Building Dependability Arguments for Software Intensive Systems

by

Robert Morrison Seater

B.S., Haverford College (2002)
S.M., Massachusetts Institute of Technology (2005)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2009

Author . . . .
Department of Electrical Engineering and Computer Science
Jan 15, 2009

Certified by . . . .
Daniel Jackson
Professor
Thesis Supervisor

Accepted by . . . .
Professor Terry P. Orlando
Chairman, Department Committee on Graduate Students

Building Dependability Arguments for Software Intensive Systems

by

Robert Morrison Seater

Submitted to the Department of Electrical Engineering and Computer Science
on Jan 15, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

A method is introduced for structuring and guiding the development of end-to-end dependability arguments. The goal is to establish high-level requirements of complex software-intensive systems, especially properties that cross-cut normal functional decomposition. The resulting argument documents and validates the justification of system-level claims by tracing them down to component-level substantiation, such as automatic code analysis or cryptographic proofs. The method is evaluated on case studies drawn from the Burr Proton Therapy Center, operating at Massachusetts General Hospital, and on a cryptographic voting system, developed at the University of Newcastle.

Thesis Supervisor: Daniel Jackson
Title: Professor

Acknowledgments

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.
- NSF grant *0438897* ('Sod Collaborative Research: Constraint-based Architecture Evaluation').
- the Toshiba Corporation, as part of a collaboration between Toshiba's Corporate Research and Development Center and the Software Design Group of MIT's Computer Science and Artificial Intelligence Lab. We especially thank Takeo Imia.

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 especially appreciate the assistance of Jay Flanz and Doug Miller of the BPTC for devoting so much of their time to the project.

I am thankful for the many people who contributed ideas and encouragement for this work. Specifically, Daniel Jackson's advising was insightful and inspiring; without it I would have abandoned my line of work many times over. I am also indebted to my thesis committee members, Rob Miller and Ed Crawley for their insight and proofreading skills. Many other researchers have made intellectual contributions to this work, of whom I would like to especially thank Peter Ryan, Eunsuk Kang, Lucy Mendel, Derek Rayside, John Hall, Rohit Gheyi, and Michael Jackson (not the singer). I would also like to thank the organizers and attendees of IWAAPF'06 (a workshop devoted to Problem Frames research) for their warm welcome and insightful feedback. No acknowledgement would be complete without a predictable, but justified, thank you to my wife (Jessica) for putting up with my graduate student stipend (low) and lifestyle (I got more sleep than she did). Lastly, thanks to Terry Nation for creating the Daleks, who played a starring role in my thesis defense.

Contents

List of Figures

Chapter 1

Motivation

Chapter 2

Synthesis Technique: CDAD

Chapter 3

Requirement Progression

3.1 Discussion

3.1.1 Role of the Analyst

The transformation process is systematic but not automatic. The decisions of what breadcrumbs to add, how to rephrase the requirement, and which enabled pushes to enact are subjective assessments by the analyst based on experience or a related frame concern.

This approach is incremental, and justified by assertions that involve, in any step, at most assumptions about a single domain. While the process involves mostly local reasoning, the resulting guarantee is a global one – that the specification together with all the domain assumptions together imply the requirement.

Perhaps the biggest shortcoming of requirement progression is the burden placed on the analyst to come up with breadcrumbs that are both useful for moving forward with the progression but also consistent with existing knowledge of the domains.

The task of deciding what responsibilities to assign to each domain is a fundamentally a judgement call and thus not automatable. However, we envision the analyst being aided by a catalogue of common transformation patterns to help guide her in the right directions. That is, given the local structure of a problem diagram and a desired push, what are the right kinds of breadcrumbs and rephrasings to perform? Such heuristics are likely to take into account which domains control which phenomena and the type of each domain (biddable, causal, lexical) – information which we currently ignore. Such heuristics can only be properly developed over the course of many applications, although we will allude to some potential guides as we discussion the BPTC and Voting case studies in later chapters.

3.1.2 Source of Breadcrumbs

Central to this approach is the introduction of breadcrumb constraints representing assumptions about the domain behaviors. However, coming up with domain characterizations that are both useful in moving the progression forward and which will be certified by an expert can be quite an onerous task. We have considered four potential sources of breadcrumbs:

analyst's intuition – The analyst introduces whatever breadcrumbs are useful to the progression, as long as they seem reasonable. They are later checked by a domain expert and hopefully validated. If not, the progression will have to be reworked with a substitute assumption. For this method to be practical, the analyst must usually generate correct assumptions, as may be the case if the analyst is one of the system experts or if the system is simple.

explicit list – In a safety critical system, it is may be reasonable to explicitly list all of the available assumptions for each domain. Such a list might already exist, or it might be cost effective to generate. The analyst can then browse the list for useful breadcrumbs. If the list is very large, this method will not be much different from the first.

implicit encoding – Even if the explicit list of all domain assumptions is large, there may be a compact encoding of those properties. For example, a state machine might be an effective way to describe a domain, as opposed to explicitly describing all of the properties of that state machine. The analyst could use the compact encoding both as a source of inspiration and as a means of verifying desired assumptions without consulting the actual domain expert.

informal description – Full formal encodings of each of the domains is often an unfulfilled wish. Rather, the analyst faces an informal, although perhaps very detailed and precise, description of the system components. These informal descriptions might be in the form of natural language documentation or expert interviews. They suggest to the analyst what sorts of domain assumptions are likely to be validated by the experts, although, due to their informality, they will still produce some false positives.

Any of these options can be appropriate depending on the type of component in question. Cutting edge designs are most amenable to using analysts intuition, as they are not nailed down and can be adapted to fit different sets of assumptions. Simple mechanical components are likely to be amenable to explicit lists, as they have a short but well-understood set of relevant properties. Mode-based components (such as a car's gearshift) are best described with implicit state machine encodings that reflect the modal nature of the domain. Human operators are best suited to informal descriptions, since formal statements about human behavior are deceptively certain. Our experience has been primarily with the fourth case – informal descriptions based on expert interviews – and that is how we will present the examples in this thesis.

3.1.3 Progression Mistakes

The power and limitations of our technique can be appreciated by considering some mistakes an analyst might make while performing the transformations. How each mistake manifests itself reveals both strengths of our current work and indicates challenges for future work.

(1) A breadcrumb might be added that is insufficient to permit the desired rephrasing. In such a case, the analyst would be unable to discharge the required implication and the rephrasing would not be permitted.

(2) A breadcrumb might be added that represents an invalid assumption. At the very least, stating that assumption explicitly will increase the likelihood that it will be corrected by a domain expert.

(3) A breadcrumb might be added that is correct but which is stronger than necessary to justify the rephrasing. There will be no ill effect on the specification, but a stronger breadcrumb places additional burden on the domain expert attempting to validate it.

(4) A breadcrumb might be added that is weaker than necessary, forcing the rephrased requirement to be stronger than necessary. The resulting specification will be stronger than it could have been, making it harder (or impossible) to implement. The analyst would review the trail of breadcrumbs to find opportunities for weakening the requirement by strengthening the breadcrumbs.

(5) The original requirement might be too strong to be enforced by any (realistically) implementable specification. In such a case, the analyst will derive an unreasonably (but necessarily) strong specification, and the requirement will have to be rethought.

Points 3 and 4 get at the fundamental tradeoff between the strength of the domain assumptions and the strength of the specification. If a domain assumption is weakened (thus permitting more behaviors), then typically the specification will have to be strengthened (thus permitting fewer behaviors). Conversely, weaking the specification typically requires strengthening the domain assumptions.

3.1.4 Reacting to Rejected Breadcrumbs

If a domain assumption (including a specification that resulted from progressing a requirement) is rejected by domain experts, there are four actions that might be taken:

Rework System An extreme option is to drop the assumption entirely and re-negotiate the requirement so that a different assumption is made upon the domain in question. This option is typically unavailable as it is costly and probably exceeds the authority of the analyst and scope of the system project.

Alter Assumption The best case is that a similar assumption can be written that will satisfy the domain expert and still provide the needed guarantee for the argument. This might happen because the domain assumption was too prescriptive and not sufficient general. Loosening the constraint may allow it to play the needed role in the argument but to still be consistent with the current implementation of the domain. This option is often too optimistic and there really is a fundamental clash between the assumption made and the capabilities of the domain.

Shift Assumption It may be that the assumption in question is enforceable, just not by the domain to which it connects. In that case, requirement progression can be used to shift the assumption to another domain. In doing so a new (weaker) breadcrumb will be added to the old domain, the old breadcrumb will be rephrased, and the rephrased breadcrumb will be pushed onto another (adjacent) domain.

For example, this is the case in the traffic light example given earlier in this chapter, in figure **??**. The requirement states that cars will not collide – an assumption which connects only to the Cars domain. According to our progressions rules, no more need be done since the requirement already is in the form of a domain assumption. However, the expert on the Cars domain will tell us that the cars domain cannot enforce the non-collision assumption. In response, we shift the offending assumption over to the Light Unit domain (using requirement progression, as shown in Figure **??**). In doing so, we leave behind a new breadcrumb that is much weaker and is confirmed by the domain expert. However, now the Light Unit expert tells us that the rephrased requirement is not enforceable by the Light Unit domain. We repeat the process, adding a weaker breadcrumb to the Light Unit and shifting the requirement on to the Control Unit. Finally, the Control Unit expert tells us that the Control Unit can enforce that constraint, so we can stop.

By shifting the assumption to a different domain, we have satisfied the domain experts but we have increased the *traceability footprint* of the requirement. Our argument now shows that the correctness of the requirement depends on three domains (Cars, Light Unit, Control Unit). If the system cannot be altered, then this sort of sacrifice must be made.

Change Domain If the domain is a designed or machine domain (in the problem frames notation) then there is a possibility of changing the domain to match the requirement, rather than the other way around. This can be the right option if the requirement is a safety- or mission-critical property, and thus it is especially important that it have a simple and concise argument (one with a small traceability footprint). In this case, one may wish to redesign the domain rather than expand the footprint by shifting the property elsewhere.

For example, back in our traffic light example, we might decide that we cannot afford to have a footprint that includes all three domains, and that we are willing to redesign the Cars domain to keep the argument simple. We might install computer chips into the cars that prevent them from entering an intersection at the same time. We have increased the complexity of the cars domain and required that it be redesigned, but we have kept the requirement's traceability footprint contained to a single domain.

