, May 2006. ACM Press.

[53] Jeffrey M. Thompson, Mats P. E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Proceedings of the 6th European Software Engineering Conference / Proceedings of the 7th ACM SIGSOFT Symposium on the Foundations on Software Engineering (ESEC/FSE'99)*, number 1687 in LNCS, pages 163–179, September 1999.

[54] Eric S. K. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*, pages 226–235, Washington DC, USA, Jan 1997.

[55] Marc Zimmerman, Mario Rodriguez, Benjamin Ingram, Masafummi Katahira, Maxime de Villepin, and Nancy G. Leveson. Making formal methods practical. In *Proceedings of the 19th Digital Avionics Systems Conferences*, October 2000.