Activities and Findings:
Safety Mechanisms for Medical Software

Software Design Group
Daniel Jackson
Robert Seater

December 14, 2007

1 Summary

In this project, we have been addressing the pressing question of how to improve the safety of medical software. Our concern is not only how to make the software safe in an absolute sense, but also to provide means by which one can *know* that the software is safe. Our work is therefore focused on the construction of a *safety case* or *dependability case* that can serve as concrete and compelling evidence for a system's safety, allowing healthcare providers to deploy such software with confidence. This work is rather unconventional for software engineering research: rather than focusing on a single technique for analysis or specification, it attempts to integrate a collection of techniques. Its novelty is not primarily in the development of new techniques (although we have developed a new technique for decomposing arguments at the requirements level) but in the framework that combines them into a cohesive end-to-end argument.

The research has been conducted on two parallel tracks: the theoretical development of the ideas, and a realistic application to the proton radiation delivery system of the Burr Proton Therapy Center (BPTC) at Massachusetts General Hospital. As we have developed new ideas and refined our framework, we have tested them by applying them in the context of the case study, and our metric of success is that we demonstrate effectiveness of our ideas in this challenging application.

The research has involved four distinct activities: structuring dependability arguments, decomposing system requirements into domain specific assumptions, performing code analysis to discharge software related domain assumptions, and using Problem Frames to separate and organize top level system concerns. All 4 areas are described in detail below, along with our experiences applying these techniques to the BPTC application.

2 The Structure of Dependability Cases

There is widespread interest in a new approach to the development of software for critical systems that support dependability claims with direct, reproducible evidence, rather than process measures. Rather than taking the traditional approach of relying on adherence to a strict development process, in conjunction with results from system testing, one instead constructs an explicit *dependability case* that gives a complete argument explaining why the desired end to end properties follow from the properties of the components (software or hardware) in combination with assumptions about the behaviour of users, operators, and physical plant. The premise behind this approach is that it can offer a higher degree of confidence, because the evidence is direct rather than indirect, and that is can be scrutinized by a third party, because the evidence is structured and reproducable.

Despite increasing support amongst certification agencies for this approach (especially in Europe), little is known about how to construct a dependability case. This research addresses that problem head on. Arguably, it addresses the problem in a more challenging context than the one that would be expected in practice. For dependability cases to be economical, it will usually be necessary to construct the case hand in hand with the system itself; the structure of the system can then be exploited to simplify the dependability argument, for example by achieving certain separations of concerns. The context of our case study, however, is a large and complex system that has already been built (and, indeed, has been in operation for several years). The dependability case is therefore more complicated and less clearly structured than it might have been had the system design been informed by it. From a research perspective, this has been a boon; it has allowed us to discover areas in which architectural decisions had negative impact on the construction of the case, and it means that our work has more general application, since it can be applied to legacy systems.

2.1 Our Progress

The research problems are twofold: (1) To find a way to represent the gross structure of dependability cases; to decompose the argument into subarguments that can be discharged effectively (either manually, or using tools). (2) In particular, to handle the apparent mismatches between levels of abstraction – for example, where the behavior of low level code must be integrated with arguments about how operators behave.

The results of the research activity so far are: (1) the development of a general model for understanding the elements of dependability arguments and how they fit together, and (2) an instantiation of this model for the proton therapy system.

2.2 Granularities

Figure 1 shows a classic decomposition of a system description, accompanied by examples drawn from the BPTC. An artifact at one granularity comprises finer granularity pieces plus some additional information about the structure of those pieces. For example, a system 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 The coarsest granularity is the *world* outside the system in question, including the stakeholder and the system itself. For BPTC, the world contains domains such as investors, doctors, and FDA regulators, as well as the delivery system itself. The internals of the systems 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.

system The next finer granularity is the *system*, in which we look inside the system domain from the previous granularity and consider its architecture. The communication and control channels between such domains are represented at the system granularity. Refining our view of the BPTC system 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 see a software *component* of the system described as a collection of software modules. The use and call structure between modules is represented at the software component granularity, as are any communication channels between modules (e.g. shared global variables). The BPTC treatment manager component contains modules such as messaging procedures and data structure definitions.

module Each *module* can be decomposed into blocks of straight line code. The non-linear flow between such blocks, such as loops and conditionals, is represented at the module granularity. 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 Within each *block* are actual lines of code, the finest granularity of description of the program. Like any other granularity, a block is a collection of lines of code plus the structure between those lines. In this case, structure between the lines is defined by the semantics of the programming language, and includes things like the order of the lines of code and the data flow between read and write statements.