3.1.5 Progression Uniqueness

One consequence of a human-guided process is that not all humans will produce the same argument when applying the process. Roughly speaking, deviations can happen through the selection of different breadcrumbs or through the selection of different global heuristics.

Adding Different Breadcrumbs

Requirement progression guarantees that the derived breadcrumbs will be *sufficient* to enforce the original requirement, but it does not guarantee that they will be *necessary* or *minimal*. In the course of performing progression, it is legal to add breadcrumbs that are stronger than what is necessary to proceed. Doing so does not violate the guarantee that the breadcrumbs are strong enough, but it can introduce undesired implementation bias and more expensive validation.

It is important that a human be *permitted* to add assumptions that are stronger than needed, as a slightly stronger assumption might be much simpler to express and therefore easier to interpret by a domain specialist. A logically minimal constraint may not be minimal in complexity or length, and may not form a coherent statement to a human reader. Our assumptions are only as good as our ability to discharge them, so it is acceptable to sacrifice minimality in order to improve clarity.

However, within the set of clear and meaningful assumptions, it is better to pick the weakest, as that incurs less validation work and less implementation bias. Requirement progression encourages more minimal statements, even though it permits stronger ones. When adding a breadcrumb, the analyst is not asking "What do I know about this domain?" but rather "What would let me push the requirement onward?". In general, one should assume that a human will apply as little (intellectual) effort as possible to complete the argument. If the task is phrased as listing facts that are true of the domain, then it is less effort to just list everything known. If the task is phrased as making progress pushing the goal towards the machine, then it is less effort to just list the facts needed to make one more step.

We saw this happen in the traffic light example. The first time through, we omitted assumptions about the red lights, since we found that we only needed assumptions about green lights in order to make progress. If one believes the assumptions we introduced (challenged in Section **??**), then the reduced assumptions only mentioning green lights are easier to enforce, understand, and validate.

Choosing Different Targets Domains

In this chapter, we have guided progression with the heuristic of shifting the requirement towards the machine domain. The correct execution of progressions does not rely on that heuristic. There need not be a (unique) machine domain, and one could pick any target domain to guide progression. For example, in the BPTC logging example, we chose the TCS as the target. We could instead have chosen any of the other domains as the target and still have performed progression.

Having a target articulates the task in the form “Make assumptions that these domains will handle the parts of the requirement that the target domain cannot handle”, thus helping to determine what sorts of domain assumptions are likely to be helpful. Different choices of targets will not change what steps are legal but may affect which ones are selected by the analyst. From a logical standpoint, given any target, it is possible to produce the same set of breadcrumbs on the domains. However, from a process standpoint, different targets may bias humans towards introducing different assumptions and performing different rephrasings.

Our experience is that it is best to target the domain under design, especially if it is a software domain. One tends to produce relatively weak breadcrumbs, and leave much of the strength of the requirement in the goal itself. Breadcrumbs are often equivalence claims of the form “phenomenon p carries the same information as phenomenon q if interpreted in this manner...”. Such breadcrumbs lends themselves to easy rephrasings – just replace references to p in the goal with references to the indicated interpretation of q. The breadcrumbs introduced in this chapter and in our later case studies are almost entirely equivalence statements.

Because of this tendency, the harder and more complex parts of the requirement end up being left over in the final specification constraint (on our target domain), rather than being spun off as breadcrumbs. This works well if the target is the domain under design, so that we can ensure the tricky parts are enforced, while only making weak assumptions about the environment.

3.1.6 Automatic Analysis

It is not necessary to combine this approach with automatic analysis tools (such as Alloy), although in practice it is extremely difficult to construct valid arguments without tool support. The same process could be performed using informal reasoning or a different formal logic and still be helpful for structuring the argument, making domain assumptions explicit, and providing a trace of the analyst’s reasoning. The language for representing domain properties and the method for discharging the rephrasing implications should be chosen based on the analyst's experience, the type of requirement being analyzed, and the level of confidence desired.

3.1.7 Are These Examples Too Small?

One might think that requirement progression will only work on small examples such as the ones shown in this section. Our experience is that most problems, even very complex ones, can be represented by relatively simple problem diagrams but that those diagrams do not quite fit existing frames and frame concerns. For example, in our work with the BPTC, we have never needed a problem diagram with more than a dozen domains. As we will see in Chapters 4 and 5, even very complex systems can have small problem frame diagrams for critical cross-cutting concerns. While these technique may well scale to more complex diagrams, our experience is that simple diagrams are preferable and provide sufficient detail to build dependability arguments.

3.1.8 Related Techniques

Central to our efforts to build dependability cases is the use of problem progression to derive checkable specifications from system requirements. While progression has proved to be the right technique in the context of the other techniques we are composing, other techniques might better fill that gap in the context of other component techniques. In a different context, one might use similar work that has been done on synthesizing problem frames with assurance cases [13, 9, 8]. That work does not integrate as well with relational code analysis tools (like Forge [2]), and we find it to permit less intuitively phrased requirements (during elicitation and designation). As such, it does not fill the niche we need filled in our end-to-end argument.

Chapter 4

Case Study: BPTC Dose Delivery

4.1 Discoveries

In the course of our analysis, we identified both *current* and *future* vulnerabilities – undocumented assumptions that are critical to system safety. Some of these were discovered directly by our analysis, and other were discovered simply through the act of articulating the system architecture and requirements. Our experience is that much of the safety gains from building a dependability argument come from the mere act of building the argument, apart from the actual results of the analysis itself. Here, we make a more general assessment of the primary vulnerabilities of the system.

Current vulnerabilities represent assumptions made in the dependability argument which are not properly enforced by the relevant components.

SQL injection: While performing separability analysis on the dose information stored in the database, we discovered that the system is vulnerable to SQL-injection attacks. The comment field of a patient entry in the database is permitted to contain arbitrary text, and provides a place for doctors and other hospital personnel to write free-form comments about the prescribed treatment. If the comment field contains fragments of SQL syntax, those fragments will be executed when a query is made of the patient, in turn causing arbitrary changes to the prescription database.

Such an attack is unlikely, since the system is on a closed network, does not have public terminals or access points, and is operated by non-malicious users. Were a hospital employee malicious, there would be easier forms of sabotage. An attack could be accidentally introduced if a programmer used the patient comment field to jot down a note about how to query that patient. However, the existence of such an attack is more of a concern because it indicates a lack of care in checking the effects of queries before they are executed. For example, one might want to include access control to the database, so that only certain employees can overwrite prescriptions. Doing so would protecting against the scenario in which the treatment manager generates a bad query that overwrites prescriptions, as the treatment manager would not have write access and thus could not corrupt the database.

network delays: If a message is delayed on the network and delivered an hour or more later, then it might arrive during a different treatment session. If this happens to a message carrying the current patient's ID, then the system might load the last patient's dose to the next patient.

We were not able to ascertain from the network documentation whether or not it guaranteed timely delivery of messages: The network is proprietary, so we cannot directly analyze its sourcecode. The network is no longer commercially supported, so we cannot ask the network providers. The race conditions and cache heuristics present in networks make blackbox testing of the system of limited value.

One could address this concern by adding additional information to each message, so that old messages can be discarded by the receiver. For example, messages could include the session ID, and recipients would discard any message from a prior session. Simply having messages expire after a short time on the network would help, but would provide less confidence than a direct check – it would not, for example, protect against expert operators who can send and resend messages very rapidly.

patient identification: Our largest concern lies in the process by which the human therapist identifies a patient and selects that patient from a list displayed by the GUI. As described earlier in this chapter, there are a number of scenarios whereby a therapist might select the wrong patient, especially if there are a large number of active patients and if several patients have similar names.

Protecting against such errors is difficult, but there are safeguards that could be added to the GUI itself. For example, the GUI might recognize similar patient names (especially ones that are currently not visible on screen), and raise a warning to the therapist to double check the selection. Alternatively, one might have an automatic scan of a barcode on the patient ID tag, in parallel to the human identification process, and halt if the two do not agree.

Future vulnerabilities represent assumptions made in the dependability argument that were not previously documented, but which turned out to hold when inspected. They represent properties that might be violated when the system is modified, and thus should be properly documented in order to permit safe maintenance. For example:

network: We assume that the network does not drop messages, or that it detects and resends dropped messages. We assume that the network does not corrupt messages, or that it has error detecting codes to catch corruptions and resend the data. The current network infrastructure (RTworks) provides these guarantees in its documentation.

database: Queries generated about the database make assumptions about the format and organization of information in the prescription database. It makes assumptions about the names and orders of columns, and that dose information is stored in certain units (e.g. joules versus rads). Changing the database format, even slightly, would require changes to many portions of the treatment manager code involved in send, receiving, and processing queries and network messages pertaining to queries.

GUI: The GUI was automatically generated with a commercial tool. If it were re-generated, it would need to be re-evaluated (unless the generate tool itself were proven correct). Specifically, we rely heavily on the authenticity of the information shown to the therapist and the influence of mouse and keyboard clicks upon the internal state of the GUI and the messages it sends to the treatment manager.

code structure: The code is overall poorly structured and lacks useful documentation. As a result, the code is much less transparent than it could have been, limiting the value of manual code reviews.

The code is written in C and manipulates memory references directly. C is not a memory safe language, although there are subsets and coding styles that reduce the risk of memory conflicts.

The code makes extensive use of global variables that are shared between portions of the code with widely varying functions. There is no access control to the globals, so non-critical portions of the software can corrupt the data used by critical portions. As such, the entire code base must be considered critical.

The code has unnecessary redundancy in its data and algorithms. For example, some data about patient dose is stored in two different global variables, one of which is used in some procedures and other of which is used in other procedures. They are currently kept in synch, but such redundancy is a recipe for introducing errors during modification. Similarly, portions of the algorithmic code are repeated in different locations, producing a dual-maintenance problem if the algorithm is updated.

