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 the *Pret a Voter* 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

Chapter 1

Motivation

1.1 Introduction: System Failure

It is widely understood that system failures often result not from component failures but from inadequate component specifications – the components behaved according to their specifications but the system failed as a whole due to unforeseen component interaction [7, 16, 29, 33, 35, 49, 50, 51]. Even when a system failure can be tracked back to a bad decision made by a particular component, usually the component made that decision in accordance with its specification. That specification, in conjunction with other component specifications, was not sufficiently strong to enforce correct system behavior. It is the system as a whole, not any one component, that produced the failure.

For example, a chemical engineer might provide a specification to a software engineer writing code to control an automatic valve, but omit assumptions that all chemical engineers take for granted (e.g. that input valves should always be closed before output valves). The software engineer then provides a piece of software in accordance with the written spec, but which violates the implicit intention of the chemical engineer [49].

Specification inconsistencies stem from two sources: a shortcoming on the part of the system engineer to decompose the system requirement into component specifications, and a failure to unambiguously communicate the specification to the engineers and specialists for each component. Our work strives to address both concerns, especially as they arise in software-intensive systems.

1.1.1 Software Intensive Systems

Software components differ from electro-mechanical components in ways that intensify the burden put on system-level analysis and requirements engineering. As software is increasingly deployed in contexts in which it controls multiple, complex physical devices, this issue is likely to grow in importance. Systems incorporating software components are likely to become increasingly resistant to traditional methods of analysis, such as testing, manual inspection, redundancy, and functional decomposition.

Software components do not break. Current practices, such as FMEA [6, 59], focus solely on component failure. While such a focus is appropriate for electro-mechanical systems, where parts wear out and must be replaced, it does not address the concerns of software components. Software does not wear out or break. Any error in a software component is present from installation. [1]

Software engineers are more vulnerable to omitted assumptions. The more different types of engineers involved in a system, the greater the chance of miscommunication. The inclusion of software engineers, whose training is often disjoint from other engineering professions, exacerbates the issue and increases the risk of miscommunication and omitted assumptions. Software engineering education programs that share common coursework with other engineering disciplines are only 5-10 years old [63, 62].

Software is given more complex tasks. Software components are often given

[1] One could consider memory leaks or caches filling up as examples of software wearout, although those concerns are more analogous to a physical motor overheating. The part has not worn out and broken, but rather it requires a reset or idle time. If it overheats more than is acceptable for the current application, then one faces a design problem not a component failure. A self-modifying computer system which is never reset or rebooted (such as a self-configuring network) could indeed have failures in the traditional sense – over the course of auto-configuring and maintaining caches, the system might evolve itself into a bad state, and effectively have broken – requiring a full replacement. For most software systems, this scenario is a stretch of the imagination, and the primary concern lies with logic errors in the design, not decay over the system's lifespan.

the most complex and subtle specifications on the grounds that software is flexible and cheap to update. This means that the software components are more likely to be sensitive to subtlies of the system architecture, and thus more vulnerable to incomplete analysis and documentation.

Software is complex and non-continuous. A computer program can behave completely different on one set of inputs than on a similar set of inputs, reducing the confidence gained from past performance and testing. Furthermore, because of the size and complexity of most software, it thwarts manual verification at the source level. The result is a components which is hard to verify statically or empirically.

1.1.2 Dependability, Auditability, and Traceability

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

A *dependability argument* [19] is one that justifies the use of a particular component for a particular role in a particular system. It is not an argument about absolute correctness, and it is not about preventing component failures. Rather, it is about understanding the interaction of components, and encurring that the individual component specifications are adequate to prevent system-level failure.

Building a *correct* argument is not enough; the argument must also be *auditable*. It might be reviewed by a certification authority (such as the FDA [61], FAA [2], or NRC [3]), a system engineer deciding if the system is suitable for a slightly different operating context, an engineer wanting to make a change to the system, or even an engineer new to the project. As the system evolves, the dependability argument must be *maintainable*, as reconstructing a thorough dependability argument after each change to the system is impractical.

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

An argument that provides both forms of traceability is termed *end-to-end*; it connects the high level system concerns down to the low level component properties, based on an explicit description of the structure of the intervening layers.

The research community has approached dependability along four, largely independent routes. Individually, these styles of approach provide insufficient breadth, depth, confidence and/or are not economical on complex systems. Our approach brings together techniques developed in these different academic fields to build a composite argument with an appropriate tradeoff of those factors.

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

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

Testing can provide the breadth of requirements engineering and the depth of program analysis, but fundamentally cannot provide the *coverage* needed to build a dependability argument. Testing a software-intensive system assures that the system operates correctly in the tested scenarios, but provides no guarantees about scenarios not specifically tested – not even if those scenarios are similar to those that were tested.

Formal Methods (FM) have the potential to provide ample coverage, but are too costly to economically apply to large legacy system. FM have only scaled to large systems when the systems have been built from scratch in a controlled manner by specially trained developers [25, 56]. Applied to an existing complex system, they do not scale adequately to build end-to-end arguments. As a result, the tend to be used in a manner that provides depth but not breadth, if they are used at all.

Unfortunately, while requirements engineering and program analysis each provide sufficient confidence at acceptable cost, the specifications generated by requirements engineering techniques often do not match up with the types of properties that program analysis techniques can validate. The two halves are typically connected only informally by a intuition that certain properties about the code (such as the lack of buffer overruns) will correspond to system properties (such as the system being protected from security attacks). There is not a systematic, auditable argument for why the properties checkable by program analysis are sufficient to ensure the properties called for by requirements engineering.

Our approach is to combined techniques from requirements engineering and program analysis to harness the best of both worlds. To connect them, we draw heavily upon techniques from both those fields and from formal methods.

1.2 Contributions

This research is organized around four interworven contributions.

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

Chain of Techniques - We have identified a set of techniques for building pieces of a dependability argument. We fit these techniques together using CDAD, producing a composite technique suitable for building dependability arguments for a particular class of software-intensive system properties.

Requirement Progression - Where necessary, we have developed techniques to connect existing techniques and thereby completing the end-to-end argument. Most prominently, we developed *Requirement Progression*, a technique used to connect problem diagrams with code specifications. Requirement progression became a central part of all our dependability arguments.

Case Studies - We have applied that technique to two systems: (a) The Burr Proton Therapy Center (BPTC), a medical system currently being used to treat cancer patients as Massachusetts General Hospital (MGH). The BPTC analysis shows how we use CDADs to integrate requirement preogression with automatic code analysis of software components. (b) The *Pret a Voter* cryptographic voting system developed at the University of Newcastle. The voting analysis shows how using requirement progression to build a fidelity argument make the construction of secrecy and auditability arguments for the same system easier, more thorough, and more transparent to review.

For each of the two case studies, we contribute the following:

- A safety case for the dependability of the system with respect to mission-critical requirements. This involves both a description of the assumptions and conditions under which the software is suitably dependable, and a verifiable argument for why those conditions and assumptions are sufficient.
- A list of undocumented dependencies, assumptions, and vulnerabilities of the system, and an analysis of their effect on safety. These assumptions will hopefully be added to the official documentation for the system, making the it easier and safer to maintain.
- A description of our experience building the dependability argument, including analysis of which parts worked well, which need improvement, and at what stage during the process different problems were discovered.

1.2.1 Hypothesis

In this thesis, we will motivate and substantiate our belief that requirement progression and CDADs are effective and cost-effective techniques for guiding and structuring end-to-end dependability arguments. CDADs provide a means for showing the overall structure of a dependability argument, and requirement progression provides a key link in that argument, providing confidence that the component specifications do indeed enforce the system requirements.

Chapter 2

Synthesis Technique: CDAD

Given a collection of techniques, each of which provides a narrow piece of a dependability argugment, how does one connect them together to build a single end-to-end argument? To answer this question, we first show how to classify component techniques according to the breadth of the claim they support and the depth of the evidence they provide. We will then show now that classification guides composition, and demonstrate one such composition that we have found to be effective.

2.1 Dependability Arguments

In this section, we introduce the Composite Dependability Argument Diagram (CDAD), a structured classification of argument styles used to analyze and document system dependability. This classification shows what approaches are appropriate for addressing different types of concerns about the system at different levels of granularity. More importantly, it shows how different approaches can be connected together to build a unified dependability argument for an end-to-end system concern.

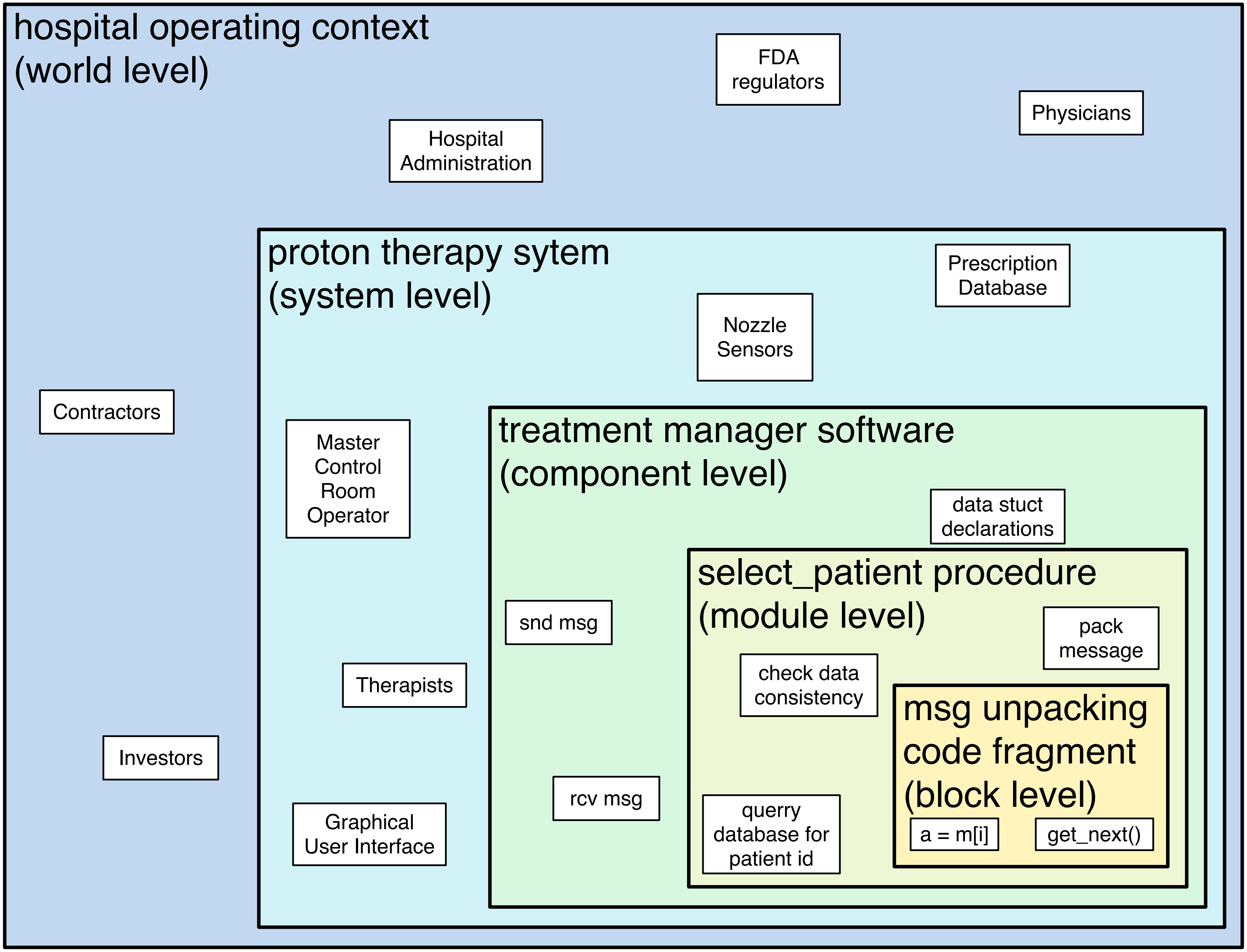

Figure 2-1: Granularities at which one can view a system: the context of the surrounding world, the system under analysis, and components of that system. A software component can be viewed as an entire component, as procedure modules, linear blocks of code, or as individual lines of code. Each granularity provides a different level of abstraction, hiding some details while revealing broader patterns and connections.

2.1.1 Granularities

The first part of understanding Composite Dependability Argument Diagrams (CDADs) is to understand the axes. Both axes use the same scale – a hierarchy of granularities at which one can view the system.

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

Figure 2-1 shows a classic decomposition of a system description, accompanied by examples drawn from the BPTC.

Context - The coarsest granularity regards the system architecture as a black box interacting with the surrounding world and stakeholders.

For the BPTC, the world contains domains such as investors, doctors, and FDA regulators, as well as the delivery system itself. The internals of the architecture are hidden from view, but their interactions, communications, and goals are shown. Legal and financial concerns are expressed at this granularity, although our work focuses solely on safety concerns.

System - The next finer granularity regards the components of the system architecture as black boxes, and examines how those components communicate and interact.

Refining our view of the BPTC architecture reveals components such as operators, prescriptions, and the treatment manager. It is at this granularity that we state safety concerns, such as accurate dose delivery, consistent logging, and safe shutdown.

Component - At the next granularity, we regard modules within a components as black boxes, and examine how those modules interact. In the case of a software components, the modules might correspond to procedures that are connected by function calls and shared data.

The BPTC treatment manager component contains modules such as messaging procedures and data structure definitions.

Module - At an even finer granularity, blocks within a module are treated as black boxes, but the structure within the module that links together those blocks is exposed. For a software module, the blocks might be linear fragments of code, linked together by conditionals and other non-linear control flow.

For example, the "set equipment" procedure includes a block that initializes some variables, the code inside the loop that constructs an array of data, and a block that constructs a message from the array and sends it to the hardware device driver.

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

2.1.2 The Space of Arguments

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

An argument's *breadth* is the granularity of the stated goal, while its *depth* is the granularity into which it recasts that goal. For example, a system refinement argument might state a claim about the system architecture as a whole and recast

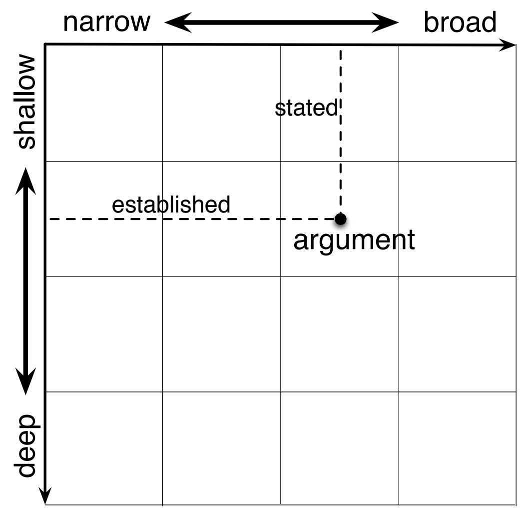

Figure 2-2: An argument states a property at a certain breadth and establishes it by examining the system at a certain depth.

that claim into a set of assumptions about the components of the system. As we will see later, a collection of arguments can be strung together to build a composite argument with greater breadth (further to the right) and greater depth (further down) than any one of the components.

x-axis: 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.

y-axis: 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.

statements: An $<x,\ x>$ point on the main diagonal is a statement about the granularity x. A <*system, system*> point is a statement about the system – a requirement. A <*component, component*> point is a statement about a component of the system – a domain assumption.

arguments: An $<x,\ y>$ point below the main diagonal is an *argument* that the statement at $<y,\ y>$ holds as long as a certain set of statements at $<x,\ x>$ hold. For example, <*system, component*> is an argument that a system requirement is enforced by a set of component assumptions.

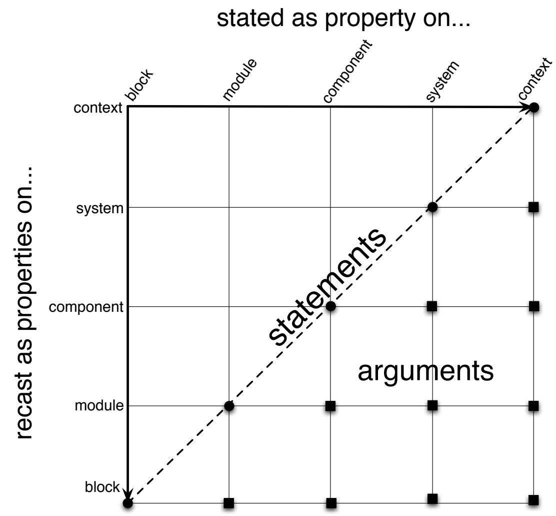

Figure 2-3: Points on the main diagonal represent statements about the system at a particular granularity. Points below the main diagonal represent arguments that establish one statement based on a set of statements (assumptions) at a lower granularity.

invalid points: The upper-lefthand triangle of the diagram is empty – breadth is always greater than or equal to depth. A property cannot be established at a higher granularity than it is stated. For example, one cannot show that a component obeys its specification by noting that the system has a certain requirement, whereas one can do the reverse – argue that a system has a requirement because a component obeys its specification.

2.1.3 Sample Arguments

Consider two particular points in this diagram: design refinement, at <*world, system*>, and whole program verification, at <*component, block*>.

Arguments at <*world, system*> are *design refinement* arguments; they recast claims/goals stated about the world surrounding the system into claims/goals stated about the system under analysis (treated as a black box) and claims/goals stated about other systems interacting with the system under analysis.

For example, a high level hazard analysis for a chemical tank would be a *design refinement* argument. It recasts safety constraints (that the tank does not harm surrounding equipment) into constraints about the chemical tank (that it does

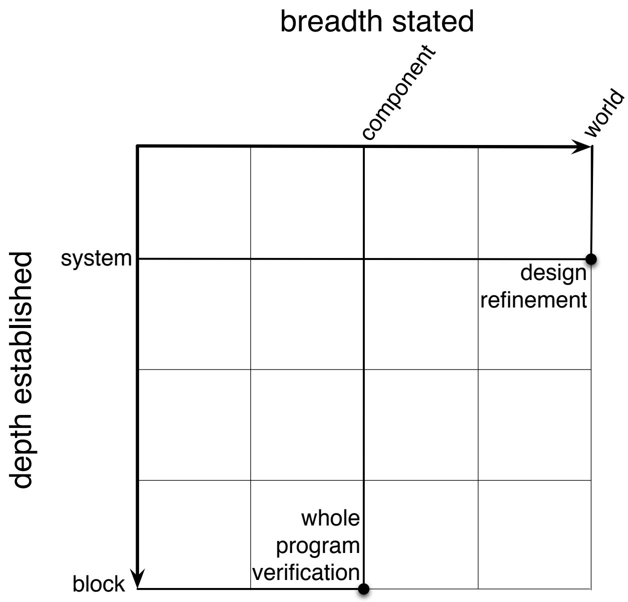

Figure 2-4: The breadth and depth of two sample argument styles.

not vent more than X grams of corrosive gas per day) and constraints about the surrounding equipment (that they will not be damaged by exposure to X grams of corrosive gas per day). A world goal (damage to surrounding assets) has been decomposed into system goals (how many grams of gas can be vented per day).

In contrast, at <*component, block*> we have *whole program verification* arguments, which recast goals about an entire software component and establish them in terms of the semantic of individual block of code.

For example, a thorough manual review of the software that controls a chemical valve would be *whole program verification*. It would take a property stated about the system as a whole, that it correctly send signals to the value according to a prescribed pattern, and recasts it as properties about the semantics of the language used (e.g. that the `send_open_signal` does indeed send an open signal to the value, and that a `wait_1_second` really does pause for 1 second). Of course, manual exhaustive review might be too costly or too error prone to be suitable for a particular analysis, but it certainly fits the mould of a *whole program verification* argument.

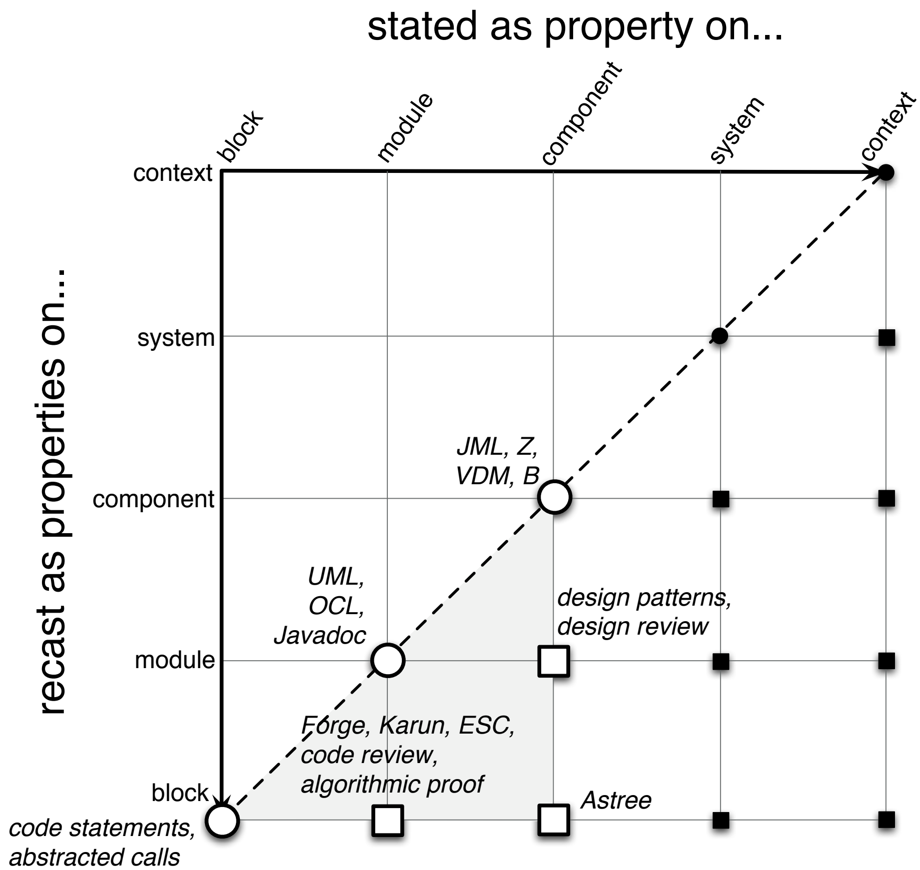

Figure 2-5: Program Analysis (PA) techniques reside in the lower lefthand region, providing depth but not much breadth.

2.1.4 Existing Techniques

CDADs do not directly represent the cost (both human and computational) of building the different kinds of arguments, although we discuss extensions of the CDAD notation to express such information in Chapter 7. In general, moving deeper (down) and broader (right) raises cost and/or lowers confidence.

The fields of program analysis (PA), requirements engineering (RE), and testing are represented by clusters of argument types in the CDAD.

PA: Program analysis techniques (PA) occupy the lower-left-hand region, as shown in Figure 2.1.3 – the properties are stated and established at a low granularity. PA techniques rarely address properties stated at or above the system granularity, as such properties are too broadly stated to be amenable to automatic analysis. We

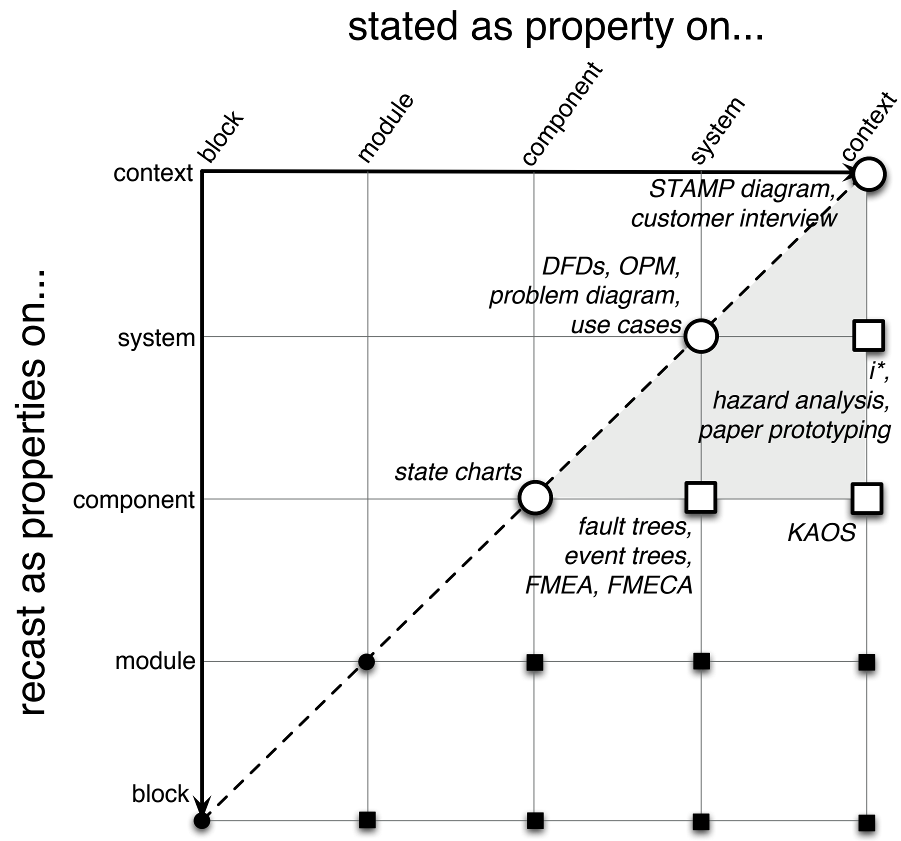

Figure 2-6: Requirements Engineering (RE) techniques reside in the top right corner, with great breadth but limited depth.

indicate this obstacle with the vertical *system complexity barrier* in Figure 2.1.3.

RE: Requirements engineering techniques (RE) occupy the upper-right-hand region, as shown in Figure refcdad-RE – the properties are stated and established at a high granularity. RE techniques rarely establish properties below the system granularity, as doing so produces descriptions that are too large and complex to be reasoned about. We indicate this obstacle with the horizontal *component complexity barrier* in Figure 2.1.3.

Testing: Testing techniques occupy the bottom row; they provide deep analysis at various breadths, as shown in Figure 2.1.3. Testing can provide the breadth of RE and the depth of PA, but fundamentally cannot provide the *confidence* needed to build a dependability argument. Testing assures that the system operates correctly in the tested scenarios, but provides no guarantees about scenarios not specifically tested.

The ultimate of software engineering is to develop a high-confidence economical

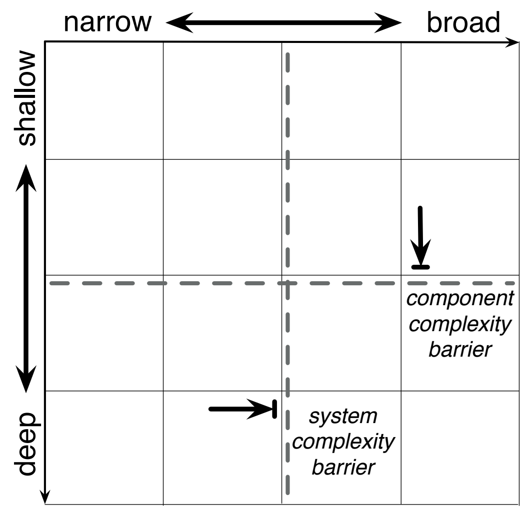

Figure 2-7: Requirements engineering techniques tend to stay above the *component complexity barrier*, to avoid introducing too much detail about the operation of the underlying system. Program analysis techniques tend to stay left of the *system complexity barrier*, to avoid introducing the details of too many interacting components of the system.

technique at the lower-right-hand-most corner – one that states a property at the highest (world) granularity and establishes it at the lowest (block) granularity. Unfortunately, getting anywhere near that goal requires crossing both complexity barriers.

2.1.5 Dependability Arguments

Most tasks do not require the holy grail and can make do with more modest approaches. For example, verifying that libraries obey their contracts requires only a <*module, block*> style argument, and can be established using program analysis techniques such as Greg Dennis's Forge [23] or Patrick Lam's HOB [46]. Similarly, determining if a given software specification is sufficient to enforce a given system requirement requires only <*system, component*> or better, and can thus be satisfied by requirement progression [81]. However, the important class of *end-to-end dependability arguments* lies outside the ranges of conventional PA and RE techniques.

Dependability arguments for software intensive systems should state properties

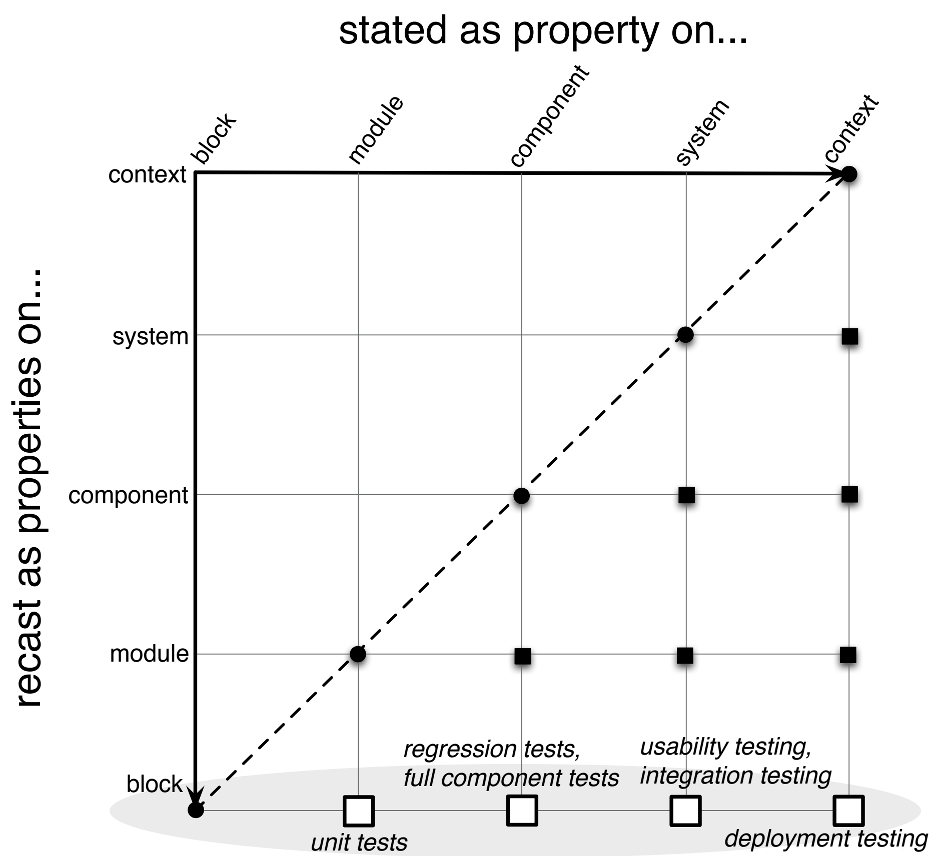

Figure 2-8: Testing techniques reside on the bottom row, establishing properties at the depest level and a variety of breadths. However, testing alone does not provide the confidence needed for a dependability argument.

at the system granularity (or higher) and establish those properties at the module granularity (or lower). For example, part of the BPTC dependability argument (described in Chapter 4) is to establish that patients do not receive more radiation than their prescriptions indicate. Such an argument should be grounded in the code, so that if the requirement is changed (e.g. to say that the patient cannot receive less than their prescription either) or if the system is changed (e.g. to include an additional firing mode), one can determine which parts of the code need updating, if any.

Figure 2.1.4 shows the space of solutions that are appropriate for building this kind of dependability argument. While we do not necessarily need to achieve the

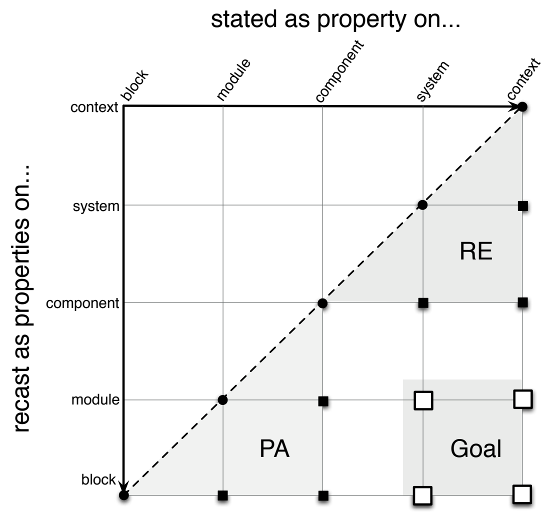

Figure 2-9: Neither requirements engineering (top right) nor program analysis (bottom left) techniques have enough breadth and depth to reach the lower right area, where dependability arguments reside. However, composition of RE and PA techniques can get us there.

bottom-right corner to build dependability arguments, we do need something more than we have – neither PA nor RE techniques have sufficient breadth and depth to land in the target region. We can, however, compose existing PA and RE techniques, together with some additional work, to create a composite technique that falls within the target region, as shown in Figure 2.1.4. The challenge of building composite techniques is to keep the cost from rising too high without letting the confidence drop too low.

2.1.6 Composition

Building composite arguments take more than picking two techniques that, between them, have sufficient breadth and depth.

(a) *The techniques must match up.*

We can't reach the bottom right corner (<*world, block*>) with just hazard analysis (<*context, system*>) and a code review (<*module, block*>). While those arguments have sufficient breadth and depth between them, they do not

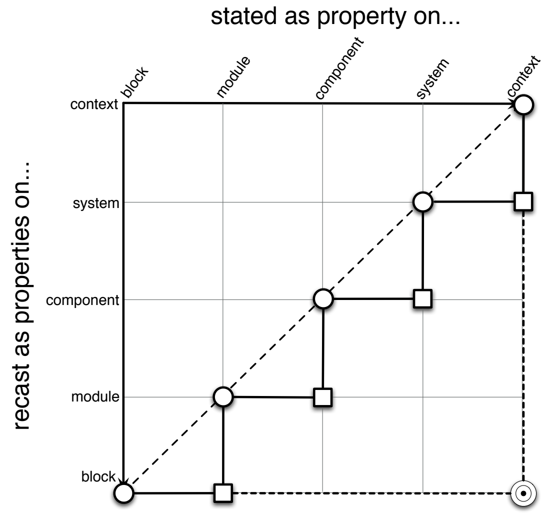

Figure 2-10: A composite argument built out of smaller arguments can reach the bottom right position (max breadth, max depth), even when no individual technique can reach that point.

connect together. An additional argument is needed to connect the system statements used to establish the hazard analysis with the module statements established by the code review.

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

We can't reach <*system, block*> with just KAOS [21, 22, 18, 8] (<*system, component*>) and Astree [9, 10] (<*component, block*>). The claims generated by KAOS are at the right level to be established by an Astree analysis, but they may not be in the right form. In order to connect up the two arguments, a glue argument may be needed – an argument that actually rests on the main diagonal, and serves only to rephrase a statement within the same granularity. In this case, we need glue at <*component, component*>) to recast the claims generated by KAOS into claims that can be established by the Astree.

(c) *The component techniques must provide sufficient confidence and coverage at acceptable cost.*

We could 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 coverage needed to certify a complex system with confidence. We discuss ways to add confidence and cost information to CDADs as future work, in Chapter 7.

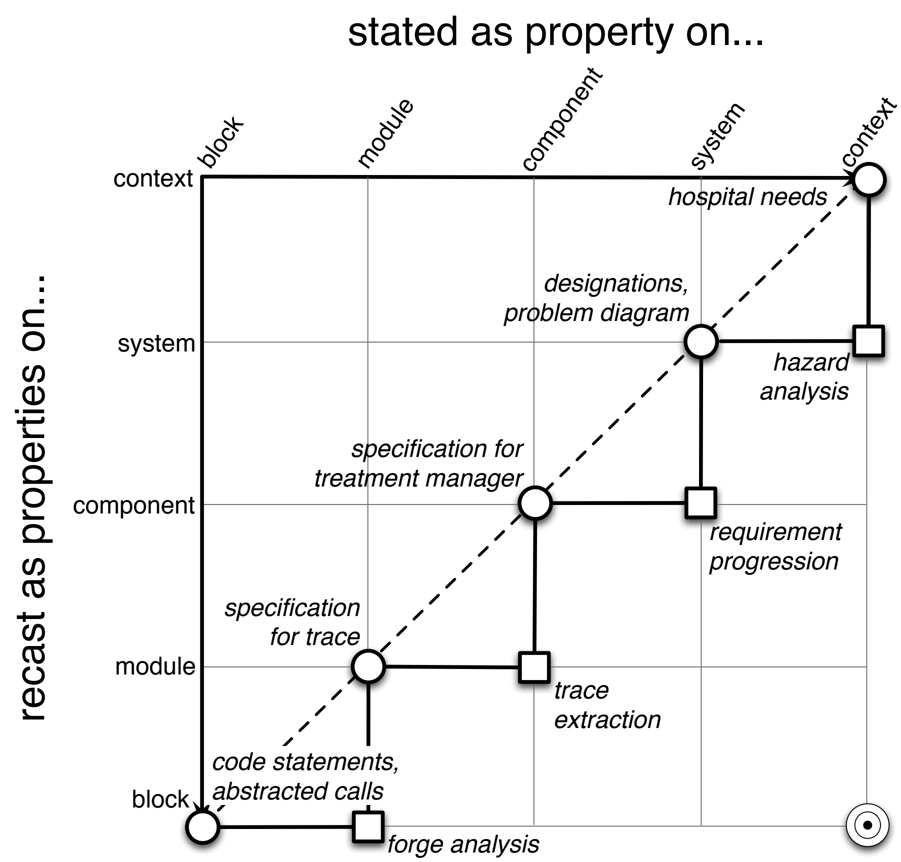

Figure 2-11: A composite technique used to analyze the BPTC.

2.2 Structuring a Dependability Argument

Our general approach to constructing composite arguments can be applied to the BPTC case study by instantiating it with a particular set of component techniques. Figure 2.1.6 shows the techniques we combine, as arguments and statements, to form the composite technique we use in Chapter 4.

Hospital Needs : An informal discussion with hospital administrators about the role of the BPTC at MGH.

This statement is at <*context, context*>.

Hazard Analysis : A characterization of dangerous states that could be induced by the system, including a classification of each hazard's danger level (low, medium, high) [49].

This argument is at <*context, system*>.

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

This statement is at <*system, system*>.

Problem Diagram : The system requirement is initially expressed with a problem diagram, from the Problem Frames approach [40, 38]. This step recasts the requirement from its original (possibly informal) statement into a form that is be amenable to requirement progression. It identifies the domains relevant to the subsystem under consideration, and the phenomena through which those domains interact, using the formal terms introduced by the designations.

This statement is at <*system, system*>.

Requirement Progression : The system requirement is transformed into a software specification using *requirement progression* [81]. The resulting diagram is called an *argument diagram*, which is the problem diagram annotated with a collection of domain assumptions (*breadcrumbs*) sufficient to enforce the original system requirement. Domain assumptions about software components can be used as specifications for those components.

This argument is at <*system, component*>.

Argument Validation : Along with the argument diagram is an Alloy [30] model which mechanically confirms that the breadcrumbs do indeed enforce the desired system property.

This argument is at <*system, component*>.

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

This statement is at <*component, component*>.

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

This argument is at <*component, component*>.

Specification for TM : A specification of the correct behavior of the Treatment manager, resulting from interpreting one of the breadcrumbs derived during requirement progression. It is now phrased in terms of code terminology, and is thus amenable to analysis.

This statement is at <*component, component*>.

Trace Extraction : The process of identifying the subset of the code relevant to the TM specification. It is identifying by using a Flow Diagram, an informal annotation of the problem diagram, indicating the flow of information through the system. The flow diagram is used to label (and thus implicitly order) the domains and the letters assigned to arcs. These labels are purely for the sake of bookkeeping and help us to systematically develop the argument.

This argument is at <*component, module*>.

Specification for Trace : A specification of what it means for the identified subset to be correct. In this case, it is the same claim as the specification for the treatment manager, but now applied to a small chunk of code.

This statement is at <*module, module*>.

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

This argument is at <*module, block*>.

Code Statements : Individual lines of code in the code base, and the assumption that they have the semantics assigned to them by Forge.

This statement is at <*block, block*>.

Together, these component techniques provide an argument at <*context, 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.

Chapter 3

Requirement Progression

3.1 Overall Approach

The *problem frames* approach offers a framework for describing the interactions amongst software and other system components [38, 40]. It helps the developer understand the context in which the software problem resides, and which of its aspects are relevant to the design of a solution [31, 39, 44, 47]. In this approach, a requirement is an end-to-end constraint on phenomena from the problem world, which are not necessarily controlled or observed by the machine. During subsequent development, the requirement is typically factored into a specification (of a machine to be implemented) and a set of domain assumptions (about the behavior of physical devices and operators that interact directly or indirectly with the machine).