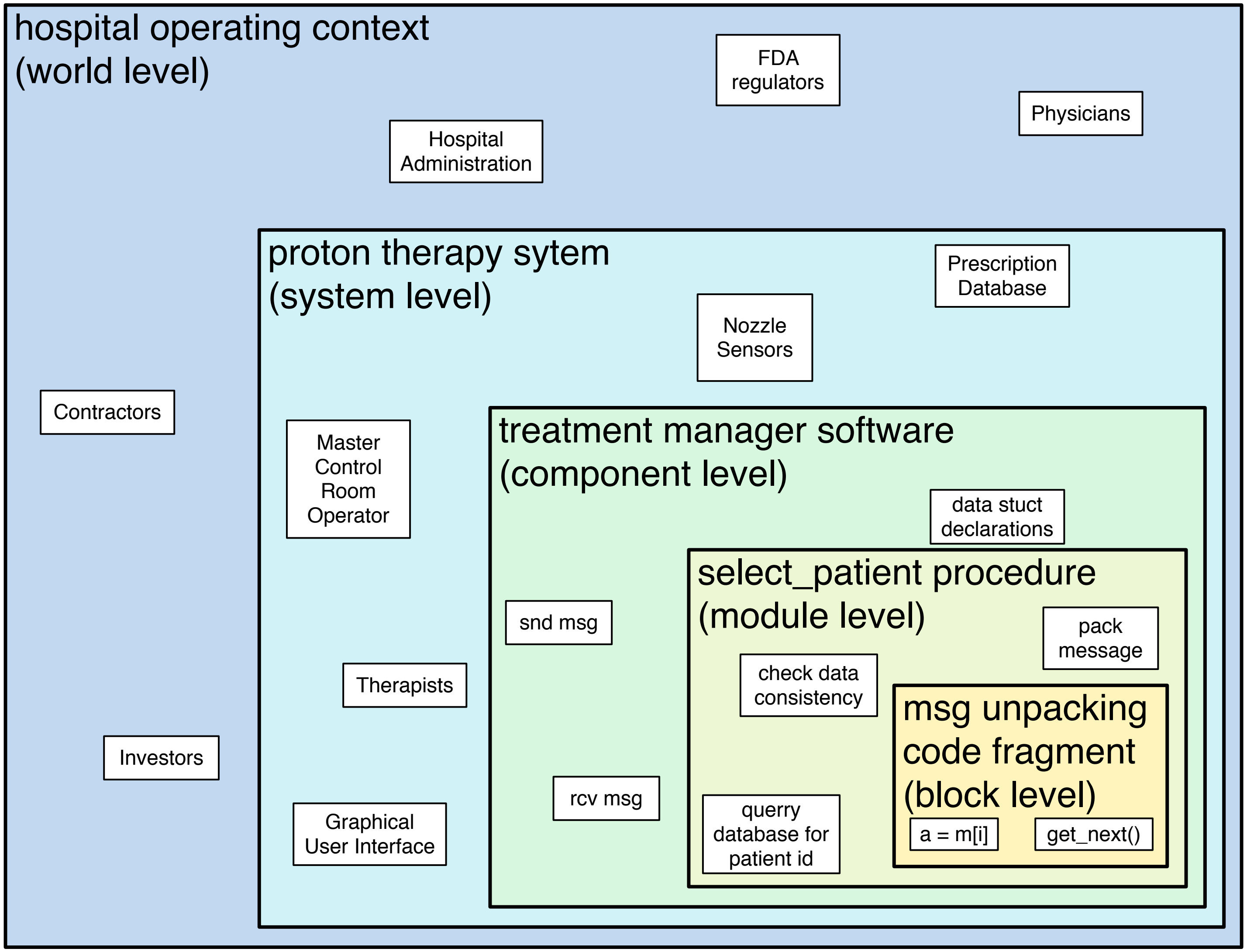

Figure 1: Granularities at which one can view a system: the context of surrounding world, the system architecture under analysis, and components of that system. A software component can be viewed as an entire component, as procedure modules, linear blocks of code, or as individual lines of code. Each granulairty provides a different level of abstraction, hiding some details while revealing broader patterns and connections.