Future vulnerabilities would be a minor concern if the system were never modified. However, there are a couple of likely scenarios in which the system will be significantly modified.

discontinued systems: The network infrastructure currently used (RTworks) has not been commercially supported for about 5 years. At the time that it was installed, it was a reputable system that provided the necessary guarantees for safe operation. However, if new functionality is needed, or if an error is discovered, an entirely new network infrastructure might need to be added. At that time, it will be essential to know what guarantees about message delivery are important to system safety.

hospital additions: The BPTC has plans to add a new firing mode to the system. Under current firing modes, the beam is fired in a broad and fairly low-intensity pattern, bathing the tumor in radiation. The proposed mode, called *pencil beam scanning*, would rapidly sweep a narrow, high-intensity beam back and forth across the tumor. Pencil beam scanning provides a more precise boundary around the tumor and thus causes less collateral damage. Apart from adding new failure modes to the system (the beam moving too slowly or halting in place), adding a new firing mode would involve significant changes to the Treatment Manager and other components in the system. When such a change is implemented, it will be vital know the set of assumptions that must be maintained as the components are altered.

In the long term, the BPTC plans to add new treatment rooms, to accommodate the high demand for proton therapy. Doing so requires minor changes to the software running in the MCR and to the shared database. These changes would be less pervasive than adding a new firing mode, but would still require a clear set of assumptions, lest those assumptions be inadvertently violated during modification. However, the last time that a room was added (room number three), it violated the emergency stop button's ability to halt the beam [11].

Further reflections on our analysis of the BPTC are described in Chapter 7.

4.1.1 Effort

Our analysis required about two months of person-time, counting both the time spend by our research group and the time spend by MGH employees. Figure 4.1 uses the BPTC CDAD to break this time down, and reveals that almost half of this time (three of the eight weeks) was spent on manual translation tasks that have since been rendered obsolete by automation such as CForge and JForge [2]. Of the remaining five weeks, one was spent gathering a basic understanding of the system – work that can be re-used on future dependability arguments for the BPTC. The remaining time is one person-month of work, and we estimate that matching dependability arguments could be built for the other high priority concerns for the BPTC in about one month

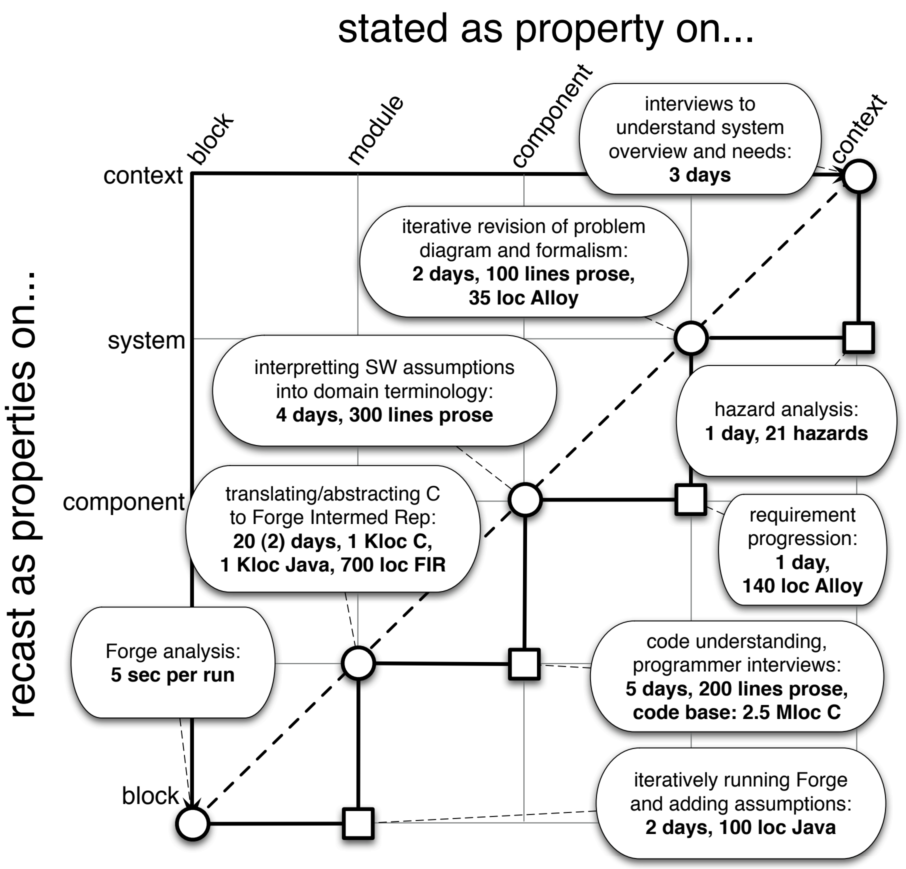
stated as property on...
recast as properties on...
block
module
component
context
context
system
component
block
interviews to understand system overview and needs: 3 days
iterative revision of problem diagram and formalism: 2 days, 100 lines prose, 35 loc Alloy
interpretting SW assumptions into domain terminology: 4 days, 300 lines prose
hazard analysis: 1 day, 21 hazards
translating/abstracting C to Forge Intermed Rep: 20 (2) days, 1 Kloc C, 1 Kloc Java, 700 loc FIR
requirement progression: 1 day, 140 loc Alloy
Forge analysis: 5 sec per run
code understanding, programmer interviews: 5 days, 200 lines prose, code base: 2.5 Mloc C
iteratively running Forge and adding assumptions: 2 days, 100 loc Java

Figure 4-1: Time spent building the BPTC dependability argument for dose delivery.

each, adding up to around a year of work. This cost is a bit high, but is a fraction of the cost of building and testing the system. Less critical systems would justify using lighter weight analyses, as they could tolerate dependability arguments with lower confidence.

Chapter 5

Case Study: Voting Auditability

5.1 Achievements

We articulated the existing intuition for the fidelity, secrecy, and auditability of the *Pret a Voter* system, using an unambiguous formal model, and confirmed those three arguments through automatic analysis of our model. Our analysis not only demonstrates that the desired properties hold, but it also provides structured, traceable, and readable argument describing *why* the system satisfies its requirements.

5.1.1 Clean Division

In building the argument, we separated the system-level arguments from low-level crypto-graphic arguments. The designers had previously conflated the arguments together, making reasoning about the system more difficult. Requirement progression provided such a boundary – it argues why a certain set of assumptions enforce the requirement, separate from the argument that those assumptions are provided by the proposed cryptographic protocols. Put another way, we identified the appropriate level of detail for the system argument – exposing certain properties about the cryptographic theorems and protocols while hiding others. The automatic analysis confirms that we exposed an appropriate level of detail.

We also re-enforced our belief that requirement progression is faster and easier when supported by an expert-provided intuitive outline for how we expect the argument to look. That argument guided progression, and allowed us to finish the process in just a few hours.

5.1.2 Leveraging Fidelity for Secrecy and Auditability

We demonstrated how building the fidelity, secrecy, and auditability arguments in tandem can make them not only easier but also more thorough.

- We built the *fidelity* argument using requirement progression, which provided automatic analysis to confirm the argument. The resulting set of breadcrumb assumptions were encoded in an Alloy model.
- When we built *secrecy*, we leveraged that Alloy model in two ways: We used the structure of the data (the sets and relations) to build the structure of the adversary's knowledge base. We used the breadcrumb assumptions to derive a core set of inferences. Those inferences can be expanded, but we found that just the derived inferences were enough to model basic attacks against the system's secure information.
- The *auditability* argument rests upon identifying the correct set of properties to audit. We showed how to read that list off the fidelity and secrecy arguments – one must audit any assumption in the fidelity argument that references phenomena that are initially hidden in the secrecy argument. The means by which one audits those assumptions is a domain-specific question, but we easily produce a complete list of *what* needs to be audited.

5.1.3 Discoveries

For the most part, our analysis confirmed the system as proposed. In a couple of cases, we discovered some minor surprises.

- The *Pret a Voter* system as proposed uses onions to encode not only the list of candidate names but also the position of the marking on the re-encrypted receipts. Our analysis shows that encoding just the list of names is sufficient to provide the three goals (fidelity, secrecy, auditability). Encoding the markings does not interfere with those goals, but adds unnecessary complication. In an actual implementation, it may be useful to encode the markings simply to more fully automate the re-encryption process – so that the entire receipt is encoded in an onion and there are no slips of paper to pass around.

- Before our collaboration, Peter Ryan had proposed a method for obfuscating what list of candidates was given to a particular voter, but he was not sure if that mechanism was necessary. Our analysis shows that it is indeed necessary to provide secrecy.

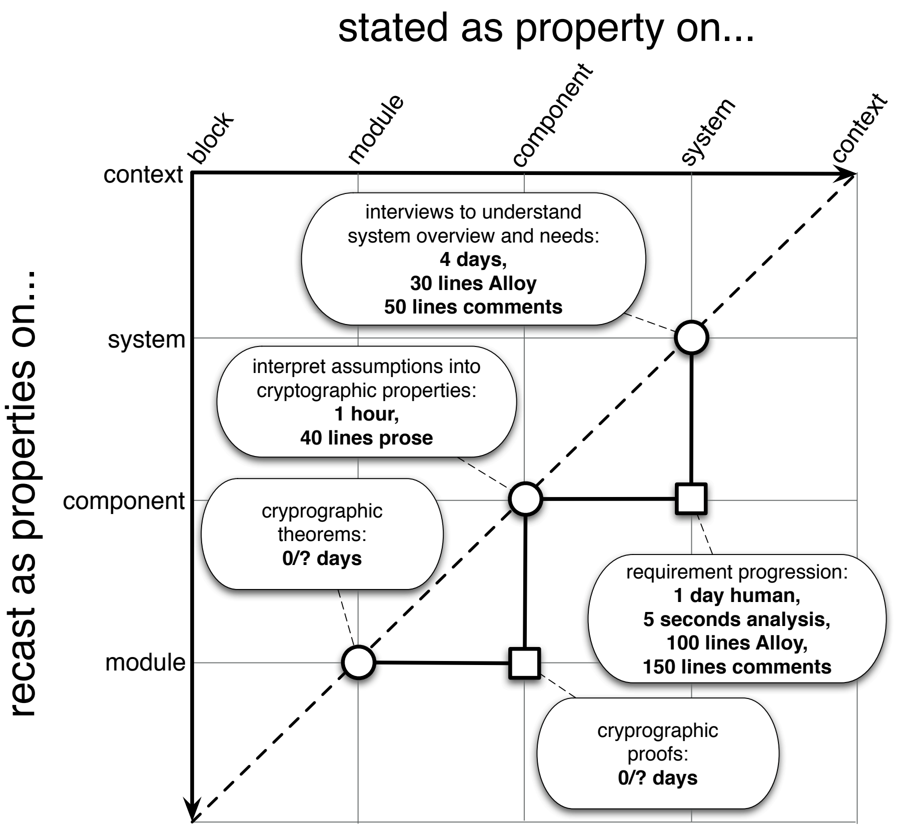
stated as property on...
block
module
component
system
context
recast as properties on...
context
system
component
module
interviews to understand system overview and needs: 4 days, 30 lines Alloy 50 lines comments
interpret assumptions into cryptographic properties: 1 hour, 40 lines prose
cryprographic theorems: 0/? days
requirement progression: 1 day human, 5 seconds analysis, 100 lines Alloy, 150 lines comments
cryprographic proofs: 0/? days

Figure 5-1: Time spent building the fidelity argument.