3.1.1 The Need for Progression

A key advantage of the problem frames approach is that it makes explicit the argument that connects these elements. In general, this argument takes a simple form: That the specification of the machine, in combination with the properties of the environment, establishes the desired requirement. When the environment comprises multiple domains, however, the argument may take a more complicated form. The problem frames representation allows the argument to be shown in an *argument diagram* – the problem diagram embellished with the argument.

In the problem frames book [40], a strategy for constructing such arguments, called *problem progression*, is described. But, since each step in a problem progression involves deletion of domains from the diagram, the strategy does not result in an argument diagram; rather, it produces a series of diagram fragments. The approach described in this paper, which we call *requirement progression*, likewise aims to produce an argument diagram. Its steps produce accretions to the diagram, never deletions, and the diagram resulting from the final step is an argument diagram in the expected form.

Often, the problem diagram fits a well established pattern (a *problem frame*), and the argument required will be an instantiation of an archetypal argument. As our logging example will illustrate, not all problems match existing frames, and an argument diagram must be specially constructed using progression.

3.1.2 Our Approach

Our approach relies upon the analyst's ability to accurately distill, disambiguate, and formalize the requirement. One of the benefits of problem oriented software engineering [44], of which problem frames is an example, is that the analyst is permitted to formulate the requirement in terms of whatever phenomena are convenient for describing the actual system requirement. For example, a designer of a traffic light might write a requirement saying "cars going different directions are never in the intersection at the same time". The analyst then methodically transforms the requirement so that it constrains only controllable phenomena, making sure that the new version is sufficiently strong to enforce the original requirement. For example, the traffic light designer might reformulate the requirement to say "the control unit sends signals to the traffic lights in the following pattern...", and justify the reformulation by appealing to known properties about how cars and traffic lights behave. Attempting to write the reformulated version from scratch is error prone. As with other progression techniques (e.g. [70]), our goal is to provide support for performing that transformation systematically and accurately. Our technique is most appropriate when the requirement can be phrased in a formal language, although the methods we describe could also guide reasoning about informal requirements.

We demonstrate our technique on two examples. The first example is of a two-way traffic light similar to the one described in the problem frames book [40]. It demonstrates the use of our technique to specialize the correctness argument of the problem frame that matches the problem diagram. The second example is a simplified view of the logging facility used in a radiation therapy medical system. It demonstrates the use of our technique when no single existing problem frame matches the entire problem. These examples are perhaps not sufficiently complex to properly demonstrate the need for systematic requirement progression, but they do illustrate the key elements of our approach and indicate its strong and weak points.

In both examples, the various constraints are formalized in the Alloy modeling language, and the Alloy Analyzer [30, 34, 37] is used to check that the resulting specification and domain assumptions do indeed establish the desired system-level properties. The Alloy Analyzer can check the validity of a transformation with a bounded, exhaustive analysis. Our transformation technique is not tied to Alloy; we chose Alloy because it is simple, was familiar to us, provides automatic analysis, and allows a fairly natural expression of the kinds of requirements and assumptions involved in these examples.

In Chapters 4 and 5 we apply requirement progression to reals systems, demonstrating its applicability to complex system. There, we see that these techniques scale to reals systems, and are not actually much more difficult to use there than on the toy examples shown in this chapter.

3.2 Problem Frames

Before one can establish a system requirement, one must articulate it.

The Problem Frames approach is a technique for describing and analyzing desired system properties. The Problem Frames approach offers a framework for describing the interactions amongst software and other system components [38, 40]. It helps the developer understand the context in which the software problem resides, and which of its aspects are relevant to the design of a solution [31, 39, 47]. Once the requirement is articulated as a problem diagram, it becomes amenable to more systematic analysis (Section 3.3).

The problem frames approach is an example of *problem oriented software engineering* [44], meaning that it focuses on the context in which a system operates rather than the internal architecture of the system. It emphasizes the distinction between phenomena one wishes to indirectly constrain (the system requirement) and phenomena the software can directly control (the component specifications and domain assumptions). System analysis is a matter of understanding the indirect links between those two sets of phenomena.

One benefit of this approach is that the analyst formulates the system requirement in terms of whatever phenomena are most appropriate and convenient. As a result, we have a higher confidence that the written requirements accurately reflect the intended requirements. Attempting to directly write the requirement in terms of controllable phenomena can be subtle and error prone. The problem frames approach separates the articulation of the requirement from the transformation of that requirement into a form usable as a specification.

In this chapter, we will describe *Requirement Progression*, a technique for systematically decomposing a system requirement (written in terms of the phenomena to be controlled) into a set of specifications (written in terms of controllable phenomena). First, however, we will examine Problem Frames in a bit more detail.

3.2.1 More Detail

An analyst has, in hand or in mind, an end-to-end requirement on the world that some machine is to enforce. In order to implement or verify the machine, one needs a specification at the machine's interface. Since the requirement typically references phenomena not shared by the machine, it cannot serve as a specification. The Problem

Frame notation expresses this disconnect as shown in Figure 3-1. [1]

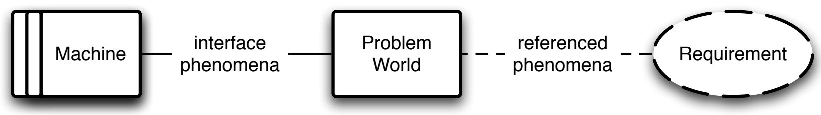

Figure 3-1: A generic problem frames description showing the disconnect between the phenomena controlled by the machine (the interface phenomena) and those constrained by the requirement (the referenced phenomena).

The analyst has written a *requirement* (right) describing a desired end-to-end constraint on the *problem world* (center). The requirement references some subset of the phenomena from the problem world (right arc). A *machine* (left) is to enforce that requirement by interacting with the problem world via *interface phenomena* (left arc).

For example, in a traffic light system, the problem world might consist of the physical apparatus (lights and sensors) and external components (cars and drivers), the requirement might be that cars do not collide, and the specification would be the protocol by which the machine generates control signals in response to the monitoring signals it receives. The machine and its specification only have access to the phenomena pertaining to control and feedback signals, whereas the requirement is a constraint on the directions and positions of the cars.

The problem world is broken into multiple *domains*, each with its own assumptions. Here, for example, there may be one domain for the cars and drivers (whose assumptions include drivers obeying traffic laws), and another for the physical control apparatus (whose assumptions describe the reaction of the lights to control signals received, and the relationship between car behavior and monitoring signals generated). A *problem diagram* shows the structure of the domains and phenomena involved in a particular situation. One possible problem diagram for the traffic light

[1] We deviate slightly from the standard problem frames notation when drawing an arc indicating that domain `D` controls phenomenon `p`. Rather than labeling the arc `D!p`, we label it `p` and place an arrow head pointing away from `D`. When not all phenomena shared by two domains are controlled by the same domain, separate arcs are used. Most of our diagrams omit indications of control altogether, as it is not currently relevant to our approach.

system is shown in Figure 3-2.

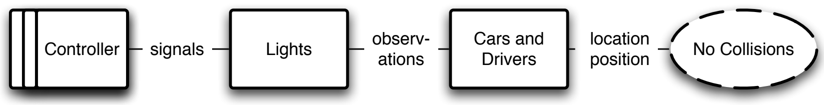

Figure 3-2: A problem diagram describing the domains and phenomena for a two-way traffic light. The arc connecting two domains is labeled by the phenomena shared by those domains – those phenomena that both domains involve. The arc connecting the requirement to a domain is labeled by the phenomen referenced (constrained) by the requirement.

To ensure that the system will indeed enforce the requirement, it is not sufficient to verify that the machine satisfies its specification. In addition, the developer must show that the combination of the specification and assumptions about the problem world imply the requirement. To argue that the machine, when obeying the specification, will enforce the requirement, we must appeal to assumptions about how the domains act and interact – how lights respond to control signals, how monitoring signals are generated, how drivers react to lights, and how cars respond to driver reactions. Those behaviors are recorded as domain assumptions, as shown in Figure 3-3.

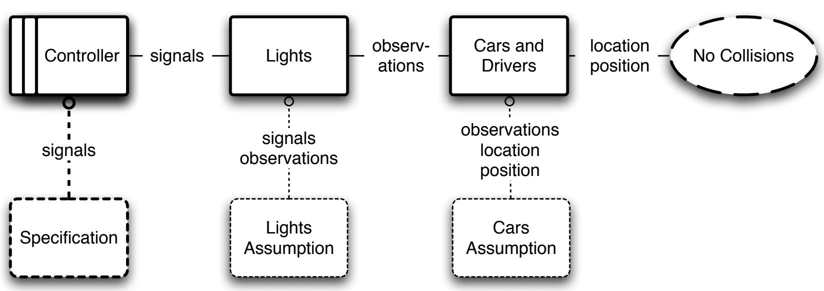

Figure 3-3: Assumptions about the intervening domains are expressed as partial domain descriptions in the form of constraints on their behaviors. These assumptions help us relate the machine specification to the system requirement. As with a requirement, the arc connecting an assumption or specification to its domain is labeled with the phenomena referenced by that assumption.

A problem diagram serves to structure the domains and their relationships to the machine and the requirement, and is accompanied by a *frame concern* that structures the argument behind this implication. The traffic light system, for example, matches the *required behavior* shown in Figure 3-4 [40].

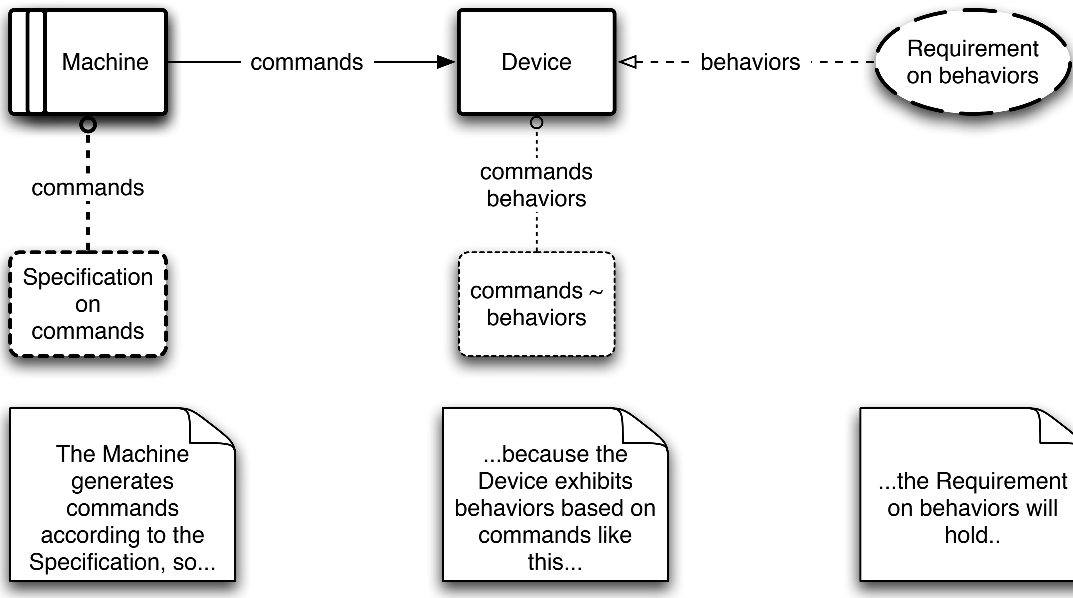

Figure 3-4: An informal argument diagram for the *required behavior* frame.

Because the required behavior frame concern is general enough to match many situations, it only gives an outline of the correctness argument and serves primarily to focus attention on the kinds of domain properties upon which the completed correctness argument is likely to rely. Applying it to the traffic light problem diagram suggests the argument structure shown in Figure 3-12.

This information is a valuable aid in building the full argument, but would greatly benefit from a systematic approach for determining exactly which properties of the domains are relevant, deriving an appropriate specification for the machine, and providing a guarantee that the specification and domain properties are sufficient to establish the requirement. This chapter describes such an approach.

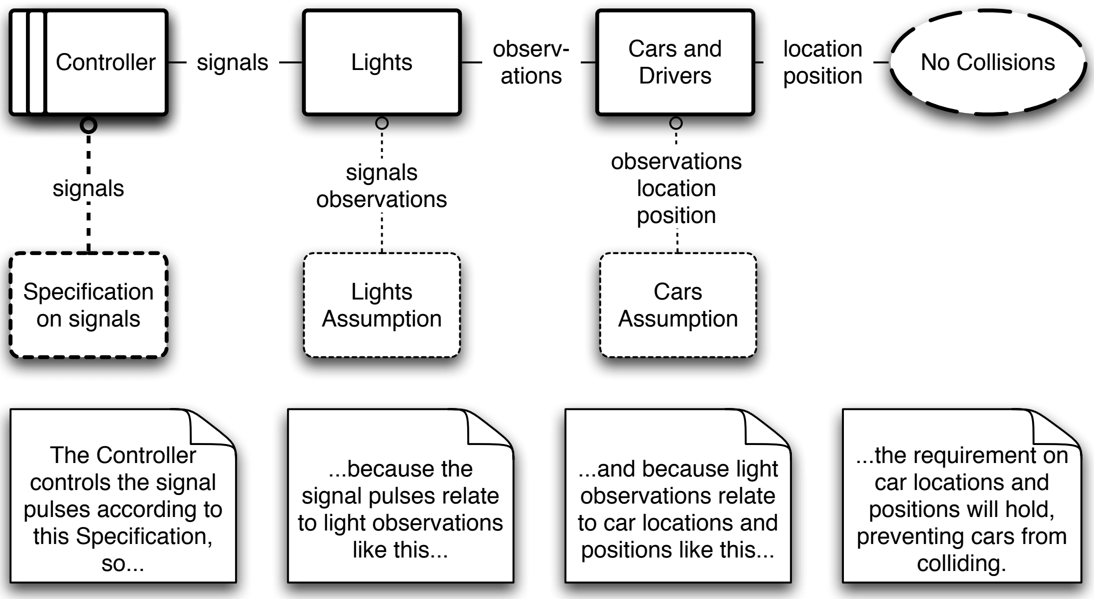

Figure 3-5: The informal argument diagram that results from applying the required behavior frame to the two-way traffic light problem diagram. It provides an outline for arguing that the specification enforces the requirement, and it indicates what sort of domain assumptions will be needed to build that argument.

3.3 Requirement Progression

In this section, we introduce an incremental way of deriving a specification from a requirement via requirement progression. A byproduct of the progression is a trail of domain assumptions, called *breadcrumbs*, that justify the progression and record the line of reasoning that lead to the specification.

Requirements, specifications, and breadcrumbs are three instances of *domain constraints*. Requirements can touch any set of domains but usually touch only non-machine domains; specifications touch only the machine domain; and each breadcrumb touches only a single non-machine domain. The only thing barring the requirement from serving as a specification is that it mentions the wrong set of phenomena. Unfortunately, altering it to mention the right set of phenomena (those at the interface of the machine domain) is no easy matter and requires appealing to properties of the intervening domains. The transformation process we describe is an incremental method for achieving such an alteration and recording the necessary domain properties. [2]

3.3.1 Available Transformations

There are three types of steps in the transformation process: *adding* a breadcrumb permits the requirement to be *rephrased*, which in turn enables a *push* to change which domains it touches. Figure 3-6 shows an archetype of how these steps can turn a requirement into a specification. In that example, there is one interface phenomenon controlled by the machine (`p1`) and one phenomenon mentioned by the requirement (`p2`). The intervening domain involves both of those phenomena.

(a) *Add* a breadcrumb constraint, representing an assumption about a domain in the problem world. The breadcrumb must touch a single domain that is currently touched by the requirement (and no other domains), and therefore only mention phenomena from that domain (e.g. `p1` and `p2`). [3] It is chosen so as to enable a useful rephrasing (step b). The breadcrumb must be validated by a domain expert to ensure that it is a valid characterization of the constrained domain.

(b) *Rephrase* the requirement so that it represents a different constraint. The new version of the requirement must touch the same domains, but it may mention (and thereby constrain) a different subset of the phenomena of those domains (e.g. mention `p1` instead of `p2`). The rephrasing is chosen so as to enable a useful push (step c).

The analyst must verify that existing breadcrumbs are sufficiently strong to permit the rephrasing by establishing the implication

$$(\text{breadcrumb} \wedge \text{new requirement}) \Rightarrow \text{prior requirement}$$

[2] During the progression, the requirement will undergo a sequence of changes until it has become a specification. It is useful to distinguish the initial version of the requirement (that the system designers actually want enforced) from the intermediate versions of the requirement (that are only meaningful within the progression process). We will thus use the term *goal* to denote an intermediate version of the requirement, and reserve the term *requirement* to refer to the original system requirement. At any given point in the process, there is exactly one goal; the goal is initially the requirement and will eventually be a specification.

[3] The phenomena mentioned by a breadcrumb might be shared amongst several domains, but there must be a single domain that involves all of them. It is this domain that the breadcrumb touches.

The means of establishing this implication will depend on the language used to express the breadcrumb and requirement constraints.

(c) *Push* the requirement so that it touches a different set of domains but still represents the same constraint over the same phenomena. A push is only permitted if it will preserve the fact that each phenomenon mentioned by the requirement is involved in some domain touched by the requirement, and that every domain touched by the requirement involves some phenomenon mentioned by the requirement.

Typically, a push changes the requirement to touch some domain d' (e.g. the machine) instead of some domain d (e.g. the non-machine domain) such that all the phenomena of d mentioned by the requirement are also phenomena of d' (e.g. `p1`). Diagramatically, this means that only one of the arcs emanating from the requirement is altered, and the phenomena labeling that arc must be shared between d and d'. [4]

The analyst continues to perform these transformations (in any order) until the requirement touches only the machine domain. At that point, it only mentions phenomena at the interface of the machine and is thus a valid specification.

In theory, one might want to express an assumption that mentions phenomena that are not involved in any single domain – the constraint representing such an assumption would necessarily touch two or more domains and would therefore be an invalid breadcrumb. Such assumptions inhibit local reasoning and are hard to validate, as there may not be any single domain expert who can certify them. In practice, we have not found (or been able to construct) an example where such an assumption is needed. We therefore only allow assumptions about intra-domain properties; inter-domain properties must be factored into several intra-domain properties (and incorporated as a set of breadcrumbs).

[4] If a requirement mentions a phenomenon that is shared between domains, we consider the diagram to be well formed as long as the requirement touches *either* of those two domains. It is good style, but not necessary, for the requirement to touch the domain that controls the mentioned phenomenon. A push transformation will violate that good style but leave the diagram well formed. Note that the problem frames notation, as given in the problem frames book [40], is ambiguous about this issue.

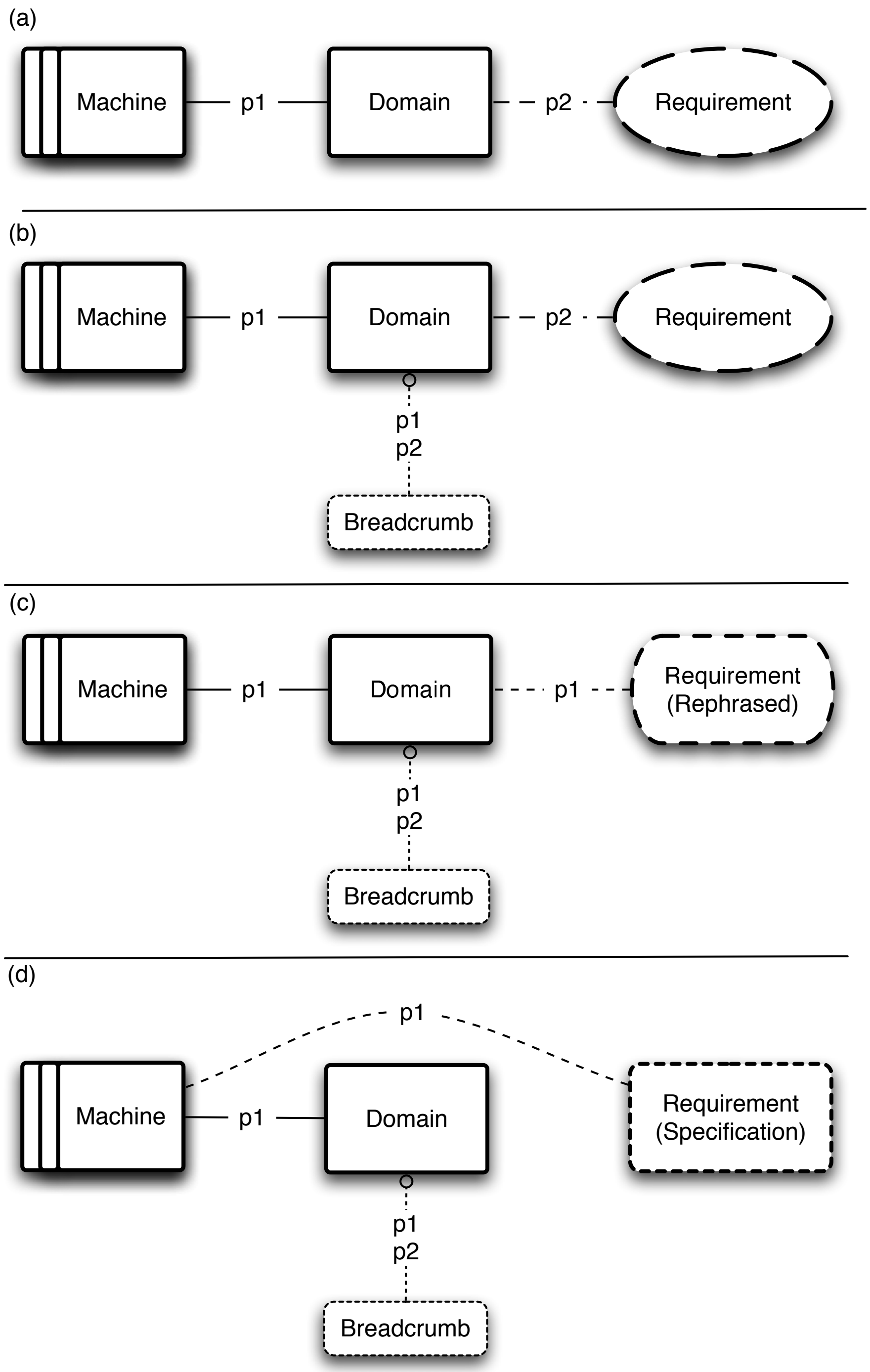

Figure 3-6: An archetypal requirement progression: (a) Prior to the transformation (b) A *breadcrumb* constraint is added, representing an assumption about how the domain relates phenomena `p1` and `p2`. (c) That breadcrumb permits the requirement to be *rephrased* to reference `p1` instead of `p2`. (d) The rephrasing enables a *push*, moving the requirement from the problem-world domain onto the machine.

3.4 Two-Way Traffic Light

Our first example is of a two-way traffic light, similar to the one described in the problem frames book [40]. A cartoon of this situation is shown in Figure 3-7. A two late road has been reduced to 1 lane for a short stretch, perhaps because of construction on one of the lanes or because of a narrow bridge. A red-green light has been placed at either end of the stretch, with a computer system synchronizing the two units. The task is to provide a specification to the computer control unit that will prevent head-on collisions from occurring on the road segment. Of course, to provide that spec, we will have to make some assumptions about how the light units behave (in response to signal pulses sent to them) and how the cars behave (in response to the red and green lights displayed by the light units). Requirement progression will guide and validate the discovery of those assumptions and specification.

This scenario is a good example of a problem frame with a *linear topology*: the machine and requirement are on opposite ends of a linear sequence of domains. Requirement progression is simply a matter of shifting the requirement down that sequence and onto the machine. Later, in Section 3.5, we will see how requirement progression works on a *branching topology*.

The two-way traffic light is also instructive because it is a prototypical instance of the *required behavior* frame, one of the five problem frames presented in the problem frames book [40]. It is thus a good example of how to use our requirement progression technique to specialize the correctness argument suggested by that frame.

The two-way traffic light problem frame is shown in more detail in Figure 3-9, along with the requirement we will focus on in this example. To make sense of the phenomena names used in that diagram, consult the designations, given in Figure 3-8.

The Light Unit has four physical lights: a red light and a green light in each direction. The control unit sends signal pulses to the light unit to individually toggle the four lights on and off. The cars moving in each direction observe those traffic signals, and then decide whether or not to enter the road segment. The requirement is that cars do not collide, which we will interpret to mean that no two cars are ever

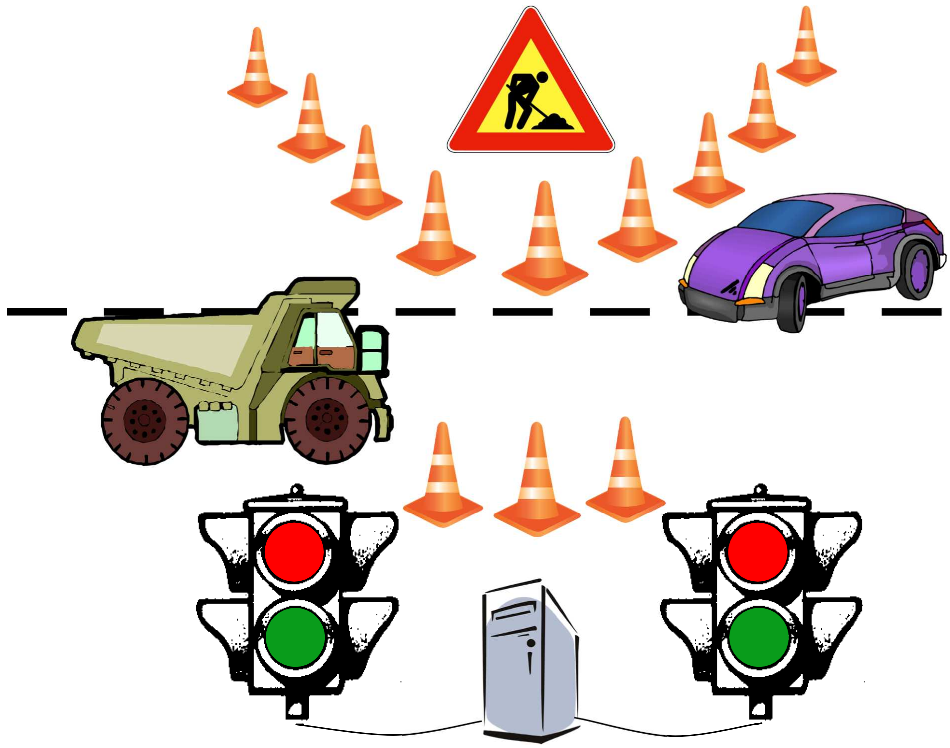

Figure 3-7: A cartoon diagram of the traffic light problem. Cars forced to share a common lane of traffic with oncoming traffic are controlled by a set of red-green lights, synchronized by a computer control unit.

on the road segment at the same time going opposite directions. However, the control unit has no knowledge of, or control over, the cars; it can only send signal pulses to the light units and observe the history of what signals it previously sent.

3.4.1 Basic Declarations

For completeness, we shall include, in addition to the constraints, the Alloy [30, 34, 37] declarations needed to complete the model.

There is a set of cars and two relations about cars: `onSeg` is a binary relation mapping each car to the set of times at which that car is on the road segment. That relation is wrapped by the predicate `CarOnSegment[c, t]`, which determines if a

`NRpulse[t]`	⇔	A red signal pulse is sent to the north light unit at time t.
`NGpulse[t]`	⇔	A green signal pulse is sent to the north light unit at time t.
`SRpulse[t]`	⇔	A red signal pulse is sent to the south light unit at time t.
`SGpulse[t]`	⇔	A green signal pulse is sent to the south light unit at time t.
`NRobserve[t]`	⇔	The northward red light is lit up and can be observed by cars.
`NGobserve[t]`	⇔	The northward green light is lit up and can be observed by cars.
`SRobserve[t]`	⇔	The southward red light is lit up and can be observed by cars.
`SGobserve[t]`	⇔	The southward green light is lit up and can be observed by cars.
`CarDirection[c,t]`	⇔	Car c is on the shared road segment at time t.
`CarDirection[c] = North`	⇔	Car c is moving northward at time t
`CarDirection[c] = South`	⇔	Car c is moving southward at time t

Figure 3-8: Designations for a two-way traffic light.

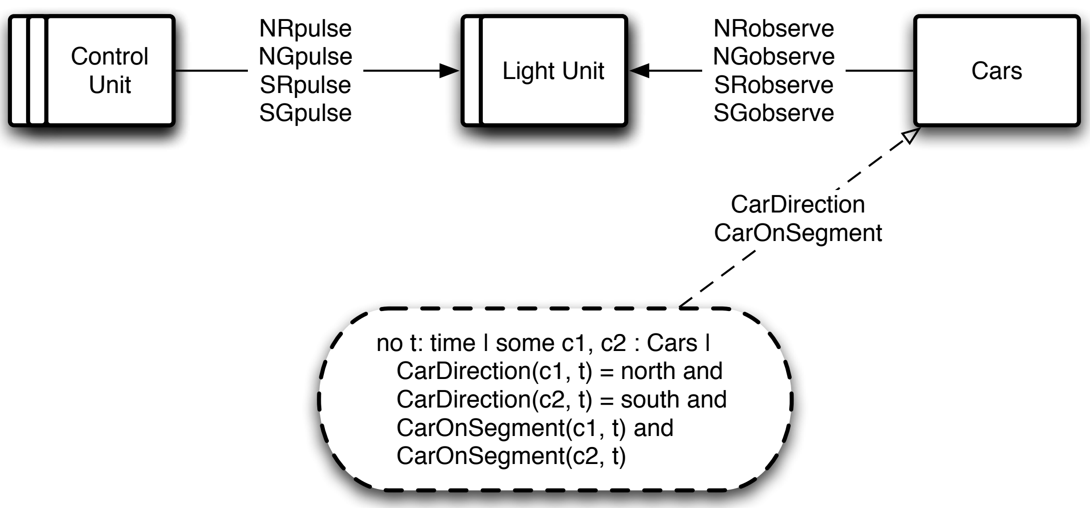

Figure 3-9: A more detailed problem diagram for the two-way traffic light problem. The constraint has been formalized and expressed using the Alloy language, a relational first-order logic.

car `c` is on the segment at time `t`. `dir` is a ternary relation mapping each car and direction to the set of times at which that car is moving in that direction. This relation is wrapped in the function `CarDirection[c, t]` which returns the direction a given car is moving at a given time. For the rest of this example, we will use the predicate and the function, rather than their equivalent relations, in order to give our constraints a more natural syntax for readers who are not familiar with relational logic.

```
sig Cars {
  onSeg: set Time,
  dir: Direction → Time }
pred CarOnSegment[c: Cars, t: Time] { t in c.onSeg }
fun CarDirection [c: Cars, t: Time] : Direction { [c.dir].t }
abstract sig Direction { }
one sig north extends Direction { }
one sig south extends Direction { }
```

There is a set of times, divided into 8 non-exclusive subsets. For example, `NRO` represents the subset of times at which the northern red light is observed, and `NRP` represents the set of times at which a signal pulse is sent to the northern red light. These 8 subsets are wrapped by 8 predicates. For example, `NRobserve[t]` determines whether or not the northern red light is observed at time `t`, and `NRpulse[t]` determines whether or not there was a signal pulse sent to the northern red light at time `t`. From now on, we will use the predicates, rather than the subsets, to make our constraints more readable.

```
sig Time { }

sig NGO, SGO, NRO, SRO in Time { }
pred NGobserve[t: Time] {t in NGO}
pred SGobserve[t: Time] {t in SGO}
pred NRobserve[t: Time] {t in NRO}
pred SRobserve[t: Time] {t in SRO}

sig NGP, SGP, NRP, SRP in Time { }
pred NGpulse[t: Time] {t in NGP}
pred SGpulse[t: Time] {t in SGP}
pred NRpulse[t: Time] {t in NRP}
pred SRpulse[t: Time] {t in SRP}
```

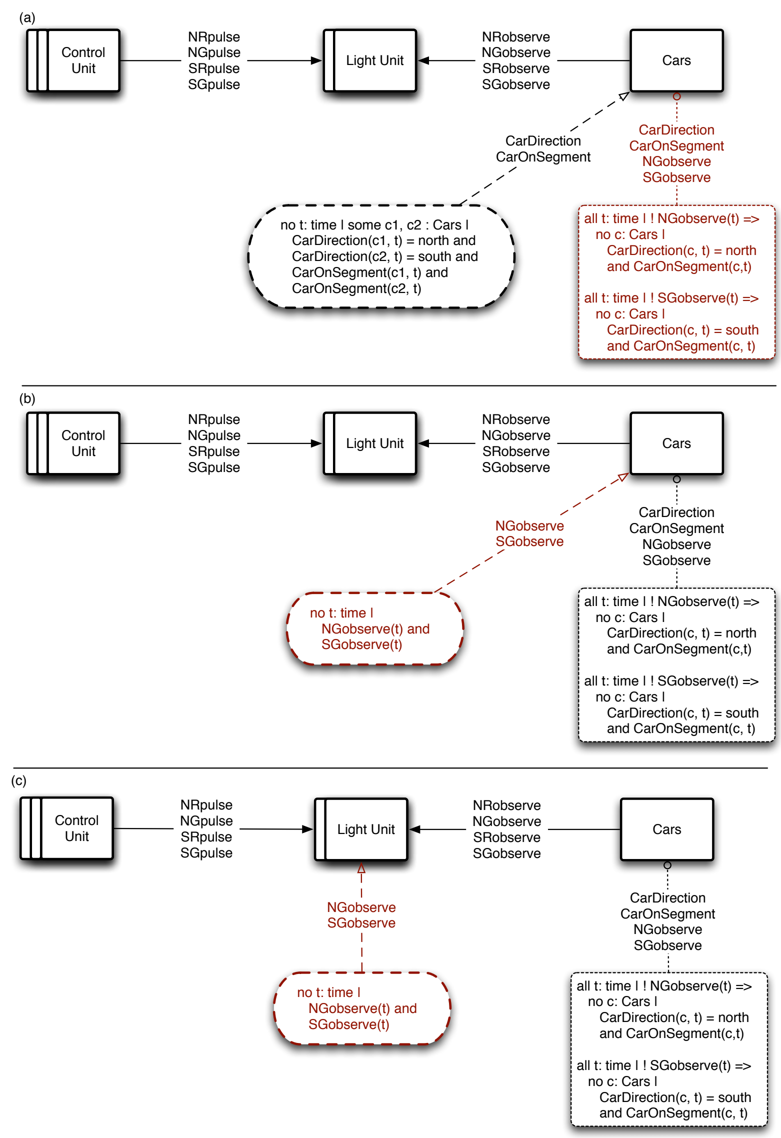

Figure 3-10: The first transformation: (a) A *breadcrumb* constraint is added to the `Cars` domain, representing the assumption that car behavior can be determined by knowing what traffic signals were observed. (b) Taking advantage of that assumption, the requirement is *rephrased* so that it refers to observations instead of car behaviors. (c) Because the requirement refers only to phenomena shared between the `Cars` and `Light Unit` domains, it can be *pushed* from one to the other.

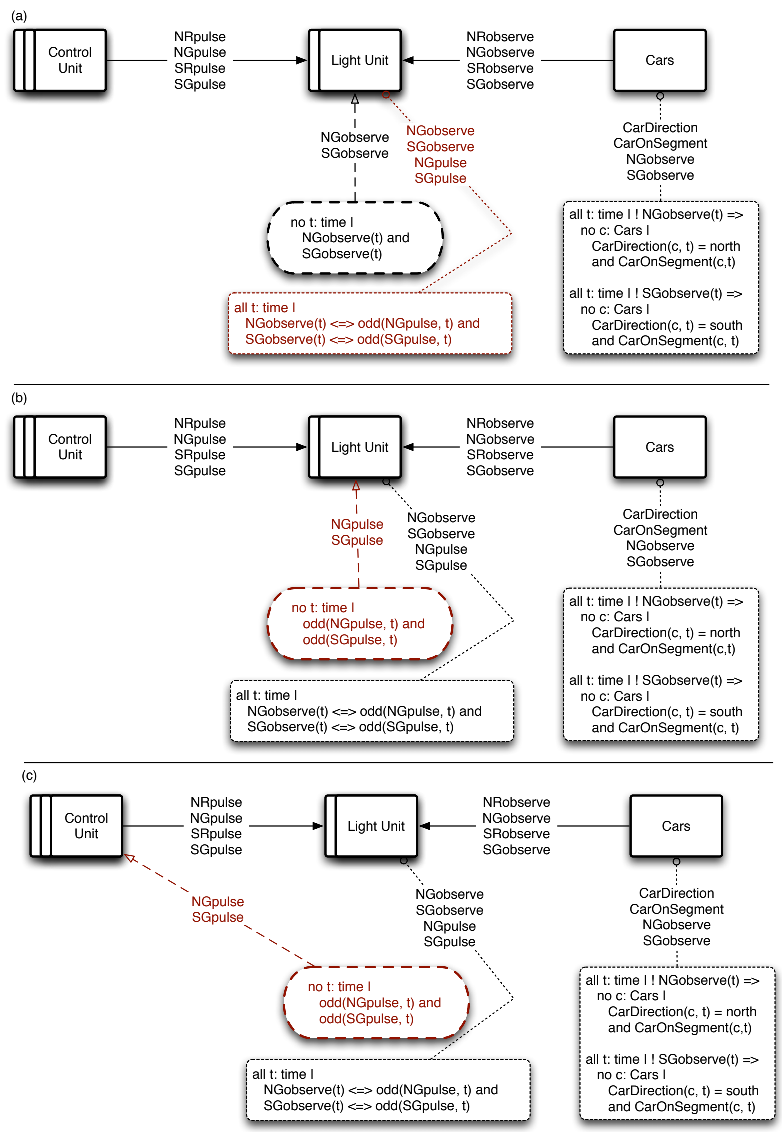

Figure 3-11: The second transformation: (a) a *breadcrumb* constraint is added to the `Light Unit` domain, representing the assumption that signal pulses completely determine how the cars observe the traffic light. (b) Taking advantage of that assumption, the requirement is *rephrased* so that it refers to signal pulses instead of observations. (c) Because the requirement refers only to phenomena shared between the `Light Unit` and `Control Unit` domains, it can be *pushed* from one to the other. The problem diagram is now an argument diagram.

3.4.2 The Requirement

The initial requirement that cars do not collide can now be expressed as follows:

```
pred Requirement1 [ ] {
  no t: Time | some c1,c2: Cars |
     CarDirection[c1, t] = north and
     CarDirection[c2, t] = south and
     CarOnSegment[c1, t] and
     CarOnSegment[c2, t]
}
```

The initial problem diagram with this requirement is shown in Figure 3-9.

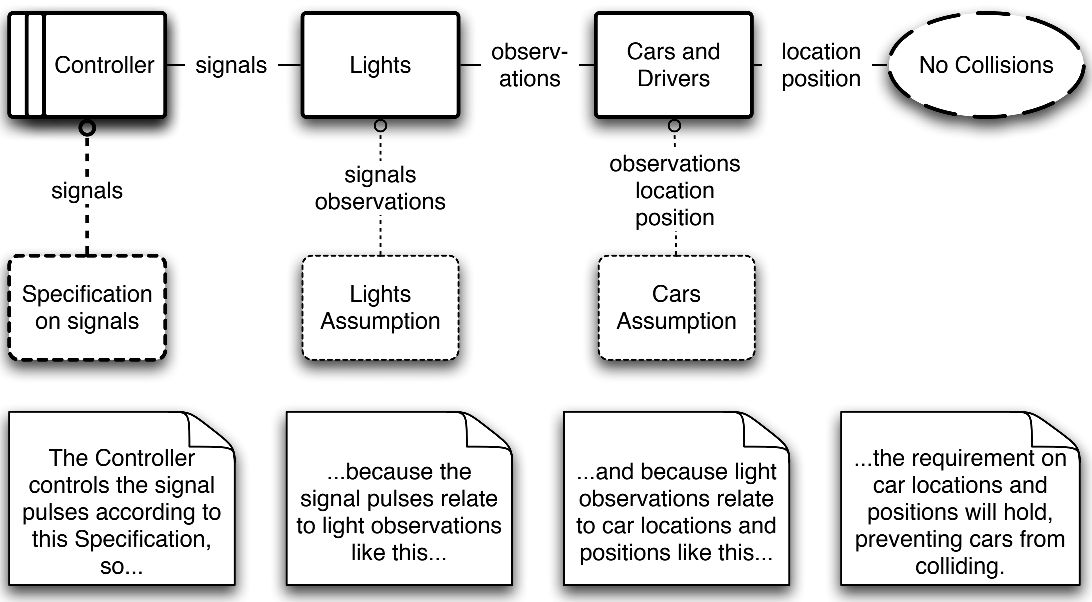

Figure 3-12: The informal argument diagram that results from applying the required behavior frame to the two-way traffic light problem diagram. It provides an outline for arguing that the specification enforces the requirement, and it indicates what sort of domain assumptions will be needed to build that argument.