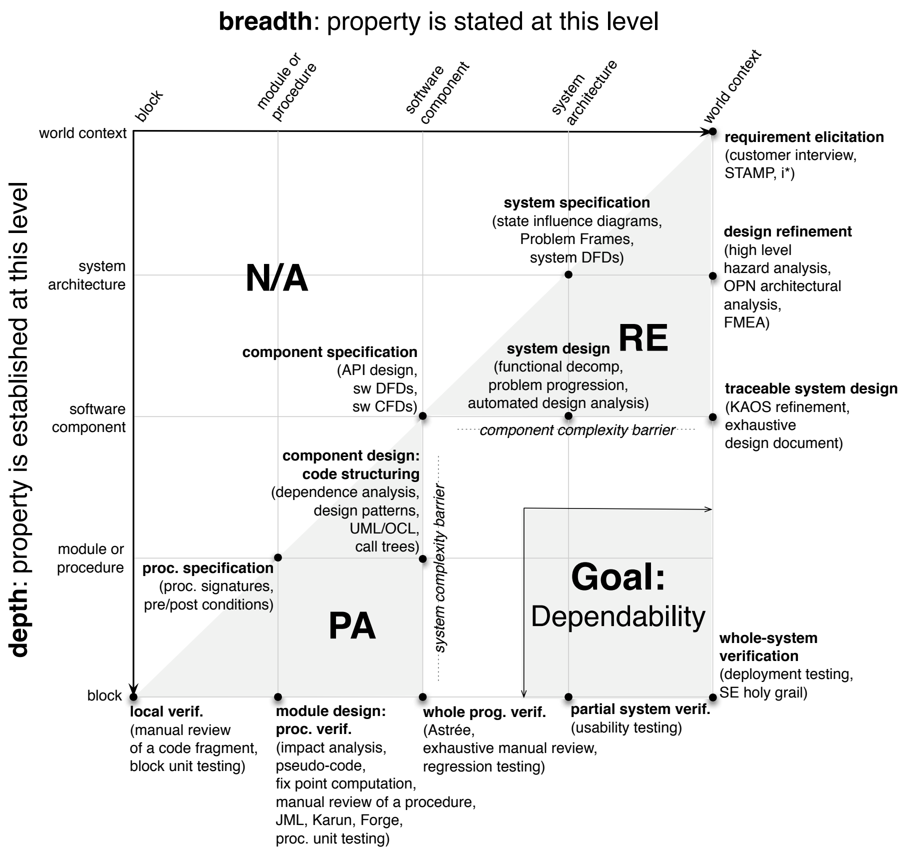

Figure 2: The space of arguments that can be constructed about a software system and the traceability each provides. Each point represents a style of argument in which a property is stated at one level (x-axis) and established at another (y-axis). Each point is labeled with sample techniques often used 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).

2.3 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 believe 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. A collection of arguments can be strung together to build larger, composite arguments that connect more varied granularities.

Figure 2 characterizes a wide array of argument styles that might be used when analyzing or designing a software-intensive system. 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 system 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.

2.4 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, there must be glue to bind them, 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 <*system, module*> using just functional decomposition (<*system, 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.