5.1.4 Effort

Our analysis required two weeks (10 days) of work, counting time spent by all participants. The fidelity argument took five days of work, four of which were spent just understanding the system and one of which was spend performing the actual requirement progression. See Figure 5.1.3. The secrecy argument took four

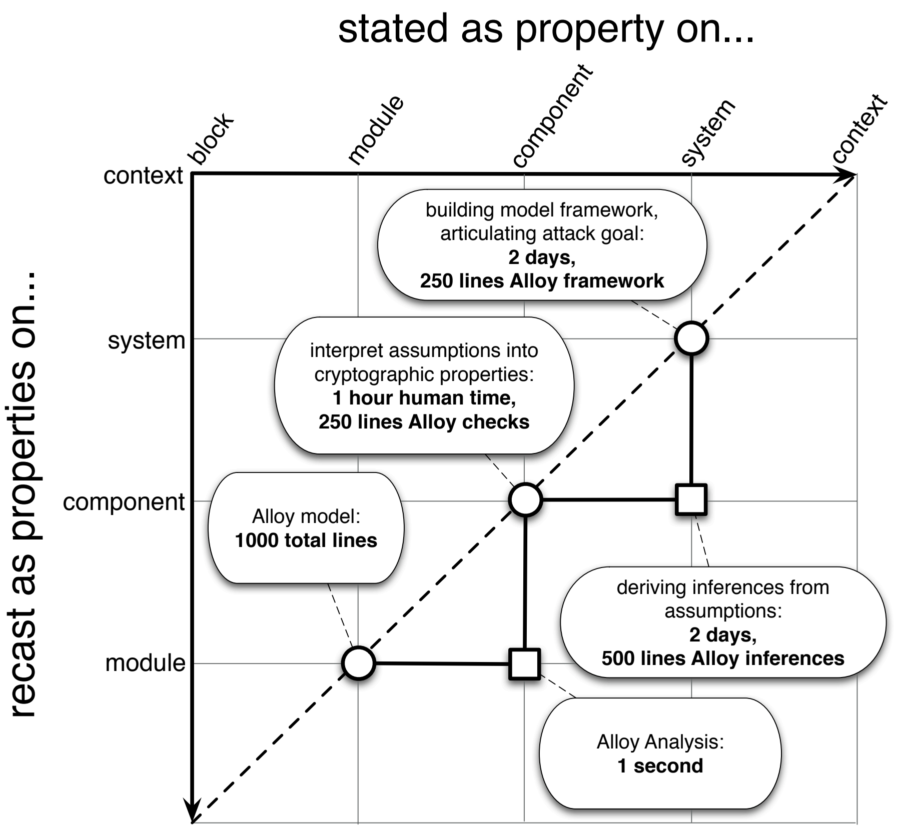
stated as property on...
block
module
component
system
context
recast as properties on...
context
system
component
module
building model framework, articulating attack goal: **2 days, 250 lines Alloy framework**
interpret assumptions into cryptographic properties: **1 hour human time, 250 lines Alloy checks**
Alloy model: **1000 total lines**
deriving inferences from assumptions: **2 days, 500 lines Alloy inferences**
Alloy Analysis: **1 second**

Figure 5-2: Time spent building the secrecy argument.

days, two of which were spent building the model framework and two of which were spent deriving inferences for the adversary. See Figure 5.1.3. Identifying the list of properties to audit required less than one day.

These counts do not include the time spent by Peter Ryan and his collegues to establish the assumptions with cryptographic protocols. That work had already been completed when we performed our analysis, and this timing data only reflects the additional work needed to build the dependability argument on top of prior work.

Chapter 6

Related Work

Chapter 7

Conclusions

We have proposed and applied a methodology with which a skilled analyst can build end-to-end dependability arguments for complex, software-intensive systems with reasonable human effort. These arguments not only validate the system design as whole, but they also provide traceability – linking system level requirements to low level assumptions about individual components in the system.

7.1 Contributions and Achievements

We introduced *requirement progression* (Chapter 3), a systematic, guided method for decomposing a system requirement into a set of component assumptions. A system requirement articulates the needs of the overall system, but no one engineer or specialist is qualified to confirm or deny that broad of a requirement, since it references aspects of many components. The component assumptions (*breadcrumbs*) generated by requirement progression articulate important properties about individual components, which can be independently assessed by appropriate domain specialists.

The progression process is incremental and local – each step of a progression only requires the analyst to reason about one domain and its interfaces. We provide a set of guidelines to help the analyst develop the progression efficiently, which are based on the structure of the system's Problem Diagram [6]. The analyst can automatically check the steps of the progression (using Alloy [5]), ensuring that the resulting set of domain assumptions will indeed be strong enough to enforce the original requirement.

We introduced the *Component Dependability Argument Diagrams* (Chapter 2), a notation for classifying analysis techniques and for composing them together to form an integrated end-to-end argument. CDADs show how requirement progression links into other analyses from related fields of study, helping the analyst select and compose techniques to build an end-to-end dependability argument.

In the proton therapy case study (Chapter 4), we saw how requirement progression can be combined with automatic code analysis to discharge domain assumptions on software components. The resulting argument, illustrated with a CDAD, constitutes a dependability argument for a critical aspect of a working radiation therapy medical device.

In the voting case study (Chapter 5), we saw how to analyze a system with multiple, apparently contradictory, requirements – fidelity, secrecy, and auditability. By using requirement progression to build an Alloy model of fidelity, we saw how it was then easier and more systematic to build secrecy and auditability arguments. The resulting analysis validates the design of the *Pret a Voter* election scheme [12].

7.2 Limitations

To understand an approach, one must understand its limits – both the incidental limitations of the particular approach and the inherent limitations of all approaches in that style. The limitations we discuss below are reasonable restrictions if one wants to build end-to-end confidence in a system, but it is important to be aware of the sort of investment one must make and results one can obtain. Much of the future work (Section 7.4) revolves around reducing or eliminating these limitations.

7.2.1 Vulnerabilities Versus Errors

This sort of system analysis fundamentally discovers *vulnerabilities* rather than *errors*. One sometimes discovers errors in the course of building the argument and understanding the needs for the system, but the focus is on the discovery and documentation of component assumptions. Sometimes the mere act of building a dependability case will increase dependability simply by focusing attention and prioritizing concerns about the different components. More typically, errors are discovered when one attempts to discharge and validate component assumptions, and the ability to perform that validation limits the errors that can be uncovered in this manner.

7.2.2 Human Domains

In some domains, discharging assumptions is relatively easy and thorough. For example, in the BPTC case study (Chapter 4), I showed how to link requirements progression and system analysis to automatic code analysis. Other technical domains, such as electro-mechanical devices, can similarly be analyzed by well-established means.

However, it was hard to analyze human domains – such as a therapist who identifies a BPTC patient and selects the matching name from a list in the GUI. Interpreting assumptions made about human processes is low cost but not as systematic as the rest of our analyses. Even our ad-hoc analysis of such components, in the BPTC example, revealed a large number of vulnerabilities and critical undocumented assumptions (Section **??**). However, it was unclear now to build proper confidence that more such assumptions and vulnerabilities do not exist.

Even with trained, experienced operators, it is hard to build confidence. One of the big concerns in human controlled systems is *habituation* – experienced operators get used to the normal modes of operations, and thus become less likely to notice deviations from the norm. As such, systems with human operators can actually become less safe the longer they operate, even as the humans become more experienced. We suspect that extending the type of classification proposed by Donald Norman [10] would help to provide such confidence.

7.2.3 Support from Domain Specialists

Building an argument focused on identifying and assessing component assumptions requires that the analyst have access to experts on the system components – probably engineers and operators working on particular components of the system under analysis. We found that the analyst did not need a lot of time from those specialists, but did need to meet with them in a few key capacities:

initial interviews: The analyst will up-front specialist interviews for each component component, to build a rough understanding the basic structure of the domains and their roles in the system. This helps the analyst to build the initial problem diagram, provides intuition for the overall shape of the argument, and gives an idea of what sorts of assumptions can be reasonably made about each component.

assisting analysis: If the analyst directly participates in the analysis of the domain (as we did with the software of the Treatment Manager at the BPTC), then additional time will be incurred, depending on the efficiency of the techniques and confidence demanded.

identifying interface: A skilled analyst must separate the internal details of the domain (the realm of the specialist) from the interface of the domain (the realm of the generalist). Assumptions are made about (and phrased in terms of) the interface, but exactly what internal details are relevant to the interface is not always obvious. The analyst must resist the pressure from specialists to expose in inner workings of a domain, and be able to abstract the interface out of the Specialists (much more detailed) explanation of the entire component.

interpreting assumptions: For each assumption made about a domain, the analyst must interpret that assumption back into the language of the domain, thus putting it in a form that the specialists can understand and evaluate. Doing so requires an understanding of the language and terminology used by the domain experts, at least at a high level. For example, interpreting a code assumption involves phrasing it in terms of input and output variables in the code. In contrast, assumptions about physics devices (e.g. the cyclotron) are best phrased in terms of the properties of the beam generated (e.g. intensity and duration).

discharging assumptions: As the argument takes form, the analyst begins to need to discharge assumptions made about the components, which involves frequent (but small) questions to be answered by particular specialists.

If no expert is available for one of the components, there are several options available to the analyst:

(1) Accept lower confidence in the system as a whole. If one cannot confidently discharge assumptions about one of the domains, then that becomes a weak link in the argument that will reduce confidence in the dependability of the system as a whole.