3.4.3 Step 1: from Cars to Light Units

The first thing we would like to do is to push the requirement from the `Cars` domain onto the `Light Unit` domain, following the heuristic of trying to shift the requirement closer to the `Control Unit`. In order to justify such a push, we will add a breadcrumb constraint on `Cars` which permits us to rephrase the requirement so that the only phenomena it mentions are `NRobserve`, `NGobserve`, `SRobserve`, and `SGobserve`. We will then be able to push the requirement from `Cars` onto `Light Unit`. These three tasks are illustrated in Figure 3-10 and narrated below.

(A) Add a Breadcrumb

The frame, shown in Figure 3-12, suggests that we characterize how the `Cars` domain relates `CarDirection` and `CarOnSegment` with the four observation phenomena. We do so by adding the following breadcrumb constraint to `Cars`, expressing the assumption that cars never disobey red lights. In Alloy, we represent each breadcrumb as a predicate.

```
pred CarsBreadcrumb [ ] {
  all t: Time | not NGobserve[t]
    ⇒ no c: Cars | CarDirection[c,t] = north and CarOnSegment[c,t]
  all t: Time | not SGobserve[t]
    ⇒ no c: Cars | CarDirection[c,t] = south and CarOnSegment[c,t]
}
```

This constraint further characterizes the `Car` domain: at any given time, if a car does not observe a green light in its direction, then it cannot be on the road segment. [5] We discuss later why red lights do not appear in this assumption – see Section 3.4.5. The result of this addition is shown in Figure 3-10a.

(B) Rephrase the Requirement

Instead of requiring that no two cars be in the intersection moving in opposite directions at the same time, we can instead require that opposing green lights are never both observed to be green at the same time.

[5] For the sake of simplicity, we will ignore the delays between when a light observation is made and when car positions change in response to that change. There is no time allowed for the intersection to clear, and there is no yellow light. These assumptions are more reasonable, if one considered a time t to represent a period of time, rather than a moment in time. A signal at time t means that the signal is sent at the beginning of time period t. A car moving northward at time t means that the card is moving northward at any point during time period t. Of course, a car that changes direction would be considered to be moving both north and south during that time period.

```
pred Requirement2 [ ] {
  no t: Time | NGobserve[t] and SGobserve[t]
}
```

The result of this rephrasing is shown in Figure 3-10b.

To validate the rewrite, we are obliged to show that the new requirement, conjoined with the new breadcrumb, implies the prior requirement.

```
assert Step1 {
  Requirement2 and CarsBreadcrumb ⇒ Requiremet1
}
check Step1 for 10
```

In general, how such implications are discharged will depend on the problem domain and the level of confidence needed in the requirement. Since our constraints are written in first-order relational logic, we used the Alloy Analyzer to perform a bounded, exhaustive check [37, 30]. The check passed for a scope of 10, meaning that the property is not violated by any situation with up to 10 cars and up to 10 points in time[6].

(C) Push the Requirement

The only phenomena mentioned by the new requirement are `NGobserve` and `SGobserve`. Since those phenomena are shared by both the `Cars` and `Light Unit` domains, we are permitted to push the requirement from one to the other. The result of this push is shown in Figure 3-10c.

3.4.4 Step 2: From Light Unit to Control Unit

The requirement is now one step away from being a specification. We repeat the process to shift the requirement the rest of the way onto the `Control Unit` domain (the machine). In order to do so, we will need add another breadcrumb and perform another rephrasing of the requirement. This process is illustrated in Figure 3-11 and narrated below.

[6] Each execution of the Alloy model was solved in under 1 second on a 133MHz G4 PowerMac with 800Mb of RAM, using the freely available version of Alloy 4 [30]

(A) Add a Breadcrumb

Once again, we appeal to the frame (Figure 3-12) for guidance on what breadcrumb to add. This time, we need to make an assumption about the `Light Unit` domain that will help us reconcile the observation and signal pulse phenomena. If we assume that the parity of signal pulses determines how the lights are observed, then we can substitute mentions of signal pulses for mentions of observations. We do so by adding the following breadcrumb constraint to `Light Unit` about the electrical wiring of the unit and about the reliability of observations:

```
pred LightUnitBreadcrumb [ ] {
  all t: Time |
    NGobserve[t] ⇔ odd[NGpulse, t] and
    SGobserve[t] ⇔ odd[SGpulse, t]
}
```

where `odd` is a function that determines the parity of the number of occurrences of the given phenomenon up to the given time. The most recent breadcrumb therefore says that, at any point in time, if an odd number of signal pulses have been sent to a particular light, then that light is on and will be observed. If an even number have been sent, then it is off and will not be observed. The result of this addition is shown in Figure 3-11a.

(B) Rephrase the Requirement

In light of that breadcrumb, we rephrase the requirement to mention signal pulses instead of light observations:

```
pred Requirement3 [ ] {
  no t: Time | odd[NGpulse, t] and odd[SGpulse, t]
}
assert Step2 {
  Requirement3 and LightUnitBreadcrumb ⇒ Requirement2
}
check Step2 for 10
```

We use the Alloy Analyzer to verify that the new requirement plus the breadcrumb imply the prior requirement. It passes for a scope of 10, so the breadcrumb is strong enough to justify the rephrasing. The result of this rephrasing is shown in Figure 3-

11b.

(C) Push the Requirement

The requirement now mentions only phenomena shared by both the `Light Unit` and `Control Unit` domains, so we can push it from one to the other. The result of this push is shown in Figure 3-11c.

Now that the requirement has been pushed all the way onto the machine domain, it only mentions phenomena known about by the machine and is a legal specification for that machine. We have derived a specification for the control unit (the final version of the requirement), a correctness argument for why it enforces the original requirement, and a set of assumptions about the world upon which we are relying (the breadcrumbs). The designer can hand that specification off to an engineer to guide or validate an implementation, knowing that (as long as the breadcrumb assumptions hold) the specification is, by construction, sufficient to enforce the original requirement.

3.4.5 Lessons Learnt

One of the primary benefits of problem frames is that it forces the designer to be explicit about what assumptions are being made. Those assumptions can then be checked by domain experts, rather than being left hidden inside of the designer's head. In fact, there is a possible mistake in this example, which might have escaped attention had the breadcrumbs not been explicitly recorded in a formal language as part of our technique.

Recall that the first breadcrumb (`CarsBreadcrumb`) states that a car will not enter the road segment if the green light in its direction is off. Upon closer inspection, suppose the designer realized that this is not true – if neither the red nor the green lights are on, then cars might assume that the system is off and enter the road segment. That breadcrumb needs to be strengthened to mention red observations as well as green ones. The corrected breadcrumb and resulting specification is shown in

Figure 3-13.

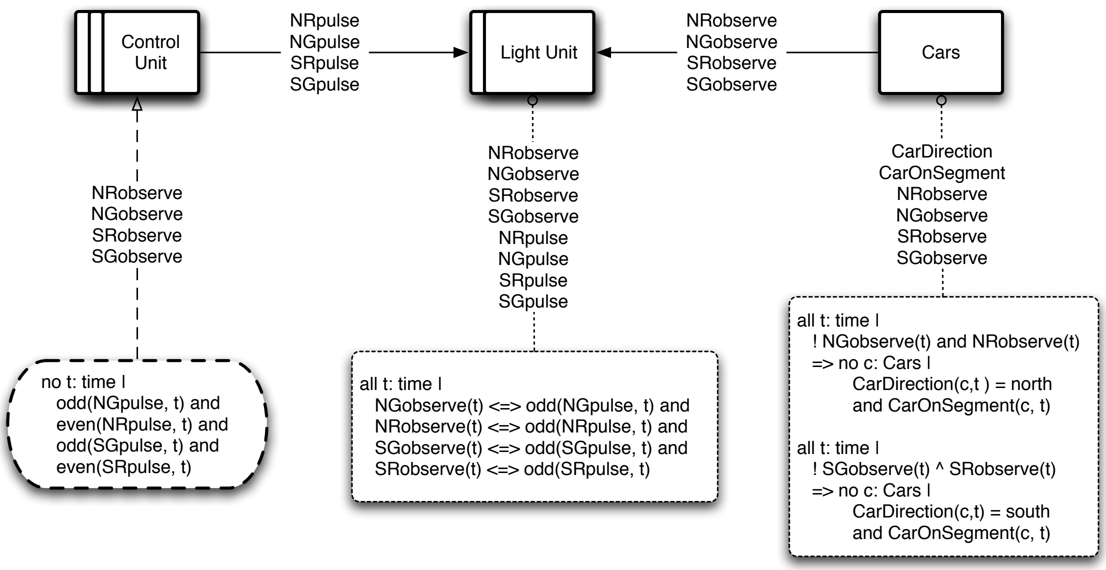

Figure 3-13: The argument diagram that results if we change the breadcrumb on the `Car` domain to permit cars to enter the intersection when neither a red nor a green light shows. In this version of the argument, both red and green lights are relevant.

If, however, the designer decides that the cars breadcrumb is reasonable, then we have learned something about the system: red lights do not play a role in establishing the original safety requirement. Had we gone straight to writing a specification, rather than deriving it incrementally, we would probably have missed this insight and have written an over-constrained specification – we would probably have written one that requires both red and green lights to be turned on and off in a certain pattern, rather than one that just constrains green lights. While sufficient to enforce the original requirement, such a specification would needlessly restrict the design of the control unit.

3.5 Proton Therapy Logging

Our second example is a simplified version of the logging system used in the BPTC system. It is a good example of a problem frame with a *branching topology*: the requirement connects to two different problem-world domains, which in turn connect (either directly or indirectly) to the machine. Requirement progression will involve shifting both of the requirement's arcs onto the machine. Each of the arcs is progressed in a manner similar to what we saw in the traffic light example (Section 3.4), and will be handled independently.

The logging problem is also an instructive example because it does not match any single standard problem frame; one part matches the *information display* frame, and another part matches the *required behavior* frame [40]. While those frames will still provide us with some guidance, neither of them captures the full essence of the logging requirement. Requirement progression can still be used to construct a correctness argument for the system, and will still ensure that we are not relying on implicit domain assumptions. However, we will not be able to rely on existing frames to guide our choice of domain assumptions and will instead introduce assumptions based on existing domain knowledge provided by the BPTC engineers.

3.5.1 System Requirements

The BPTC system is considered to be safety critical primarily due to the potential for overdose — treating the patient with radiation of excessive strength or duration. The International Atomic Energy Agency lists 80 separate accidents involving radiation therapy in the United States over the past fifty years [72]. The most infamous of these accidents are those involving the Therac-25 machine [49, 53], in whose failures faulty software was a primary cause. More recently, software appears to have been the main factor in similar accidents in Panama in 2001 [26].

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

There are two top-priority requirements in the BPTC system: *overdose avoidance* and *logging*.

Overdose Avoidance: At no time should the radiation received by any part the patient's body exceed the dose stipulated in the treatment plan.

Logging: The system should write a log that accurately reflects the dose delivered to the patient.

Without an accurate log, clinicians cannot resume an interrupted treatment without risking an overdose.

Each such requirement is handled, in the problem frames approach, as a distinct *subproblem*. The proton therapy development involves several other subproblems, such as that of positioning the patient accurately [36]. We shall consider only the logging subproblem in this chapter, although we consider other other BPTC concerns in Chapter 4.

3.5.2 Logging Subproblem

The BPTC provides us with some knowledge about the domains that, together with the two partially-relevant frames, suggest some domain properties that are likely to be relevant to our argument (and that will therefore manifest themselves as breadcrumbs).

The challenge presented by the logging problem is that neither the physical machine producing the beam nor the logging disk are completely reliable. For example, the beam equipment could be shut off by a hardware interlock, or the logging database might reach its capacity or its disk might crash. If the log cannot be written, the treatment must be halted.

We assume, however, that the Treatment Control System (TCS) *is* a reliable component and will therefore be given the responsibility of enforcing the requirement in the face of known unreliabilities of the other components. If the TCS is found to be unreliable in ways that prevent it from fulfilling the derived specification, then the process must be repeated to find a looser specification. Doing so is likely to entail stronger assumptions about the reliability of other components, or weakening the requirement we are able to guarantee.

We assume a standard failure model for the disk subsystem and the network. Disk writes are atomic – they either complete successfully, or fail, leaving the disk unaffected. Messages sent on the network may be dropped, delayed, or reordered, but are never corrupted or duplicated.

The radiation hardware may fail like a disk, but presents a harder challenge. A disk write can be made atomic, by regarding it as not having occurred until a single commit bit is flipped, until which point the write can be revoked. The delivery of radiation, in contrast, is irrevocable.

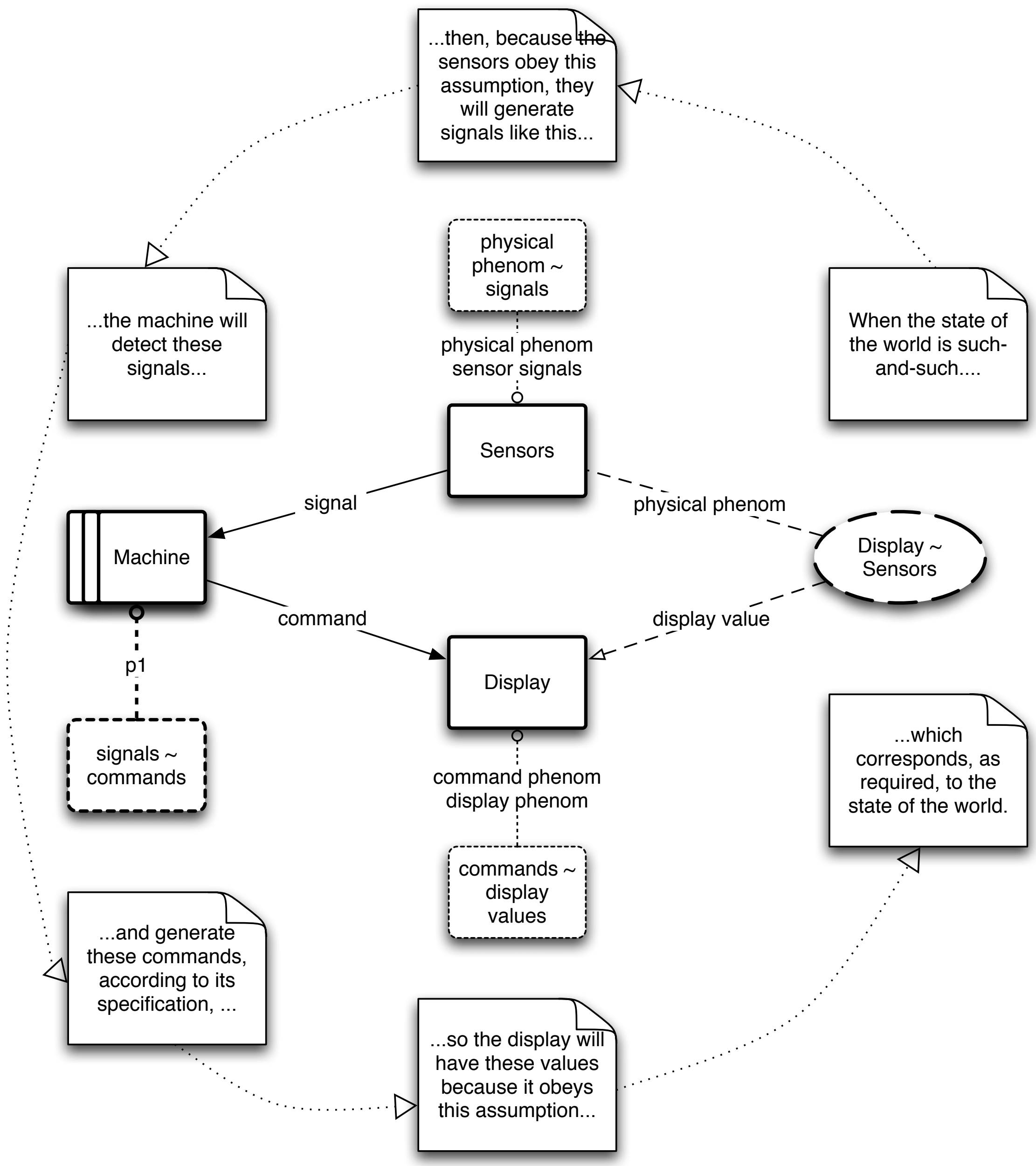

Figure 3-14: An informal argument diagram for the *information display* frame.

`d = #DoseUnit`	$\Leftrightarrow$	Upon the completion of treatment, the patient's body has exactly d units of radiation.
`e = #Entry`	$\Leftrightarrow$	Upon the completion of treatment, there are exactly e entries in the log.
`b in DelivBurst`	$\Leftrightarrow$	At some point during the treatment, a burst of radiation was delivered, associated with the burst b.
`b in ReqBurst`	$\Leftrightarrow$	At some point during the treatment, a request was made for burst b to be delivered.
`b in AckBurst`	$\Leftrightarrow$	At some point during the treatment, an acknowledgement was made that burst b was delivered.
`b in ReqWrite`	$\Leftrightarrow$	At some point during the treatment, there was a request for burst b to be written.
`b in AckWrite`	$\Leftrightarrow$	At some point during the treatment, there was an acknowledgement that burst b was written.

Figure 3-15: Designations for the dose logging problem diagram.

The strategy, therefore, is to deliver the beam in short bursts, logging each burst as it is occurs. If the disk fails, no further bursts are delivered. If the delivery mechanism fails, no further log entries are written. Although the log might not match the treatment exactly, we are assured that they deviate by at most a single burst.

The analysis we perform shows how this approach is justified, and how it reveals a distribution of small but subtle assumptions across the various components of the system.

3.5.3 The Phenomena

Figure 3-16 shows a problem diagram for the logging sub-problem. In it, the informal logging requirement has been formalized using the Alloy language [30, 34, 37]. Designations[7] for the phenomena used in that diagram are given in Figure 3-15.

A `Patient` is prepared to receive radiation from the `Beam Equipment`. The `Treatment Control System` (TCS) issues a series of `ReqBurst` requests to the `Beam Equipment`.[8] Each `ReqBurst` instructs the equipment to deliver a single burst of radiation to the patient, `DelivBurst`, which in turn raises the total radiation delivered to the patient by one `DoseUnit`. After a successful `DelivBurst`, the `Beam Equipment` sends an `AckBurst` acknowledgement back to the `TCS`.

Whenever the `TCS` issues a `ReqBurst`, it attempts to write a record of that dose to the `Log` by issuing a `ReqWrite` request. The `Log` may then create an `Entry` recording that a `DoseUnit` has been delivered to the patient. Upon successfully creating an `Entry`, the `Log` sends an `AckWrite` acknowledgement back to the `TCS`.

Both the `Beam Equipment` and the `Log` are known to be partially unreliable. The `Beam Equipment` will never perform a `DelivBurst` without first receiving a `ReqBurst`, but it may ignore some `ReqBurst`s. Similarly, the `Log` will never write erroneous

[7]A `designation` is an association between formal terms in some description and informal properties of the real world. This is in contrast to a `definition`, which relates formal terms to other formal terms. [40]

[8]The number of such requests is based on the patient's treatment plan. The treatment plan has thus omitted from the problem diagram, since it is not relevant to the logging requirement. It would be included in the problem diagram for the overdose avoidance requirement.

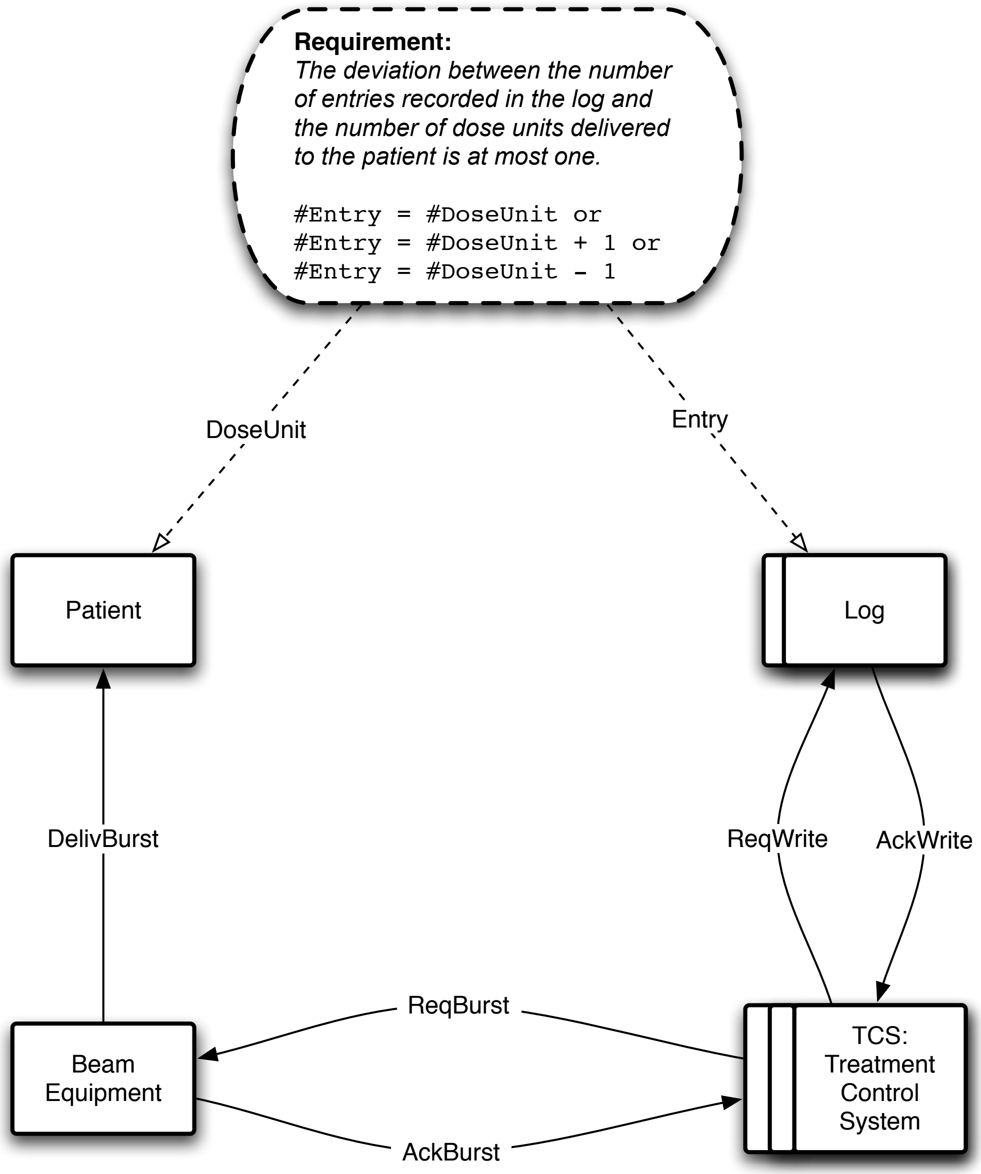

Figure 3-16: The problem diagram for the logging requirement. At any point in time, the doses recorded in the log entries should match the total dose actually delivered to patient, up to a known margin or error.

`Entries`, but it may ignore some `ReqWrite` requests (if, for example, the log has reached its capacity or the disk crashes).

This knowledge about the domains is not initially represented in the problem diagram, as we are not yet sure which parts of it will be relevant to the progression. We will not actually add any of this information into the diagram until it is needed for the progression. Rather, these informal descriptions are used to help the analyst know what domain properties are available for introduction as a breadcrumb.

In this way, the breadcrumbs are only those domain properties relevant to the argument that the derived specification enforces the original requirement, and they are uncluttered by unnecessary (albeit correct) domain assumptions. If the domains are later changed in ways that do not affect the breadcrumbs we used, then the argument represented by the requirement progression will still hold. Including unnecessary, but true, assumptions increases the chance that changes to the domain will require the progression to be reworked.

3.5.4 Matching Problem Frames

No single existing problem frame matches the logging subproblem, although we can draw some insight from two frames that match pieces of the problem.

Logging partly matches the *information display* frame, shown in Figure 3-14. In an information display frame, a Machine resides between Sensors that detect phenomena in the physical world and a Display that encodes some representation of those phenomena. The requirement is that the display values correspond, in some prescribed way, to the state of the physical world. The frame concern focuses our attention on the following characteristics of the three domains: how the Sensor domain relates physical phenomena to signals sent to the machine; how the Machine reacts to those signals by issuing commands to the Display; and how the Display reacts to those commands by rendering display values. The correctness argument will follow this chain to argue that any physical world phenomenon will result in the appropriate display values.

The Logging facility is an information display problem in the following sense: The `DoseUnits` are the physical phenomena that we are attempting to represent. The `Patient` and `Beam Equipment` together constitute the Sensor, which detects increases in `DoseUnits` and sends AckBurst signals to the TCS. The `TCS` is the Machine, which receives `AckBurst` signals and generates `ReqWrite` commands. The `Log` is the Display, responding to `ReqWrite` commands and generating `Entries`. Our requirement is that

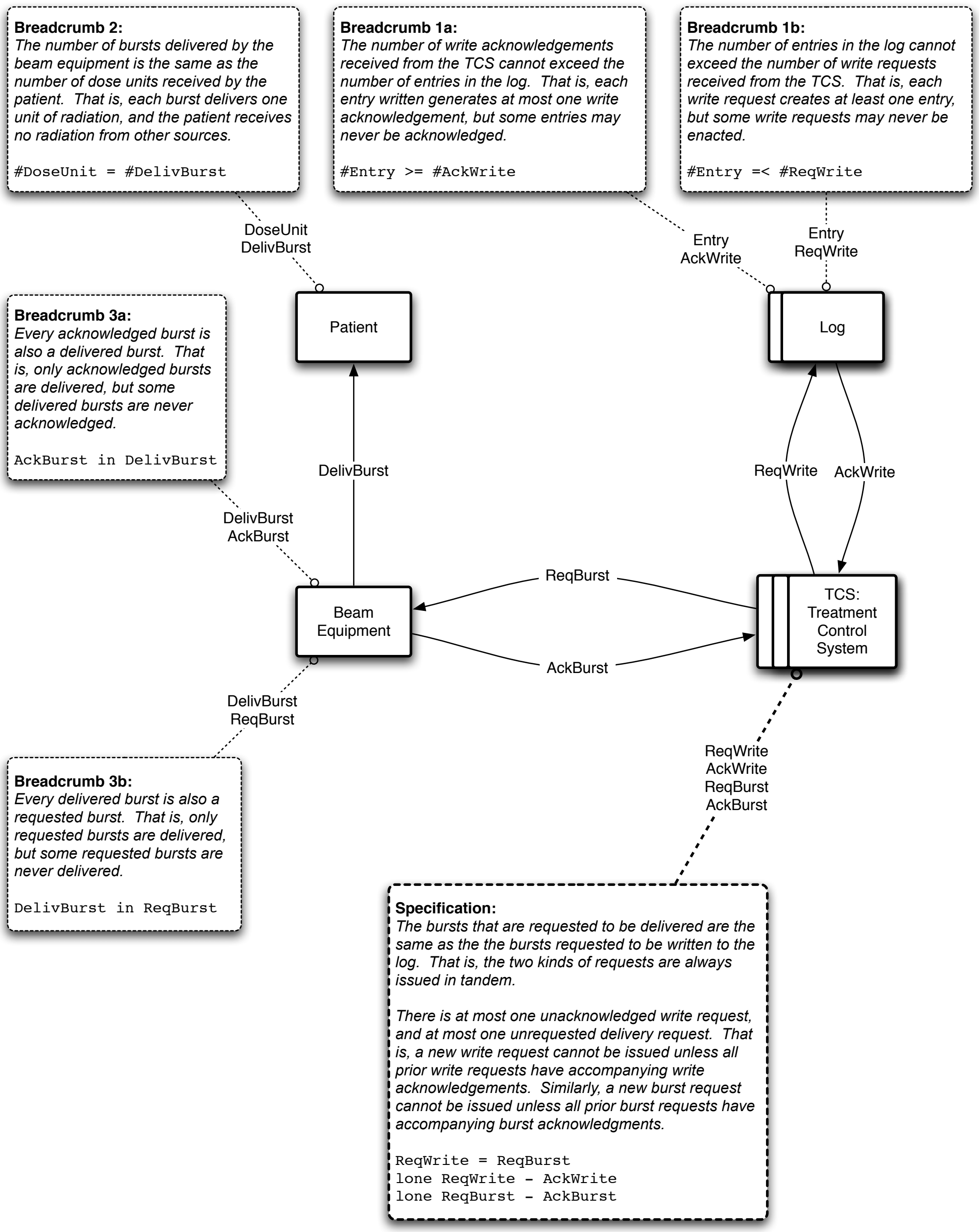

Figure 3-17: The argument diagram that results from transforming the requirement into a specification. Each breadcrumb constraint has a formal description of a partial domain property and an informal interpretation of that formula. The conjunction of the breadcrumb formulae and the specification formula logically imply the requirement formula. The Alloy keyword *lone*, used in the TCS specification, indicates that a set has a cardinality of zero or one.

`Entries` correspond to `DoseUnits`.

The TCS does not just passively watch the patient and react to changes in `DoseUnits` by updating the Log, as suggested by the information display frame. The TCS is also permitted to write a log entry and then deliver a burst of radiation to match it. (Stopping the TCS once the prescribed dose of radiation has been delivered and ensuring that it eventually delivers a sufficient dose is part of the overdose requirement, not the logging requirement.)

The failure to match is also apparent from the diagrams by taking note of the arrow heads on the requirement arcs. A requirement arc with an arrow head indicates that the phenomena labeling that arc are the ones that should change in order to satisfy the requirement. Requirement arcs without arrow heads indicate that those phenomena should not be changed. In the information display frame, only the arc to the Display has an arrow head, indicating that only the Sensor's phenomena will not be changed. In contrast, the logging problem diagram has arrow heads on both the Log and the Patient domains, as both entries and dose units can be changed in order to satisfy the requirement.

Logging also partly matches the *required behavior* frame, shown in Figure 3-4. In a required behavior frame, a Machine issues commands to a Device domain, which in turn exhibits certain behaviors. There is a requirement on what sorts of behaviors should occur. The frame concern focuses our attention on characterizing how the behaviors exhibited by the Device domain depend on the commands issued by the Machine.

The Logging facility is a required behavior problem in the following sense: The `TCS` is the Machine, which issues `ReqWrite` and `ReqBurst` commands. The `Log`, `Beam Equipment`, and `Patient` together constitute the Device domain, whose exhibited behaviors are `DoseUnit` and `Entries`. The requirement on valid behaviors exhibited by the Device domain is that the `DoseUnit`s match the `Entries`.

The TCS also does not control a single Device domain, as suggested by the required behavior frame. The controlled device is really three different domains, one of which (the Log) has no direct connection to the other two (the Beam Equipment and the Patient). Lumping those three domains together into a single Device domain hides the very trait that makes the problem hard – the fact that the Log and Patient cannot directly communicate with one another. It suggests that we could introduce a domain assumption that says "the Device keeps the Entries and DoseUnits the same", missing the key challenge of the Logging problem.

Neither frame alone captures the nature of the pro-active logging problem that we are analyzing. One might argue that the system ought to be designed so that one machine delivers successive doses (required behavior) and a separate machine passively maintains the log (information display). However, with an unreliable log, there needs to be a communication channel between the log and the delivery mechanism, as each needs to react to the acknowledgements of the other. Eliminating that dependence would require changes to the system itself, a luxury not available when the system is already in place, and forcing the system into a mold that fits poorly will only produce a correctness argument that fits equally poorly. Rather, we must approach the system anew.

3.5.5 The Requirement

From the user's perspective, there are two fundamental sets – a set of radiation dose units and a set of log entries.

```
sig DoseUnit { }
sig Entry { }
```

The initial requirement is that the number of dose units delivered to the patient matches the number of entries in the log, with a margin of error of one unit.

```
pred Requirement1 [ ] {
  #Entry = #DoseUnit or
  #Entry = #DoseUnit + 1 or
  #Entry = #DoseUnit − 1
}
```

This requirement is loose enough to permit behaviors in which a burst is both delivered and logged (first line), logged but not delivered (second line), or delivered but not logged (third line). However, in either of the latter two cases, further logging and treatment cannot continue until the imbalance has been corrected.

The essence of the interaction is that various messages are exchanged about bursts delivered by the beam machine (or requested of it). Since each message is about a particular burst, there is no need to introduce a separate notion of a message. Rather, we simply introduce a set of bursts

```
sig Burst { }
```

and a classification into a collection of (possibly overlapping) sets, consisting of bursts that are delivered, requested, and acknowledged, and bursts associated with log entries that are requested and acknowledged.

```
sig DelivBurst, ReqBurst, AckBurst, ReqWrite, AckWrite in Burst { }
```

That is, a burst in the `ReqWrite` set is one for which a write request has been issued. If a write acknowledgement has been issued for that burst, then it will also be in the set `AckWrite`.

Our task is to establish a relationship between `Entries` and `DoseUnit`s, as per the requirement. We will introduce domain assumptions about the `Patient` and `Beam Equipment` to relate `DoseUnit` to `ReqBurst`. Domain assumptions about the `Log` will be added to relate `Entries` to `ReqWrite`. The `TCS` specification will then constrain `ReqBurst` and `ReqWrite` requests, thus indirectly enforcing the original requirement. Figure 3-16 shows the problem diagram before requirement progression begins, and Figure 3-17 shows the same diagram at upon completion.

3.5.6 Transformation and Derivation

We begin with the requirement we want to enforce. The derivation happens in three stages: First, we push the requirement from the `Log` to the `TCS`, and add a breadcrumb and rephrase the requirement as needed to permit that push. Second, we push the requirement from the `Patient` to the `Beam Equipment`, adding another breadcrumb and performing another rephrasing. Finally, we push the requirement from the `Beam Equipment` to the `TCS`, adding a third breadcrumb and performing a third rephrasing. At that point, the requirement only touches (only mentions phenomena involved in) the machine domain, and has thus been transformed into a specification. Figure 3-17 shows the final state of the Problem Frame description, after the transformation process is complete.

Step 1: from Log to TCS

Our first task is to *push* the requirement from the `Log` domain onto the `TCS` domain. We cannot do so because the requirement mentions the `Entry` phenomenon, which is not involved in the `TCS`. We will thus need to *rephrase* the requirement to reference phenomena shared with the `TCS` (`ReqWrite`, `AckWrite`) instead of those known only to the `Log` (`Entries`). However, we first need to introduce a breadcrumb, characterizing the log, to justify such a rephrasing. That breadcrumb needs to relate the phenomena that the requirement constraint currently mentions to those that we would like it to reference. To that end, we add the following breadcrumb representing our domain assumptions about `Log`:

```
pred LogBreadcrumb [ ] {
  #Entry >= #AckWrite
  #Entry =< #ReqWrite
}
```

The first constraint says that the number of entries written is greater than or equal to the number of write acknowledgments; it allows entries to be written without corresponding acknowledgments. The second constraint says that the number of entries written is less than or equal to the number of write requests; it allows write requests to be ignored. With this assumption in hand, we rephrase the requirement as follows:

```
pred Requirement2 [ ] {
  lone ReqWrite - AckWrite and
  (#ReqWrite = #DoseUnit or #ReqWrite = #DoseUnit + 1)
}
```

The Alloy keyword `lone` indicates that the following expression has a cardinality of zero or one. Thus, the formula `lone ReqWrite - AckWrite` means that there can be at most one write request for which there is no write acknowledgement.

To confirm that the new breadcrumb and the new requirement together imply the prior requirement (the original requirement), this is presented to the Alloy Analyzer as an assertion to be checked:

```
assert Step1 {
  LogBreadcrumb and Requirement2 ⇒ Requirement1
}
check Step1 for 10
```

Now that the requirement only mentions phenomena from the recipient domain, it can be pushed from `Log` to `TCS`.

Step 2: from Patient to Equipment

We repeat the process to push the requirement from `Patient` to `Beam Equipment` by characterizing the `Patient` domain. First, we add the following breadcrumb:

```
pred PatientBreadcrumb [ ] {
  #DoseUnit = #DelivBurst
}
```

which is motivated by the fact that each `DelivBurst` event delivers exactly one `DoseUnit` to the patient, and that the patient receives no `DoseUnit`s of radiation from other sources. The breadcrumb permits the requirement to be rephrased as follows:

```
pred Requirement3 [ ] {
  lone ReqWrite − AckWrite and
  (#ReqWrite = #DelivBurst or #ReqWrite = #DelivBurst + 1)
}
```

To confirm that the new breadcrumb and the new requirement together imply the prior requirement, we present the Alloy Analyzer with the following assertion to check:

```
assert Step2 {
  PatientBreadcrumb and Requirement3 ⇒ Requirement2
}
check Step2 for 10
```

We can now push the requirement from `Patient` to `Beam Equipment`.

Step 3: from Equipment to TCS

We repeat the process a third time to push the requirement from `Beam Equipment` to `TCS`. First add the following breadcrumb:

```
pred EquipBreadcrumb [ ] {
  AckBurst in DelivBurst
  DelivBurst in ReqBurst
}
```

which says that an acknowledgement must be sent only when a burst is delivered, and that a burst may only be delivered when it is requested. Limited unreliability is permitted; some requests have no matching delivery and some deliveries have no matching acknowledgement. The requirement can now be rephrased as follows:

```
pred Requirement4 [ ] {
  ReqWrite = ReqBurst
  lone ReqWrite − AckWrite
  lone ReqBurst − AckBurst
}
```

The first line of the derived specification says that a write must be requested of the log whenever the beam equipment is requested to deliver a burst and vice versa. The second line says that no new write requests can be made if any write request remains unacknowledged. The third says that no new burst request can be made if any burst request remains unacknowledged. The machine must wait for both acknowledgements before issuing another pair of requests.

We present the Alloy Analyzer with the following assertion to check that the final rephrasing was justified by the following breadcrumb:

```
assert Step3 {
  EquipBreadcrumb and Requirement4 ⇒ Requirement3
}
check Step3 for 10
```

Finally, we push the requirement from `Beam Equipment` to `TCS`. At this point, the requirement mentions only phenomena from `TCS` and has become a specification. If the TCS issues requests according to this specification, and the other three domains satisfy their domain assumptions, then the original requirement will be preserved. The problem diagram resulting from the entire is shown in Figure 3-17.

3.6 Handling Time: Automatic Door Controller

In this section, we demonstrate requirement progression on a system with highly temporal aspects – an automatic door, as one might find in a supermarket. Here is the *Door Controller* problem, as defined by Nick Ourusoff [64]:

> We wish to specify a software system to control an automatic door. The automatic door contains a motor, which may either be ON or OFF and has a polarity, which is either OPEN (indicating that the door will move to the OPEN position if the motor is ON) or CLOSE (indicating that the door will move to the CLOSED position if the motor is ON). The door also contains two sensors: one registers OPEN, when the door is within 3 cm. of being fully open; the other registers CLOSED, when the door is within 3 cm. of being fully closed. In addition, there is a motion sensor. It sends a signal to the controller if the sensor detects motion 6 feet away from door. It isn't important how it works.
>
> We wish to write a Door Controller to open the door whenever a person wishes to walk though it; and to keep the door closed when someone isn't passing through it.

3.6.1 Designations and Context

First, we build a context diagram describing the automatic door situation (Figure 3-20) and an accompanying set of designations for the domains and phenomena (Figures 3-19 and 3-18).