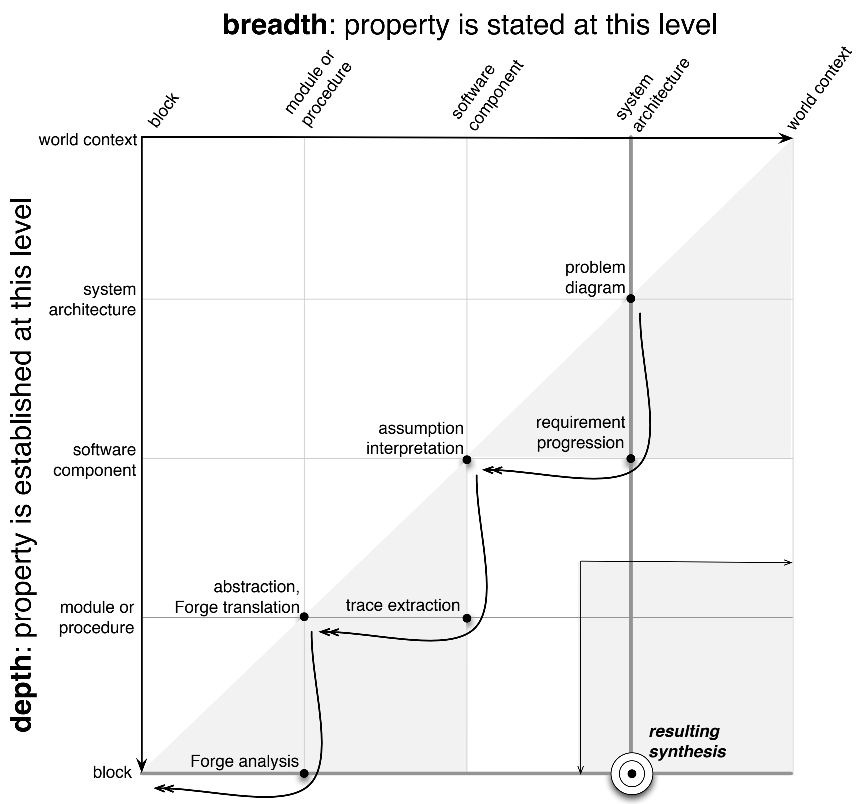

Figure 3: The components of our synthesis that allow us to produce practical certification arguments.

2.5 Our Approach

Figure 3 shows the techniques we combine, using our general framework , to form a composite technique for building dependability arguments for the BTPC.

Problem Diagram : <*system, system*>

> The system requirement is initially expressed with a problem diagram, from the Problem Frames approach. This step recasts the requirement from its original (possibly informal) statement into a form that is be amenable to requirement progression.

Requirement Progression : <*system, component*>

> The system requirement is transformed into a software specification using *requirement progression*. This step recasts the system requirement into a collection of domain assumptions about components.

Interpretation : <*component, component*>

> The domain properties inferred by Requirement Progression are interpreted back into the languages of their domains, making them amenable to domain specific analysis. In the case of software properties, we translate the assumptions into the logic of the Forge Intermediate Language.

Trace Extraction : <*component, module*>

> The problem diagram is used to guide a human in determining the relevant portions of the code base relevant to a particular assumption.

Translation & **Abstraction** <*module, module*>

> 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.

Forge Analysis : <*module, block*>

> 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 with Forge.

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.

3 Requirement Progression

An important feature of system-level dependability arguments, which distinguishes them from the more traditional kinds of arguments made in computer science (in reasoning about code for example), is the important role of assumptions. These assumptions represent properties of (typically non-software) components of the system that are assumed by other components, and which, if they fail to hold, will undermine the end-to-end properties of the systems as a whole. Unlike assumptions made in code reasoning, these assumptions may be articulated and elaborated on the fly as the system is developed or the argument is constructed. For a component such as a human organization or operator, these assumptions are formalizations of informal facts, and are drawn from a potentially unbounded pool. There is therefore a much more fluid relationship between the articulation of assumptions and their use in an argument.

In order to make it easier to identify, articulate and elaborate the necessary assumptions as a dependability argument is constructed, we have developed a reasoning strategy that we call "requirement progression" that allows end-to-end system arguments to be decomposed into component assumptions in an incremental and systematic way.