(2) Rework or replace the component, effectively building a new component for which you now have an expert. This option can be costly, but is can also fit with an iterative development process, such as those adhering to Fred Brook's advice:

> Plan to throw one away. You will do that, anyway. Your only choice is whether to try to sell the throwaway to customers.
> *Fred Brooks* [4]

(3) Use components that are transparent, clear, or simple. That is, use components for whom anyone can become an expert through careful examination. Some components are fundamentally too complex to make transparent to outsiders, but the general engineering experience is that simpler components are better.

> Everything should be made as simple as possible, but not simpler.
> *Albert Einstein*

Confidence in the system relies on both confidence in the *system argument* and confidence in the *component assumptions* that underly that argument. Without both, confidence is impaired.

7.2.4 Analyst Expertise

The role of analyst – actually building the dependability argument – should, itself, be treated as a specialized task demanding proper background and training. Only a small number of analysts are needed, perhaps as few as just one, but that analyst must have a certain technical aptitude

In general, the analyst must be capable of system level reasoning – a generalist not a specialist. It is the analyst's job to to communicate with different kinds of engineers and extract the relevant information – both a technical skill (getting past domain specific terminology) and a social skill (convincing engineers to help build the safety argument and managers that it is a worthwhile expenditure of resources).

For our approach, the analyst must be capable of using and interpreting formal notation. We used Alloy as our formal language, although other formalisms can also be used to articulate the assumptions made during requirement progression. However, some sort of formal language is needed, both to unambiguously communicate and record the assumption, and also so that the system argument is amenable to automatic analysis. The analyst must both translate assumptions and requirements into the formal language (based on informal descriptions provided by specialists), as well as being able to interpret the assumptions discovered during requirement progression back into language that makes sense to the specialists (whose job it is to confirm or deny those assumptions). The analyst must also be comfortable at rephrasing the requirement (during the requirement progression process) and structuring/debugging the associated model.

Relational logic provided us with a useful formalism. We found it to be a fairly intuitive way to precisely describe requirements (and found that most technical people, even from other engineering disciplines, could make sense of Alloy statements). It also fit nicely with the Forge analysis tool [2], which was capable of automatically discharging relational claims about code fragments.

Domain Knowledge During Validation

When analyzing and interpreting assumptions into domain language (e.g. the code analysis for the BPTC case study), the analyst must be aware of the kinds of failures possible in that domain. That is, how might the domain violate the assumption made about it? This knowledge can come partly from talking to domain experts (personnel working on the system) but the analyst needs to have a basic idea of what to look for. Put another way, an analyst should be a *generalist* capable to talking to a range of *specialists* in order to gain an understanding of the relevant domains.

For example, one vulnerability we found on the BPTC involved an SQL injection attack. We discovered the attack while we were performing a separability analysis to determine if the data read out of the data base could have been overwritten. While we did not initially look for SQL injection attacks, we only discovered the attack because were were (peripherally) aware of the existence of such attacks. The analysis uncovered the assumption – the database values are currently the same as when they were initialized – but the analyst had to come up with the particular failure mode that could violate that constraint – SQL injection attacks.

This observation ties into our philosophy of providing a technique that is *systematic* but not *automatic*. No tool or technique can substitute for domain expertise, although our technique helps an analyst decompose a system requirement (that no one person is qualified to confirm) into a set of domain assumptions (that individual domain experts are qualified to confirm). We aid the analyst in identifying what question to ask what specialist, but do not replace the need for the specialists nor do we replace the need for an analyst who can reason about abstract, system-level concerns.

7.2.5 Code Analysis

A particular instance of relying on expert specialists to validate assumptions is the reliance on expert software engineers when analyzing software. We found that, while automatic analysis eased the process and make it more thorough, it did not substitute for a well structured or well explained code base. Our analysis of the BPTC code was dependent on the head programmer (Doug Miller) and his broad understanding of how the code fit together. When he moved away, our ability to discharge code assumptions with confidence went down, as it became much harder to identify the subset of the code relevant to particular assumptions (which was necessary, since our analysis tools could not scale to the entire code base).

In order to keep performing analyses without an expert on the code base, we would either have needed a tool that scaled better than Forge (but which could still discharge arbitrary relational claims), or the code base would have had to be better structured, so that we could have more easily identified the relevant subset. We suspect that tools (like Forge) that are expressive enough to handle relational claims about real code (including loops, recursion, conditionals, arithmetic) will never scale well enough to handle millions of lines of code without some amount of human assistance.

The more reasonable path is to demand that the code be structured to reflect the safety argument and execution modes, thus making it possible to easily and confidently identify a small portion of the code relevant to a particular concern. While in theory this might not be possible for an arbitrary algorithm implemented in code, in practice our impression was that there were no such obstacles facing the BPTC code. Having written the code once, a complete rewrite of the code (i.e. iterative design) would have produced code that was transparently correct to an outside observer.

For example, the BPTC code uses many globals that are only used in a few places (and thus could instead be passed around, have access control, or have not be global to the entire code). The code also lumps all data of one type together, rather than all data of one purpose together. For example, all messages are listed in one huge case statement (which must be kept synchronized with other lists which declare the valid types of messagers). Modes and sub-modes are kept as separate variables, with no assertions to maintain the invariant that you are not in mode A while you are also in a submode of mode B.

In spite of these limitations, this work does show that (even without automatic tool support or better code structure) it is possible to link requirement progression to code analysis, thereby building deep end-to-end arguments. The cost of manual analysis was still a fraction of the cost of building and testing the system, and was quite reasonable for a safety critical system like the BPTC. If one reduced the cost, then our techniques would be applicable to a wider range of systems.

7.3 Experience and Reflections

While the research focuses on the technical aspects of building and checking an argument, a lot of the skill involved is communicating effectively with the specialists involved in the system.

> The art of fortifying does not consist of applying rules or following a procedure, but of good sense and experience.
> *Marechal Sebastien le Prestre de Vauban*

(1633-1707, Military Engineer to King Louis XIV)

7.3.1 Types of Personnel

In the course of building the BPTC dependability argument, we talked with the following types of specialists (ordered with the most frequently accessed personnel first):

- The lead software engineer and programmer – Doug Miller. Extensive contact and support during the code analysis. Provided overview of code fragments and answers to particular questions about blocks of the code.
- The head of the BPTC, responsible for managing, certifying, and providing funding for the project – Jay Flanz. Extensive contact early in the project, but less as the analysis moved to lower levels. Useful for identifying whom we should speak to, and determining the correct set of requirements.
- The head physicist, who works both on calibrating the system, performing research on it, and helping physicians translate their prescriptions into radiation treatments. Moderate contact early in the project. Limited contact late in project. Useful for understanding the precise definition of a correct dose, including the somewhat subtle definition of location. He also helped describe the overall system structure.
- Operators who work in the Master Control Room (MCR), coordinating the therapists in the individual treatment rooms. Moderate contact mid-way through project. Helpful in understanding day-to-day process and what normal operating conditions are like, and what sorts of minor errors occur routinely.
- Therapists who directly contact patients and prep them for treatment. Limited contact due to hospital restrictions about access during operating hours. Potentially helpful to analyzing patient identification protocol, but not helpful in practice due to limited availability.
- Physicians who write prescriptions for patients undergoing radiation treatment. Limited contact during analysis of database. Relevant to the initial assignment of criticality to hazards, to determine the danger posed by different failure modes. Would be key to building a more thorough hazard analysis or requirement elicitation phase.
- Patients undergoing treatment. No contact due to privacy restrictions. Might have helped understand the patient identification process better, to better understand the likelihood of different sorts of false-identification scenarios.

For the voting case study, I spoke almost exclusively with Peter Ryan, who originally proposed the system and is currently one of the leading researchers developing it. It is a much smaller system than the BPTC, involving fewer different types of engineers, and our total analysis took about a quarter of the time (two weeks instead of two months). Peter Ryan is an academic researcher, with a background in cryptography and voting systems, and a side interest in system analysis. He thus played both the role of a specialist (knowing what assumptions could be guaranteed by cryptographic proofs) and a generalist (giving a summary of the overall system). Before our collaboration, he already had an intuitive safety argument, which proved helpful in guiding progression.

7.3.2 Mediums of Communication

Initially, we used the problem diagrams themselves as a means of guiding communication with the BPTC personnel. However, this proved to be less fruitful than using the assumptions (generated via requirement progression), as isolated concrete questions. When shown a high-level overview of the system, the specialists tended to trust the diagram's accuracy more than we wanted, and thus not provide proper feedback on our understanding of the system structure. In contrast, concrete claims or questions produced elaborate and informed responses.

For example, an early version of the BPTC problem diagram had a direct connection between the GUI and the prescription database. At one point, the software lead made an aside along the lines "I guess that's some sort of abstracted view of dataflow" when actually it was a mistake – the database information only gets to the GUI via the TM and network (which were also on that diagram). However, when shown the matching domain assumption that the messages sent by the GUI are received by the DB, he immediately pointed out that no such message existed, and explained the indirect path of communication between those two points. Furthermore, he pointed to the particular parts of the code relevant to passing that message along and processing it.

In general, we found that using breadcrumbs as a medium of communication was more productive, as they provide concrete questions. The engineers and specialists tended to be concrete thinkers who were deeply grounded in their particular component. As such, they were very able to answer very hard (and slightly vague) questions about their components, but were not able to give us a useful overview of how the component worked and what key properties it provided.

When we did end up showing problem diagrams to the programmers, we ended up just pretending they were dataflow diagrams – a more concrete and familiar notation for a programmer. For the most part, phenomena in our diagrams represented the flow of information (or the issuing of commands) between components, and so viewing them as dataflow diagrams was fine for checking our broad understanding.

In the voting case study, Peter Ryan was able to directly understand the Problem Frame notation, but still needed help in making sense of the details of larger Alloy models.

7.3.3 Styles of Thinking

While we interacted with only a small sample number of engineers, a few patterns did emerge about how the different types of engineers tended to describe their components. The physicists were more apt to think declaratively than the programmers – they were more apt to give a declarative statement about the system (such-and-such a property will always be true of the beam) and less likely to make an operational statement (X happens and so Y then happens). In contrast, programmers were more able to separate abstraction layers, describing the overall shape of information in the system without diving into the details of the code (the A-related stuff happens in this part of the code, and the B-related stuff happen in this chunk of code). The physicists seemed comfortable with thinking about non-temporal invariants (X is always greater than Y), but less comfortable deciding what details to leave out of an explanation. Roughly speaking, physicists told us too much, programmers told us too little, and we had to adapt our questions accordingly.