Time	$\Leftrightarrow$	A moment in time, and the associated state of the world at that time. Measured in seconds.
DistanceToDoor	$\Leftrightarrow$	The distance (in feet) between the door and the person closest to the door at a given point in time. If there is no person, the distance is considered to be infinite.
DistanceToSensor	$\Leftrightarrow$	The distance (in feet) between the motion sensor and the person closest to the motion sensor (at a given point in time). If there is no person, the distance is considered to be infinite.
DoorGap	$\Leftrightarrow$	The percentage the sliding door is open at a given point in time.
MotionDetected	$\Leftrightarrow$	The presence of a signal generated by the Motion Sensor indicating that it has detected nearby motion. This variable has the value "Motion" when the sensor is sending a signal indicating motion and the value "NoMotion" when no signal is being sent.
MotorPolarity	$\Leftrightarrow$	The direction in which the motor has been instructed to run. It either has the value "Opening" or "Closing".
MotorPower	$\Leftrightarrow$	Whether or not the motor has been instructed It either has the value "MotorOn" or "MotorOff".
DoorGapMeasure	$\Leftrightarrow$	A value reported by the Position Sensor. A value of "AlmostOpen" means that the door is 90 percent open or more. A value of "almostClosed" means that the door is 10 percent open or less. In all other cases, the sensor reports a value of "UnknownGap".
AppliedForce	$\Leftrightarrow$	The force currently being applied to the doors, directly causing them to open or close. It either has the value "OpeningForce" or "ClosingForce".
MotorSpeed	$\Leftrightarrow$	The percentage that the motor can open/close the sliding doors in 1 second, measures in increments of 10 percent per second.

Figure 3-18: Designations for an automatic door controller.

`Door`	⇔	A pair of sliding doors that the automatic door system is in place to control.
`Position Sensor`	⇔	A sensor on the doors that reports on the status of the door.
`Motor`	⇔	A machine that applies force the door.
`People`	⇔	Humans and other moving objects in the vicinity of the door.
`Motion Sensor`	⇔	A sensor that detects and reports on nearby motion.
`Controller`	⇔	The component we are designing to coordinate the system.

Figure 3-19: Domains for an automatic door controller.

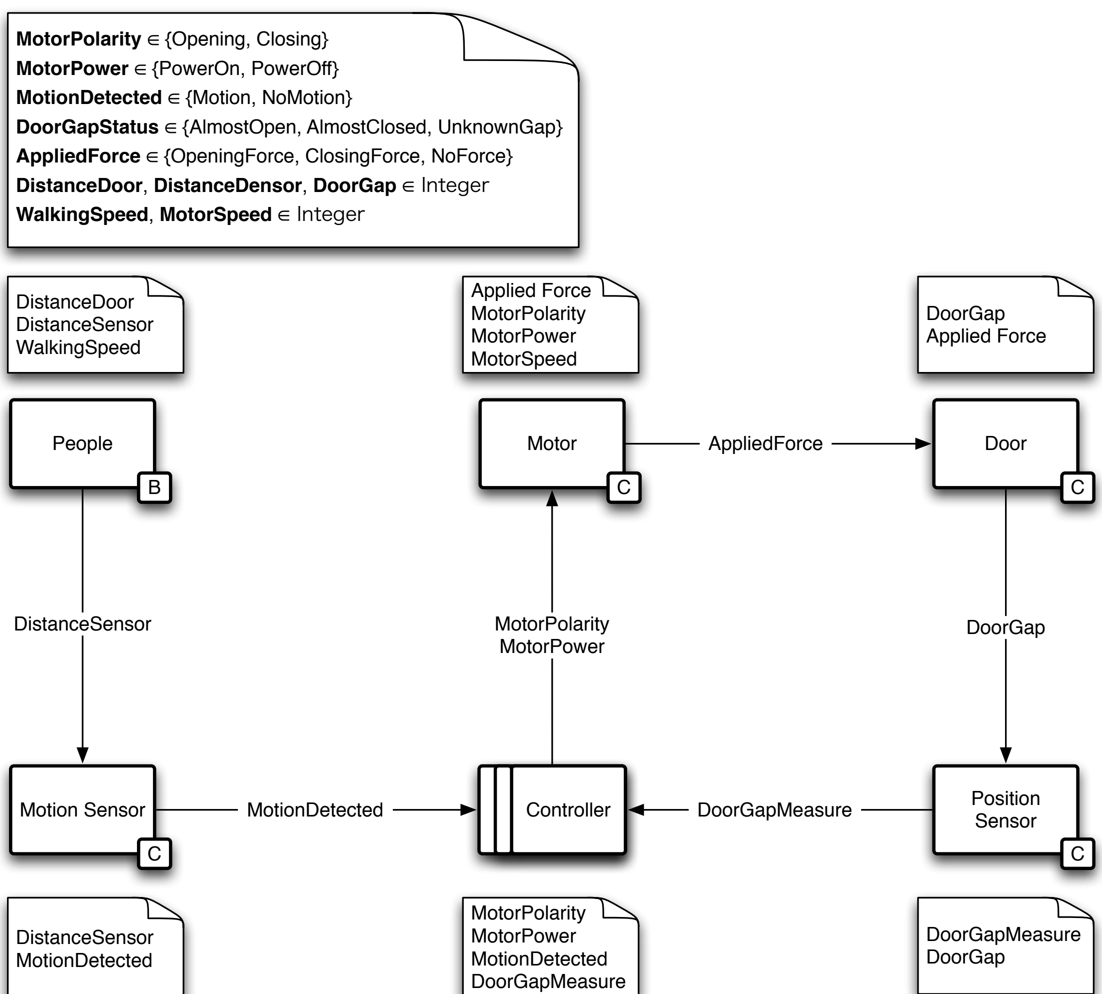

Figure 3-20: Context diagram for an automatic door.

3.6.2 Formalizing the Requirement(s)

As is often the case when writing requirements, a number of subtleties arise in the course of simply writing down what it means for the system to operate properly. There are actually 3 requirements here, which should be addressed independently. Informally, they are as follows:

(1) **Service Provided** The door must be open when a person is close enough to walk through, or at least open far enough to allow someone through. The door remains closed the rest of the time.

(2) **No Motor Damage** The motor does not try to close the door when it is fully closed or open the door when it is fully open.

(3) **No Door Damage** The door must not be forced open when already open, or forced closed when already closed.[9]

Formally, we interpret and encode those requirements in the Alloy language as follows:

(1) **Service Provided** Whenever someone is within 1 foot of the door, the door must be at least 90% open. If nobody is within 11 feet of the door, then the door must be at most 10% open. Otherwise, the door can be any amount open.

```
all t: Time | (DistanceDoor[t] =< 1) ⇒ (DoorGap[t] >= 9)
all t: Time | (DistanceDoor[t] >= 11) ⇒ (DoorGap[t] =< 1)
```

Filling in specific values is necessary to build a working model, but those details must be confirmed or provided by a domain expert. This is a case where the act of formalization revealed an important omission in the design requirement. It turns out that the details of what it means for the door to be mostly open/closed and what it means for a person to be near/far are not details that can be left for later; they are relevant to high level design. In the absence of an expert, we have provided plausible placeholder values.

(2) **No Motor Damage** The Motor is never on and opening when the door is completely open. The Motor is never on and closing when the door is completely closed.

[9] Note that the second and third requirements are subtly different. Door damage and motor damage are only the same if the motor gets damaged under exactly the same circumstances that the door is damaged. That assumed both (a) that force is only applied to the door as a result of the motor running and (b) that the motor only serves to open/close the door. While those properties may be true, they are not inherent in the problem context, and would need to be introduced as domain assumptions.

```
no t : Time |
  MotorPower[t] = MotorOn
  and MotorPolarity[t] = Opening
  and DoorGap[t] >= 10
no t : Time |
  MotorPower[t] = MotorOn
  and MotorPolarity[t] = Closing
  and DoorGap[t] =< 0
```

(3) **No Door Damage** The door is never forced open when it is 100% open or forced closed when it is 0% open.

```
no t : Time |
  AppliedForce[t] = OpeningForce
  and DoorGap[t] >= 10
no t : Time |
  AppliedForce[t] = ClosingForce
  and DoorGap[t] =< 0
```

These formal requirements, when added to the context diagram, form the problem diagram shown in Figure 3-21.

The **No Door Damage** requirement only mentions phenomena from the `Door` domain. According to requirement progression, there is nothing to be done here, as the requirement is already a domain assumption.

The **No Motor Damage** and **Service Provided** requirements each reference phenomena from multiple domains. The service requirement refences phenomena from `People` and `Door`, and the motor requirement references phenomena from `Motor` and `Door`. We then use requirement progression on both requirements to decompose them into localized domain assumptions. We have omitted the intervening steps for brevity. The resulting argument diagram is given in Figure 3-22.

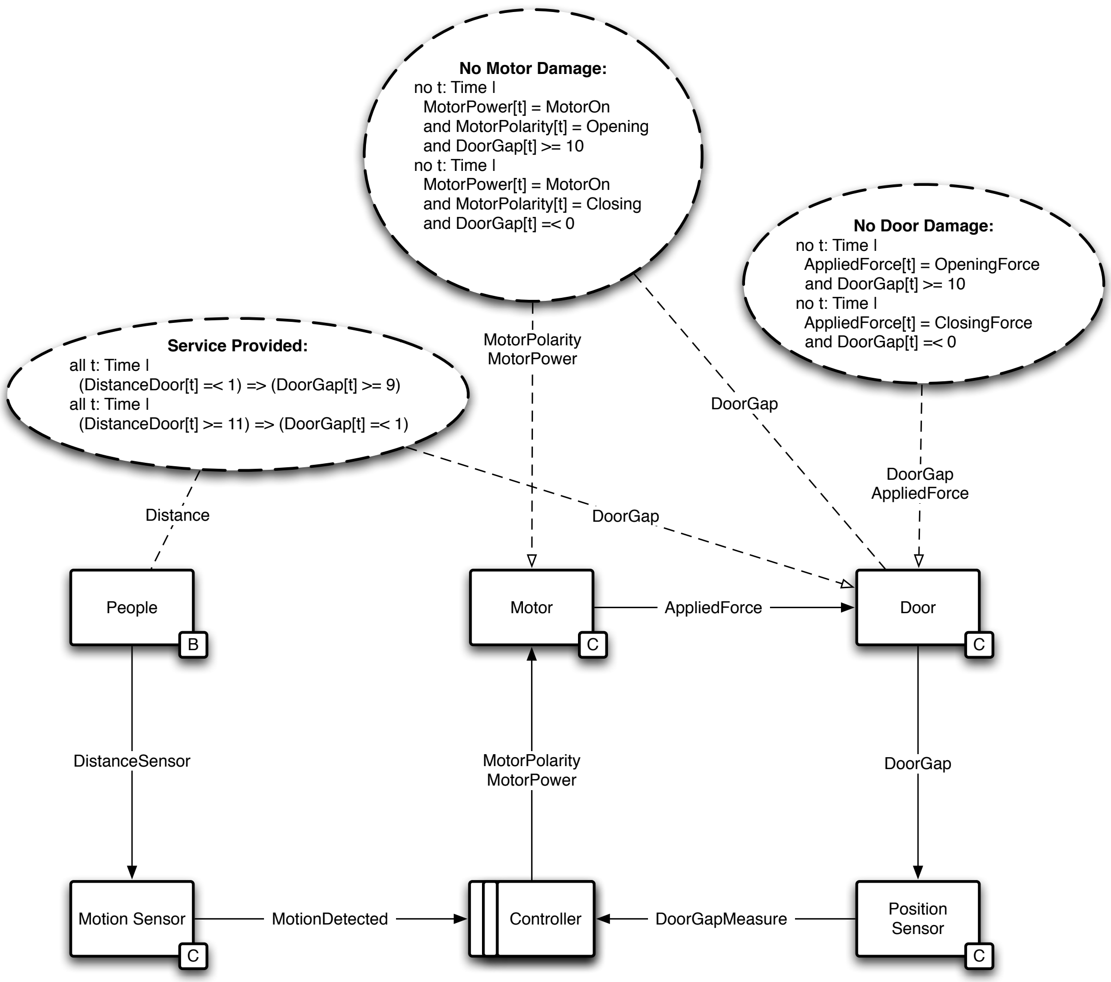

Figure 3-21: Problem diagram for an automatic door.

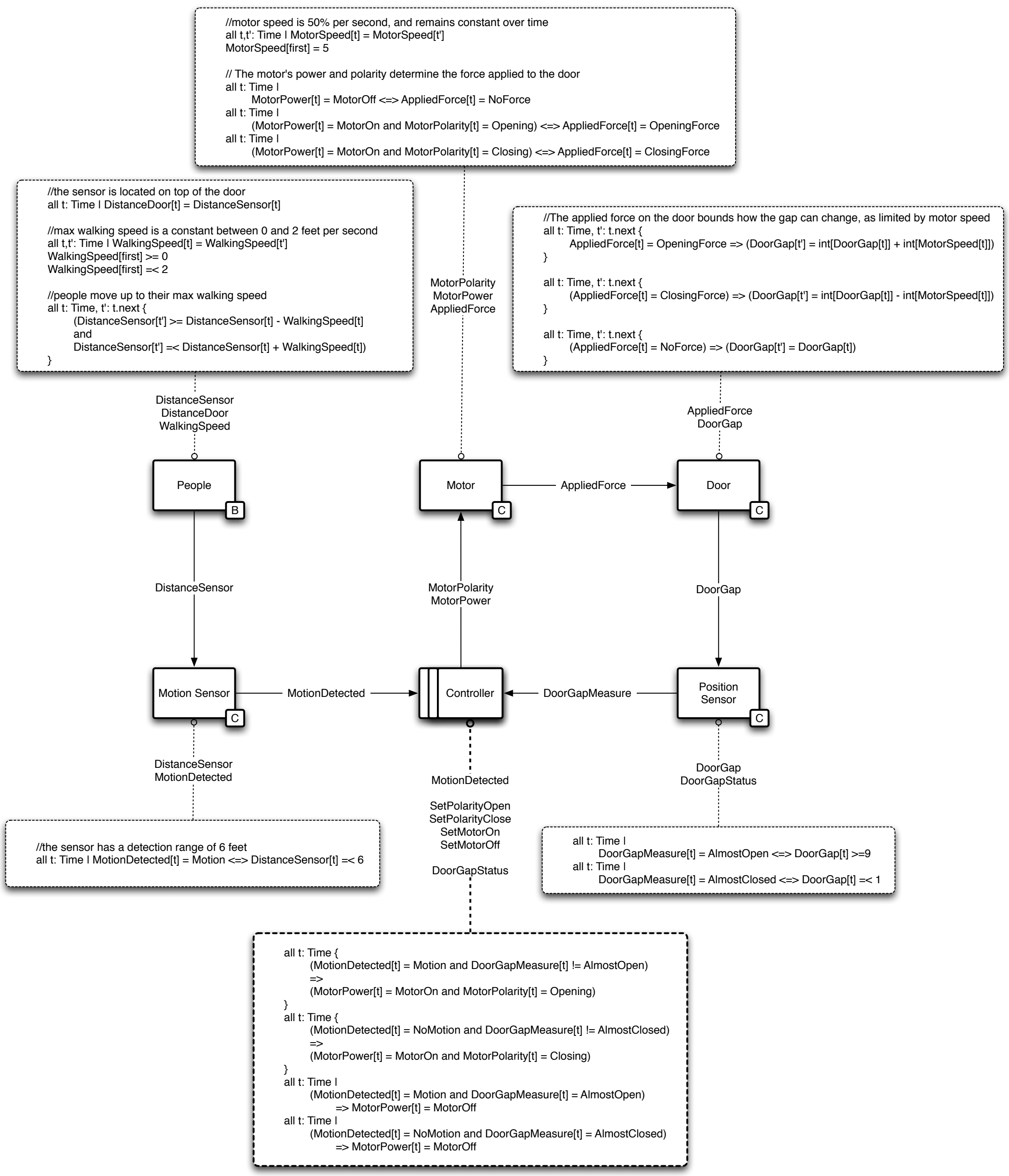

Figure 3-22: The argument diagram generated by applying requirement progression to service and safety requirements for an automatic door controller.

3.6.3 Lessons Learnt

Simply writing down the problem in the PF framework and using a formal language (in this case, Alloy) revealed a number of subtleties. We later worked through the progression and checked it with the Alloy Analyzer, and discovered another set of complexities. Only the analysis and consistency checking of an automatic tool revealed the full depth of the problem, and provided us with a verifiable argument for correctness. This experience is what Daniel Jackson refers to as the humbling nature of analysis [35].

Designation Documentation

When writing down all of the designations (shown in Figure 3-18), we discovered that there were really 2 distances that had been conflated in our original conception: the distance from a person to the door and the distance from a person to the sensor. The system requirement is that the door opens and closes according to the distance to the door, however, our feedback system (the motion sensor) only reports on distance to the sensor. There is an implicit assumption (which we made explicit) that the two distances are the same (e.g. that the sensor is located on top of the door). Without this explicit assumption, one could place the sensor far from the door (e.g. on the ceiling above it) – doing so would satisfy the domain assumptions but violate the service requirement.

Building the Problem Diagram

Building the problem diagram, showing which phenomena the requirement referenced, revealed the fact that there were really 3 separate requirements, each with a different set of phenomena. By separating the requirements, each one became much more manageable, and the resulting argument was more structured.

Performing Requirement Progression

Our initial attempt at formalizing the requirement was not enforcible by a motor that was not infinitely fast. We discovered this during attempted progression, when we were unable to push the requirement across certain arcs without adding unacceptable domain assumptions.

Initially, we allowed time for the motor to open the door while a person walked towards it – the sensor detects them at distance 6 but isn't required to have opened until distance 1. However, we did not allow time for the motor to close the door as they walked away – the sensor detects their absence when they are at distance 6, but we required the door to be closed when distance was 6. We needed to loosen the requirement to only have the door closed when the closest person is 11 or more distance units away, thus providing the same margin for response.

Automatic Alloy Analysis

The initial (unchecked) controller specification was too weak. It allowed (but did not require) that the door oscillate open and closed when nobody was nearby. This error escaped simulation (since the correct behavior was permitted and often occurred) but was caught by the automatic check. The incorrect version looked like this:

```
pred ControllerBC [] {
  all t: Time {
    (MotionDetected[t] = Motion and DoorGapMeasure[t] != AlmostOpen)
    ⇒
    (MotorPower[t] = MotorOn and MotorPolarity[t] = Opening)
    }
  all t: Time {
    (MotionDetected[t] = NoMotion and DoorGapMeasure[t] != AlmostClosed)
    ⇒
    (MotorPower[t] = MotorOn and MotorPolarity[t] = Closing)
    }
  all t: Time {
    DoorGapMeasure[t] = AlmostClosed ⇒
      !(MotorPower[t] = MotorOn and MotorPolarity[t] = Closing)
    }
  all t: Time {
    DoorGapMeasure[t] = AlmostOpen ⇒
      !(MotorPower[t] = MotorOn and MotorPolarity[t] = Opening)
    }
}
```

The corrected specification is more tightly constrained and looks like this:

```
pred ControllerBC [] {
  all t: Time {
    (MotionDetected[t] = Motion and DoorGapMeasure[t] != AlmostOpen)
    ⇒
    (MotorPower[t] = MotorOn and MotorPolarity[t] = Opening)
  }
  all t: Time {
    (MotionDetected[t] = NoMotion and DoorGapMeasure[t] != AlmostClosed)
    ⇒
    (MotorPower[t] = MotorOn and MotorPolarity[t] = Closing)
  }
  all t: Time |
    (MotionDetected[t] = Motion and DoorGapMeasure[t] = AlmostOpen)
      ⇒ MotorPower[t] = MotorOff
  all t: Time |
    (MotionDetected[t] = NoMotion and DoorGapMeasure[t] = AlmostClosed)
      ⇒ MotorPower[t] = MotorOff
}
```

The difference lies in the second two constraints, which more tightly constrain the acceptable behaviors when the door is already in the desired position. The incorrect version allows the door to do anything when it is in the correct position, and only requires it to correct itself once in the wrong position. It can thus close when it is open (and should be) as long as, in the next time step, it opens again.

The initial (unchecked) formalization of the problem contained an inconsistent representation of motor speed. When we had to make the model pass automatic tests, we discovered that they were nonsensical and had to be reworked.

The initial (unchecked) formalization of the motor breadcrumb had a conditional where it needed a bi-conditional[10]. The weaker version permitted the system to violate the service requirement while satisfying the domain assumptions. The intent of the original phrasing was right, but the actual written constraint was incorrect. The modeler had 5 years of experience with logical modeling (and Alloy in particular), but still needed the automatic analysis to get the constraints right.

[10]A conditional is of the form "A holds if B hold" or "A hold only if B hold". A bi-conditional is the stronger statement "A hold if and only if B holds"

During Assumption Confirmation

Suppose that the domain expert on doors tells us that the Door Damage requirement is not enforceable, since the door domain cannot control whether or not force is applied to the door at the wrong time. However, no other domain involves both of the phenomena referenced by the requirement (AppliedForce and DoorGap). As things stand, it is impossible to generate local domain assumptions to enforce the requirement. In terms of the problem diagram, this difficulty corresponds to the fact that the AppliedForce phenomenon is mentioned, but not controlled, by the Door domain.

This reveals the need for an additional shared phenomenon between the Motor and Door domains. The phenomenon would represent feedback from the door to the Motor, and would be controllable by the Door. This way, the Door Damage requirement can be decomposed into two modular assumptions; that the feedback is generated correctly (assumption about the Door) and that the feedback is reacted to accordingly (assumption about the Motor).

Constant Phenomena

In the case where `WalkingSpeed` and `MotorSpeed` are known constants, the context diagram shown in Figure 3-20 is an accurate description of the problem context. The assumptions about their (constant) values appear as domain assumptions on their controlling domains (`People` and `Motor`), as is done in the Alloy model given in Appendix 8. However, suppose we want to replace our constant-valued assumptions with a relative-value assumption, such as the following:

```
MotorSpeed >= WalkingSpeed
```

Doing this might be desirable to permit the same argument to apply to automatic door controllers used in different contexts. However, we cannot simply add this new assumption as a breadcrumb, since it is non-local – it references phenomena from both the `Motor` and `People` domains and is thus not a valid breadcrumb.

When we attempt to use progression to decompose that assumption into breadcrumbs, we hit a blockade; there is not an appropriate information channel between the `Motor` and `People` domains to synchronize `MotorSpeed` and `WalkingSpeed`. Furthermore, it is clearly the case that people and motors do not directly share either of those phenomena between them – people walking through an automatic door do not communicate with the motor controlling that door! Thus we cannot simply add another arc to the context diagram linking those domains. What we *can* add to solve the problem is a calibration domain, as shown in Figure 3-23.

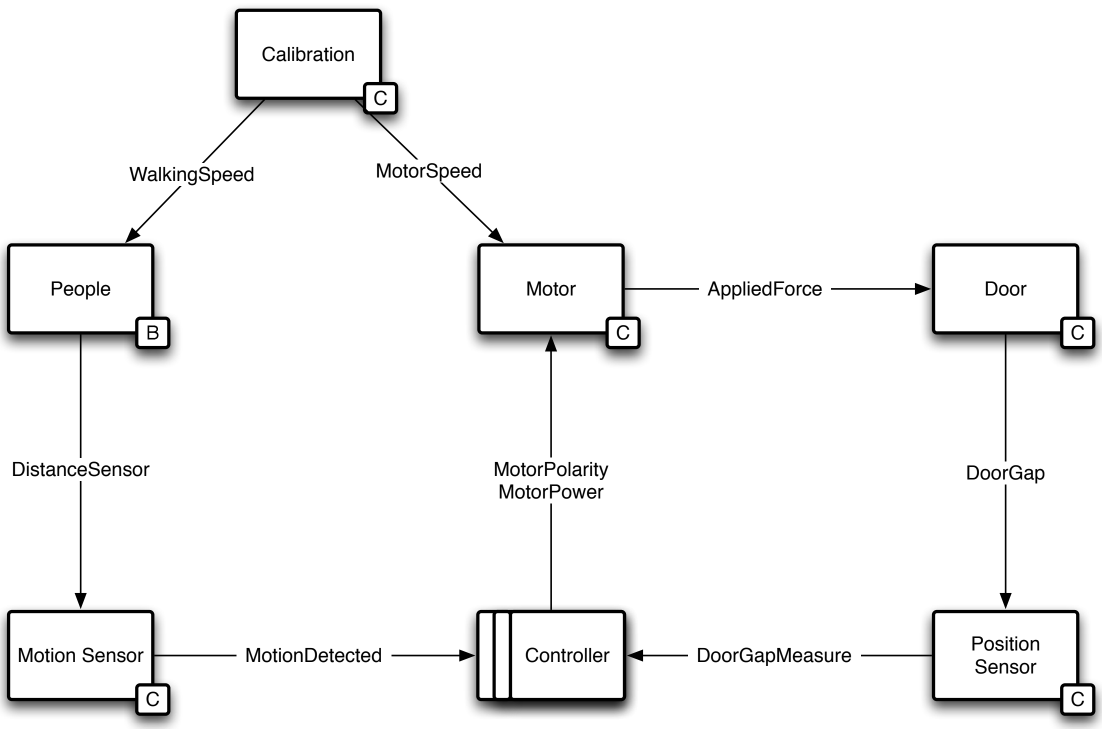

Figure 3-23: Context diagram for an automatic door with a Calibration domain.

Such a domain represents the fact that, in order to enforce the relation between `WalkingSpeed` and `MotorSpeed`, someone has has to observe the walking speeds of people and provide adequate speed to the motor. This is a domain that can support our desired breadcrumb, and thus provides an avenue for progression – we would push the requirement from `People` to `Calibration` and then on to the `Motor`, leaving the breadcrumb behind on the calibration domain. Of course, adding a domain to the problem diagram is not something that can be done merely to ease progression; it must be confirmed with the system experts and is likely to require a significantly different implementation.

3.7 Encoding Problem Diagrams in Alloy

In this section, we describe an Alloy model of problem diagrams, and define what it means for a problem diagram to be well formed. In Section 3.8, we extend the model to describe our method for requirement progression (adding breadcrumbs, rephrasing goals, and pushing goals). Key parts of the model are introduced in these sections, and the entire model (including all referenced predicates) is shown as a single unit in the Appendix. [11]

3.7.1 Sets and Relations

The key sets and relations that define a problem diagram are shown in object model notation [75] in Figure 3-24. Each constraint mentions a set of phenomena and touches a set of domains. Each domain involves a set of phenomena and connects to a set of domains. There is a special machine domain and two special kinds of constraints, specifications and requirements.

To express the anatomy of a problem diagram in Alloy, we start by defining three sets: the set of phenomena, the set of domain, and the set of constraints. These are the building blocks of problem diagrams.

```
sig Phenomenon, Domain, Constraint { }
```

Next we define set `Diagram`, each element of which represents a complete problem diagram.

```
sig Diagram {
  phenomena: set Phenomenon,
  domains, machines: set Domain,
```

[11] We use Alloy to formalize problem diagrams and the effect of our transformations on them (Sections 3.7 and 3.8) and also to express the constraints in particular examples (Sections 3.4 and 3.5). We use the same language only to reduce the number of logics that the reader must keep track of, not to suggest a connection between the two uses. The two kinds of models are not currently put together, and need not be written in the same language.

```
    constraints, requirements, specifications: set Constraint,
    connects: Domain → Domain,
    involves: Domain → Phenomenon,
    touches: Constraint → Domain,
    mentions: Constraint → Phenomenon
}
```

A problem diagram comprises a set of `domains`, a set of `phenomena`, and a set of `constraints`. There is a special kind of domain called a machine, and two special kinds of constraints, called requirements and specifications. The first three lines encode these as relations. For example, if `x` is a `Diagram`, then the expression `x.domains` denotes a set of `Domain`s.

Problem diagrams structure their domains, phenomena, and constraints. Each domain in a diagram involves a set of phenomena and connects to a set of other domains. Each constraint in a diagram mentions a set of phenomena and touches a set of domains. The last four lines encode these as relations. For example, if `x` is a `Diagram`, then the expression `x.mentions` denotes a binary relation that maps `Constraint`s to `Phenomena`. More generally, we can get the set of phenomena mentioned by a constraint `c` in a diagram `x` by writing `c.(x.mentions)` or by the equivalent expression `x.mentions[c]`.

3.7.2 Well Formedness

Not any collection of domains, phenomena, and constraints constitute a meaningful description. If the predicate `wellFormedDiagram` holds on a diagram, then we know that the diagram has a meaningful structure. Later, we will use this predicate to check whether or not certain transformations preserve well formedness.

```
pred wellFormedDiagram [x: Diagram] {
    selfContained[x]
    one x.machines
    connectIffShare[x]
    nonTrivial[x]
    all c: x.constraints | wellFormedConstraint[c,x]
}
```

A well formed diagram satisfies five properties.

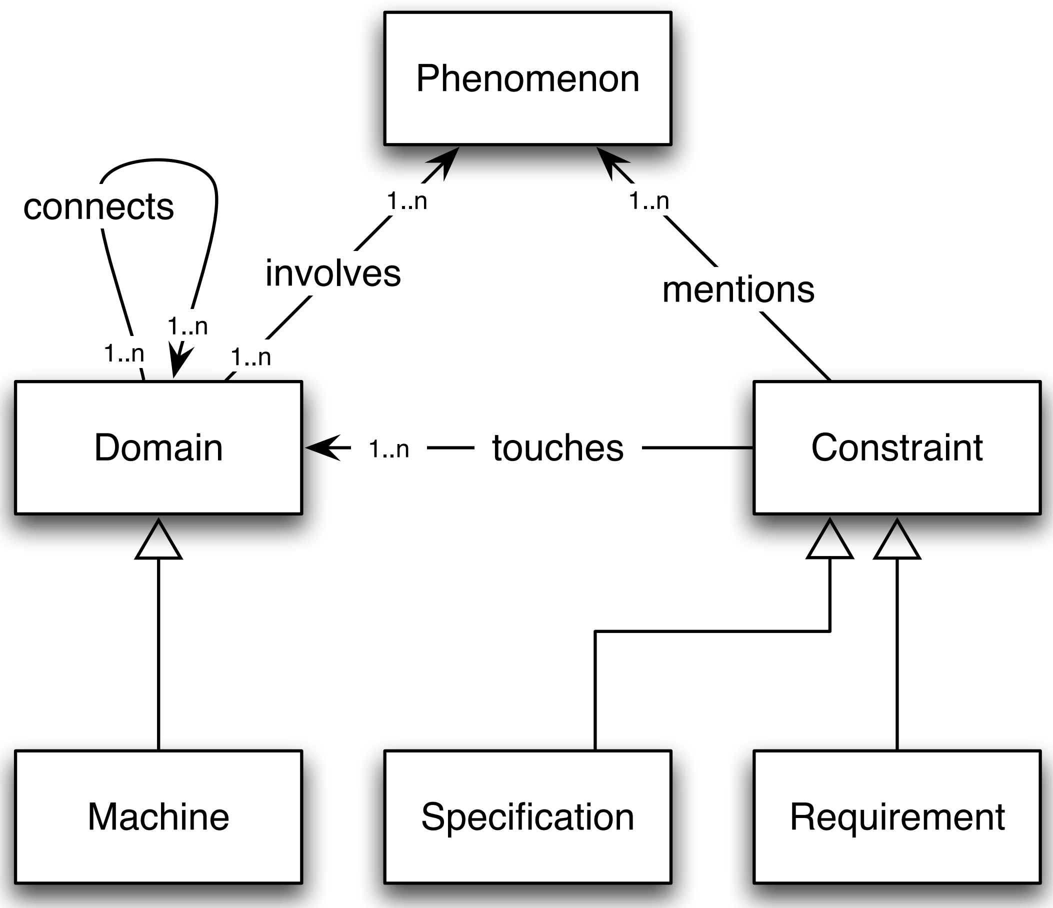

Figure 3-24: A metamodel of problem diagrams, expressed using standard object model notation.

(1) Diagrams must be self contained. For example, the domains in a diagram cannot connect to domains in a different diagram. Full definitions of all predicates can be found in the Appendix.

(2) There must be exactly one machine.

(3) Every domain must be reachable from every other domain by following the `connects` relation zero or more times.

(4) Trivial diagrams are forbidden, such as disconnected diagrams or domains that contain no phenomena. Non-triviality is not technically a requirement of a problem diagram, but we include it for the sake of not having to worry about uninteresting corner cases.

(5) Every constraint must be well formed.

```
pred wellFormedConstraint [c: Constraint , x: Diagram] {
  c in x.constraints
  all p: x.mentions[c] | some d: x.touches[c] | p in x.involves [d]
  all d: x.touches[c] | some ( x.involves[d] & x.mentions[c])
  c in x.specifications ⇔ x.touches[c] in x.machines
  x.touches[c] in x.domains
  x.mentions[c] in x.phenomena
}
```

A well formed constraint satisfies four properties.

(1) Any phenomenon mentioned by the constraint must be involved in at least one of the domains touched by the constraint. That is, every phenomenon used in a constraint must come from somewhere.

(2) Any domain touched by the constraint must involve at least one phenomenon mentioned by the constraint. That is, a constraint cannot touch a domain for no reason.

(3) A constraint is a specification if it touches only the machine.

(4) A constraint must be completely contained within the diagram. For example, it cannot touch domains that are not in its own diagram or mention phenomena that are not it its own diagram.

The Alloy Analyzer can automatically generate sample solutions to the above constraints by executing a `run` command:

```
run wellFormedDiagram for 4
```

The "for 4" specifies a *scope* for the execution. It tells the Alloy Analyzer to only consider solutions in which each signature has 4 or fewer elements. That is, we will only generate solutions with up to 4 diagrams, up to 4 domains, up to 4 phenomena, and up to 4 constraints.

3.8 Encoding Requirement Progression in Alloy

Now that we have laid the groundwork with a description of well formed problem diagrams, we will formalize what it means to perform requirement progression on such diagrams. We do so by extending our previous model to include descriptions of add, rephrase, and push operations.

Since we will be talking about sequences of problem diagrams, we use one of Alloy's library modules to impose a total ordering on Diagrams. We can write `first[]` to denote the first `Diagram` in the ordering and `next[x]` to denote the next `Diagram` after a `Diagram x`.

```
open util/ordering[Diagram] as ord
```

3.8.1 Requirement Progression Invariant

In requirement progression, only constraints change; the underlying structure of the domains and phenomena remains constant. We express this invariant as a predicate.

```
pred structureEquivalent [x,y: Diagram] {
  x.domains = y.domains
  x.machines = y.machines
  x.phenomena = y.phenomena
  x.connects = y.connects
  x.involves = y.involves
}
```

Two diagrams are structurally equivalent if and only if their domains, machines, and phenomena are the same, as well as the connections between domains and the phenomena involved in each domain. No restriction is placed on constraints, requirements, or specification, nor on the touches and mentions relations.

3.8.2 The Transformations

The addition of a breadcrumb to a diagram is modeled as a predicate. The only change to the diagram is the addition of a single constraint. That constraint touches a single domain, is well formed, but is neither a requirement nor a specification. The domain structure remains the same, as do all other constraints.

```
pred addBreadcrumb [before, after: Diagram] {
  structureEquivalent[before, after]
  some bc: Constraint {
    addConstraint[bc, before, after]
    one after.touches[bc]
    wellFormedConstraint[bc, after]
    bc !in after.requirements + after.specifications
  }
}
```

The rephrasing of a requirement is modeled as another predicate. The only change to the diagram is the replacement of one requirement (`r`) by another (`r'`). The new requirement must be well formed, mention at least one different phenomenon than the only one, and touch the same phenomena. The constraints in the final diagram (comprising the new requirement and the old non-requirement constraints) must logically imply the old requirement.

```
pred rephraseRequirement [before, after: Diagram] {
  structureEquivalent[before, after]
  some r: before.requirements, rafter.requirements {
    wellFormedConstraint[r', after]
    replace[r,r',before,after]
    onlyChanges[r, r.touches[r', before, after]
    before.mentions[r] != after.mentions[r']
    before.touches[r] = after.touches[r']
    implication[after.constraints, r, after]
  }
```

A third predicate defines a requirement push. The only change to the diagram is that one requirement changes what it touches but remains well formed.

```
pred pushRequirement [before, after: Diagram] {
  structureEquivalent[before, after]
  onlyTouchesChanges[before, after]
  some r: before.requirements & after.requirements {
    before.touches[r] != after.touches[r]
    before.touches − (r → univ) = after.touches − (r → univ)
    wellFormedConstraint[r, after]
  }
}
```

3.8.3 Well Formedness Preservation

With formal descriptions of the transformations in hand, we can check our belief that these transformations preserve well formedness. We write an assertion that, if any of the three operations is performed on a well formed diagram, the resulting diagram will also be well formed.

```
pred someTransformation [x,y: Diagram] {
  addBreadcrumb[x,y] or
  rephraseRequirement[x,y] or
  pushRequirement[x,y] or
  commonTransformation[x,y]
}

assert wellFormednessPreservation {
  all x,y: Diagram |
    wellFormedDiagram[x] and someTransformation[x,y]
      ⇒ wellFormedDiagram[y]
}
check wellFormednessPreservation for 4
```

The check passes for a scope of 4, so we know that the transformations preserve the well formedness invariant for all small problem diagrams.

3.9 Discussion

3.9.1 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, assumptions about a single domain and its interface. 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.

3.9.2 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.

3.9.3 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.

3.9.4 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 3-9. The requirement states that cars will not collide – an assumption which connects only to the Cars domain. According to our progression 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 3-10). 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.

3.9.5 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 3.4.5), 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 external environment.

3.9.6 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.

3.9.7 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.

3.9.8 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 [83, 58, 57]. That work does not integrate as well with relational code analysis tools (like Forge [23]), 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 The Burr Proton Therapy Center

The Burr Proton Therapy Center (BPTC) is a radiation therapy facility associated with the Massachusetts General Hospital (MGH). In this section, we demonstrate our techniques on a key concern of the BPTC radiation delivery stystem. Background of the BPTC and our collaboration with it are given in Appendix 9.

```
Exposure of Humans to Life Threatening Conditions
   General Exposure
      Physical Danger from Moving/Mechanical Parts
         swinging parts (dangling control pad) --------------------- (Low)
         pinching parts (nozzle joints) ---------------------------- (Medium)
         crushing force (gantry rotation) -------------------------- (High)
      Uncontrolled Electrical Current ------------------------------ (High)
   Patient Exposure
      Radiation Overdose
         major overdose (relative to absolute limit) --------------- (High)
         minor overdose (relative to prescription) ----------------- (Medium)
         locational overdose (wrong focus or shape) ---------------- (High)
         collateral overdose (surrounding organs) ------------------ (Medium)
      Patient Not Fully Treated
         aware of a certain underdose ------------------------------ (Low)
         aware of an uncertain underdose --------------------------- (Medium)
         unaware of an underdose ----------------------------------- (High)
         treatment visit cancelled --------------------------------- (Low)
         many treatment visit cancellations ------------------------ (Medium)
         all treatment visits cancelled ---------------------------- (High)
   Non-Patient Exposure (staff, maintenance crew, neighbors)
      Radiation Exposure
         major one-time dose --------------------------------------- (High)
         accumulation of minor doses: adult ------------------------ (Low)
         accumulation of minor doses: child/fetus ------------------ (High)

Exposure of Critical Machinery to Destructive Conditions
   Camera and Sensor Wearout / Accumulated Radiation --------------- (Low)
   Physical Mechanisms (e.g. gantry, nozzle arm)
      Stress and Wear ---------------------------------------------- (Low)
      Broken or Inoperable ----------------------------------------- (Medium)
   Permanent Damage to Cyclotron ----------------------------------- (High)
```

Figure 4-1: High level hazards for the BPTC radiation therapy system. The hazards are organized according to what assets they endanger and the sources of that endangerment. Each hazard is assigned a severity rating of High, Medium, or Low.

4.2 BPTC Hazard Analysis

Before we can analyze the sytem, we must examine the system's context. We will first perform a high level hazard analysis of the BPTC system, and then focus on one of those hazards and apply our technique to it.

A *Hazard* is the exposure of an asset to a dangerous situation, although not necessarily the realization of the dangers of that situation. Hazards are not the ways that the particular system may fail, but rather ways that *any* system filling the role might put valuable assets in danger. Figure 4-1 gives a high level hazard analysis for the Burr Proton Therapy Center. Each hazard is assigned a severity (High, Medium, Low) based on its worst case realistic outcome. The severity levels are interpreted as follows:

High - loss of human life, complete failure to treat life a threatening condition, damage to irreplaceable equipment

Medium - human harmed non-fatally, damage to expensive but replaceable equipment, reduction in effectiveness of treatment, significant treatment delays

Low - minor reduction in effectiveness of treatment, increased but tolerable maintenance costs, minor treatment delays

We organize the hazards into two broad categories: exposure of humans to life threatening conditions, and exposure of critical machinery to destructive conditions. The hazards to humans are classified into general risks to all humans, risks that pertain only to patients, and risks that pertain only to non-patients (such as therapists, support staff, or personnel in adjoining buildings). The severities for some hazards are a bit tricky to assess, and are discussed below. [1]

Types of Underdose

Delivering an underdose to a patient – less radiation than prescribed – is not immediately harmful to the patient. However, the severity can range from Low to

[1] Our assessments are based on our discussion with BPTC personnel, but have not been certified by them.