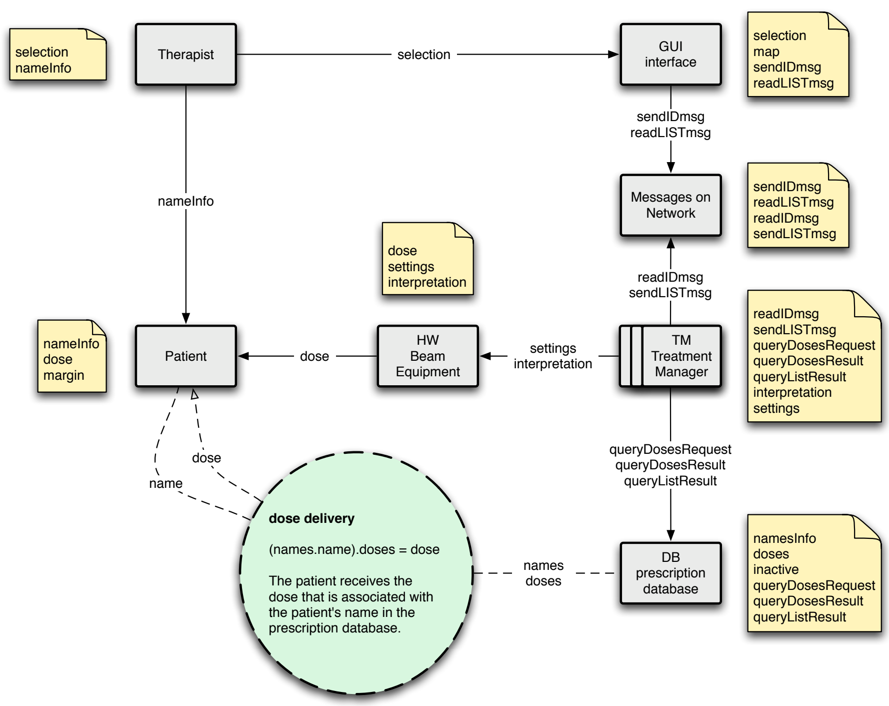

Figure 4: Problem Frames problem diagram for the patient identity subproblem.

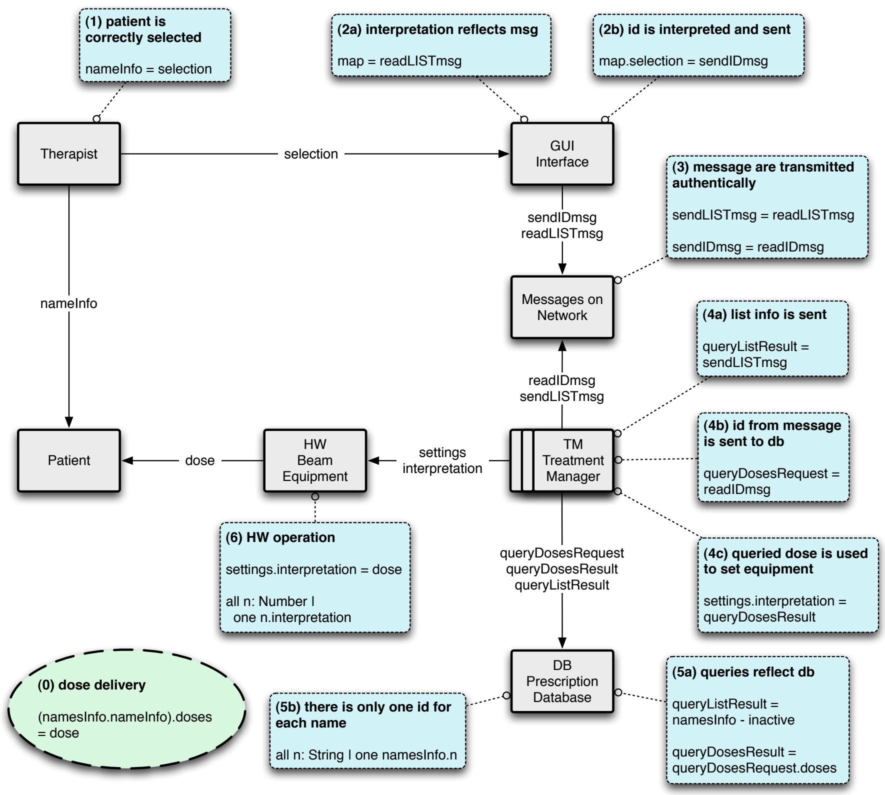

Figure 5: Argument diagram for the patient identity subproblem.

```
sig String {}
sig Number { interpretation: set Number }
sig ID {
  map, namesInfo, inactive, queryListResult, sendLISTmsg, readLISTmsg: set String,
  doses: set Number
}{inactive in namesInfo}

one sig nameInfo, selection in String {}
one sig settings, dose, queryDosesResult in Number {}
one sig sendIDmsg, readIDmsg, queryDosesRequest in ID {}

pred Requirement [] { (namesInfo.nameInfo).doses = dose }
pred allBreadcrumbs [] {
  Therapist[] and GUI[] and Network[] and TM[] and DB[] and HW[] }

pred Therapist [] {
  nameInfo = selection }
pred GUI[] {
  map = readLISTmsg
  map.selection = sendIDmsg }
pred Network[] {
  sendLISTmsg = readLISTmsg
  sendIDmsg = readIDmsg }
pred TM[] {
  queryDosesRequest = readIDmsg
  queryListResult = sendLISTmsg
  settings.interpretation = queryDosesResult }
pred DB[] {
  queryListResult = namesInfo - inactive
  queryDosesResult = queryDosesRequest.doses
  all n: String | one namesInfo.n }
pred HW[] {
  settings.interpretation = dose
  all n: Number | one n.interpretation }

assert end2end {allBreadcrumbs[] => Requirement[]}
check end2end for 6
```

Figure 6: An Alloy model verifying that the argument diagram is consistent; if the breadcrumbs hold, then the requirement will hold.