7.3.4 BPTC Safety Culture

The BPTC specialists tended to have broad and deep understandings of their own domains. The overall system was small enough that there were only a few specialists of each type, and thus individual people could answer fairly broad questions about a component. This made it easy to find a specialist qualified to validate a given domain assumption, or at least to help us in validating it.

However, while the individuals were knowledgeable, the system documentation was too vague and too sparse. It gave little or no overview of the system nor any argument for why the system would work, and simply described details of how the system actually operated. As such, a lot of the relevant knowledge to maintaining safety is in the heads of the specialists, and is lost when those specialists are replaced or retire.

The head of the center, Jay Flanz, was very concerned with safety issues, and very supportive of our efforts to analyze the system. He was unsure of how to build an appropriate safety argument, and was concerned that the FDA certification process did not provide the confidence he wanted in the system. He knew that the testing was not enough, but he was not sure what to do other than add additional safety interlocks in response to incidents as they occurred.

Overall, the personnel had a conscious understanding of the safety-critical nature of their device. They understood the different types of dangers presented, reinforced by their physical proximity to the device (and thus immediate personal concern in the safety of the proton beam). They had proper respect not only for the immediate dangers of overdosing a patient, but also the dangers of poor logging or non-graceful failure modes. While they lacked the techniques and expertise to build a safety argument for the system, they were motivated and skilled enough to support the construction of such an argument.

7.3.5 BPTC Conceptual Mistakes

While maintaining an overall strong safety culture, there were some particular points wherein the BPTC personnel and management made conceptual mistakes about how to reason about a complex system.

Criticality Classification

Components were not always properly classified as critical or non-critical, and thus their reliability was not always appropriated prioritized. Some components were classified as non-critical, even though they could (if they were replaced by a malicious or careless implementation) violate safety concerns.

For example, the network was not deemed safety critical, even though emergency stop commands were transmitted across it [11] and corrupt network messages could result in patients receiving someone else's treatment (Chapter 4). Similarly, in earlier work [3], we analyzed the automatic beam scheduler, responsible for allocating the proton beam between the treatment rooms. It was classified as non-critical, since the instruction to fire the beam was controlled by the therapists in the individual rooms. However, a bad scheduler could cause the beam to turn on or off at unpredictable times, causing underdoses and treatment delays (and potentially harming confused therapists or technicians).

In general, the devices classified as non-critical are the devices that we felt *should* be non-critical. However, the realities of the system architectures did not not always provide sufficient separability and modularity, meaning that the safe operation of the system ended up relying upon a wider range of components than necessary. This indicates a general need to provide better separation between critical and non-critical components, so that one can better assign effort to the critical ones and ignore the less critical ones without undermining confidence in the critical concerns.

Planning for Change

A lesser concern was with the provision of misguided generality in the software. While planning for change is difficult, as one does not know exactly how requirements will change, we found a few cases where a little more forethough would have made the system much more amenable to safe and easy modification.

For example, the code written to allocate the proton beam to one of the three rooms [3] also provided a notion of priority, so a therapist could indicate that he or she has a small child who is getting restless and needs the beam right away. The priority queue included a three-tiered system for determining which room to allocate next, including nine total possible priority levels. However, there were only three rooms, and in practice there are only two priorities – "any time is fine" and "sooner is better". The code provided generality for adding more priority levels and more types of priorities at each level, even though the current priority levels already far exceeded the system's needs.

However, the scheduler code did not provide generality for how many rooms there were. It was originally written for exactly two rooms, and had to be retro-actively (and inelegantly) extended to handle the 3rd room, when it was later added. The new code included a lot of duplicated functionality, requiring dual maintenance when modifications are made. As the center grows to meet the high demand for proton therapy, the hospital is likely to add more rooms, which will require further extensions of the code in ways it does not easily accommodate.

Human Versus Machine

The BPTC includes redundant checks and safety interlocks, combining automatic hardware checks, automatic software checks, and manual human checks. However, as the center evolved, some portions were over-automated due to inadequate requirements elicitation.

The Automatic Beam Schedule [3] implements a priority queue, used to automatically decide which room should currently have access to the proton beam.

This process was previously handled by live communication (via a telephone) between the therapist and the Master Control Room (MCR) operator. There are only three treatment rooms, and a treatment takes about an hour to complete, so the beams scheduling was not much of a burden on the MCR operator. The system was automated in response to complaints that the therapists had that they were not sure if their request was being processed or if their room had been forgotten. As such, what they needed was better visibility of the current queue, not automatic prioritization of that queue. A simple system could have provided feedback on the current queue without adding the risks and complexities of an automatic priority queue.

In contrast, we would like to see more automatic checks in the patient-identification process, to support the existing human checks. For example, scanning a barcode on a patient ID rather than reading text by eye would reduce the risk of selecting a patient with a similar name (and thus delivering the wrong dose).

7.4 Future Work

7.4.1 Tool Support for Progression

The requirement progression process is fundamentally a human process, requiring a human to guide the introduction of meaningful assumptions. However, tool support can certainly improve human processes. We currently support the human with automatic checks of proposed requirement rephrasings. We would like to extend this support to include automatic suggestions of how to proceed in the progression process, using a combination of heuristics (such as pushing the requirement arcs towards the machine domain) and mathematical inferences (such as using *prime interpolents* to propose breadcrumbs [1]).

Aside from generating suggestions, simply providing a GUI for building and maintaining problem diagrams and progressions would make the process more accessible. Such a GUI could integrate with a back-end Alloy analysis, linking the constraints in a diagram with the accompanying Alloy model that analyzes those constraints.

7.4.2 Code Analysis

The current code analysis required a fairly large amount of human effort, although only a fraction of that spent on building and testing the system. The introduction of automatic translation tools, such as CForge and JForge [2], helps to reduce this time cost. However, the scalability limitations of the Forge analysis still requires that a human invest time in building an abstraction barrier of specification stubs to isolate the relevant portion of code. However, we remain tied to Forge for our analysis because of its unique ability to check arbitrary relational claims (written in Alloy) against code. This feature permits us to smoothly integrate the code analysis with the assumptions generated by our Alloy-based requirement progression.

We feel that the gains from smoothly integrating the code analysis (Forge) with the requirements analysis (requirement progression) justifies the additional human investment. For costly or safety-critical applications, this tradeoff is sensible. However, reducing the time investment would broaden the appeal of our techniques, and make it applicable to a wider range of systems.

As mentioned earlier, one solution is to require better structured code, so that it is easier to identify the relevant subset. Another approach would be to improve Forge-like tools to scale better. A third option is to provide better tool support for automatically identifying the relevant subset. For example, one might run a slicing algorithm over the code to identify a subset small enough to hand off to Forge.

7.4.3 Integration with STAMP

Our current approach uses hazard analysis to justify the set of requirements analyzed, but that technique is not as systematic as other component arguments, and thus weakens the overall confidence of the dependability argument. For example, we believe that Leveson's STAMP [7] notation would link requirement progression to requirements elicitation, justifying why the requirements analyzed by progression are indeed the right requirements to be establishing.

7.4.4 Lightweight Techniques

We would like to experiment with applying these techniques to less critical applications, where the analysis must be cheaper but need not provide as much confidence. Working on that sort of case study would likely involve

- adding more automation and tool support so that existing techniques are lower cost,
- using CDADs to select a lighter-weight set of component techniques, and
- being more conscious about the tradeoff, not only between breadth and depth, but also between cost incurred and confidence provided.

One idea we have begun to develop to help manage that tradeoff is the *waterglass model* – an extension of the CDAD notation that guides the distribution of effort or budget across those techniques based on the confidence they provide and costs they incur. We provide a glimpse of the waterglass model in the next section.

7.5 Waterglass Model of Budget Allocation

Suppose you have selected a set of techniques that fit together to build an end-to-end dependability argument, as shown in the CDAD in Figure 7-1. Now you have to allocate effort amongst those techniques, given a limited budget.

7.5.1 Representing Component Techniques

Think of each component argument as a glass of water, as shown in Figure 7-2. The height of the glass represents the maximum confidence you could gain from the technique, the height of water within a glass shows how much confidence you are gaining given your current investment in the technique, and the diameter of the glass

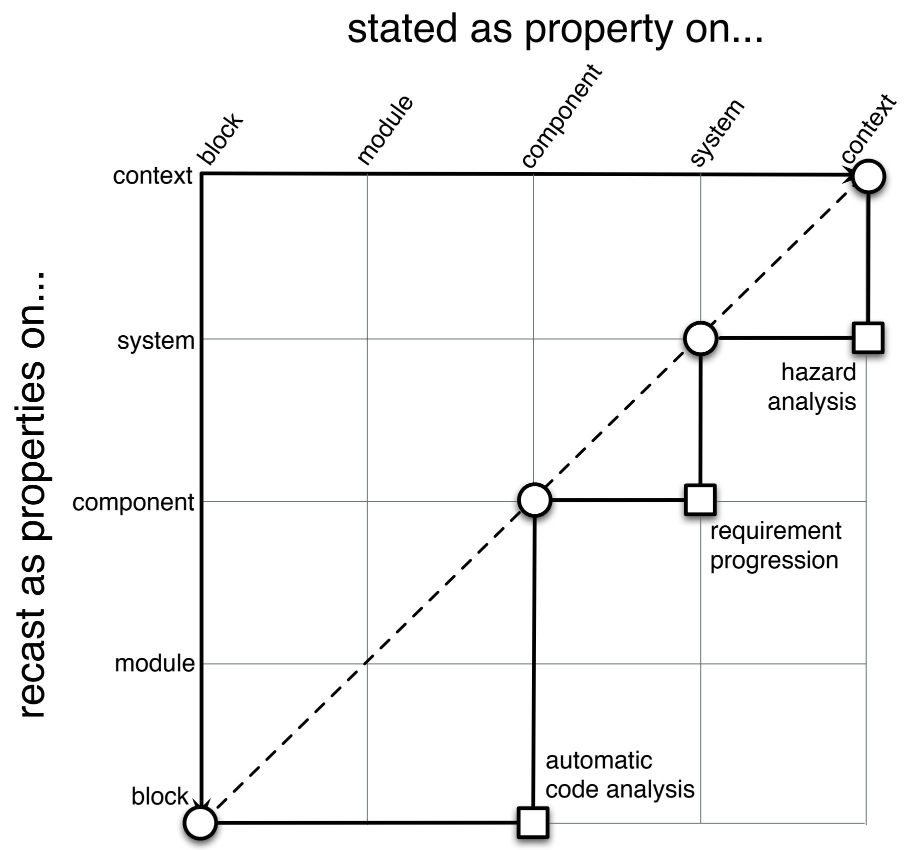

Figure 7-1: Techniques linked together to form an end-to-end dependability argument.

represents return on on investment – it takes more water to raise the level of a wider glass. Your budget is a pitcher of water, which is to be poured into the glasses.