High depending on two factors: (a) how aware the therapists are that an underdose occurred, and (b) how certain the therapists are of the intensity of the underdose. If the patient is massively underdosed but the therapists are not aware of the problem, then that patient's cancer will go untreated (High severity). If the therapist is aware that the patient was underdosed, but doesn't know by how much, then treatment must be interrupted, to avoid overdosing the patient by repeating a dose (Medium severity). If the patient is underdosed and the therapists are certain of the intensity delivered, then the remaining dose can be accurately delivered (Low severity).

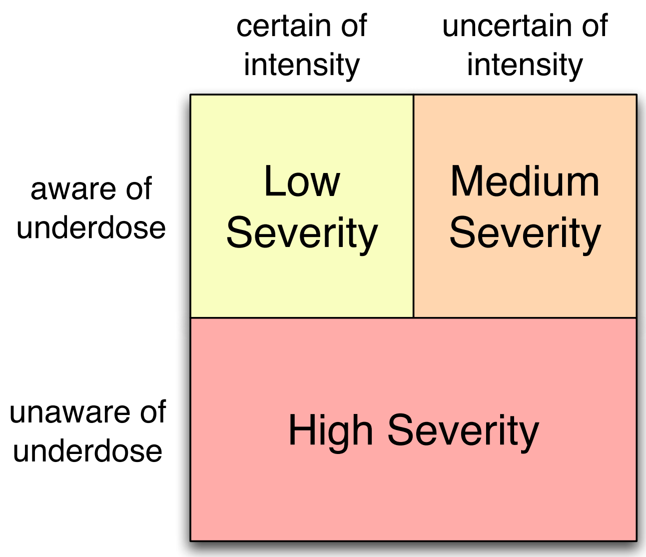

Figure 4-2: The severity of a treatment underdose depends on whether or not the therapist is aware of the underdose, and, of so, whether or not the intensity of the underdose is certain.

Severity of Interrupted Treatments

Physical machinery is threatened by the accumulation of radiation over time. In contrast, human tissue is threatened by single high doses of radiation, but is largely resilient to many low doses.

Suppose human tissue will regenerate from radiation damage (and a patient can be safely treated again) after X days, and suppose that BPTC cancer patients typically have Y days remaining until the cancer becomes untreatable. The increased risk to the patient of receiving an interrupted (partial) treatment depends on the ratio of X to Y. If X is much smaller than Y, then preventing interrupted treatments are of Low or Medium severity. If X and Y are close in value, then interrupted treatments are High Severity.

If a patient can be retreated after 1 week ($X = 7$) and the patient needs treatment within 2 years ($Y = 730$), then the increased risk of an interrupted treatment to the patient's life is $7/730 \approx 1\%$ (Medium Severity). In contrast, if the patient cannot be safely retreated for 6 weeks after an interrupted session, and the patient must be treated within 4 months, then the increased risk to the patient's life is $42/120 \approx 30\%$ (High Severity).

Separation of Equipment and Patient Hazards

The hazards in Figure 4-1 concerning equipment damage are evaluated based on the cost of repairing or replacing the equipment. The increased risk to the patient's well being of operating with damaged components is captured in separate patient hazards. These are separated both because they have different severities and because they are often mitigated in different ways.

For example, a broken gantry might fail to stop rotating when instructed, and end up crushing a patient. The "Broken or Inoperable" hazard has a medium severity, as gantries are expensive but replaceable. The "Crushing Force" hazard to patients is high severity, as it can be fatal.

These hazards are kept separate, as the mitigation tactics are likely to be quite different for the two concerns. Avoiding damage to the gantry might be achieved by building in bracers so that the gantry motor is not powerful enough to damage the rotational mechanism. Doing so will protect the gantry, but not the patient. The patient hazard might be addressed by ensuring that the patient is not in the line of rotation of the extended nozzle. Doing so will keep the patient out of harm's way, but will not stop the machinery from damaging itself by hitting some other surface.

4.3 Dose Delivery Argument

Our technique is best explained by application to a real problem. Consider the *dose delivery* subproblem:

> If the therapist instructs the proton beam to fire, and no explicit error message is presented to the therapist, then the patient under the nozzle of the beam will receive the prescription stored in the database for that patient.

This is a concern about matching the identity of the patient to the prescription in the database, and then delivering that dose to the patient. Safe error handling, database initialization, and unsafe prescriptions are separate subproblems and will not be addressed here. This argument is focused on the problem of coordinating the therapist's instructions (entered via a GUI), the database values, and the hardware device drivers. As a result, we focus primarily on the treatment manager software at the center of this coordination.

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

4.3.1 Designations

The designations of a subproblem relate the formal terms used in the argument to their informal counterparts in the real world and/or code base.

Domains

Patient - The person who has been positioned under the beam nozzle. This is the person who will be receiving the bulk of the radiation generated by the device, but it is not necessarily the patient who should be receiving it and is not necessarily the only person receiving it. We assume that there is exactly one person under the nozzle, and the wording of the other designations reflects that assumption.

Therapist - The hospital employee who identifies the patient, confirms his or her prescription, operates the positioning mechanisms, and initiates delivery. The therapist spends part of the time in the treatment room with the patient and part of the time in the adjoining treatment control room.

GUI Interface - The graphical user interface used by the therapist to select the patient, read back the patient's prescription, and instruct the device to deliver the selected treatment to the patient under the nozzle. The source code for this software is in the "hci" directory of the code hierarchy.

TM Treatment Manager - The software that receives GUI commands and requests, and which sets the beam equipment device drivers to deliver a certain intensity of radiation for a certain amount of time. The source code for this software is in the "app/treatmentmgr" directory in the code hierarchy.

Messages on Network - The communication channel between the GUI and the TM. Communication is handled by atomic messages packed by the sender, sent via *RTworks*, and unpacked/interpreted by the receiver. RTworks is third party software, and provides certain (commercial, not formal) guarantees about message ordering and persistence. RTworks is no longer an active product, although it was a reputable one.

DB Prescription Database - The database containing patient information. Each patient entry includes an identifier (patient id), the patient name, and information about the prescription. For our purposes, we use the following relational abstraction for the database:

$DBnamesInfo : id \rightarrow string$ - the name and personal information for the patient with this id

$DBdoses : id \rightarrow value$ - the dose value(s) for the patient with this id. An abstraction of all prescription information.[2]

$inactive \in names$ - the subset of the *names* mapping for those patients who are currently active

HW Beam Equipment - The electrical and mechanical systems which generate and deliver radiation to the patient under the nozzle. This domain ncludes devices such as the cyclotron (which generates the proton beam) and gantry (which rotates the nozzle into position above the patient).

Phenomena

nameInfo - Name and personal information of the patient under the nozzle. It is assumed that any two patients can be distinguished by the personal information included in nameInfo, such as admission date and home address.

patientDose - The sum total of radiation that is delivered to the patient during the course of the treatment session. Dose does not include location and distribution, which are not relevant to this subproblem.

selection - The patient selected by the therapist from the list displayed by the GUI, with the intention of matching the patient's name. The mechanism for selection is not specified in this diagram. Currently, the therapist uses a mouse and keyboard to select a patient tname from a list of active patients displayed on a terminal.

Actually, a therapist makes several sequential selections to pull up treatment data for a delivery. The therapists selects a patient, then a treatment, then a component of that treatment (if it is compound). For the purposes of this analysis, we abstract all of those selections into the act of selecting a single

[2] We use the following naming convention in our designations: a singular phenomenon corresponds to a single piece of data; e.g. `dose` refers to a single dose. Plural phenomenon names correspond to a mapping; e.g. `doses` is a mapping from each `dose` to some `id`. More lucid names are perhaps desirable in the future, but consistent naming conventions are essential.

treatment entry for the patient under the nozzle. There is really another subproblem here about selecting the wrong treatment for the right patient. This subproblem considers the possibility of getting the wrong patient.

read id msg - The execution of code that receives a message, interprets it as containing information about the selected patient, extracts a patient id from the message, and stores that id.

send id msg - The execution of code that encodes information about the id of the selected patient into a message and sends that message.

send list msg - The execution of code that encodes the mapping from each active patient's nameInfo to that patient's id, and then sends over the network.

read list msg - The execution of code that receives a message, interprets it as containing a mapping from patient nameInfo to patient id's, and stores that mapping.

query doses - A query formulated, submitted, and answered by the database. A query has two parts: the request and the response. In this case, the query is given a patient id and returns the dose prescribed for that patient.

query list - A query as before, but in this case it is a query requesting the set of active patients, and their name/id mapping. This list only includes the patients who have an active status flag in the database.

settings - The execution of the code that sets device drivers for the beam equipment. Among other things, those settings determine the intensity and duration of the delivery.

interpretation - A function that maps machine settings to dose delivered by a machine with those settings. This is really just a unit conversion. The HW embodies this interpretation, and the TM explicitly encodes it in a translation routine.

DBdoses - A mapping from id numbers to prescription dose information. We abstract all prescription information stored in the database into this phenomenon. This map is stored in the database, and is intended to describe the dose that the patient's physician wants to have delivered to the patient.

DBnamesInfo - A mapping from id numbers to names and personal information of patients as recorded in the database. We abstract all patient identification information (e.g. name, address, admission date) into this phenomenon. Names are not necessarily unique, but personal information is assumed to be so.

Requirement

The requirement given in Figure 4-3 is an interpretation of the informal requirement that patients receive their prescribed doses, within a certain safe margin of error.

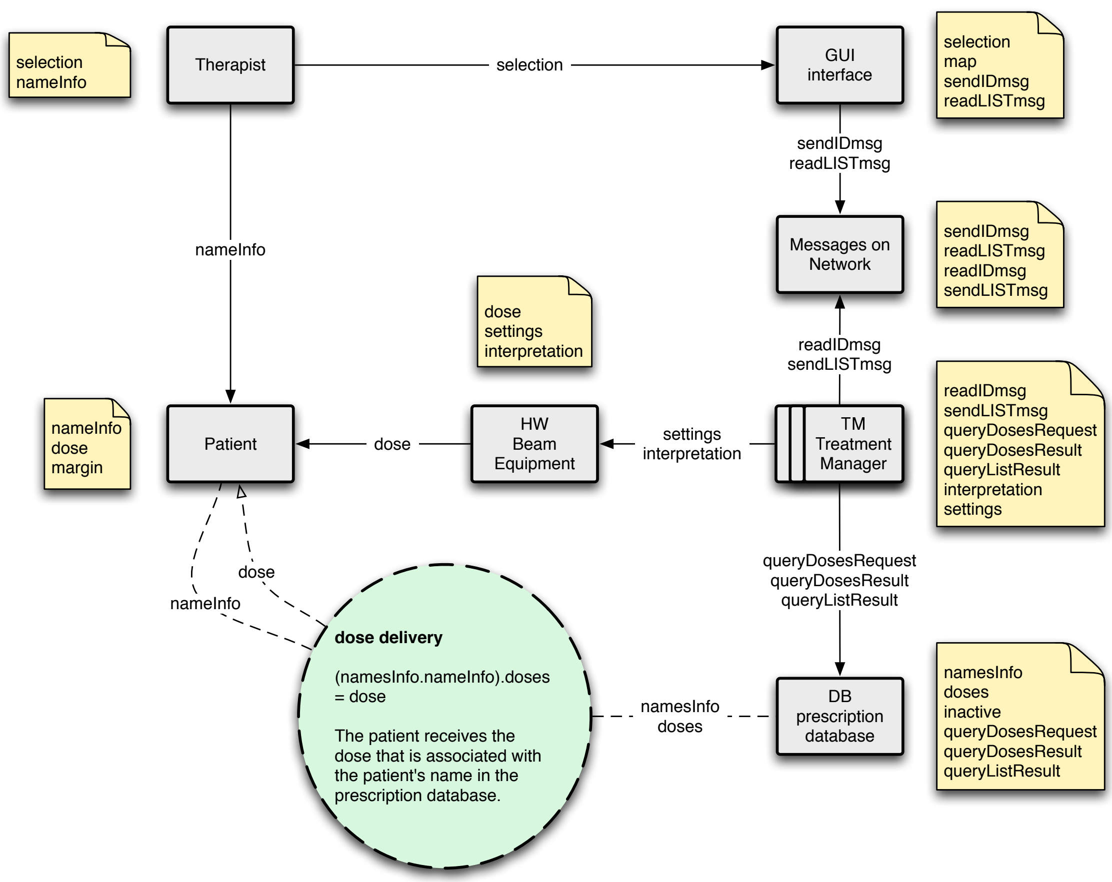

Figure 4-3: Problem Frames problem diagram for the patient identity subproblem.

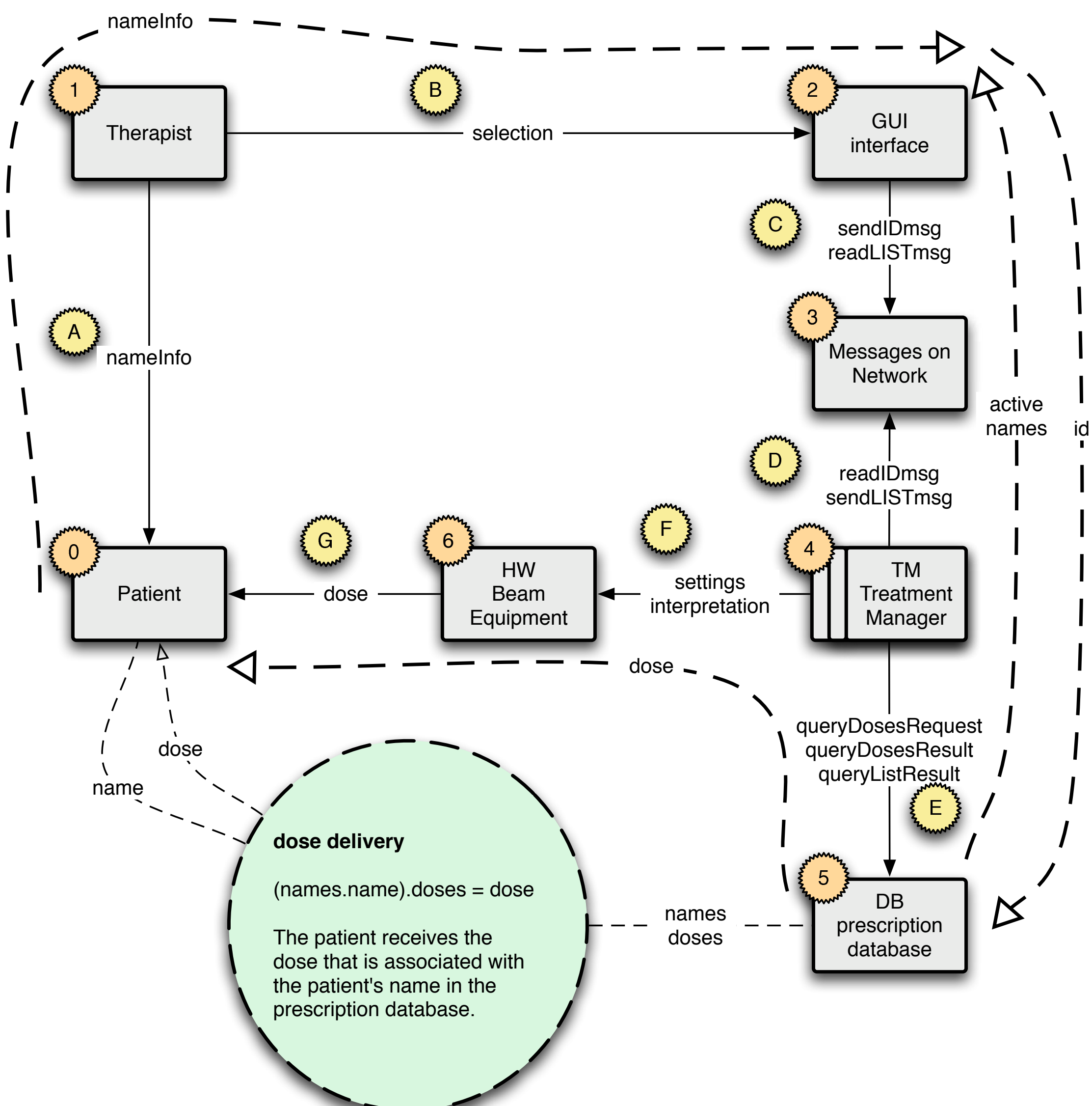

Figure 4-4: Flow diagram for the patient identity subproblem.

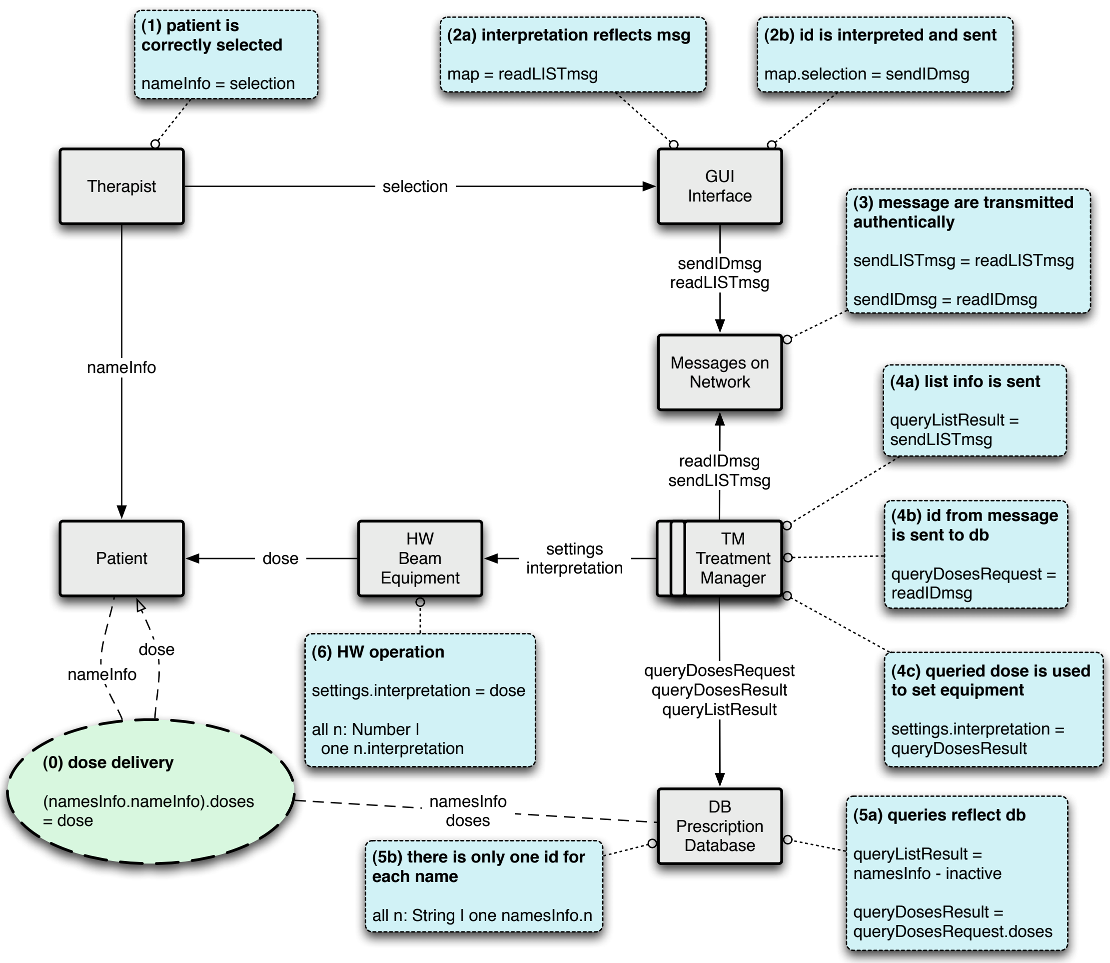

Figure 4-5: Argument diagram for the patient identity subproblem.

```
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 4-6: An Alloy model verifying that the argument diagram is consistent; if the breadcrumbs hold, then the requirement will hold.

4.3.2 Problem Diagram

Figure 4-3 shows the problem diagram for the dose delivery subproblem. It shows the cross-cutting slice of the BPTC system relevant to the dose delivery concern, lists the relevant phenomena for each domain involved, and indicates how the domains interact via shared phenomena.

The *Dose Delivery* concern relates the names and doses stored in the database to the name of the Patient under the beam and the dose delivered to that Patient. Using the Alloy Language formal notation, it specifies that the patient receives the dose associated with that patient in the prescription database.

4.3.3 Flow Diagram

Figure 4-4 indicates the pattern of information flow through the system with an annotated version of the problem diagram. The diagram's semantics are informal, but it guides the construction of the dependability argument in later stages.

Identifying information about the Patient (nameInfo) flows from the Patient to the GUI interface via the Therapist. The mapping between names and id's for active patients is passes from the database to the GUI via the Treatment Manager and messaging Network. Those two pieces of information are reconciled to select the Patient's id, which is send back to the database via the messaging Network and Treatment Manager. The database maps the id into a dose, which is transferred to the patient via the Treatment Manager and Beam Equipment.

4.3.4 Argument Diagram

Figure 4-5 shows the argument diagram derived from the application of requirement progression to the problem diagram. The cross cutting dose delivery property has been decomposed into a collection of *breadcrumb* assumptions, each of which only references phenomena from a single domain. These domain assumptions can be handed off to domain experts and independently validated. In our work, we focus on the software-related assumptions made about the Treatment Manager.

4.3.5 Argument Validation

Figure 4-6 gives an Alloy model used to validate the decomposition depicted in the argument diagram (Figure 4-5). It verifies that, within the given bounds, the breadcrumb domain assumptions are sufficiently strong to enforce the dose delivery requirement.

4.3.6 Breadcrumb Interpretation

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

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

This property is checked against the code base using the Forge framework [23], along with some accompanying liveness checks to mitigate the chance of overconstraint. In order to perform these analyses, the C sourcecode must be translated into the Forge Intermediate Language. This translation is currently performed manually, but much of it could be automated, by programs such as Tochiba's CForge and Dennis's JForge [23].

4.3.7 Breadcrumb Assumptions & Hazards

We now examine each breadcrumb assumption in the argument diagram (the result of requirement progression). We interpet each using the designations relevant to its domain, thus allowing domain-specific tools and experts to be applied to validating them. As show in Figure 4.3.7, we are essentially reversing our earlier designations. The designations carried us from the world into the formalism, but we now use them to carry us back.

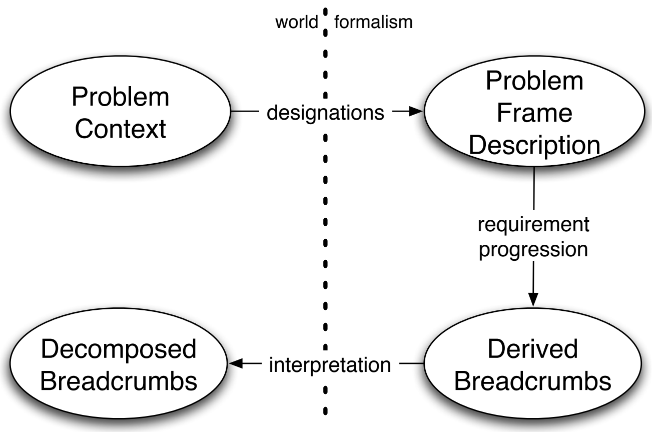

Figure 4-7: Designations carry us from the world into the formalism, where techniques such as requirement progression can manipulate the problem. The result of such manipulation is carried back into the world via interpretation.

In doing so, we decompose the breadcrumbs into component claims and classify those claims in terms of how they should be discharged.

Assumption Types

Each hazard is either entrusted to a domain expert or translated (using the designations/abstractions) into terms that can be check directly against the code. Code concerns are either correctness (the things that should happen will happen correctly) or separability (things that shouldn't happen won't happen).

(u) user interface – claims that humans understand and correctly interact with the mechanized portions of the system. We analyze such properties through an informal but systematic examination of the human domains, in an attempt to classify the failure modes.

(c) software correctness – claims that the portions of code that should provide a given function do provide that function. We analyze software correctness properties using the Forge program analysis.

(s) software separability – Claims that other portions of code do not interfere with that function. We analyze separability claims with memory safety and data flow analyses. [3]

(x) non-software – All other claims. We delegate the validation of non-software properties to domain experts. Because the claims have been mapped into the language of their domains, domain experts should be able to independently validate or decline them.

Domain Types

It is in this stage of the analysis that classification of Problem Frame domains into biddable (human), lexical (data), and causal (machine) are relevant, as those categories suggest how to substantiate each type of assumption. For example, many domains serve as data transformers; data passes through them and changes form, but does not change the piece of information it is representing. Data transforming domains often contain the following pair of assumptions:

(c) The data is correctly transformed between when it is received and when it is transmitted.

```
transmitted data =
 desired transformation ( received data )
```

(s) The temporary, internal representations of the data are not corrupted during between reception and transmission. For example, they might be write-once.

```
data stored = data recalled
```

In a causal or lexical domain, the second assumption means that the temporary intermediate variables are not overwritten and are passed along in a variable. In a biddable (human) domain, this means that the human does not forget or garble information and correctly/honestly/unambiguously writes down what was remembered.

[3] Of course, in a memory-unsafe language like C, this is not strictly possible without a much more heavyweight analysis than we are willing to apply. However, we can still perform lightweight scans of the code, to at least identify which poritions of the code are accessing certain globals. For a more thorough separability analysis, one would either need to use a safer language, or apply a much more heavyweight analysis, such as Astree [10].

The Breadcrumbs

Therapist: (1) "patient is correctly selected"

`nameInfo = selection`

(x) There is one patient on the table.

The machinery is configured to treat one patient at a time, as patients have unique treatment plans. We do not consider multiple, simultaneous treatments.

example: We do not allow a parent to hold a child during treatment. We do not allow a pregnant woman to be treated, as the mother and fetus would be two patients under the nozzle.

(c) There is one selection made in the GUI. It is not possible for a therapist to select more or less than exactly one entry and still proceed to treatment.

note: Some prescriptions are compound, and are delivered over the course of several visits to the center or with several firings of the beam. In such a case, we consider the delivery to be restarted from the beginning, including selection of the treatment. Since we are considering a single firing of the beam, we require that a single entry be selected for that firing.

example: If the therapist does not select an entry from the list, then the GUI must not permit the therapist to move on to the next stage of treatment. If the therapist selects a second entry, the GUI must either de-select the old entry, or deny the new selection. In either case, the therapist must be made aware that the selection did (or did not) actually change.

(u) The selection made in the GUI matches the patient.

The identity of the patient (nameInfo) is the identity selected from the GUI. The therapist selects exactly one patient from the list, and that patient is the one under the nozzle.

example: The therapist must not identify the patient as "Fred Smith" but select "Fred Smyth" in the GUI. One might address this concern by holding the patient tag up next to the screen, rather than remembering it as one walks across the room from the patient (and tag) to the GUI screen. Such a routine would require that the therapist always correctly returns the tag to the patient, and that there are no other tags lying around that could be confused with the patient tag.

(x) The patient matches at least one entry.

example: As long as a patient is under treatment, she must have an entry in the database indicating her prescription. If a patient does not have an entry, she should not be in the treatment room.

(u) The patient matches at most one entry.

If there are two similar entries, then (1) they must be for different patients, (2) there must be sufficient additional information to disambiguate the entries, (3) it must be apparent that disambiguation is necessary, and (4) it must be apparent when disambiguation has been sufficiently established. It is technically ok for a patient to have several identical entries, but we will establish the stronger claim that there is only one. Multiple entries would increase the likelihood of errors being introduced through dual maintenance.

example 1: If a patient is evaluated by two physicians, he must still only have one prescription in the database, and thus only one entry in the GUI list. If the patient's prescription is updated, the old one is overwritten, deleted, or made inactive. As an invariant on the database, each patient has at most one active entry. To avoid maintenance risks, it is best if the patient has at most one entry total, be it active or inactive.

example 2: If there are two "Fred Smith" patients, we assume that they have distinguishing personal information, and that the therapist is aware of that information.

The therapist might not be permitted to select one without explicitly turning the other down. If one is visible on the screen, the other must not be hidden off the screen by a scrollable list or buried in a long unsorted list. If one of the entries is selected, perhaps all similar entries should be brought to the attention of the therapist for comparison.

example 3: Consider the case where two names are identical in the primary list (that just displays names), so additional information is manually pulled up on each. In the current GUI, this is in an overlaid window showing home addresses and admission dates. When the extra info window is closed, and the therapist returns to the primary list after determining the desired patient, he/she must not accidentally select the other one (since on the primarily list, they are still identical). For example, the therapist might be allowed to select a patient from that patient's expanded info window. Or when the therapist returns from the expanded info window for a patient, that patient's entry in the primary (name only) list is highlighted.

example 4: Two patients with the same name and the same admission data are active, and one is currently being treated. If the other is first on the list, the therapist will pull up its disambiguating information (admission date) and see that it matches, and select. If the therapist does not also look at the other patient's data, it will remain hidden that they have the same admission date and thus require further disambiguation. In this case, being able to select a patient from that patient's expanded info window would exacerbate the issue. Better to identify potential ambiguities and force the therapist to explicitly eliminate them.

(2a) GUI: "interpretation reflects msg"

`map = readLISTmsg`

(c) The map from patient nameInfo's to patient id's is received in a message from the database and is correctly stored and accessed

example: The locally stored map might use a data structure that resolves ambiguities differently than the database. For example, if there are duplicate entries in the database, the local map might choose one of them arbitrarily, rather than reflecting the ambiguity.

(s) The map is not corrupted while stored.

Ideally, the map should never be altered unless completely replaced by a new list from a more recent message.

violation: In the current system, a therapist can override the data held in the treatment manager, for example to override an angle based on image feedback. This violates our assumptions about synchronization with the database dose information. The argument presented here holds as long as the therapist does not use the override feature.

(2b) GUI: "id is interpreted and sent"

`map.selection = SendIDmsg`

(c) The selected entry is stored into some temporary representation, which must be the same variable used to retrieve the patient id from the name-id mapping.

(s) The temporary representation of the selected entry is not corrupted between when it is written and when it is read. The simplest case is when it is write-once.

(c) The selected entry is the entry used to lookup the patient id in th name-id mapping. The mapping has exactly one id for that entry, and that is the id that is returned.

(s) The temporary representation of the id extracted from the map is not corrupted. Preferably, it is write-once.

(c) The id is packed into a msg accurately.

(s) The message is not modified between when it is packed and when it is sent. It might be write-once, or it might only be modified after transmission.

(3) Network: "message are transmitted authentically"

`sendLISTmsg = readLISTmsg`

`sendIDmsg = readIDmsg`

(x/s) Messages are transmitted without corruption; their contents when they leave are the same as their contents when they arrive. At this level of description, we abstract away any mechanism for detecting corruption and resending messages, and just consider the entire message transmission (and re-transmission) process to be a single event.

(c) Messages eventually get to their destinations. This might subsume a re-send process to account for dropped messages, and refers to the eventual effect of a send command.

sample problem All patient selection messages from the GUI to the TM are dropped, preventing treatment from progressing.

(s) Messages do not get sent to other destinations.

sample problem Patient data from the TM is sent to each treatment room, and two concurrent patients in two treatment rooms receive the prescription for one of them.

(s) Messages are not corrupted in transit.

sample problem A patient is identified correctly in the GUI, but the message to the TM is corrupted and the wrong patient is selected.

aside: One might want to make some claim that messages get to their destinations within a certain acceptable delay. However, for the patient id subproblem, there is no timing requirement, and the domain assumption we derived for the messaging domain is simply that messages eventually reach their destinations.

contradiction: In validating this assumption, we learned that this is not how the BPTC network operates. Here, we modeled it as a send-to-destination system.

Actually, it is a publish-subscribe system, using shared variables stored on the network and duplicated in the subsystems. There are additional relevant assumptions that we must introduce when viewing the full complexity of the messaging system, which are abstracted away in the representation here. For example, can several subsystems publish to the same shared variable? How often do readers of the shared variables update their local values?

(4a) TM: "list info is sent"

`queryListResult = sendLISTmsg`

(c) The result of the querry is stored into some internal representation.

(s) The internal representation is not overwritten between its creation and its use

(c) The internal representation is encoded into a message and sent onto the network.

TM: (4b) "id from message is sent to db"

`queryDosesRequest = readIDmsg`

elaboration: In order to describe the connection between the id read from the message and the id sent in a query to the database, we are forced to expose the fact that there is redundancy in the storage of the id value. It is stored in 3 places. One is only local to the message unpacking, and does not escape. Another escapes (it is a write to a pointer) but does not appear to ever be read. The third escapes and is used to form the database query. The dual maintenance of the two escaping representations does not appear to be necessary, but does complicate the correctness argument. It does not appear to represent a fault in the current system, but is certainly a vulnerability: if the code is later modified, it would be easy to read/write one but not both of the escaped id representations, creating a potential for stale or inconsistent data.

(c) The message is constructed correctly from intermediate representation(s) of id.

This has 2 parts: (a) that the message is stripped and stored in variables, and (b) that the variables are read and used to build a message.

We can check properties of that code using the Forge Analyzer.

(s) The message, and the data being put into it, are both write-once; all representations of id are consistent whenever read.

(4c) TM: "queried dose is used to set equipment"

```
settings.interpretation = queryDosesResult
```

(c) The dose value is translated and used correctly to set the machine.

(s) the temporary rep of dose is not overwritten

(s) The equipment settings aren't overridden.

(5a) DB: "queries reflects db"

```
queryListResult = namesInfo - inactive
queryDosesResult = queryDosesRequest.doses
```

(s/x) The database itself is not corrupted, and remains constant throughout the treatment.

example: A query contains an sql injection, either intentional or accidental, might overwrite patient prescription data. Garbage values for a dose would almost certainly result in a massive overdose, as most (32 bit) integers correspond to dangerous levels of radiation (a high severity hazard). Even an error value that is safe on an absolute scale is still likely to cause a systematic underdose (High severity).

(x) The result returned by a query is the answer to that query, according to the information stored in the database at the time.

(x) The subset of the map sent to the GUI correctly reflects the subset of patients who are currently active in the actual database.

example: If there is one patient with the name "Fred Smith", the patient is active, but the active flag is mistakenly off, then things are not so bad. The patient will not show up on the list, and the therapist will realize that there is a problem. The patient's treatment will be delayed, but not misdelivered (a low severity hazard).

example: Similarly, if there is one patient with the name "Fred Smith", the patient is no longer active, but the active flag is mistakenly on, the patient will just be a dead entry in the list. The risk that the therapist will accidentally select this name instead of the correct one is no greater than the risk that the therapist will accidentally select another active patient. The only added risk is that the list will become very long and diluted.

example: However, consider the case where there are two active patients with the same name (Fred Smith) but different personal information (addresses and admission dates). If one of them is correctly flagged as active but the other is incorrectly flagged as inactive, then the therapist will almost certainly select the wrong one. On the primary list (with just names), there will be just one Fred Smith, so the therapist will not see a need to consult/confirm the elaborated entry (with addresses). Both patients will receive the dose prescribed to the Fred Smith with the active flag on. Worse, they might each receive half of the treatment segments, causing both to receive systematic underdoses (a high severity hazard).

(5b) DB: "there is only one id for each name"

```
all n: String | one namesInfo.n
```

(c) As an invariant on the database, every name in the database maps to exactly one id under the `names` mapping.

(6) HW Beam Equipment: "HW operation"

```
settings.interpretation = dose
all n: Number | one n.interpretation
```

(x) If the machine changes settings into doses according to the interpretation mapping, then the settings made by the TM will produce the desired dose for the patient.

violation: This assumption is not quite correct, as there is a margin of error. The current model does not represent error margins in the machine, and permissible error margins in the dose delivery, as doing so does not affect patient identity. Really, interpretation maps a number to a range of numbers, and the claim is that any of the range of numbers mapped from the machine settings would be within the error margin of the patient's dose.

4.3.8 Arc Assumptions & Hazards

Every problem diagram contains implicit assumptions that phenomena arcs are atomic (non-interruptible) and accurate (non-corruptible). When we say that two domains share a phenomenon, we are assuming that they really are viewing the same phenomenon and that their two different views of that phenomenon are consistent. Here, we make such assumptions explicit.

A: Therapist → Patient

```
Therapist.nameInfo = Patient.nameInfo
```

The nameInfo that the therapist uses to make GUI selections is the same nameInfo that the patient has.

(u/x) That patient has one nameInfo, the therapist uses exactly one nameInfo, and the two match. The nameInfo the patient and therapist use must match the nameInfo the patient has been using for identification in the hospital. The nameInfo must pass from the patient domain to the therapist domain without corruption.

sample problem: The therapist might ask the patient for his or her name to double check, but should not rely on the patient's answer. The patient might not be mentally acute enough to answer such a question accurately – many alzheimer's patients will answer "yes" to any question, so asking "Is your name Fred Smith" is not adequate.

sample problem: A patient might have multiple names, and give you a different one than he or she gave to the hospital upon admission. Fred George Smith might normal go by George, and tells you that his name is George Smith. If the hospital uses the name "Fred Smith" to identify him, then the wrong treatment may be delivered.

sample procedure: The therapist might scan a bar code on the patient's ID tag, and check the photo on the tag matches the patient's appearance. We assume that a photo is sufficient to match an ID with a patient, and that the barcode system and database are accurate. We assume that the patient's appearance has not changed since the photo was taken – e.g. the patient might have gained or lost weight or hair as a result of hospitalization.

B: Therapist → GUI

```
Therapist.selection = GUI.selection
```

(u/c) The therapist selects exactly one entry from the GUI. The GUI records exactly one patient id. The label on the item selected by the therapist is the patient id that is recorded.

sample problem: The therapist might select an entry labeled "Fred Smith", but the GUI erroneously stores "Sally Queue" as the selected patient.

sample problem: The therapist might change his or her selection. In such a case the prior selection must be de-selected, and the GUI's record must reflect the new selection.

sample problem: If the therapist has not yet made a selection, then the GUI should be unable to proceed until a selection is made. The GUI must not have a default recorded value that it could interpret as a patient id.

C,D: GUI → Network, TM → Network

(s) Messages sent to the network end up on the network with the same content. Messages taken off the network were on the network and have the same content.

implication: In particular, this assumption entails the following requirements: messages are not created on the network except when a message is sent, and messages are not destroyed on the network except when a message is received. Two distinct messages have distinct labels. Messages are unambiguously identified; when a message is received, the receiver knows what type of message it is, who sent it, and for what purpose. Two messages are never confused for each other.

sample problem: The GUI sends a message to the TM indicating the current patient. This message is delayed on the network. The GUI resends the message, and the TM receives it this time, selecting the correct patient. The TM is now waiting to hear from the GUI when it should begin firing the beam. The original selection message now arrives, and the TM interprets it to be a "fire now" command, and begins treatment while the therapist is still in the treatment room.

sample problem: The GUI sends a message to the tM indicating the current patient. The message is delayed on the network, and is resent by the GUI. Later, another patient is brought in. The previous patient-selection message arrives, and the TM loads the wrong patient data.

aside: This assumption does not include corruption that occurs during network transmission, which is a domain assumption about the network message domain. These assumptions are concerned with message creation and reception.

(c) When unpacked into a datastructure, the data has the same format as when packed into a message. That is, the unpacked message is correctly interpreted.

sample problem: If the id packed into a message is stored in a different kind of struct than the one it is unpacked into, then erroneous values could be read for the id. Because the messages are not type safe (they are treated as bits, not as strings and integers), there is no way to *a priori* know that a garbage value is not a valid id.

This concern can be checked at any given point by examining the data structures used on both ends, but such checks are not very robust across maintenance.

E: TM → DB

```
TM.queryDoseRequest = DB.queryDoseRequest
TM.queryListRequest = DB.queryListRequest
```

(c) Queries constructed in the treatment manager C code are the same as the queries interpreted by the database.

sample problem: The compilation step that translates queries written in C into the database query language (SQL?) might introduce errors.

(x) The queries made by the treatment manager are authentically transmitted to the database.

sample problem: The queries might be constructed using one set of semantics but interpreteed by the database using different semantics.

F: TM → HW

```
TM.settings = HW.settings
TM.interpretation - HW.interpretation
```

(x/s) The hardware device driver settings made by the software are conferred to the hardware without corruption.

(c) The interpretation implicitly embodied by the hardware is the reverse of the interpretation explicitly applied by the treatment manager. That is, the treatment manager has been calibrated correctly to reflect the current behavior of the hardware.

G: HW → Patient

```
HW.dose = Patient.dose
```

(x) The HW beam equipment completely determines the dose of radiation that the patient receives. There are no barriers or other sources of radiation that could make the dose that ends up in the patient different from what would be induced by the radiation from the beam equipment.

sample problems: Ambient radiation in the room raises the total intensity of the dose received, so the patient receives an overdose. Lead in the patient's clothing dissipates the beam, resulting in an underdose.

contradiction: Actually, this assumption will never hold. The real assumption is that the radiation received is within a known small margin of error of the radiation expected to be received. A further assumption is then needed that the sum of all possible accumulated errors will not produce a treatment that is too much above or below the prescribed treatment.

4.4 Translation to Forge

To demonstrate our end-to-end synthesis, we use the Forge framework [23] to check our software properties against the BPTC code base. For this analysis, we only consider correctness properties, not separability properties. Our translation of the sourcecode into Forge is done manually, but much of it could be automated. The purposes of this section is not to propose this technique as the ideal method of code analysis, but rather to demonstrate how a code analysis technique can be smoothly intergrated into the larger dependability argument – the relational claims generated by requirement progression can be fed directly into the Forge analysis engine.

4.4.1 Sample Procedure Translation

In the Patient Identity subproblem, the relevant software fragments were manually abstracted and translated into Forge via a Java API. The translation process is fairly systematic, and part or all of it could be automated. In this section, we examine a single procedure from the Patient Identity case study, and follow it through the translation process.

version	lines	non-trivial lines
original C code	62	30
reformatted C code	27	26
abstracted C code	17	14
Java API calls	52	41
Forge code	13	12

The *non-trivial lines* column ignores blank lines, comment lines, and lines that contain just a curly brace. It gives a rough sense of the human cost of performing the manual translation, and of the efficiency of the encoding.