3.1 Dose Delivery Case Study

For example, Figure 4 shows a Problem Frame describing a the *dose delivery* concern in the BPTC system – the requirement that each patient be delivered the prescription stored for that patient in the prescription database. Overdoses may damage sensitive surrounding tissue, and underdoses may fail to eliminate the cancerous tissue – both are high severity hazards. The requirement – formalized in the Alloy language and represented in the dotted ellipse – stipulates that the radiation dose delivered to the patient (`dose`) matches the dose stored for that patient in the prescription database (`(names.name).doses`).

Using Requirement Progression, we derive the *argument diagram* shown in Figure 5. By revealing the structure of the overall argument, the argument diagram decomposes the end-to-end system property (dose delivery) into a collection of finer grained, domain-specific assumptions Each domain assumption is local to a particular domain – it references only phenomena from that domain and is thus amenable to analysis and verification by tools and experts specialized to that domain. For example, assumptions about the behavior of the Therapist might be verified with a human-factors analysis of the operators, and enforced with an ongoing training program informed by that analysis. In contrast, the assumptions about the Treatment Manager domain can be verified using program analysis, such as the technique we describe in the next section, and enforced with runtime checks and hardware interlocks informed by that analysis.

Figure 6 shows an Alloy model generated during requirement progression; automatic analysis of this model confirms that the domain-specific assumptions shown in the argument diagram are sufficient to enforce the end-to-end requirement given in the problem diagram. If the domain experts and local analyses are trusted, then the system as a whole can be treated as dependable.

4 Code Analysis

The dependability argument of a dependability case must establish the high-level end-to-end properties (such as "the right dose is delivered to the right patient") but it must do so by accounting for the lowest level properties of the code. Our approach therefore integrates Forge, a new code analysis developed within our research group that uses the Alloy language as a specification language and SAT solving as an analysis technique for finding counterexample traces. The integration of the code analysis into the overall dependability argument is not easy, because the codebase is large and involves many interacting functions, only a few of which are relevant to the particular safety property being established. Our approach involves identifying the relevant code fragments by a preliminary reasoning step; identifying the required properties of the code in context (which become a partial specification of the relevant code fragments); and then performing a code analysis that establishes those properties for the fragments.