To get an idea of the overall confidence provided by a dependability argument, line up the glasses side-by-side, as shown in Figure 7-3-a. As a rough approximation, the confidence provided by the entire argument is the minimum water level of any glass.[1] Confidence is maximized by equalizing the water level in all the glasses. Imagine putting a pipe between the glasses so that they even themselves out, producing the highest possible minimum (Figure 7-4-b).

[1]The actual confidence is surely a more complex function, but it is one that punishes you severely for having one glass much lower than the rest and rewards you very little for having one glass much higher than the rest. The minimum function is a good approximation for the purposes of this narration.

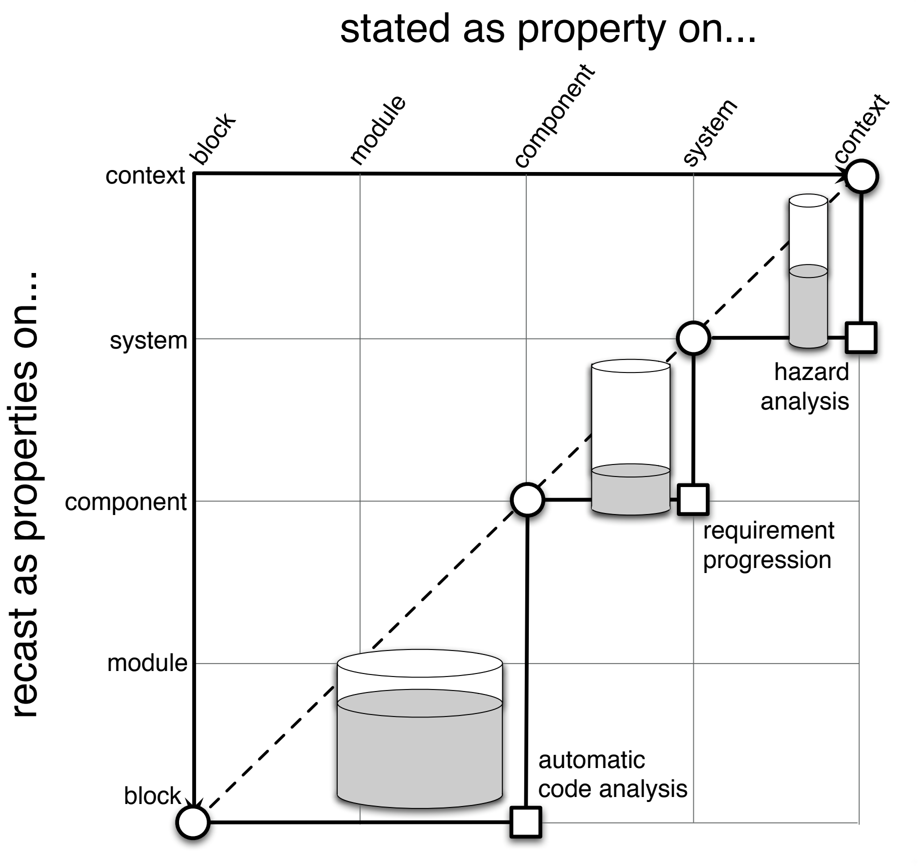

Figure 7-2: Each technique is represented by a glass of water. The height of the glass shows potential confidence gained, the water level shows the current confidence being provided, and the diameter represents return on investment.

7.5.2 Classifying Mistakes

This representation allows us to classify some of the ways that a dependability argument can go wrong.

Figure 7-4 shows cases where one of the glasses has been omitted. In part (a), requirements gathering has been omitted (right glass). A requirement has been carefully decomposed into breadcrumbs (center glass), and the breadcrumbs have been validated (left glass), but the wrong requirement might have been enforced, so overall confidence is low. In part (b), requirements were carefully gathered (right), and the system was carefully architected (center), but the components were not validated (left), leaving overall confidence low. In part (c), the requirements were

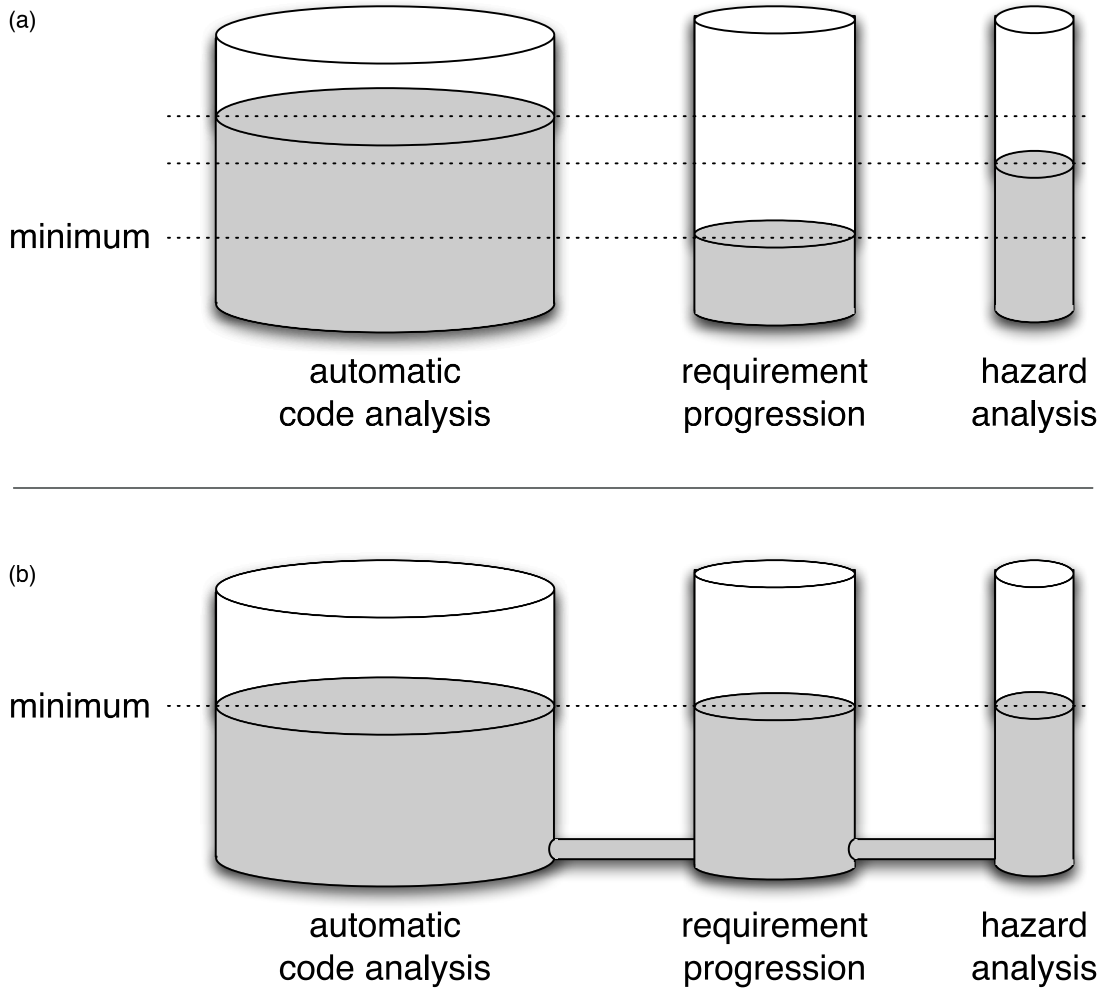

Figure 7-3: Overall confidence in the dependability argument is the minimum confidence of the component techniques.

well understood (right), and the components were checked carefully (left), but no argument was made that the component assumptions actually enforced the system requirement (center), lowering overall confidence.

Figure 7-5 shows cases where techniques were chosen that were not appropriate given the budget. Part (a) shows a case where a heavyweight theorem proving technique was used to analyze code (left), as represented by a very wide (but tall) glass. However, with a low budget, the benefits of theorem proving cannot be realized, and the wide glass just sucks the water out of the other (much thinner) glasses. Overall confidence is lower than necessary. Part (b) shows the opposite problem. A set of lightweight techniques have been used – they are narrow (fill up quickly) but short (can only ever provide so much confidence). With a high budget, all three glasses have been filled up with water to spare (and no where to spend the extra budget). Overall confidence is lower than necessary.

7.5.3 Shaped Glasses

Actual techniques do not always correspond to cylindrical glasses. For example, consider the glasses in Figure 7-6. The left-most glass represents a technique with diminishing returns, such as testing. It takes more water (more test cases) to gain confidence the higher the level already is (the more tests you have already run). Each drop of water (test case) adds less confidence than the last. The second glass represent a technique with a high overhead, such as a custom-build analysis. It takes a lot of work to setup, but then has high return on investment. The last two glasses show the tradeoff (discussed earlier) between lightweight and heavyweight techniques. Heavyweight techniques have the potential to provide high confidence, but take a large investment to achieve that confidence. Lightweight techniques have a much lower maximum confidence, but attain that maximum much more quickly. The shapes of the glasses for particular techniques would be based on empirical data and historical experience.

Building a dependability argument is thus not only a matter of picking techniques with appropriate breadth and depth (as shown on the CDAD), but also about matching the techniques to the budget at hand. The waterglass model has the potential to guide the selection of techniques and also guide the allocation of budget to those techniques.