```
/*****************************************************************************
* tpcrInSelectPatient
* PSEUDO-CODE :
*
* Extracts groups of information in array.
* Checks value of property field.
* Checks value of widgetGroupId field.
* Checks value of widgetId field.
* Extracts patient id and copies to structure scrCrtPatientData.
* Calls function eventsTPCRSelectPatient.
*
*/
BOOLEAN tpcrInSelectPatient(
            /* IN */       T_INT4 screenId,
            /* IN */       T_INT4 sizeofList,
            /* IN */       DATA_MSG_GROUP_ARRAY msgList,
            /* IN_OUT */   DBASCR_SCR_DATA_PTR_TYPE pscrData)
{
   T_INT4 num;

   /* Process list of message */
   for(num = 0; num < sizeofList; num++) {
      if(msgList[num].property != PDEF_CONTENTS_PROPERTY) {
         SW_ERROR_MSG(ERR_APP_WRONG_PROPERTY_TYPE);
         return FALSE;
      }

      switch(msgList[num].widgetGroupId) {

         case WG_PATIENT:
            switch(msgList[num].widgetId) {
               case W_PATIENT_ID:
                  if(strlen(msgList[num].value) > DBA_PATIENT_ID_LEN) {
                     SW_ERROR_MSG(ERR_APP_WRONG_STR_VALUE);
                     return FALSE;
                  }
                  strcpy(pscrData -> scrCrtPatientData.id_patient, msgList[num].value);
                  break;
               default: /* Error happens */
                  SW_ERROR_MSG(ERR_APP_WRONG_WIDGET_ID);
                  return FALSE;
            } /* widgetId */
            break;

         default: /* Error happens */
            SW_ERROR_MSG(ERR_APP_WRONG_WIDGET_GROUP_ID);
            return FALSE;
      } /* widgetGroupId */
   }

   /* call events function correspond to ``Select Patient'' button */
   if(!eventsTPCRSelectPatient(screenId, pscrData)) { /* Error happens */
      TRACE_ERROR_MSG();
      return FALSE;
   }

   return TRUE;
}
```

Figure 4-8: Original C source code for the Patient Selection routine, exactly as it apears in the BPTC code base. 62 lines, 30 of which are non-trivial.

4.4.2 Original C Code

Figure 4-8 shows the `tpcrInSelectPatient` procedure, shown verbatim from the BPTC code base. This code displays a number of conventions that are persistent throughout the BPTC code base.

Comments

The style of comment provided for the procedure is typical for the code base – it gives an overview of what the procedure does, but does not describe why it does it (and thus is of limited value for the purposes of traceability).

Exception Mechanism

Almost every called procedure returns a boolean, indicating whether or not it encountered an error If a procedure has information to return (that would normally be in a return value), then a pointer is passed in which is mutated by the procedure. In some cases, one of the parameters is mutated to contain information about why the error occurred, which is only read if `false` is returned.

In most cases, if a called procedure returns an error, then the calling procedure returns an error. At the top level, the system handles the error and (typically) halts the system with an error message. In some cases, the caller ignores the error and tries something different. These decisions do not appear to be documented in any single repository, or justified in the code comments.

This technique is a common convention for getting the effect of exception handling in C. It is not in itself a good or bad coding style. However, consistent and clear use of this (or any other) style is important if the code is to be trusted and accurately updated and maintained.

Parameter Annotations

Each parameter to a function is annotated (with a comment), with the following interpretation:

`IN` This parameter contains information passed in by the caller. It will not be read after this procedure returns, and any mutations to it are irrelevant.

`OUT` This parameter contains no information initially, and is a pointer reference. It will possibly be read after this procedure returns, so mutations to it are relevant.

`IN_OUT` This parameter contains information passed in by the caller. It will possibly be read after this procedure returns, so any mutations to it are relevant.

There is no check to verify if these comments are correct.

```
BOOLEAN tpcrInSelectPatient(T_INT4 screenId, T_INT4 sizeofList,
                            DATA_MSG_GROUP_ARRAY msgList,
                            DBASCR_SCR_DATA_PTR_TYPE pscrData) {
   T_INT4 num;
   for(num = 0; num < sizeofList; num++) {
      if(msgList[num].property != PDEF_CONTENTS_PROPERTY) {
         SW_ERROR_MSG(ERR_APP_WRONG_PROPERTY_TYPE);
         return FALSE; }
      switch(msgList[num].widgetGroupId) {
         case WG_PATIENT:
            switch(msgList[num].widgetId) {
               case W_PATIENT_ID:
                  if(strlen(msgList[num].value) > DBA_PATIENT_ID_LEN) {
                     SW_ERROR_MSG(ERR_APP_WRONG_STR_VALUE);
                     return FALSE; }
                  strcpy(pscrData -> scrCrtPatientData.id_patient,
                                     msgList[num].value);
                  break;
               default:
                  SW_ERROR_MSG(ERR_APP_WRONG_WIDGET_ID);
                  return FALSE; }
            break;
         default:
            SW_ERROR_MSG(ERR_APP_WRONG_WIDGET_GROUP_ID);
            return FALSE; } }
   if(!eventsTPCRSelectPatient(screenId, pscrData)) {
      TRACE_ERROR_MSG();
      return FALSE; }
   return TRUE;
}
```

Figure 4-9: A condensed (but semantically identical) version of the C code for the patient selection procedure. 27 lines, 26 of which are non-trivial.

4.4.3 Condensed C Code

Figure 4-9 shows the C source code condensed to eliminate all comments, blank lines, and unnecessary line breaks. This code listing is used only to provide a better baseline for comparing the lengths of the different versions and translations of this procedure.

```
void tpcrInSelectPatient (
T_INT4 screenId,
T_INT4 sizeofList,
DATA_MSG_GROUP_ARRAY msgList,
DBASCR_SCR_DATA_PTR_TYPE pscrData)
{
  T_INT4 num;
  for(num = 0; num < sizeofList; num++) {
    if (msgList[num].widgetGroupId =  WG_PATIENT)
      if (msgList[num].widgetId = W_PATIENT_ID)
        if (strlen(msgList[num].value) > DBA_PATIENT_ID_LEN)
          ERROR;
        else
          strcpy(pscrData -> scrCrtPatientData.id_patient,
                            msgList[num].value);
  }
  eventsTPCRSelectPatient(screenId, pscrData);
}
```

Figure 4-10: An abstracted version of the C code for the patient selection procedure. 17 lines, 14 of which are non-trivial.

4.4.4 Abstracted C Code

Figure 4-10 shows the C code after it has been manually abstracted.

The purpose of abstraction is to eliminate parts of the code base that are not relevant to the current concern. In doing so, we make the model simpler, increases the ability of humans to understand it and machines to automatically analyze it. Our abstraction includes two parts: eliminating parts of the code not relevant to any analysis (replacing calls to a trusted code with specifications) and eliminating parts of the code not relevant to this particular concern (focusing on a particular path through the code).

Focusing

Recall that the *Patient ID* requirement is that the prescribed dose is delivered to the patient when no explicit errors are returned. As such, our analysis is not concerned with what happens when explicit errors are generated, so we abstract away such occurrences. In most cases, we simply assume that no error is generated. In some cases, we model the code as setting an error flag, but we declare in our analysis that we are only interested in seeing bad traces in which the error flag was not set. The end effect is that our analysis will only reveal paths through the code in which the stated property is violated but no error was raised. A separate analysis (for a different subproblem) is needed to ensure that error cases are handled safely and properly.

This abstraction was partly done to reduce the branching complexity of the code, thus improving the scalability of the analysis, and partly done to the aid the manual translation process. With proper automatic support, this abstraction might be unnecessary or might be itself automatable.

Called Procedures

For scalability of analysis, and feasibility of the manual translation, we cannot fully model the behavior of every called procedure. At a certain depth in the call hierarchy, the human analyst must decide to cut off the translation, and replace the called procedures with an appropriate (partial) specification. There are a few common reasons to cut off the translation of a procedure:

trusted code base Calls into trusted code bases are a prime candidate for replacement with appropriate partial specification. For example, low level calls, language features, and trusted libraries needn't be translated. Some of these can be handled automatically with proper infrastructure (e.g. language features), the rest might be added manually (e.g. user provided specifications for trusted/in-house libraries).

For example, the codebase contains calls to a native C function that compute the length of a string, `strlen`. We have supplied a full specification for that function, rather than modeling the details of how it executes. We trust that code base enough to leave it out, thus reducing the total number of lines that Forge must analyze directly.

We also assume that correctness of a pre-compilation pass that is performed on the BPTC code. This pre-compilation pass is used to extend the C syntax to include simple calls into an SQL database. Rather than building up a full SQL query by hand each time, it permits the programmer to write a simple "`SQL EXEC querytext`" command. We did not have access to this precompilation code, and are thus forced to assume that all `SQL EXEC` commands do indeed send the given query to the SQL database and receive the answer correctly.

In general, the analyst may decide to make a judgement call about what called procedures to trust, and provide a simple specification for them. Such assumptions must be documented, but are often appropriate (for modularity) or simply unavoidable (when correctness cannot be directly evaluated). In such a case, the important thing is that the assumption be recognized, documented, and consciously accepted.

irrelevant A called procedure might be irrelevant to the property being checked, in which case it can be replaced with an unconstrained spec or simply removed.

For example, a call to a procedure "`state = get_current_state()`" might be replaced with a constant "`state = 'reading_data'`", if the property being check is only relevant when data is being read. Similarly, a call to "`check_message_content_consistency()`" might be omitted if the property being check is about message timings, not content. In both cases, we are essentially replacing a called procedure with a partial specification based on our calling context and target property. Such assumptions must be recorded and documented as part of the code analysis argument.[4]

In the running example illustrated in this section, we have performed 2 (related)

[4] Ideally, one might want to use some sort of automatic iterative refinement, in the style of Taghdiri [84, 85].

abstractions that are more elaborate than simply replacing procedure calls with partial specifications.

The property we are checking is that the correct dose is delivered if no explicit errors are generated. The behavior of explicit error handling code is not relevant to this property. We replace all error code with the flag “ERROR”, which we interpret to mean that a global error flag is set to true. When we check our property, we check that the dose is correctly set as long as the error flag is false. Abstracting away explicit error handling code greatly simplifies much of the code base. We document our assumption that error handling code always results in an explicit error reaching the user. Whether or not the system behaves safely in the presence of an explicit error is a separate (but important) safety concern.

On a related note, the procedure’s Boolean return value is only used as a mechanism for propagating errors up to the top level control loop. Since we have abstracted away the error handling code, we can drop the boolean return values. This abstraction is not strictly necessary, but helps to clean up the code and enable manual translation. An automatical translation might not need this abstraction, and dropping it would reduce the number of assumptions being made about the code base.

```
void define__tpcrInSelectPatient() {
  //signature
  final LocalVariable
  screenId = program.newLocalVariable(``screenId'', T_INT4),
  sizeofList = program.newLocalVariable(``sizeofList'', T_INT4),
  msgList = program.newLocalVariable(``msgList'', DATA_MSG_GROUP_ARRAY),
  pscrData = program.newLocalVariable(``pscrData'', DBASCR_SCR_DATA_PTR_TYPE);
  tpcrInSelectPatient = program.newProcedure(
  ``tpcrInSelectPatient'',
  Arrays.<LocalVariable>asList(screenId, sizeofList, msgList, pscrData),
  Arrays.<LocalVariable>asList(pscrData));

  //body
  final LocalVariable num = program.newLocalVariable(``num'', T_INT4);
  final AssignStmt initializenum = tpcrInSelectPatient.newAssign(num, zero);
  final BranchNode loop_head_condition = tpcrInSelectPatient.newBranch(num.lt(sizeofList));
  final LocalVariable current = program.newLocalVariable(``current'', DATA_MSG_GROUP);
  final AssignStmt assign_current = tpcrInSelectPatient.newAssign(current, num.join(msgList.join(msg_grp_array_index)));
  final BranchNode check_widgetGroupId = tpcrInSelectPatient.newBranch(current.join(widgetGroupId_field).eq(WG_PATIENT));
  final BranchNode check_widgetId = tpcrInSelectPatient.newBranch(current.join(widgetId_field).eq(W_PATIENT_ID));
  final BranchNode check_length = tpcrInSelectPatient.newBranch(((current.join(value_field)).join(strlen)).gt(DBA_PATIENT_ID_LEN));
  final AssignStmt flag_error = tpcrInSelectPatient.newAssign(error_has_occurred, program.trueLiteral());
  final AssignStmt assign__id_patient = tpcrInSelectPatient.newAssign(dba_patient_type__id_patient, dba_patient_type__id_patient.override(
  (pscrData.join(scrCrtPatientData)).product(current.join(value_field))));
  final AssignStmt loop_end_num_increment = tpcrInSelectPatient.newAssign(num, num.plus(one));
  final CallStmt call_eventsTPCRSelectPatient = tpcrInSelectPatient.newCall(
  eventsTPCRSelectPatient,
  Arrays.<ForgeExpression>asList(screenId, pscrData),
  Arrays.<ForgeVariable>asList());

  //linkups
  initializenum.setEntry();
  initializenum.setNext(loop_head_condition);
  loop_head_condition.setThen(assign_current);
  assign_current.setNext(check_widgetGroupId);
  check_widgetGroupId.setThen(check_widgetId);
  check_widgetId.setThen(check_length);
  check_length.setThen(flag_error);
  flag_error.setNext(loop_end_num_increment);
  check_length.setElse(assign__id_patient);
  assign__id_patient.setNext(loop_end_num_increment);
  check_widgetId.setElse(loop_end_num_increment);
  check_widgetGroupId.setElse(loop_end_num_increment);
  loop_end_num_increment.setNext(loop_head_condition);
  loop_head_condition.setElse(call_eventsTPCRSelectPatient);
  call_eventsTPCRSelectPatient.setNext(tpcrInSelectPatient.exit());
  }
```

Figure 4-11: A Java program that generates a Forge program that emulates the C procedure for patient selection. 52 lines, 41 of which are non-trivial.

4.4.5 Java Code to Generate Forge Code from C Code

Figure 4-11 shows the Java code that was manually written to generate API calls to the Forge framework that will generate a Forge program that imitates the abstracted C code.

```
proc tpcrInSelectPatient (screenId, sizeofList, msgList, pscrData) : (pscrData) {
  Node55: num := 0 goto Node56
  Node56: if (num < sizeofList) then Node57 else Node64
  Node57: current := (num . (msgList . msg_grp_array_index)) goto Node58
  Node58: if ((current . widgetGroupId_field) = WG_PATIENT) then Node59 else Node63
  Node59: if ((current . widgetId_field) = W_PATIENT_ID) then Node60 else Node63
  Node60: if (((current . value_field) . strlen) > DBA_PATIENT_ID_LEN) then Node61 else Nod
  Node61: error_has_occurred := true goto Node63
  Node63: num := (num plus 1) goto Node56
  Node62: dba_patient_type__id_patient :=
    (dba_patient_type__id_patient
    ++ ((pscrData . scrCrtPatientData) -> (current . value_field))) goto Node63
  Node64: eventsTPCRSelectPatient(screenId, pscrData) : () goto Node54
  Node54: terminate
}
```

Figure 4-12: The analyzable forge program written based on the C procedure for patient selection. 13 lines, 12 of which are non-trivial.

4.4.6 Generated Forge Code

Figure 4-12 shows the Forge code generated by the above Java code. Forge represents a program as a collection of nodes, each representing an atomic program statement (e.g. a condition, assignment or procedure call). When connected together with control flow edges, a Forge description compactly represents the set of all valid program executions, in a form that can be queried and constrainted using the Forge relational logic. Figure 4-13 shows a control flow view of the Forge program, indicating how the program nodes are linked together to represent legal execution paths.

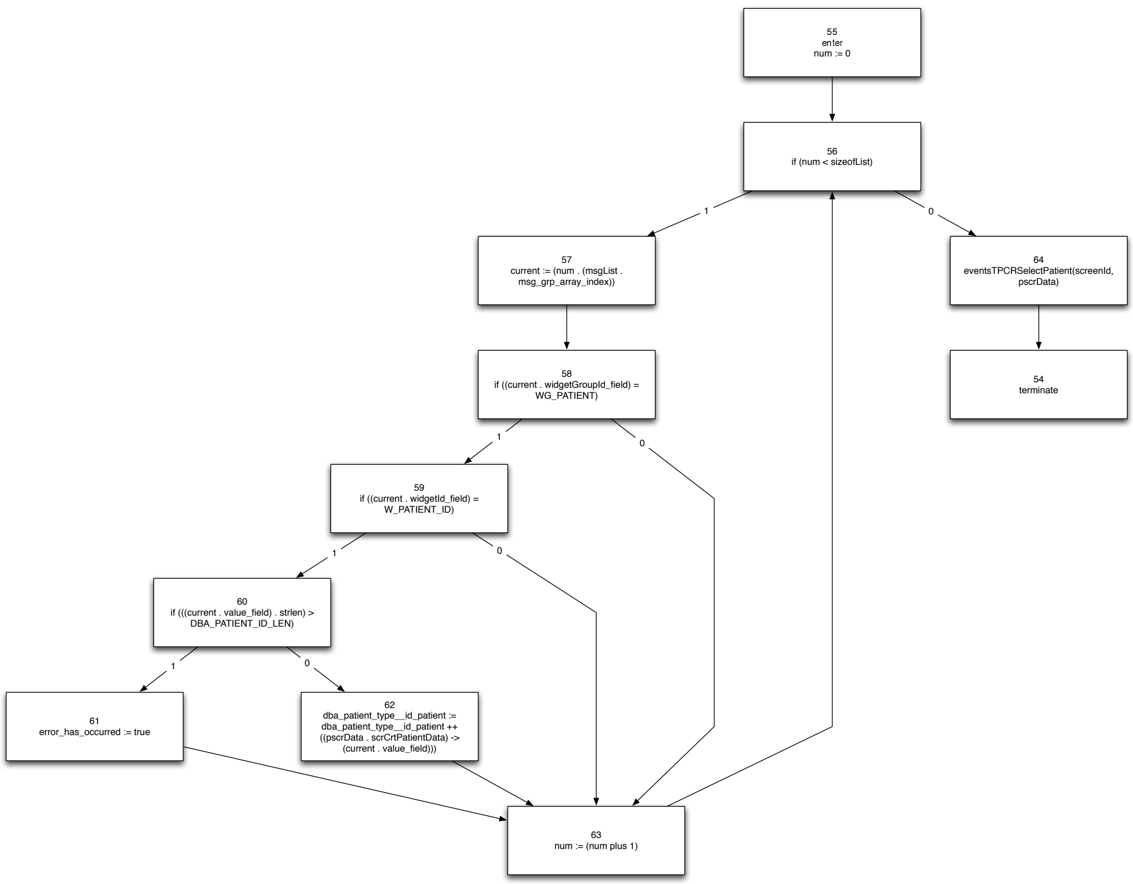

Figure 4-13: A diagramatic view of the resulting Forge program, showing the control flow between program nodes.

4.4.7 Human Burden: Abstraction & Translation

You cannot treat the software as a complete black box. If you know nothing about how your software works, then you simply cannot build a safety case for it.

In particular, the user must provide the analysis with abstraction information on what called procedures to include. A taghdiri-style refinement [84, 85] would remove the need to manually provide abstractions for some functions, but not all. Sometimes you need the human's high level understanding of the property being check to make that decision – such as ignoring error handling code or replacing function calls with constant return values to check behavior under specific contexts. The user may also provide global abstractions, such as eliminating boolean returns and conflating different integer types, which could not be inferred by a taghdiri-style iterative refinement.

Of course, an analysis that scale much better than the current Forge could analyze the software without any help from the human. However, such an analysis is not possible in the foreseeable future, so we are restricted to only analyzing programs that we can abstract – i.e. that we have a cursory understanding for.

4.4.8 Forge Analysis of Specification

Next we examine how one checks a constrain against a Forge encoding of a program. The fragment translated above is one pieces of the code relevant to the *dose delivery* concern, but is too small to be of interest by itself. That fragment, together with about a dozen other procedure definitions, describe the subset of the code related to how patient identity is received in a message from the GUI and stored in the Treatment Manager's heap.

In the language of the problem diagram, our constraint is as follows:

> If the patient select button has been pressed on the patient selection screen, then the id stored in the GUI is communicated to the treatment manager.

Before we can automatically check this claim, we need to map it into the language of the code base.

Guided by our original set of designations, we interpret this claim in the context of the treatment manager software. First, let's be precise about how the information is received, and what it means for the patient id to be correctly extracted from that information.

> A message is received as a pair of parameters, *data* and *arg*. The `data` parameter contains a message which is an array of identifiers. The 0^{th} slot of that array indicates the screen that was displayed when the message was generated. The 1^{th} slot indicates the button that was pressed to trigger the message. The `arg` parameter is a lump of data containing state information about the gui. Part of that lump of data is the identity of the patient being treated.

This spec is not yet precise enough for automatic analysis, but it *is* now expressed in the language of the program code base. In order to analyze the spec, we must now formalize it. Because the code manipulates a lot of structured data (pointers and objects), a relational logic is a good match for formally and intuitively expressing code properties. Using the Alloy language, the spec looks like this:

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

Figures 4-14 gives Java code that generates a forge expression equivalent to that Alloy expression. As written, that code performs a liveness check, not a safety check. It verifies that safe traces can occur, and generates sample safe traces, but does not check if all traces will be safe. Safety checks can also be performed, in an analogous manner. Liveness checks increase the confidence that the formal model matches the actual system. Safety checks increase confidence that the formal model has a desired property. Both are necessary to gain confidence that the actual system has a desired property.

Here is the safety check (the java that generates the forge that looks for bad behaviors). The analysis returns no counter-examples, increasing our confidence that the code obeys the desired assumption.

```
callingContext.newAssume(
  no_error.and(
  correct_result.not()).and(
  sensible_result)
);
```

It tells forge to solve for an execution in which no error message is generated, an incorrect result is returned, and the result is in a form that could be processed by the system. This represents a dangerous situation in which the machine invisibly delivers the wrong dose to a patient.

4.4.9 development process

A human developing a Forge model can follow the *counterexample driven precondition discovery* process – check the desired property against the Forge encoding of the code, find a counterexample, add a precondition (assumption about the problem context) to remove counterexample, repeat. All assumptions must then be verified by an appropriate specialist.

We followed this process and eventually found a set of reasonable (but undocumented) assumptions that made the checks pass. Most pertained to the format of received messages, the initial values of certain global variables defining the system mode, and the behavior of functions that are not defined in the code to which we had access. These assumptions were true of the current system, but since they were undocumented they could easily have been violated as changed were made to the system.

```
final ForgeExpression correct_result =
  current_id_patient
    .eq(arg.join(scrCrtPatientData)
    .join(dba_patient_type__id_patient))
  //arg screen data contains the final patient id

  .and(
    one.join(data.join(data__msg)
    .join(mixed_array_index))
    .eq(W_PATIENT_SELECT_BTN)
  ) //data input contains correct button id

  .and(
    zero.join(data.join(data__msg)
    .join(mixed_array_index))
    .eq(SCR_A1_PATIENT_SELECTION)
  ) //data input contains correct screen id
;

final ForgeExpression sensible_result =
  (current_id_patient).one()
  .and(original_id_patient
  .eq(current_id_patient).not());

final ForgeExpression no_error =
  error_has_occurred.eq(program.falseLiteral());
final SpecStmt postconditions =
  callingContext.newAssume(
    no_error
    .and(correct_result)
    .and(sensible_result)
  );
```

Figure 4-14: A Java program that generates a Forge spec to check that patient id's are correctly extracted from the patient selection message sent by the GUI.

4.5 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. The major current vulnerabilities we discovered for the BPTC are the following:

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 on 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 protect 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 deliver 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 many active patients and several 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 sending, 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 generation 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 messages the GUI 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 changes to the software running in the master control room 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. The last time that a room was added (room number three), it violated the emergency stop button's ability to halt the beam [68].

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

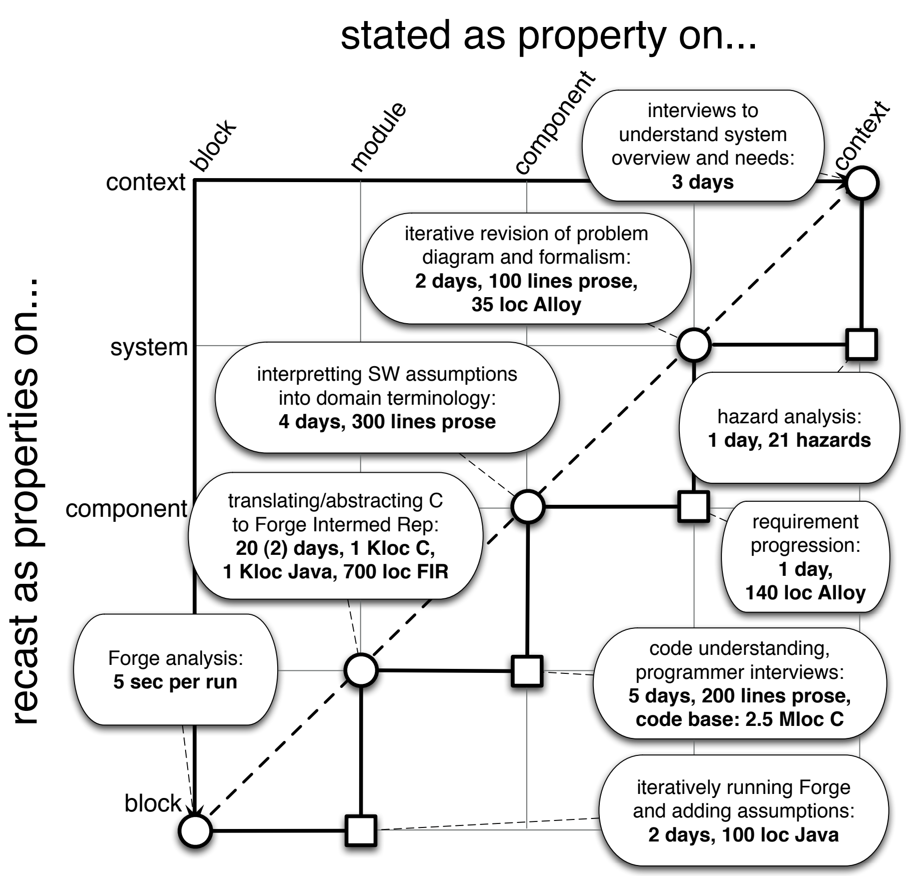

Figure 4-15: Time spent building the BPTC dependability argument for dose delivery.

4.5.1 Effort

Our analysis required about two months of person-time, counting both the time spent by our research group and the time spend by MGH employees. Figure 4.5 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 [23]. 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 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

The previous case study, and indeed requirement progressions in general, focuses on a traditional engineering notion of correctness – that the system generates correct values. However, many systems have additional types of correctness to consider besides fidelity, such as secrecy and auditability.

- A *fidelity* argument establishes a conventional engineering notion of correctness – that the system will generate well formed outputs and behaviors under normal operating conditions, and that it will fail safely and gracefully under abnormal conditions.
- A *secrecy* argument addresses security and anonymity concerns. It assures that the system protects sensative information from outsiders, even outsiders who know the design and implementation of the system.
- An *auditability* argument ensures that the working system is demonstrably correct in implementation, not just that the system is correct in theory. It provides a means by which outsiders can be confident that the system is operating correctly, and has not been replaced by a malicious or careless imitation.

In applications such as medical databases and political elections, secrecy and auditability are often considered to be of equal importance as fidelity. Legally, medical physicians are obliged not only to provide effective treatment but also to protect the patient's privacy, and technical systems involved in patient treatment and data must be certified by federal agencies, such as the FDA [61]. A voting system must not only count votes correctly, but it must also protect the anonymity of voters (to avoid coercion) and provide auditability (to avoid corrupted systems being installed in place of real ones).

Conventionally, these three types of arguments are built independently. In this Chapter, we describe a framework in which one can build these three types of arguments in tandem, which makes the overall analysis more systematic, less vulnerable to omissions, and less time-consuming to perform. The key to this framework is to build the fidelity and secrecy arguments using compatible lexicons, and then to build the auditability argument based on the intersection of those lexicons.

We illustrate our integrated approach on a the *Pret a Voter* cryptographic voting system developed at the Universities of Surrey of Newcastle [76].

5.1 Verifiable Voting

It is not enough for a voting system to be correct in a classic engineering sense of matching its specification at build-time. A leading concern in political voting systems is that of auditability [14, 82, 69] – the ability for users of the system (voters) to determine if the system works, rather than having to trust the system provider. Even if individual voters are not technically competent enough to assess the system, auditability allows them to choose whom they trust enlist a trusted auditor.

For simple systems, auditability is simply a matter of publishing the design documents for the system. A user can examine the design and confirm that the operating machine matches that design. However, secrecy concerns complicate this process. If certain information in the system is to be kept secret from users, then it becomes hard (or impossible) for users to assess if that system is operating according to its design. For example, in an non-secret election, auditability is simply a matter of making all voter's votes public, so that anyone can count them. However, once we decide that votes should be anonymous, such an audit is impossible without a much more sophisticated methodology.

We apply our technique to a cryptographic voting system proposed by Peter

Ryan in 2004, and currently being developed at the Universities of Surrey and Newcastle [76]. The system is called *Pret a Voter* – literally, "ready to vote" – and stands out among voting systems for providing auditability without compromising secrecy. It allows us to illustrate the suitability of our approach to the construction of fidelity arguments that integrate smoothly with secrecy and auditability arguments. The rest of this section describes the existing system and the intuition for why it should work. In the following sections, we apply our technique to building compatible fidelity, secrecy, and auditability arguments. In the last section of the chapter, we describe a method for leveraging the fidelity argument to help build a compatible secrecy argument.

5.1.1 Overview of the System

Prior to our analysis, the system designers had mathematically validated cryptographic properties about particular components and had an intuition for why those properties should collectively provide fidelity, secrecy, and auditability. There was no unifying system argument, and there were not precise definitions for what fidelity, secrecy, and auditability meant. The designers' intuitions proved to be very helpful in building the dependability argument, and the proven properties were indeed necessary to establish that argument. Our analysis confirms their intuition about the system, documents that intuition in precise and re-usabled fashion, and reveals some unstated (but reasonable) assumptions necessary for system correctness.

5.1.2 Flow of a Vote

Figure 5-1 shows a sample ballot that has been used to cast a vote for Candidate B.[1] Figure 5-2 illustrates the path of a vote through the system, and is narrated below.

A voter receives a ballot and takes it to a private voting booth. The ballot displays a list of candidates, with check boxes next to them. The voter checks the box next to one of the candidates, then tears off the list of boxes (along a perforation). The

[1] The system also works for ranked and multiple-vote voting systems, but for simplicity we have describe it as applied to a single-approval election (e.g. a presidential election in the U.S.).

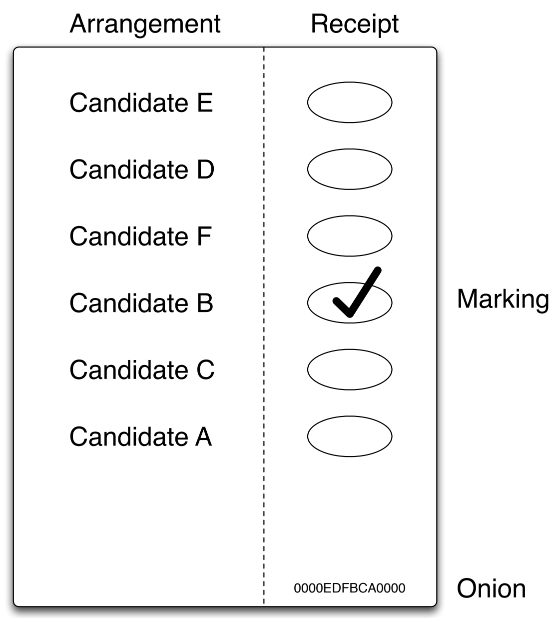

Figure 5-1: What the ballot looks like to a voter, who has just cast a vote for Candidate B.

list of candidates is discarded, and the voter turns in just the "receipt" – the list of check boxes, one of which is marked. At the bottom of the receipt is an "onion" which encodes the order in which the candidate names appeared on the ballot. It's encrypted, so no one without the private key can tell who the voter voted for.

The receipt is turned into the Voting Board, along with everyone else's votes for the day. The ballot receipt that a voter carries over to the ballot box is just a check on a blank page. Since different ballots used different candiate orderings, there is no indication which candidate the check corresponds to. The onion at the bottom of the page encodes that information, but it can only be deciphered with a private key, which is not available to observers.

The receipts then undergo a series of re-encryption steps, each of which changes the onion on each receipt to look different to the eye but to still represent the same ordering of candidate names. The vote recorded on the receipt is not changed, but voter anonymity is preserved – without the secret key, it is impossible to tell which record coming out of the machine corresponds to which receipt going into the machine. The re-encrypted receipts are then decrypted into voting records. Each voting record

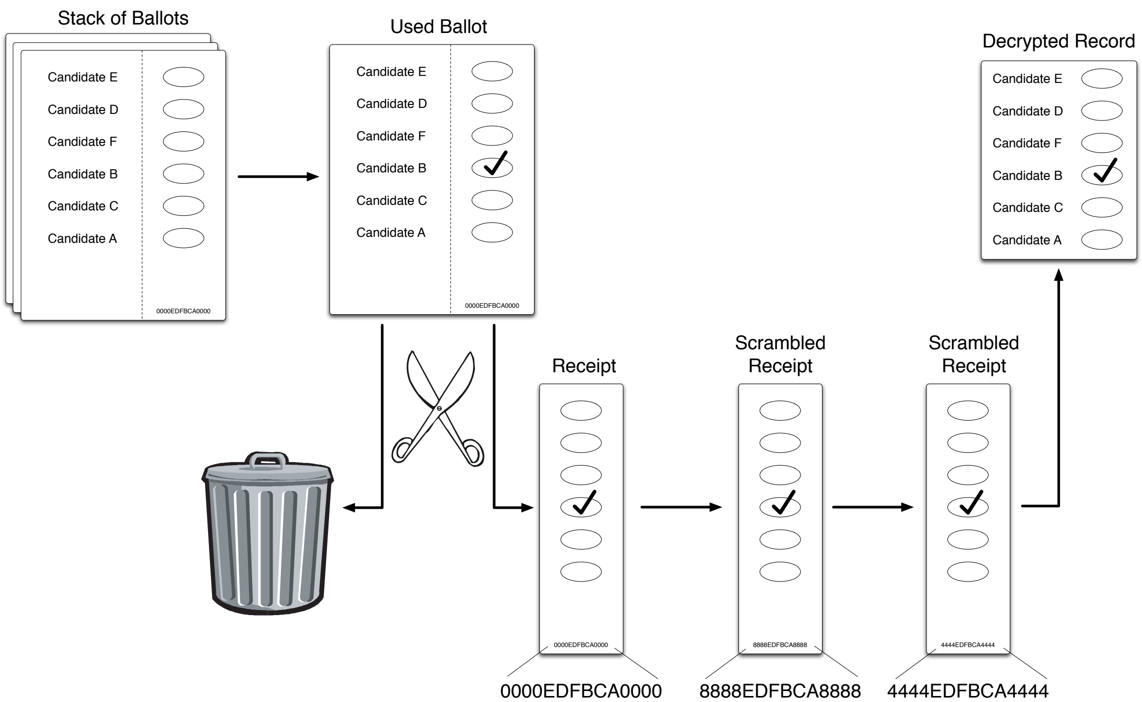

Figure 5-2: The flow of a vote through the system.

indicates the candidate ordering (from the decrypted onion) and the checked box next to one of them (from the voter). The voting records can be read by a human or a machine and tallied to determine the winning candidate.

5.2 Representing the Problem

The first step to building a dependability argument is to provide a set of desigantions, domains, and definitions. These artifacts serve to bridge the inescapable gap between the informal world of the actual system and the formal world of documented requirements [40]. Designations connecting our formalism to the informal problem domain are given in Figure 5-4. Supporting definitions are given in Figure 5-5, and supporting domains are given in Figure 5-3. Figure 5-6 gives an Alloy model that precisely records the structure of those phenomena and domains, and that model is narrated in Figure 5-7. [2]

[2] An elaboration of that model, with more extensive comments, is integrated into the fidelity model, which is given in Appendix 11 and introduced in Section 5.3.

The Alloy model declares sets and relations to represent the relevant phenomena. At this point, the constraints and assumptions about how these phenomena relate are not represented, just the problem structure. Using these phenomena (as they are formalized in the Alloy model), we build the Problem Frame context diagram shown in Figure 5-8. It shows how the phenomena relate to the domains and which phenomena are shared between which domains.

$c \in$ `Candidate`	$\Leftrightarrow$	c is a person running in this election.
$v \in$ `Voter`	$\Leftrightarrow$	v is a person who is capable of voting.
$b \in$ `Ballot`	$\Leftrightarrow$	b is a ballot – a piece of paper with a list of candidate names, attached to a receipt.
$r \in$ `Receipt`	$\Leftrightarrow$	r is a receipt – a list of checkboxes attached to a ballot.
$o \in$ `Onion`	$\Leftrightarrow$	o is an onion – a cryptographic string representing an ordering of candidates.
$r \in$ `Record`	$\Leftrightarrow$	r is a record – a piece of paper with a list of candidates and a check next to one of them.
$vb \in$ `Board`	$\Leftrightarrow$	vb is the voting board – the device that re-encrypts receipts, and then decrypts them into records.

Figure 5-3: Domains for verifiable voting

$(c \rightarrow i) \in$ `score`	$\Leftrightarrow$	Candidate c has been assigned i votes. The candidate with the most votes wins.
$(v) \in$ `RegisteredVoter`	$\Leftrightarrow$	Voter v is authorized to vote in this election.
$(b \rightarrow p \rightarrow c) \in$ `ballotArrangement`	$\Leftrightarrow$	On ballot b, candidate c's name appears at position p.
$(b \rightarrow r) \in$ `ballotReceipt`	$\Leftrightarrow$	Receipt r was originally attached to ballot b.
$(v \rightarrow b) \in$ `voterBallot`	$\Leftrightarrow$	Voter v was given ballot b to use to vote.
$(v \rightarrow c) \in$ `intention`	$\Leftrightarrow$	Voter v wants candidate c to win the election.
$(r \rightarrow p) \in$ `receiptMarked`	$\Leftrightarrow$	Receipt r is marked at position p.
$(r \rightarrow o) \in$ `receiptOnion`	$\Leftrightarrow$	Onion o is written at the bottom of receipt r.
$(o \rightarrow p \rightarrow c) \in$ `onionArrangement`	$\Leftrightarrow$	According to onion o, candidate c's name appears at position p.
$(d \rightarrow p \rightarrow c) \in$ `recordArrangement`	$\Leftrightarrow$	On record d, candidate c's name appears at position p.
$(r \rightarrow p) \in$ `recordMarked`	$\Leftrightarrow$	Record d is marked at position p.
$(r \rightarrow d) \in$ `mix`	$\Leftrightarrow$	Record d is generated by the voting board as the result of re-encrypting receipt r.

Figure 5-4: Designations for verifiable voting

$(b \rightarrow c) \in$ `ballotCandidate`	$\Leftrightarrow$	Ballot b indicates a vote for candidate c, according to the arrangement of candidates on b and the marking on b's receipt.
$(r \rightarrow c) \in$ `receiptCandidate`	$\Leftrightarrow$	Receipt r indicates a vote for candidate c, according to the arrangement of candidates defined by r's onion and r's marking.
$(d \rightarrow c) \in$ `recordCandidate`	$\Leftrightarrow$	Record d indicates a vote for candidate x, according to the arrangement of candidates on d and d's marking.

Figure 5-5: Definitions for verifiable voting.