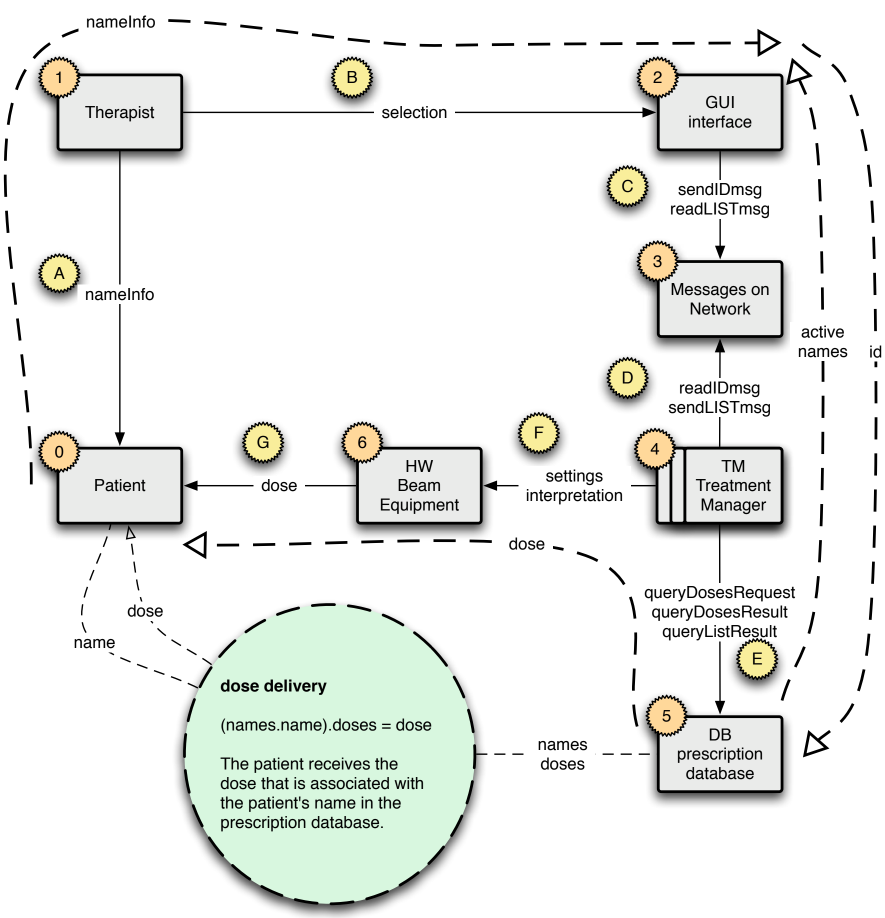

Figure 7: Flow diagram for the patient identity subproblem.

4.1 Dose Delivery Case Study

Our code analysis approach is guided by a *flow diagram*, a version of the problem diagram annotated with information about the flow of information through the system.

Figure 7 shows the flow diagram for the dose delivery concern, and indicates the subset of the code base relevant to this particular system requirement. The argument diagram, derived by requirement progression, provides us with the local assumptions that must be enforced along this flow. Together, these two pieces tell the analyst which pieces of software must obey which specifications in order for the desired requirement to hold. Each domain assumption is interpreted based on its local context in the flow diagram, mapped into an analyzable form, and discharged using analysis techniques appropriate for that domain.

In the case of software domains, such as the Treatment Manager, we discharge the assumptions using the Forge program analysis framework, developed in our research group by Greg Dennis. The derived property is translated into the Forge logic, and automatic analysis discovers counterexamples in which the assumption fails. After an iterative process of adjusting the code and strengthening precondition assumptions, the property passes, providing a greater degree of confidence that the Treatment Manager will properly play its role in the dose delivery problem.

5 Separating Concerns

A significant change in how computer scientists think about software verification is that the notion of a single, monolithic (and complete) specification against which correctness can be determined is no longer regarded as desirable or achievable. Instead, there are a variety of properties one might want to analyze software against, of varying degrees of criticality.

Michael Jackson has argued, further, that the very structure of the software development problem should be viewed as a composition of subproblems. That is, the software developer's challenge is to solve a collection of problems that not only have different degrees of criticality, but which are to some extent independent of one another. By focusing on the individual subproblems, important difficulties become clearer, and can be tackled more effectively. The solutions to the individual subproblems must still be composed in the development of the final system, but delaying the composition can reduce the cost of the entangling of the subproblems, by allowing each to be treated in a simpler context before the composition is handled.

In a collaboration with Michael Jackson, we explored the impact that this approach might have on the proton therapy system by analyzing a small but significant problem that had arisen in the deployed system, in which the gantry appeared to experience creep, so that its position would gradually change without explicit instructions having been issued for that to happen. We constructed a formal model to analyze this issue, using problem frames as a guiding structure and the Alloy language and analyzer for the model proper, and demonstrated that the problem can be easily accounted for when this separation of concerns, brought about by addressing a subproblem in isolation, is available. We were able to demonstrate in a very simple and clean model that the solution that had been adopted by the development team did in fact have the desired effect, without having to encounter the complexities that had prevented the development team itself from constructing such an argument.

6 Publications and Presentations

Publications

Robert Seater, Daniel Jackson, and Rohit Gheyi. *Requirement Progression in Problem Frames: Deriving Specifications from Requirements.* Requirements Engineering Journal (REJ'07). 2007.

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.

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.

Daniel Jackson and Michael Jackson. *Separating Concerns in Requirements Analysis: An Example.* M. Butler, C. Jones, A. Romanovsky, E. Troubitsyna (editors). Chapter of *Rigorous Development of Complex Fault Tolerant Systems.* Springer-Verlag. 2006.

Presentations

Robert Seater. *Requirement Progression in Problem Frames Applied to a Proton Therapy System.* Presented at the 14th IEEE International Requirements Engineering Conference (RE'06). Minneapolis, MN. September 2006.

Robert Seater. *Problem Frame Transformations: Deriving Specifications from Requirements.* Presented at 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). Shanghai, China. May 2006. ACM Press.