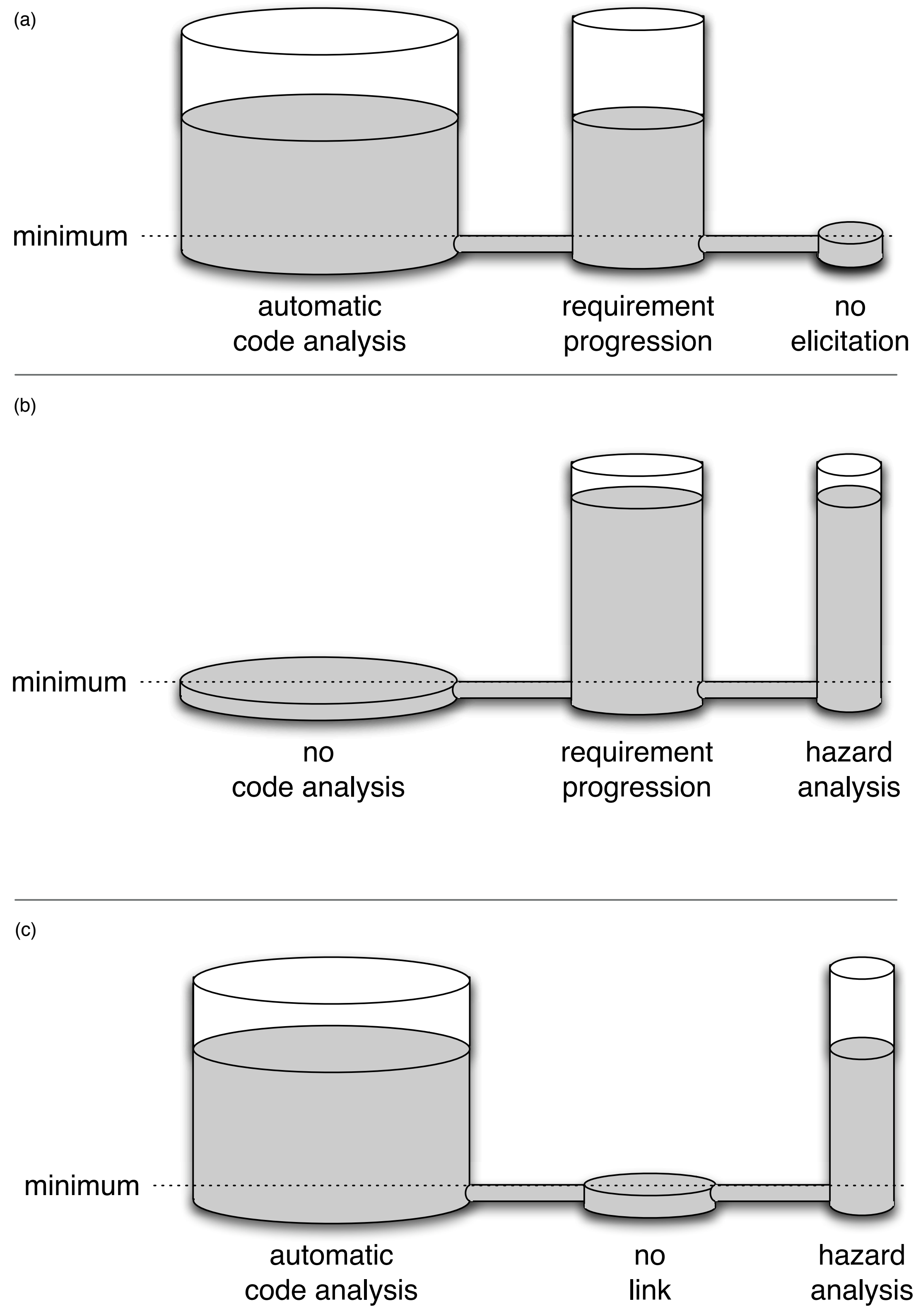

Figure 7-4: Representing errors of omission. Building an incomplete argument greatly harms confidence.

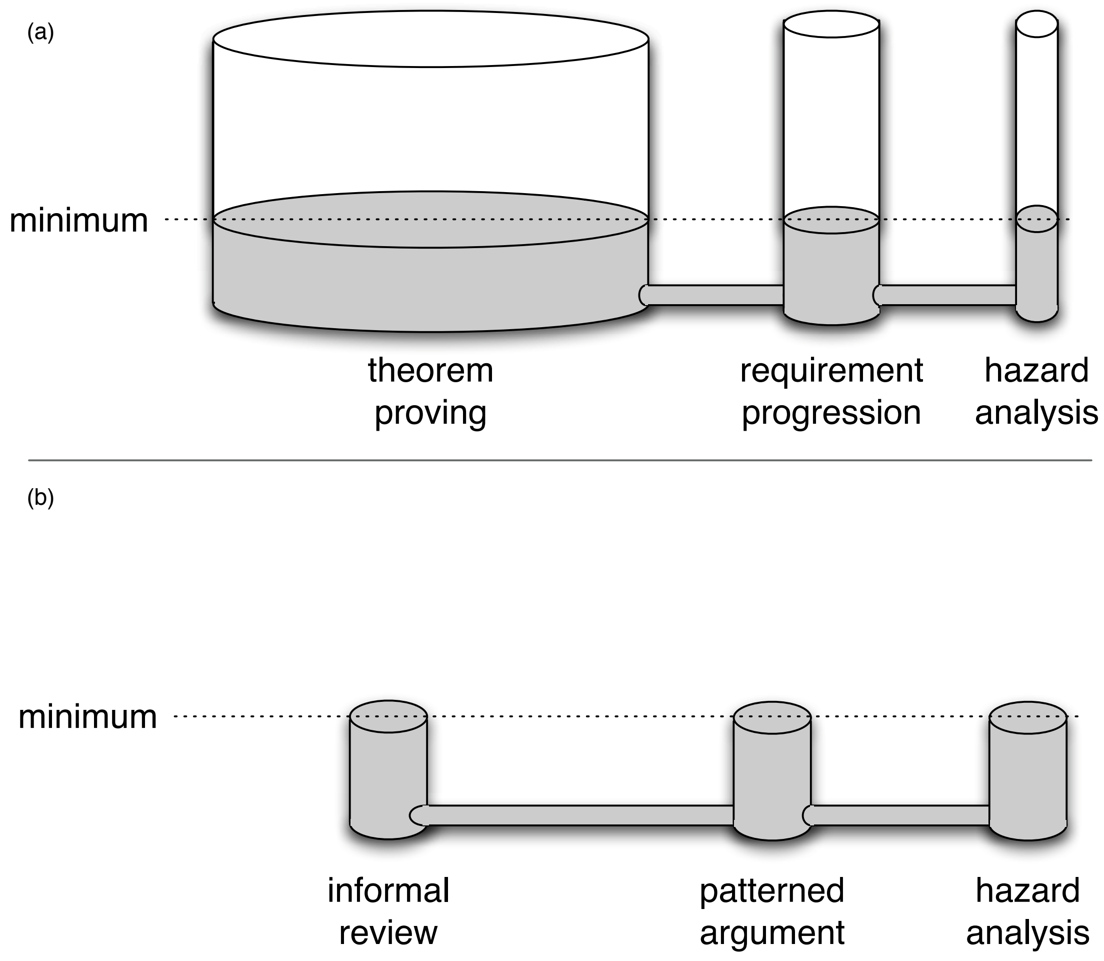

Figure 7-5: Representing errors of technique selection. In the first case, the budget is low so the heavyweight technique sucks all the water out of the other glasses, lowering overall confidence. In the second case, the techniques are lightweight but the budget is high, resulting in wasted budget and lower confidence.

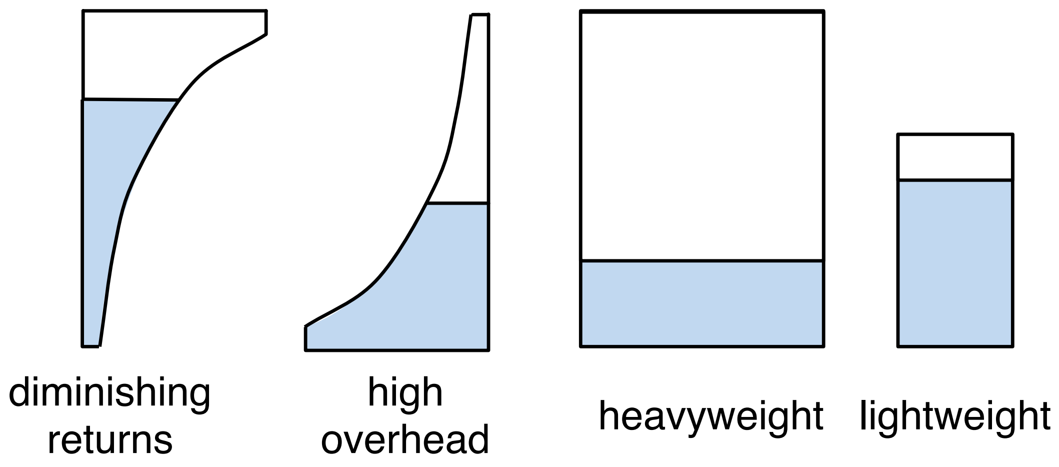

Figure 7-6: Waterglasses have different shapes depending on how their return on investment changes as investment increases.

Chapter 8

Appendix: Automatic Door Model

Chapter 9

Appendix: BPTC Case Study History

Chapter 10

Appendix: Requirement Progression Model

Chapter 11

Appendix: Voting Fidelity Model

Chapter 12

Appendix: Voting Secrecy Model

Bibliography

[1] Giovanna D'Agostino and Marco Hollenberg. Logical questions concerning the mu-calculus: Interpolation, lyndon and los-tarski. *The Journal of Symbolic Logic*, 65(1):310–332, 2000.

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

[3] 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.

[4] Jr. Fred P. Brooks. The mythical man-month. In *Proceedings of the international conference on Reliable software*, page 193, New York, NY, USA, 1975. ACM.

[5] Software Design Group. The Alloy Analyzer. website, 2007. http://alloy.mit.edu.

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

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

[8] Derek Mannering, Jon G. Hall, and Lucia Rapanotti. Relating safety requirements and system design through problem oriented software engineering. Technical Report 2006/11, Department of Computing, The Open University, 2006.

[9] Derek Mannering, Jon G. Hall, and Lucia Rapanotti. A problem-oriented approach to normal design for safety critical systems. In *Proceedings of Fundamental Approaches to Software Engineering (FASE'07). European Joint Conferences on Theory and Practice of Software (ETAPS'07)*, Braga, Portugal, 24 March - 1 April 2007.

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

[11] 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.

[12] P. Y. A. Ryan and S. A. Schneider. Pret a voter with re-encryption mixes. In *In European Symposium on Research in Computer Security, number 4189 in Lecture Notes in Computer Science*, pages 313–326. Springer-Verlag, 2006.

[13] 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.