```
sig Candidate { score: one Int }
sig Voter {
  intention: lone Candidate,
  voterBallot: set Ballot,
}
sig RegisteredVoter extends Voter {}
sig Ballot {
  ballotArrangement: Position one → one Candidate,
  ballotReceipt: one Receipt,
  ballotCandidate: lone Candidate,
}{ ballotCandidate = ballotReceipt.receiptMarked.ballotArrangement }
sig Position {}
sig Receipt {
  receiptOnion: one Onion,
  receiptMarked: lone Position,
  receiptCandidate: lone Candidate
}{ receiptCandidate = receiptMarked.(receiptOnion.onionArrangement) }
sig Onion { onionArrangement: Position one → one Candidate }
sig Record {
  recordArrangement: Position one → one Candidate,
  recordMarked: lone Position,
  recordCandidate: lone Candidate
}{ recordCandidate = recordMarked.recordArrangement }
one sig Board { scramble: Receipt lone → lone Record }
fun mix[] : Receipt → Record { Board.scramble }
```

Figure 5-6: An Alloy model formalizing the designations, definitions, and domains defining the problem context for a voting system. This model is narrated in Figure 5-7.

Line 1: There is a set of candidates. Each `Candidate` has an (integer) score, representing that candidate's score at the end of the election process.

Lines 2-6: Each `Voter` has an `intention` to vote for zero or one `Candidate` and is given a set of zero or more `Ballots`. A subset of the `Voters` are `RegisteredVoters`.

Lines 7-11: Each `Ballot` has an `arrangement` that determines an ordering on candidate names (a mapping from each `Position` to one `Candidate` and vice versa). A `Ballot` is attached to a single `Receipt`. The relation `ballotCandidate` is constrained to match the definition given in Figure 5-5, and represents the intended interpretation of the ballot.

Lines 12-17: Each `Receipt` is `marked` at zero or one `positions`. It also bears an `Onion`. The relation `receiptCandidate` is constrained (in an appended fact) to match the definition given in Figure 5-5.

Line 18: Each `Onion` encodes an `arrangement` – an ordering of candidate names. Like the arrangement on a ballot, it is represented as a bijection between `Positions` and `Candidates`.

Lines 19-23: Each `Record` has an `arrangement` and is `marked`, just like `Onions` and `Receipts`. The relation `recordCandidate` is constrainted to match the definition given in Figure 5-5.

Lines 24-25: The voting `Board` relates each receipt with zero or one `Records`, and vice versa. The `mix` function mirrors that relation, in a form that is more convenient for modeling.

Figure 5-7: A narration of the Alloy model given in Figure 5-6.

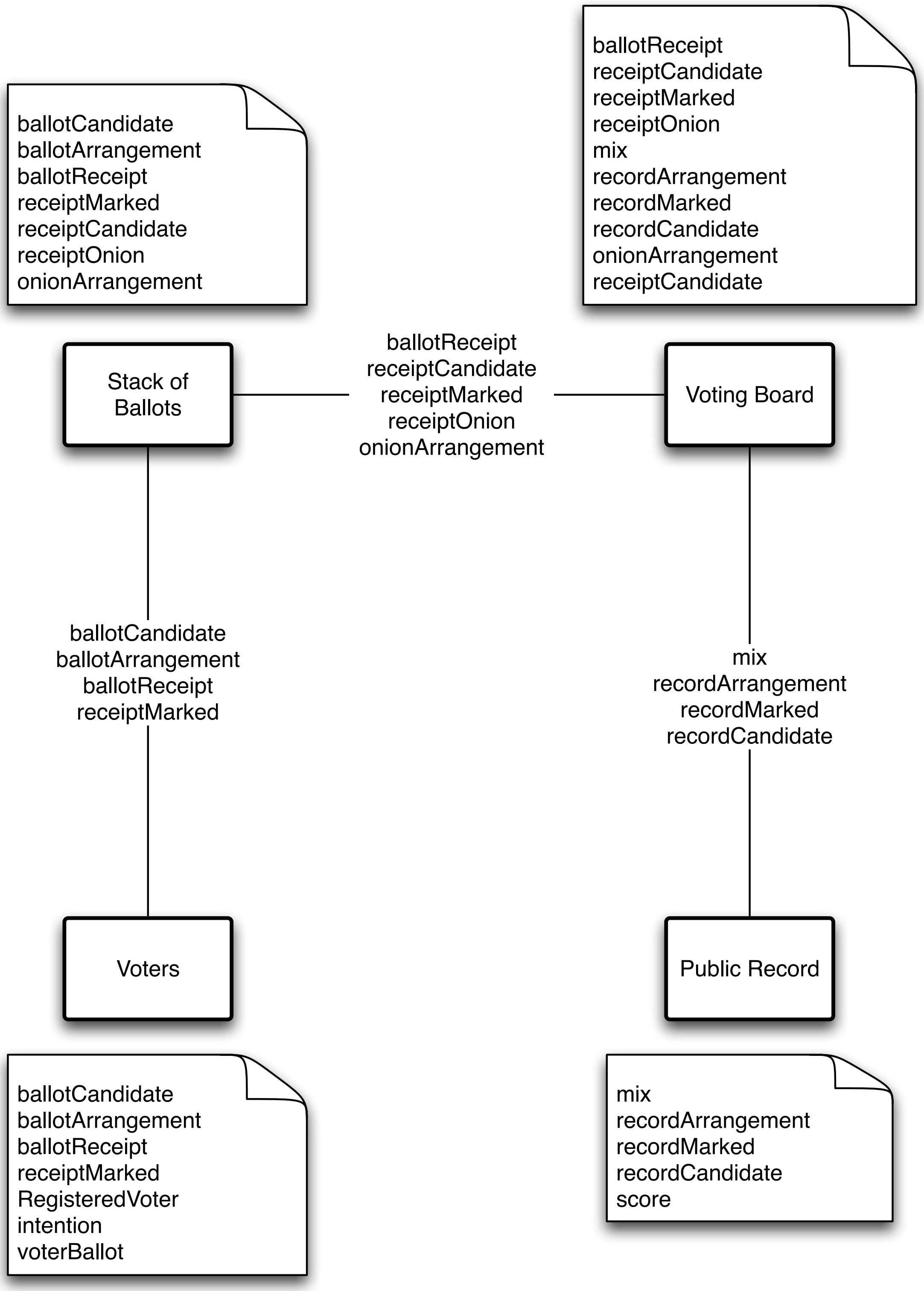

Figure 5-8: Context diagram for the voting example, showing the phenomena known to each domain, and thus which phenomena are shared between each domain.

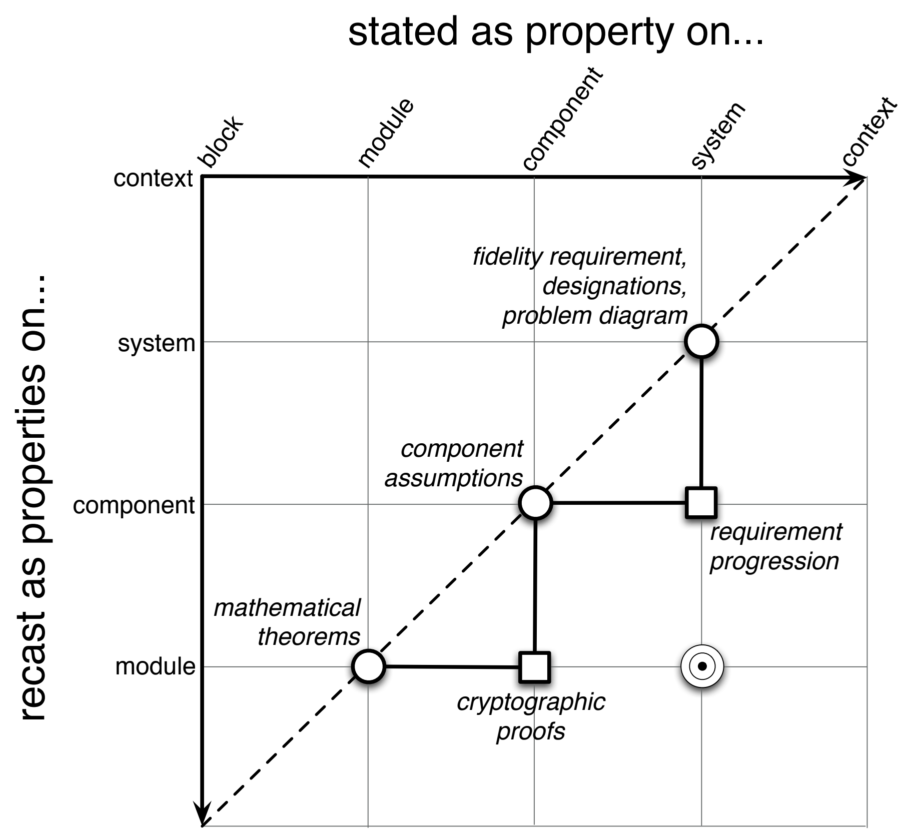

Figure 5-9: A CDAD showing how requirement progression and cryptographic proofs link together to build a composite argument.

5.3 Fidelity Goal

We approach the fidelity goal with requirement progression, as seen in our previous examples. The resulting breadcrumbs were discharged (by Peter Ryan and his collegues) using cryptographic proofs. The CDAD for this (quite simple) composition is shown in Figure 5.2.

5.3.1 Formalization of the Requirement

We begin by translating the informal fidelity requirement into precise language and then into a formal language (Alloy).

informal

The public record reflects the ballots cast.

precise

For each candidate, the number of votes the public records shows for that candidate is the same as the number of registered voters who marked their ballots for that candidate.

formal

```
all c: Candidate |
  c.score = #(RegisteredVoter & intention.c)
```

For each candidate `c`, `c`'s score should be the number of voters who are both registered and intended to vote for `c`.

5.3.2 Requirement Progression for Fidelity Goal

First, we sketch out the shape we expect the argument to take, as shown in Figure 5-10. This sketch guides progression, suggesting what breadcrumbs and pushes will be helpful. With that intuitive argument flow in mind, we begin requirement progression formally. Figure 5-11 shows the problem diagram with the formal constraint, ready for requirement progression to begin.

The initial goal is the requirement that the public record reflect the number of candidates who intended to vote for each candidate. It references phenomena from both the `Voters` and `Public Record` domains, although those domains are only indirectly related to each other. As such, progression is needed to decompose the requirement into localized domain assumptions. s Figures 5-12 to 5-21 show the steps of the requirement progression. Initially, the requirement connects to both the `Voters` and `Public Record` Domains. In a series of transformations, we shift the left arc from `Voters` to `Stack of Ballots` to `Voting Board` and finally to `Public Record`.

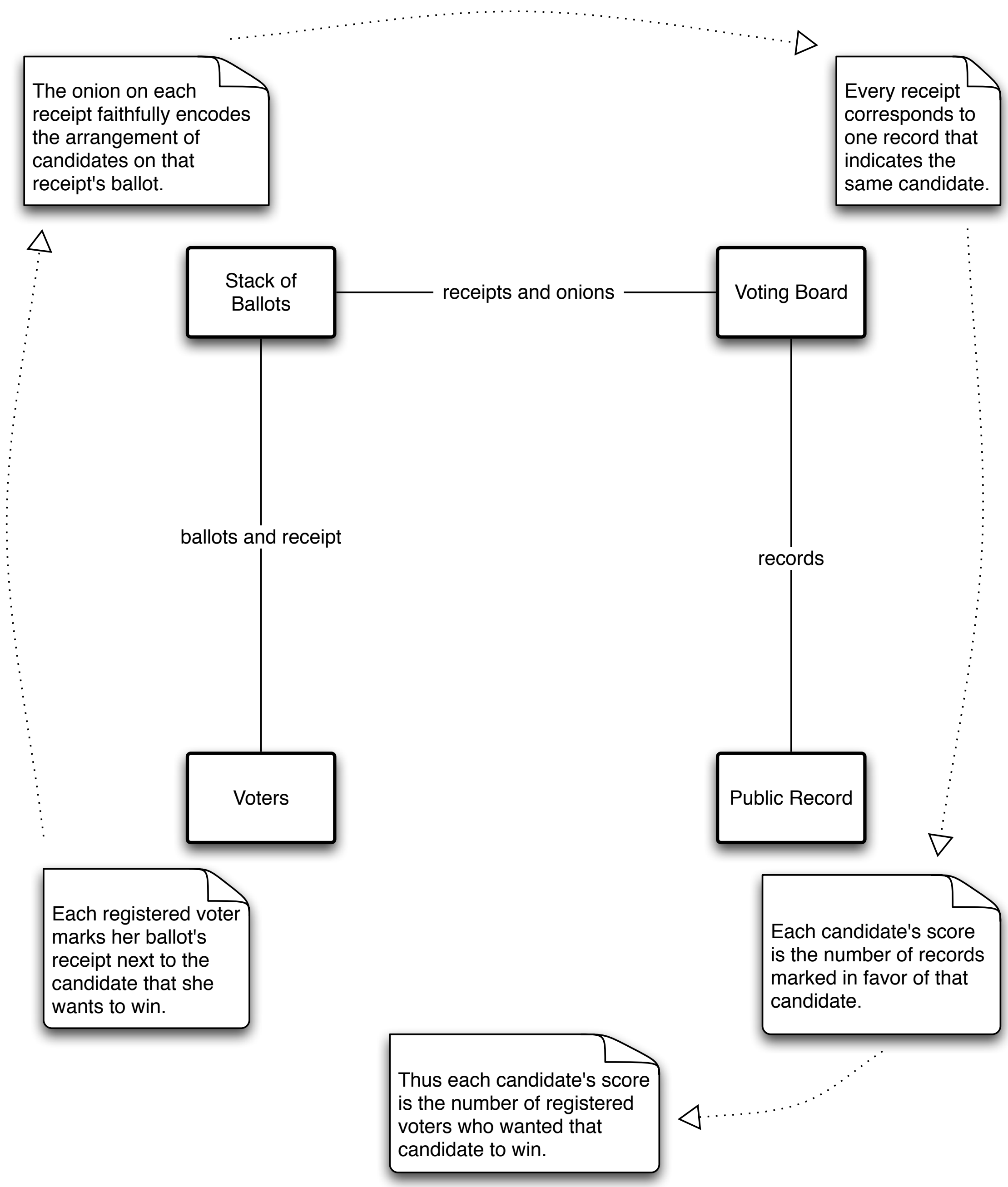

Figure 5-10: An informal sketch of how we expect the argument diagam to look, what sorts of breadcrumbs will be helpful, and how the argument will flow.

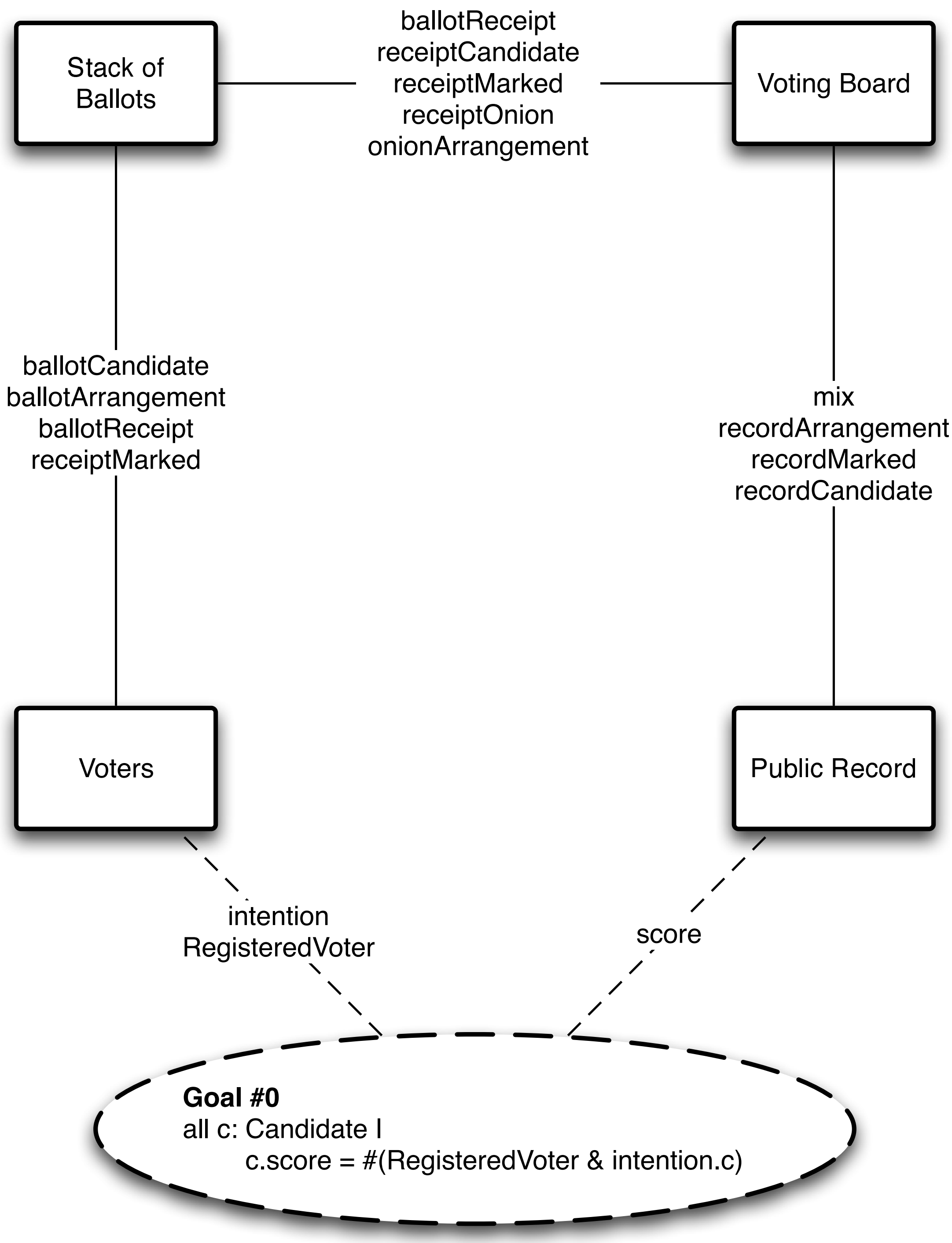

Figure 5-11: Problem diagram for the voting example, showing the fidelity requirement (`Goal0`), which relates the `Voters` and `Public Record` domains.

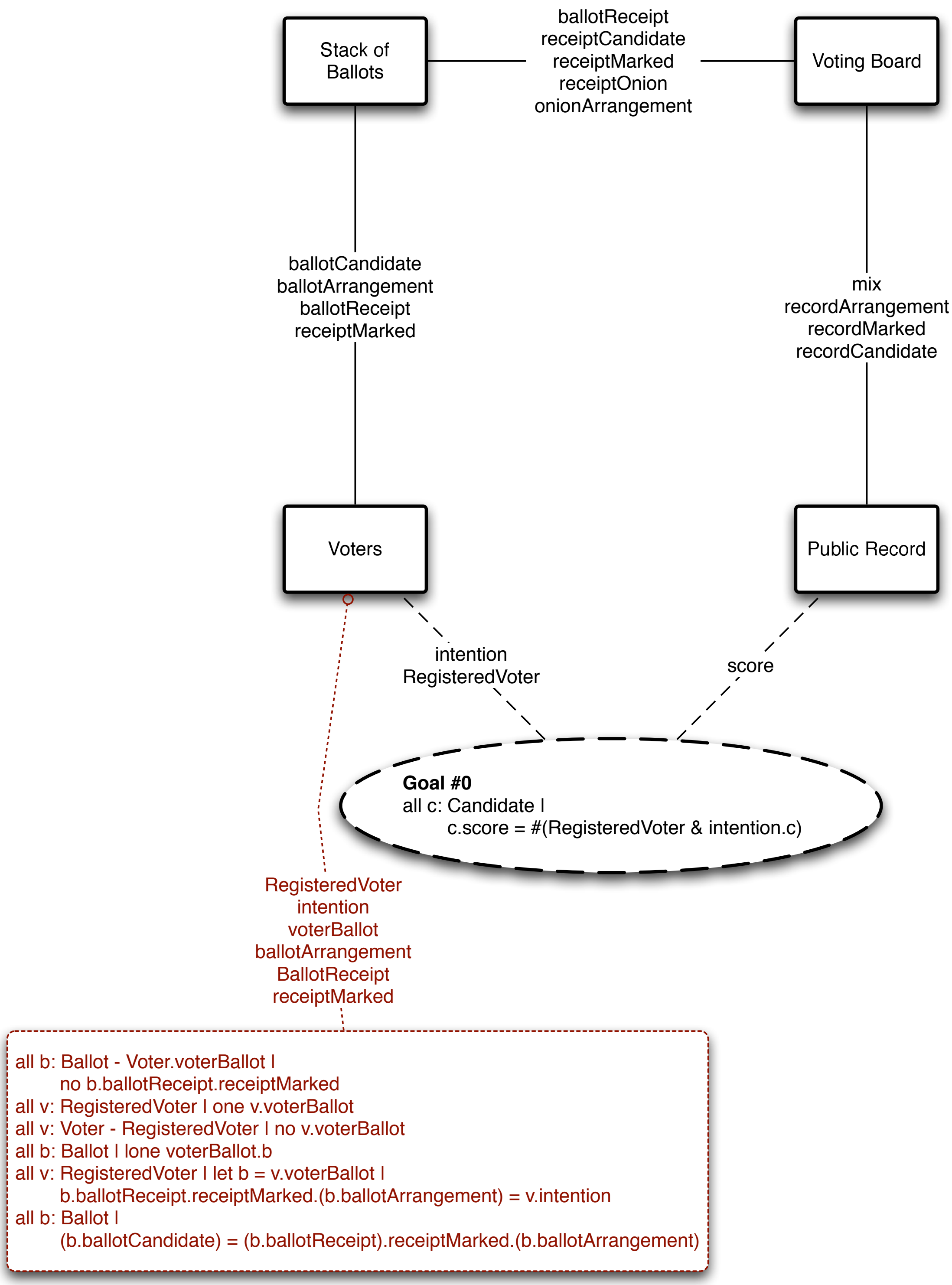

Figure 5-12: Step 1, Part 1. Adding a breadcrumb to the `Voters` domain.

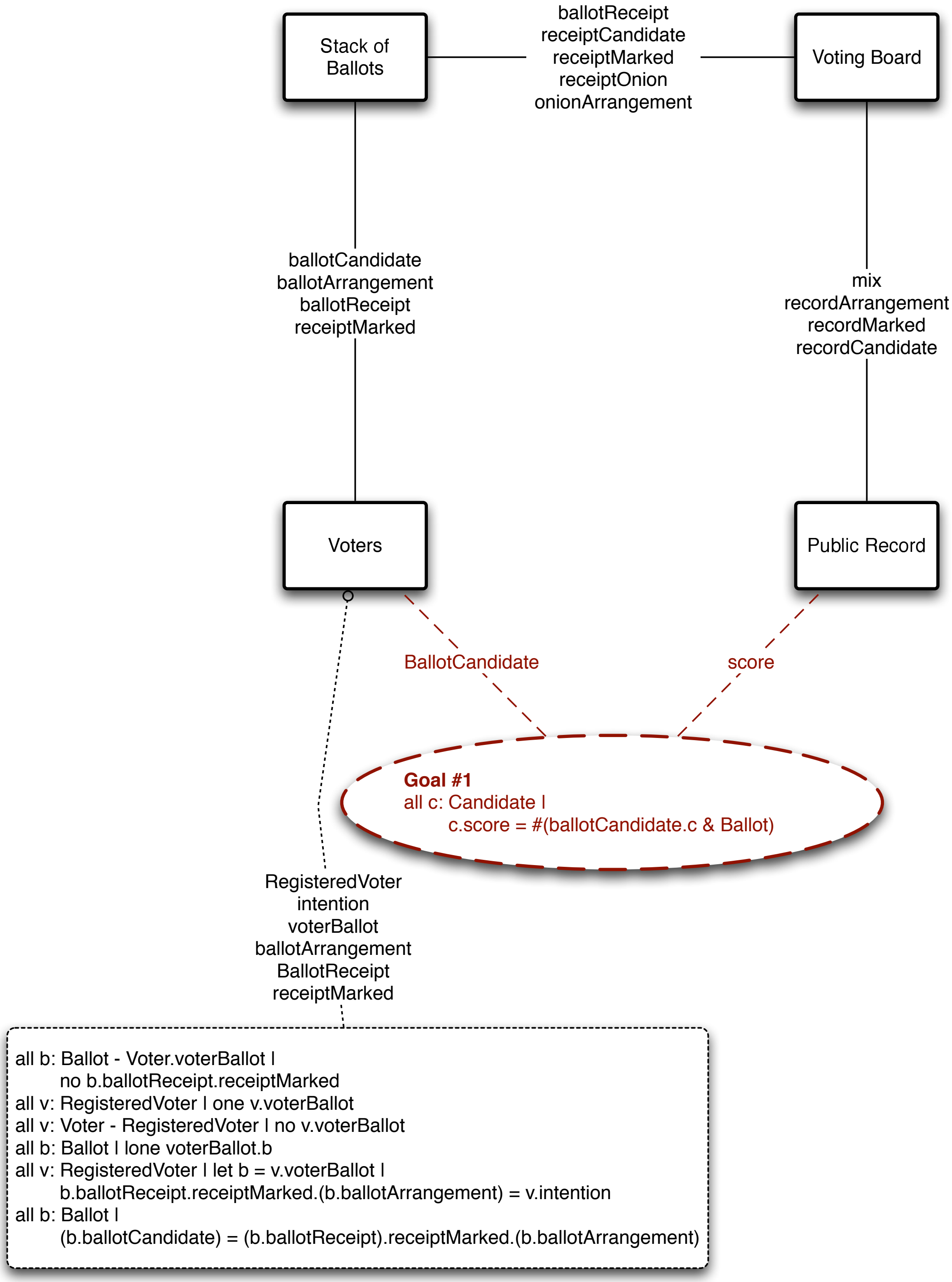

Figure 5-13: Step 1, Part 2. Rewriting the goal to reference different phenomena.

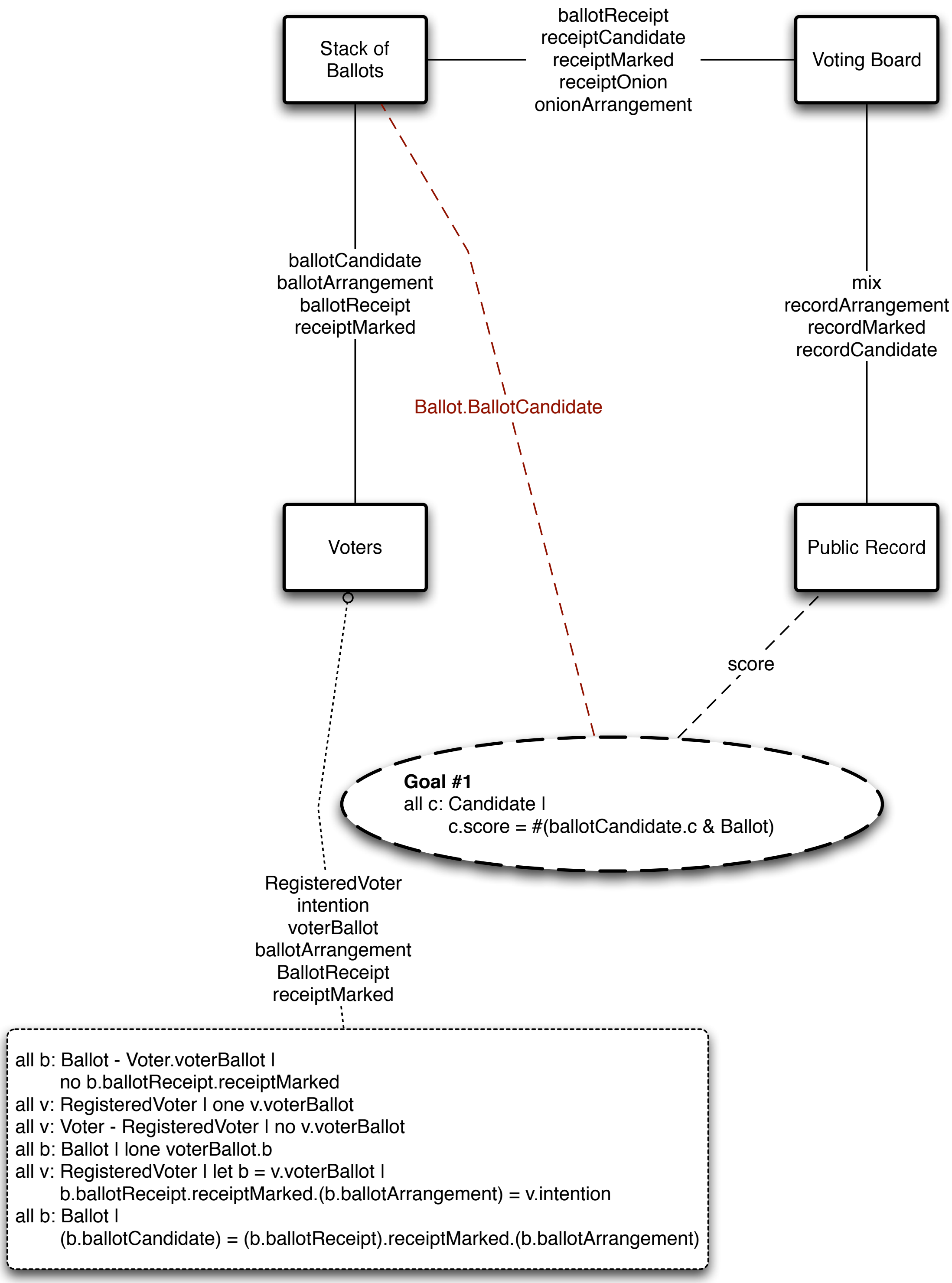

Figure 5-14: Step 1, Part 3. Pushing the goal from the `Voters` domain to the `Stack of Ballots` domain.

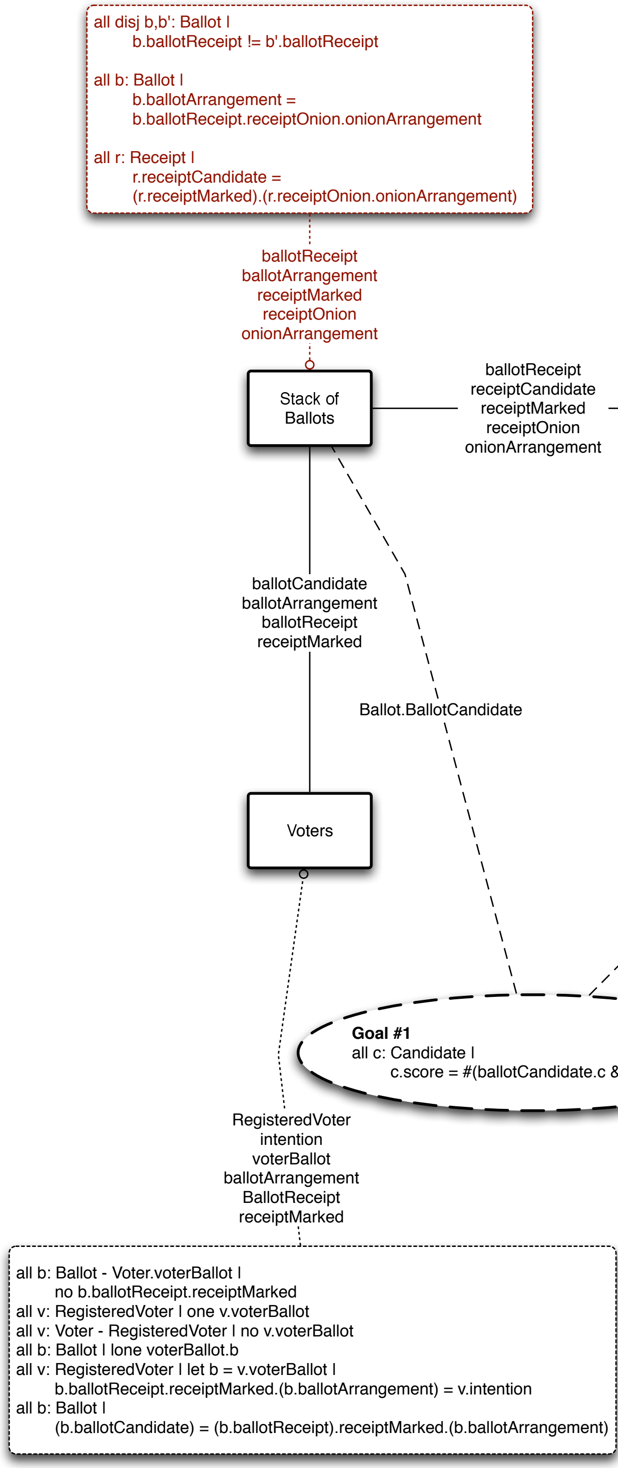

Figure 5-15: Step 2, Part 1. Adding a breadcrumb to the `Stack of Ballots` domain.

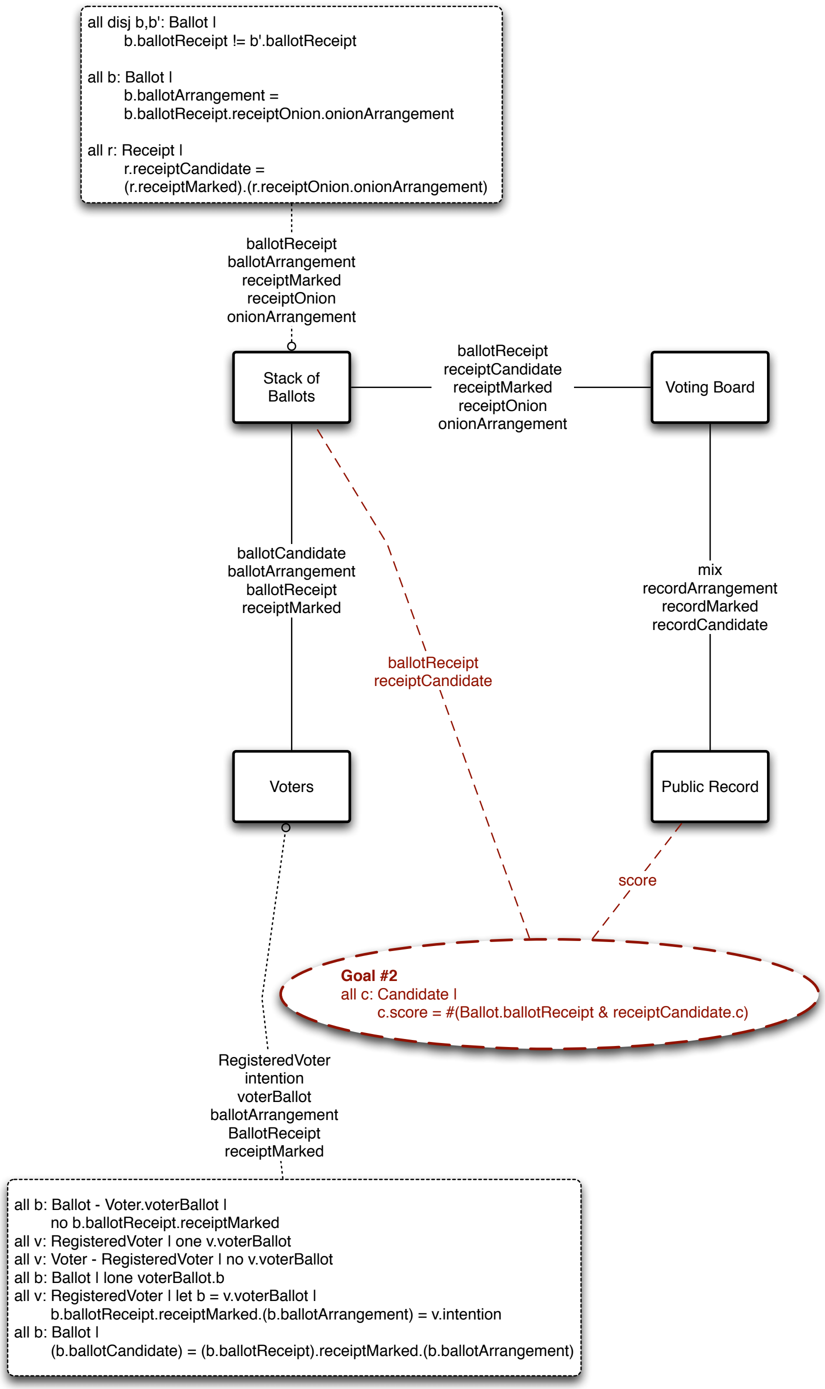

Figure 5-16: Step 2, Part 2. Rewriting the goal to reference different phenomena.

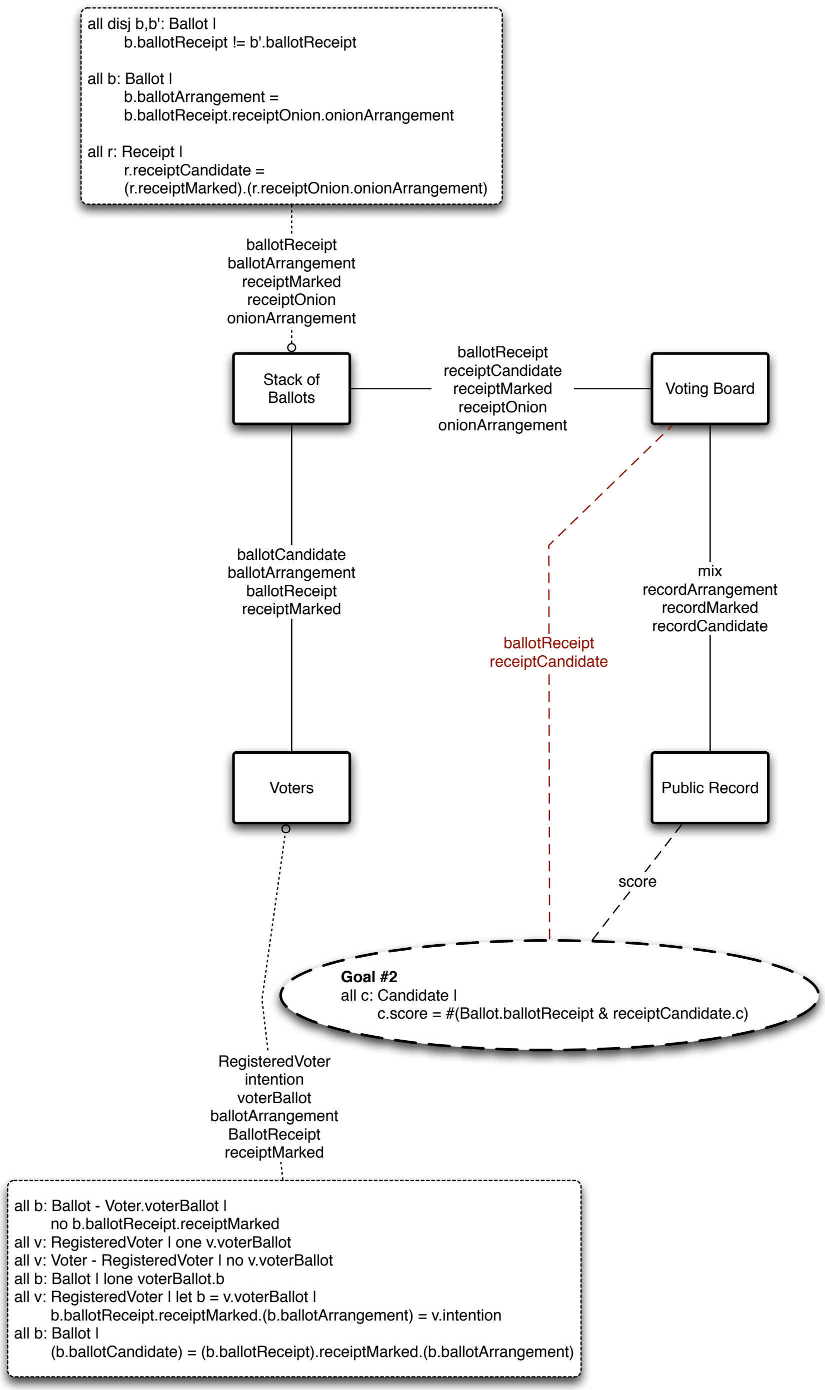

Figure 5-17: Step 2, Part 3. Pushing the goal frrom the `Stack of Ballots` domain to the `Voting Board` domain.

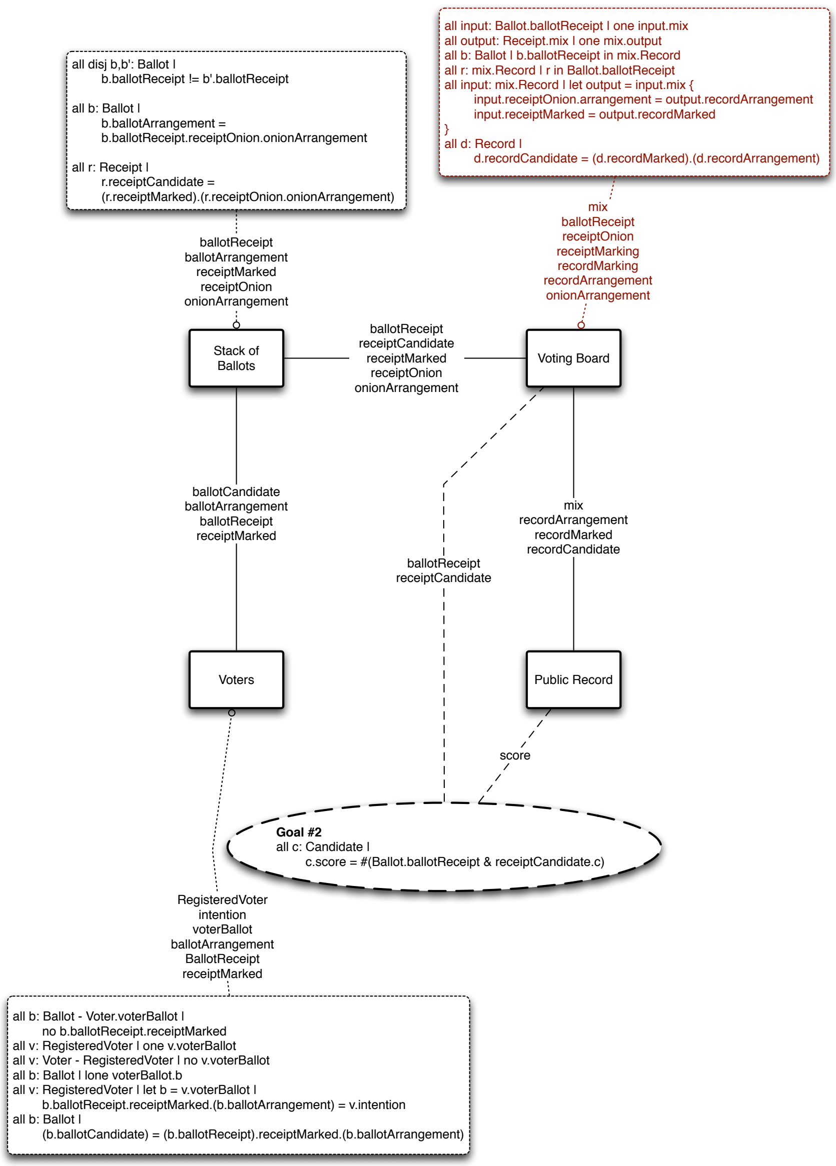

Figure 5-18: Step 3, Part 1. Adding a breadcrumb to the `Voting Board` domain.

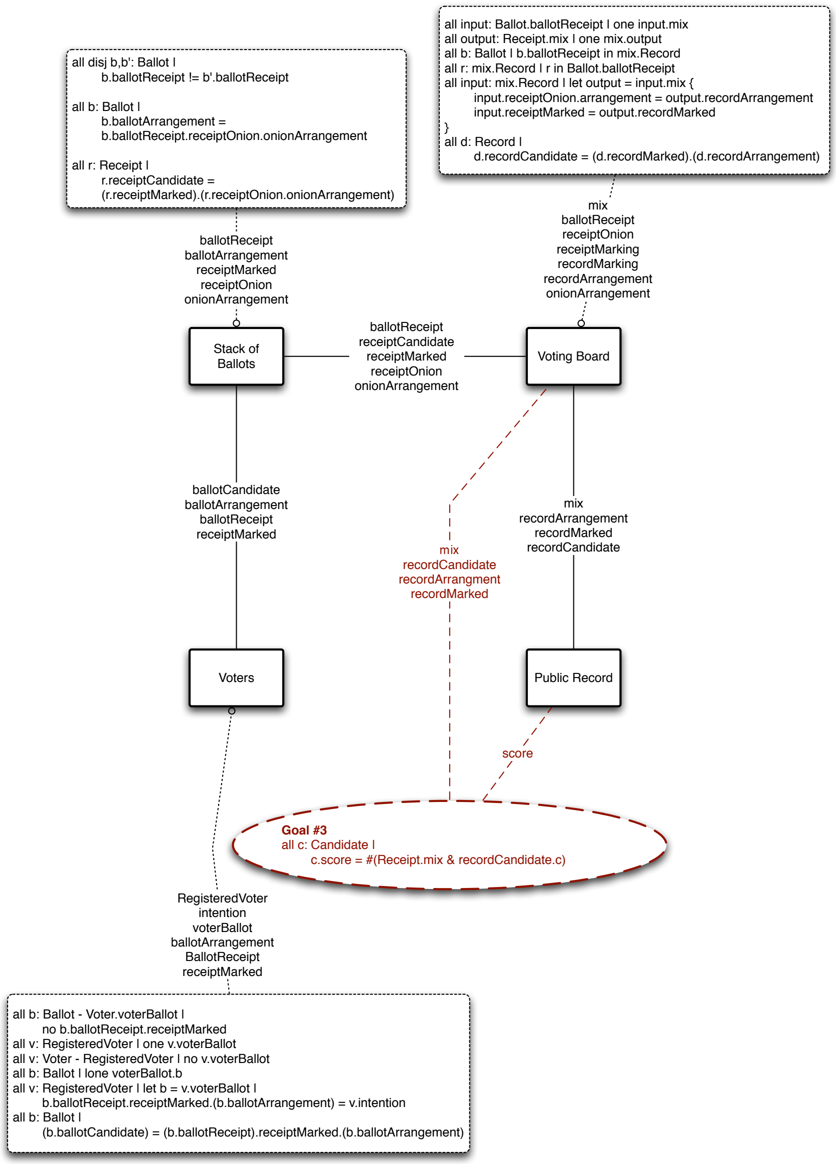

Figure 5-19: Step 3, Part 2. Rewriting the goal to reference different phenomena.

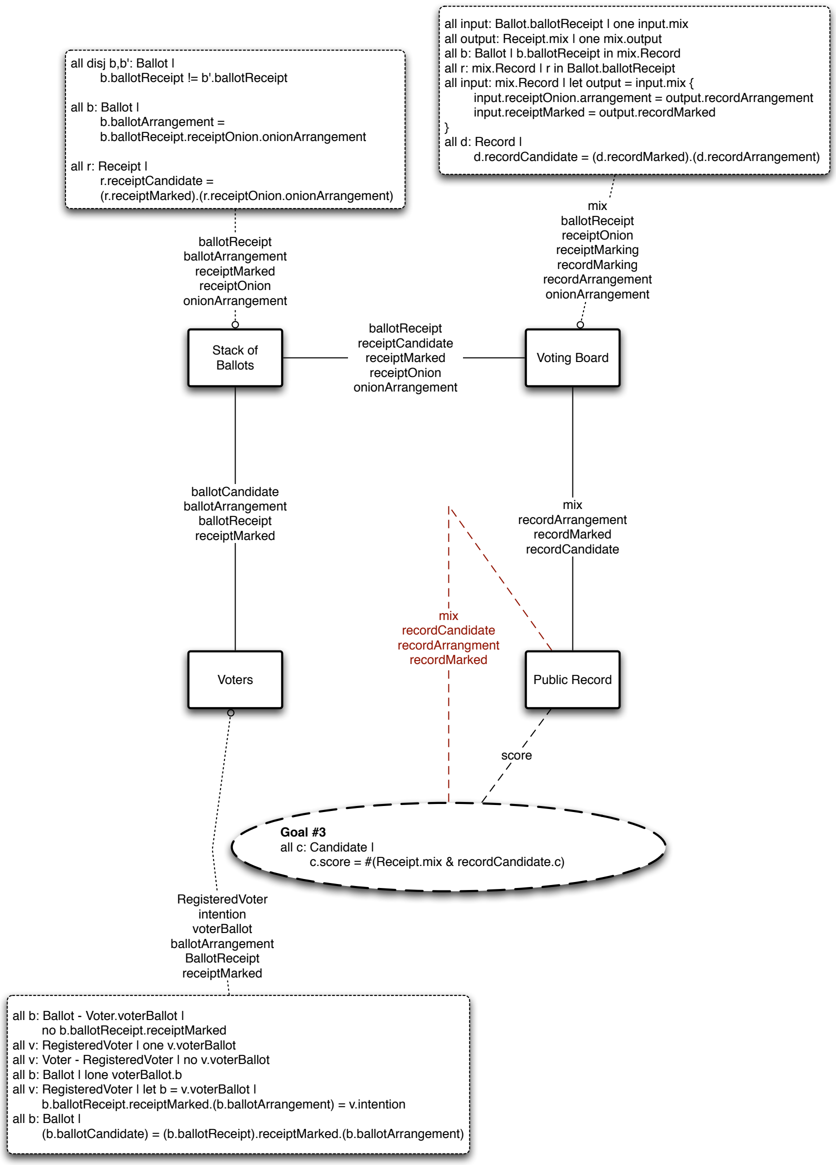

Figure 5-20: Step 3, Part 3. Pushing the goal from the `Voting Board` domain to the `Public Record` domain.

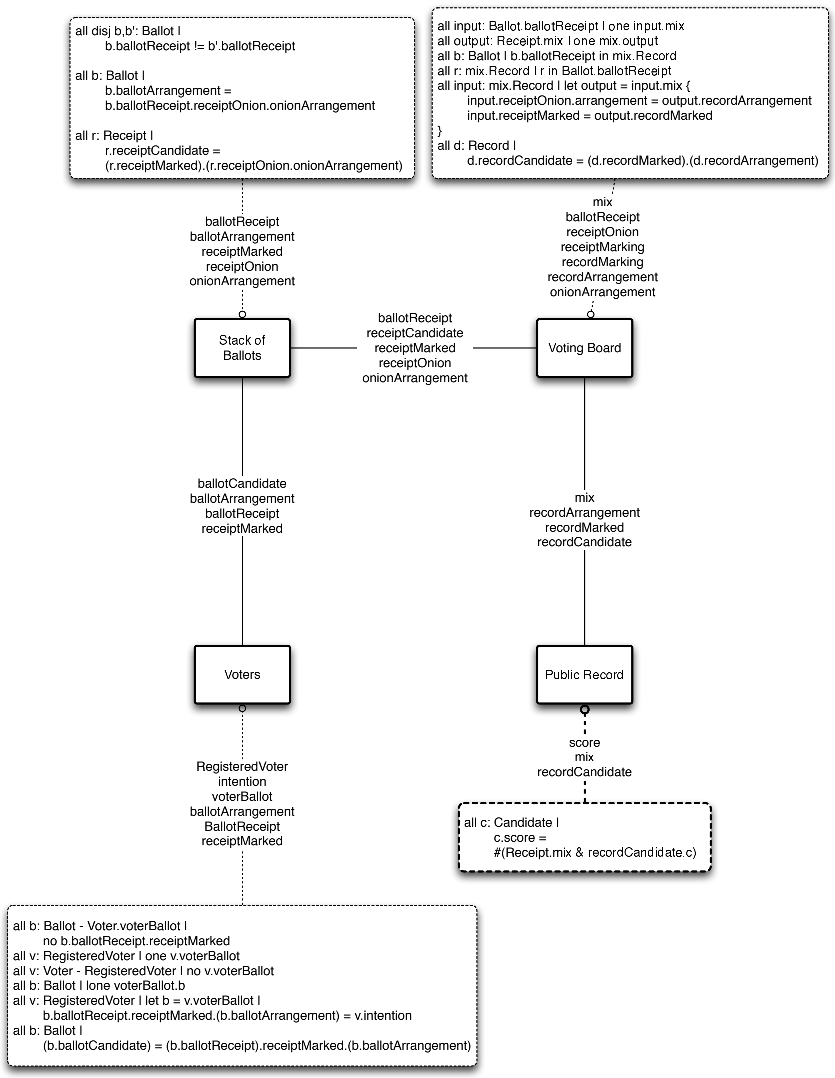

Figure 5-21: Cleaning up the final argument diagram. Each assumption references only a single domain, so progression is complete.

We extend the earlier Alloy model to check the requirement progression rewrite steps. The initial goal is the system requirement; each candidate's final score is equal to the number of registered voters who intended to vote for that candidate.

```
pred Goal0 [] {
  all c: Candidate |
    c.score = #(RegisteredVoter & intention.c)
}
```

Our first rewrite replaces the expression "registered voters who intended to vote for that candidate" to instead say "candidates marked on the ballots".

```
pred Goal1 [] {
  all c: Candidate |
    c.score = #(ballotCandidate.c & Ballot)
}
```

To justify this rewrite, we have to justify the fact that the markings on the ballots reflect the intentions of the registered voters (and only registered voters). These constraints are added as a breadcrumb on Voters.

```
1 pred VoterBreadcrumb [] {
2   all b: Ballot - Voter.voterBallot | no b.ballotReceipt.receiptMarked
3   all v: RegisteredVoter | one v.voterBallot
4   all v: Voter - RegisteredVoter | no v.voterBallot
5   all b: Ballot | lone voterBallot.b
6   all v: RegisteredVoter | let b = v.voterBallot |
7     b.ballotReceipt.receiptMarked.(b.ballotArrangement) = v.intention
8 }
```

These constraints can be interpreted as follows:

Line 2: ballots not given to voters are not marked,

Line 3: every registered voter gets exactly one ballot,

Line 4: non-registered voters get no ballots,

Line 5: each ballot is given to only one voter, and

Lines 6-7: the ballots given to registered voters are marked to reflect that voter's intention.

We check that this breadcrumb is sufficiently strong to enforce the rewrite, by showing that the breadcrumb conjoined with the new goal logically imply the prior goal.

```
assert partialClaim1 {
  VoterBreadcrumb and Goal1 ⇒ Goal0
}
check partialClaim1 expect 0
```

The check passes, so we proceed to the next rewrite. The second rewrite replaces the expression “candidates marked on the ballots” to instead say “candidates marked on the receipts of the ballots”.

```
pred Goal2 [] {
  all c: Candidate |
    c.score = #(Ballot.ballotReceipt & receiptCandidate.c)
}
```

To justify this rewrite, we add a breadcrumb to Ballots. It articulates why it is ok to refer to ballot receipts instead of the ballots themselves.

```
1 pred BallotBreadcrumb [] {
2   all disj b,b': Ballot |
3     b.ballotReceipt != b'.ballotReceipt
4   all b: Ballot |
5     b.ballotArrangement = b.ballotReceipt.receiptOnion.onionArrangement
6 }
```

These constraints can be interpreted as follows:

Lines 2-3: no two ballots have the same receipt, and

Line 4-5: the arrangement or candidate names on each ballot is the same as the arrangement encoded in the onion on that ballot’s receipt.

We check that this breadcrumb is sufficiently strong to enforce the rewrite, by showing that the breadcrumb conjoined with the new goal logically imply the prior goal.

```
assert partialClaim2 {
  BallotBreadcrumb and Goal2 ⇒ Goal1
}
check partialClaim2 expect 0
```

The check passes, so we proceed to the next rewrite. The third rewrite replaces the expression “candidates marked on the receipts of the ballots” to instead say “candidates marked on the records that result from feeding the ballot receipts into the voting board scrambler”.

```
pred Goal3 [] {
  all c: Candidate |
    c.score = #(Receipt.mix & recordCandidate.c)
}
```

To justify referencing processed records instead of unprocessed receipts, we add a breadcrumb to the Voting Board.

```
1  pred BoardBreadcrumb[] {
2    all input: Ballot.ballotReceipt | one input.mix
3    all output: Receipt.mix | one mix.output
4    all b: Ballot | b.ballotReceipt in mix.Record
5    all r: mix.Record | r in Ballot.ballotReceipt
6    all input: mix.Record | let output = input.mix {
7      input.receiptOnion.onionArrangement = output.recordArrangement
8      input.receiptMarked = output.recordMarked
9    }
10 }
```

These constraints can be interpreted as follows:

Line 2: no records are created except those that correspond to receipts,

Line 3: one record is generated for each receipt sent into the voting board,

Line 4: all receipts from ballots are sent to the voting board,

Line 5: only receipts from ballots are sent to the voting board, and

Lines 6-8: the record that result from a receipt indicates the same candidate (has the same arrangement and marking).

We check that this breadcrumb is sufficiently strong to enforce the rewrite, by showing that the breadcrumb conjoined with the new goal logically imply the prior goal.

```
assert partialClaim3 {
  BoardBreadcrumb and Goal3 ⇒ Goal2
}
check partialClaim3 expect 0
```

The check passes, so we are done. Together, the breadcrumbs are sufficiently strong to justify replacing the original goal (`goal0`) with the final goal (`goal3`).

The full Alloy model, with additional liveness simulations and inlined comments, is given in the Appendix 11.

5.4 Secrecy Goal

It is not enough for a voting system to correctly tally votes. It must also provide anonymity to the voters; an outsider should not be able to deduce for whom a given voter cast a vote (and thereby influence that voter with threats or bribes). To represent an attack on the system's secret information, we build upon our previous model of the system to create a model of what information can be known by an adversary and how an adversary might deduce that information. A secrecy model has four components:

Information – A description of what is knowable about the system. For example, it represents the fact that a candidate has a score and a receipt is attached to a ballot.

Initial Data – A list of the information that is initially available to an adversary. For example, it includes the fact that the adversary is permitted to initially know how records are marked but not permitted to (initially) know voter intentions.

Incognito Data – A list of the information that ought to be kept hidden from an adversary. It specifies that the adversary should not be able to deduce how voters marked their ballots.

Inferences – Rules describing the conditions underwhich pieces of information can be learned by an adversary. For example, they include the fact that one can deduce a candidate's score from the set of records, or that one can deduce a voter's intention by examining that voter's ballot.

Put together, these four parts allow us to automatically answer the question "Given the initial information made available, can an adversary infer incognito information?". The rest of this section describes how we model each of these four pieces in Alloy, thus permitting automatic detection of secrecy vulnerabilities. The full text of the model is given in Appendix 12.

Our goal here is not to provide a method capable of performing new analyses, but rather to perform existing analyses in a manner that integrates smoothly with the fidelity and auditability goals. As we will see in Section 5.5, if the secrecy model is built using the same phenomena as the fidelity model, then the auditability argument is easy to structure.

$(c \to i \to t)$ $\in$ `known_score`	$\Leftrightarrow$	As of inference step t, candidate c is known to have score i.
$(v \to t)$ $\in$ `known_RegisteredVoter`	$\Leftrightarrow$	As of inference step t, voter is known to be a registered voter.
$(b \to p \to c \to t)$ $\in$ `known_ballotArrangement`	$\Leftrightarrow$	As of inference step t, candidate c is known to be at position p on ballob b.
$(b \to r \to t)$ $\in$ `known_ballotReceipt`	$\Leftrightarrow$	As of inference step t, receipt r is known to have been attached to ballot b.
$(v \to b \to t)$ $\in$ `known_voterBallot`	$\Leftrightarrow$	As of inference step t, ballot b is known to have been given to voter v.
$(v \to c \to t)$ $\in$ `known_intention`	$\Leftrightarrow$	As of inference step t, voter v is known to want candidate c to win.
$(r \to p \to t)$ $\in$ `known_receiptMarked`	$\Leftrightarrow$	As of inference step t, receipt r is known to be marked as potision p.
$(r \to o \to t)$ $\in$ `known_receiptOnion`	$\Leftrightarrow$	As of inference step t, onion o is known to be written on receipt r.
$(o \to p \to c \to t)$ $\in$ `known_onionArrangement`	$\Leftrightarrow$	As of inference step t, candidate c is known to be at position p on the ordering encoded by onion o.
$(d \to p \to c \to t)$ $\in$ `known_recordArrangement`	$\Leftrightarrow$	As of inference step t, candidate c is known to be at position p on record d.
$(d \to p)$ $\in$ `known_recordMarked`	$\Leftrightarrow$	As of inference step t, record d is known to be marked at position p.
$(r \to d)$ $\in$ `known_mix`	$\Leftrightarrow$	As of inference step t, record d is known to have derived from the re-encryption of receipt r.

Figure 5-22: Designations for describing an adversary's inferrable knowledge about the voting system. For each relation in the fidelity model, there is an additional knowledge relation in the secrecy model with the same type signature but an extra time column at the end. A knowledge relation records the subset of the corresponding relation that the adversary has inferred at a given point in time.

5.4.1 Modeling Information

We build upon the designated phenomena used in the fidelity model (and given in Figure 5-4). We augment those designations with a parallel set of knowledge phenomena, shown in Figure 5-22.

Each relation in the prior model is mirrored with a copy that has the same type signature but which has an extra time column. The old relations represent the actual state of the world (how each voter voted, how the receipts were encrypted, who won the election, and so on). The knowledge relations represent knowledge the adversary has learned about the true world (the fact that voter X marked candidate Y on her ballot, the fact that Candidate Q has 10 votes, the fact that ballot A was attached to receipt B, and so on).

In our model, time represents steps during the inference process. The election (and all of the relations from the fidelity model) is a static model – it remains constant throughout the inference process. However, the knowledge relations are dynamic – they vary over time, as the adversary uses inferences to expand them.

For example, we expand the `Voter` signature as shown in Figure 5-23. The `intention` relation (line 2) maps a voter to a candidate, and has the same interpretation (designation) as in the fidelity model. The `known_intention` relation (line 3) maps a voter to a candidate to a time, representing the set of times at which the adversary knows that voter intended to vote for that candidate.

5.4.2 Modeling Initial Data

Line 13 of Figure 5-23 says that initial knowledge is never erroneous. It may be incomplete (and indeed we expect it to be), but it must not contradict the actual world. The keyword `first` refers to the first point in time, so an expression such as `known_intention.first` gives the set of known intentions at the beginning of the attack.

Lines 24-26 prevent the adversary from initially knowing any *defined* variables. Defined variables are derived properties of the world, and are not directly observable, so it would not make sense for the adversary to be able to intuit them. However, there are inferences provided in the model that permit the adversary to infer defined variables according to their definitions (as shown in Appendix 12).

For example, `ballotCandidate` represents the candidate indicated by a ballot – if you know how to interpret it by looking at the arrangement of names and how its receipt is marked. One cannot directly observe what candidate a ballot indicates (a defined phenomenon), but one *can* observe the arrangement, receipt, and receipt marking (designated phenomena) and from that infer the meaning of the ballot. The information about how to interpret a ballot to deduce its candidate is not *a priori* apparent from the world, and the extra knowledge required to make that interpretation is encoded in the inference. We discuss inferences in greater depth in Section 5.4.4.

Lines 29-32 prevent the adversary from knowing some crucial pieces of information. These are designated phenomena, so we have to justify the claim that an adversary cannot directly observe them (by appealing to domain experts).

Line 29: voter intentions cannot be observed because they are hidden inside of the voters' heads. We assume that the adversary cannot forcibly extract that knowledge from a voter (and be confident of its accuracy). A bribed or threatened voter might lie.

Line 30: the ordering of names on a ballot cannot be observed, because the ballot (with that list) is torn off of the receipt before the voter leaves the voting booth. The booth might be equipped with a stack of different half-ballots (without receipts), so a voter could carry out a different one than the one actually attached to the receipt used.

Line 31: the ordering of names encoded in an onion cannot be observed, since they are encrypted with public key cryptography (and the private key is kept secret).

Line 32: the link between receipts going into the voting board and records coming out of it is obscured by the re-encryption process. This line is commented out since it was omitted from an early version of the model, and doing so permits the attack described in Section 5.4.5.

5.4.3 Modeling Incognito Data

First, we consider the informal secrecy requirement, as described to us by the system designers. Then we restate it precisely and rewrite it formally in Alloy.

informal

> Do not allow an outside observer to deduce how an individual voter voted.

precise

An outside observer making reasonable inferences and observations must not be able to deduce (with certainty) for whom a particular voter voted.

formal

```
no v: Voter |
  some v.(known_voterBallot.last).(known_ballotCandidate.last)
```

There must not be any voter whose ballot is known if the candidate indicated by that ballot is decipherable. Appears as line 35 in Figure 5-23.

5.4.4 Modeling Inferences

The model uses a variation on the *event-based* idiom [34]. Each inference is reified in its own signature, which allows us to write constraints about inferences as first class objects. For example, we can easily write a constraint that says that only one inference is made each time step, that a particular inference is never made, or that a particular inference is only made under certain conditions. Figure 5-24 shows a typical inference signature written as a reified event.

The event based idiom allows us to build *history* and *prophecy* variables, as shown in Figure 5-25 , which make counterexamples much easier to interpret. A history variable maps a point in time to the inference that leads to that point in time (Line 2). A prophecy variable maps a point in time to the inference that will lead it to the next point in time (Line 3).

The event-based idiom also eases our use of the the *justified-change* idiom. Rather than writing constraints of the form "if inference X is used then (only) information Y is learned" (enforced-change idiom), we write them in the form "if information Y is learned then inference X must have been used" (justified-change idiom).

Lines 22-26 of Figure 5-25 show how we enforce the justified change idiom in Alloy. In order for the adversary to learn a new tuple of `known_intention`, he must satisfy the constraints given in at least one of the inference rules for intention. One such inference rule is shown in Figure 5-24, which says that the adversary can use information from a fully marked ballot to infer a voter's intention.

This pattern permits us to have several inferences trigger in the same time step, without having their frame conditions interfere with each other. Under the enforced-change idiom, an inference X1 cannot allow the adversary to learn information Y1 in the same time step that inference X2 allows the adversary to learn information Y2, because the frame condition on inference X1 says that only information Y1 is learned.[3] However, the justified-change idiom allows any number of pieces of information to be learned as long as each addition is justified by some inference. The benefit of this is that it allows complex attacks to be made in fewer time steps, meaning that an analysis in a given scope provides a stronger guarantee and thus greater confidence in the secrecy of the system. When examining counterexamples, it is helpful to prevent simultaneous inferences rule (Lines 14-17 of Figure 5-25), but when checking assertions we relax that constraint to provide a stronger guarantee.

5.4.5 Identifying an Attack

Lines 19-38 of Figure 5-23 instruct Alloy to find a sequence of inferences that successfully attack the system's secret data.

Lines 20-21 import some predicates that constrain how learning happens and which make the resulting solutions easier to interpret. All but one have been elided from this excerpt; the remaining constraint says that only one inference is made each time step (which makes counterexamples easier to interpret). Lines 23-32 define restrictions on the adversary's initial knowledge, as discussed earlier.

Line 35 defines what it means for the adversary's attack to succeed. We force it to be true, thereby telling Alloy to find us a solution showing a successful attack.

Lines 37-38 are `run` statements that instruct Alloy to search for solutions to the predicate within the stated bounds. The first one has a solution, indicating

[3] If X1 did not say that only Y1 changes, then X1 would be implicitly allowing anything at all to change when it triggers, which is clearly not a valid inference. The problem centers around the need for so called *frame conditions* in declarative logic – statements of the form "and nothing else changes".

a successfully attack (discussed below). The second one does not have solutions, indicating that there are no viable attacks that take only 2 inferences.

5.4.6 Interpreting the Solutions

Running the analysis on Line 37 of Figure 5-23 returns a solution – a case where the adversary succeeds at making an attack despite obeying the restrictions placed on his initial knowledge. Figures 5-26 through 5-29 show one such solution as displayed by the Alloy visualizer. The solution has been projected over Time, meaning that we will see a sequence of diagrams each representing the state of the world at a particular point in time.

We can see the actual election information in any of the diagrams, since it is not time dependent. There is one voter, and that voter intended to vote for the only candidate. That voter is given the only ballot, which has a receipt with an onion. That receipt is sent to the voting board where is is re-encrypted (the `mix` relation) and turned into a record. Both the receipt and record are marked at the same position, next to the sole candidate in the race. The candidate thus has a score of 1. These relations have no time column, so they are are the same in each Time-dependent diagram.

By looking at the sequence of diagrams, one for each time step, we see the sequence of inferences the adversary used. In each diagram, the trapezoid node `pastAttractions` is a history variable that gives the inference that result in that time step. The trapezoid labeled `comingAttractions` is a prophecy variable giving the inference that is about to happen.

```
sig Voter {
  intention: set Candidate,
  known_intention: Candidate → Time,
  voterBallot: set Ballot,
  known_voterBallot: Ballot → Time,
  ...
}

...

//initial knowledge is correct, but possibly incomplete
pred seededKnowledge [] {
  known_intention.first in intention
  ...
}

...

pred successful_hard_attack [] {
  ...
  seededKnowledge

  //you cannot initially know defined phenom, only designated ones
  no known_ballotCandidate.first
  no known_receiptCandidate.first
  no known_recordCandidate.first

  //domain-specific restrictions
  no known_intention.first //no telepathy
  no known_ballotArrangement.first //tear-off receipts
  no known_onionArrangement.first //encryption
  --no known_mix.first //re-encryption scrambling

  //malicious goal
  some v: Voter | some v.(known_voterBallot.last).(known_ballotCandidate.last)
}
run successful_hard_attack for 2 but 3 Inference, 4 Time, 3 int, 1 Record expect 1
run successful_hard_attack for 2 but 2 Inference, 3 Time, 3 int, 1 Record expect 0
```

Figure 5-23: Selections from the voting secrecy model that define the information and temporal structure. Full text is given in Appendix 12.

```
abstract sig Inference {
  pre, post: one Time
}
sig pause extends Inference {} {} //the trivial inference that learns nothing

abstract sig intention_Inference extends Inference {
        used_voter_from_intention: one Voter,
        used_candidate_from_intention: one Candidate,
}
sig intention_inference_1 extends intention_Inference {}{
  //what you learn
  (used_voter_from_intention → used_candidate_from_intention)
    in known_intention.post
  (used_voter_from_intention → used_candidate_from_intention)
    not in known_intention.pre

  //when you can learn it
  used_voter_from_intention in known_RegisteredVoter.pre
  let b = used_voter_from_intention.(known_voterBallot.pre) |
    b.(known_ballotReceipt.pre).(known_receiptMarked.pre)
      .(b.(known_ballotArrangement.pre)) = used_candidate_from_intention
}
```

Figure 5-24: A typical inference written in the event-based idiom. Adding inferences under this idiom is modular. To add an inference, only one signature paragraph need be added – the rest of the model remains untouched. This particular inference is for deducing a tuple in the intention relation – learning that a particular voter definitely intends to vote for a particular candidate by examining that voter's ballot.

```
sig Time {
  comingAttractions: set Inference, //prophecy
  pastAttractions: set Inference, //history
}
fact history_matches_prophesy {
  all t: Time | t.comingAttractions = t.next.pastAttractions
  no first.pastAttractions
  all t: Time - last | t.comingAttractions.pre = t
  all t: Time - first | t.pastAttractions.post = t
}

...

pred sequentialInferences [] {
  all t: Time | lone t.comingAttractions
  all t: Time | lone t.pastAttractions
}

...

pred explainAdditions {
  all t: Time - first, v: Voter, c: Candidate |
    (v → c) in known_intention.t - known_intention.(t.prev)
      ⇒ some inf: intention_Inference & t.pastAttractions |
         inf.used_voter_from_intention = v
         and inf.used_candidate_from_intention =c
  ...
}
```

Figure 5-25: Selections from the voting secrecy model that define predicates that constrain how inferences are made. Full text is given in Appendix 12.

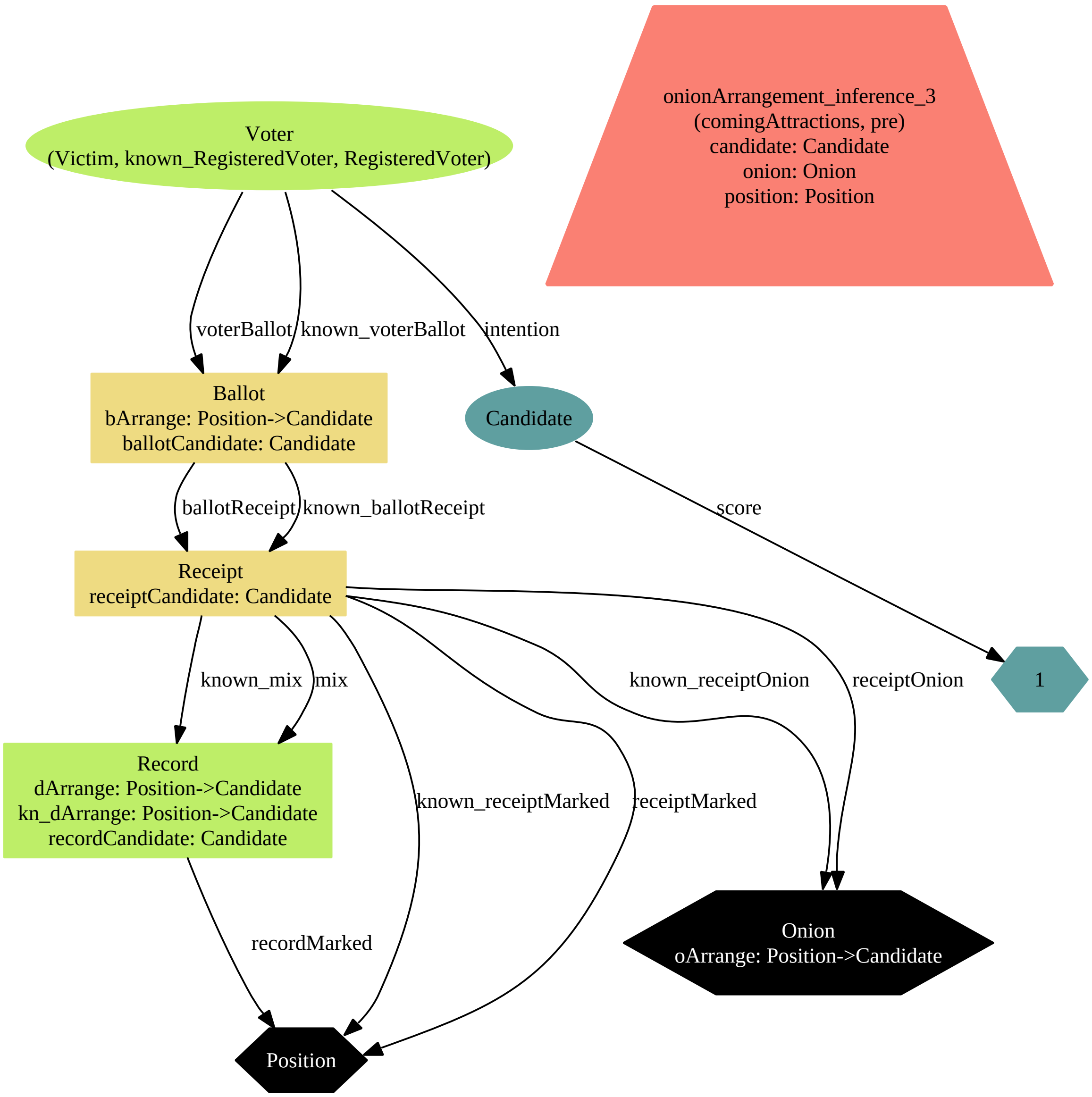

Figure 5-26: The first time step (Time0) in a successful attack on the voting system's voter records. This timestep represents the static structure of the election that happened, and the initial knowledge available to the adversary.

Initially (Time0), the adversary knows which ballot the voter was given, the receipt attached to that ballot, the onion for that receipt, which record that receipt turned into, and how the receipt was marked. This is not yet enough information to deduce for whom the voter voted, since it is not (yet) apparent how the ballot was arranged, how the receipt was marked, or how the voter intended to vote. In fact, in this attack, the final score for the candidate isn't known and doesn't need to be inferred.

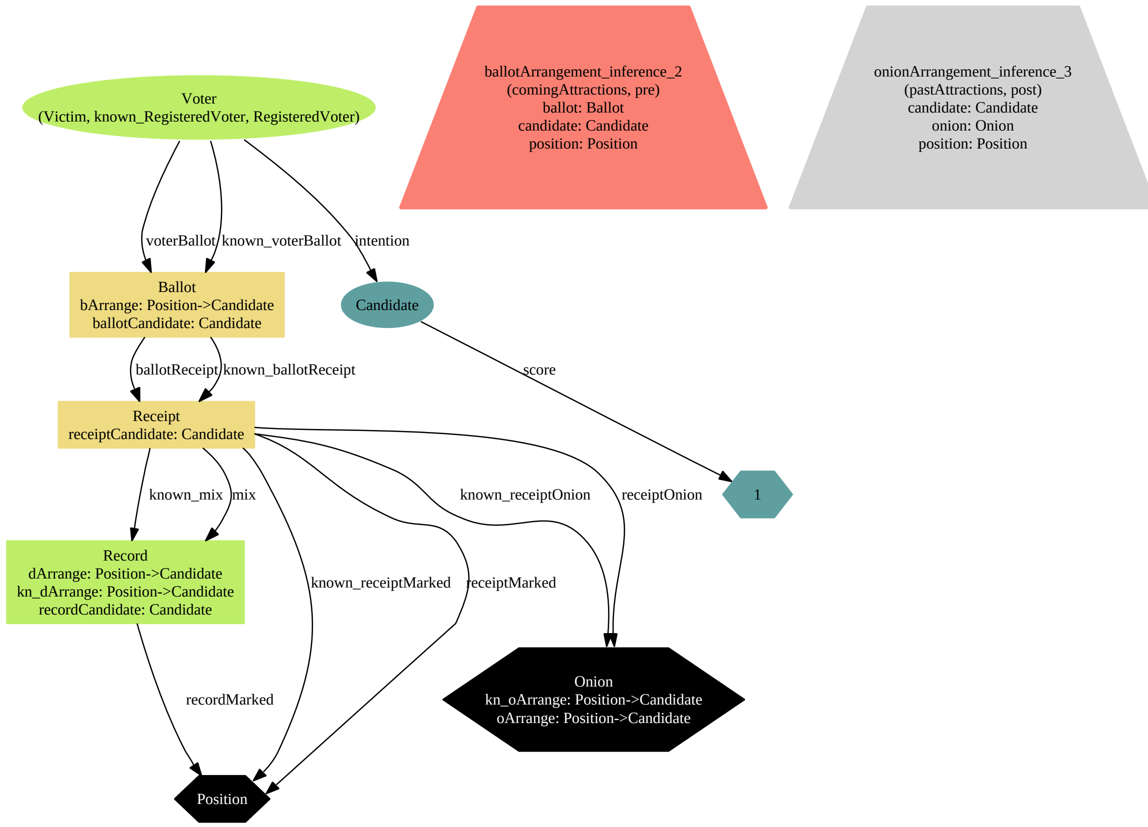

Figure 5-27: The second time step (Time1) in a successful attack on the voting system's voter records. In this step, the adversary has just inferrred the ordering represented by the onion by looking at the ordering of the record associated with that onion's receipt.

The first inference used is `onionArrangement_Inference_3`, which says that one can deduce the arrangement of an onion if one knows the arrangement of the record that the onion's receipt turned into when fed into the voting board. Specifically, it relies on the fact that those two orderings must be identical (as stated in the voting board bread crumb). In the second time step (Time1), we see the result of this inference – that the adversary now knows the onion's arrangement (a `kn_oArrange` entry has been added to the onion's pentagonal box, which mirrors that portion of the actual onion arrangement, `oArrange`).

From the second to the third time steps, the adversary uses `ballotArrangement_inference_2` to deduce the arrangement of candidates on the ballot based on the arrangement of candidates encoded in the now-revealed

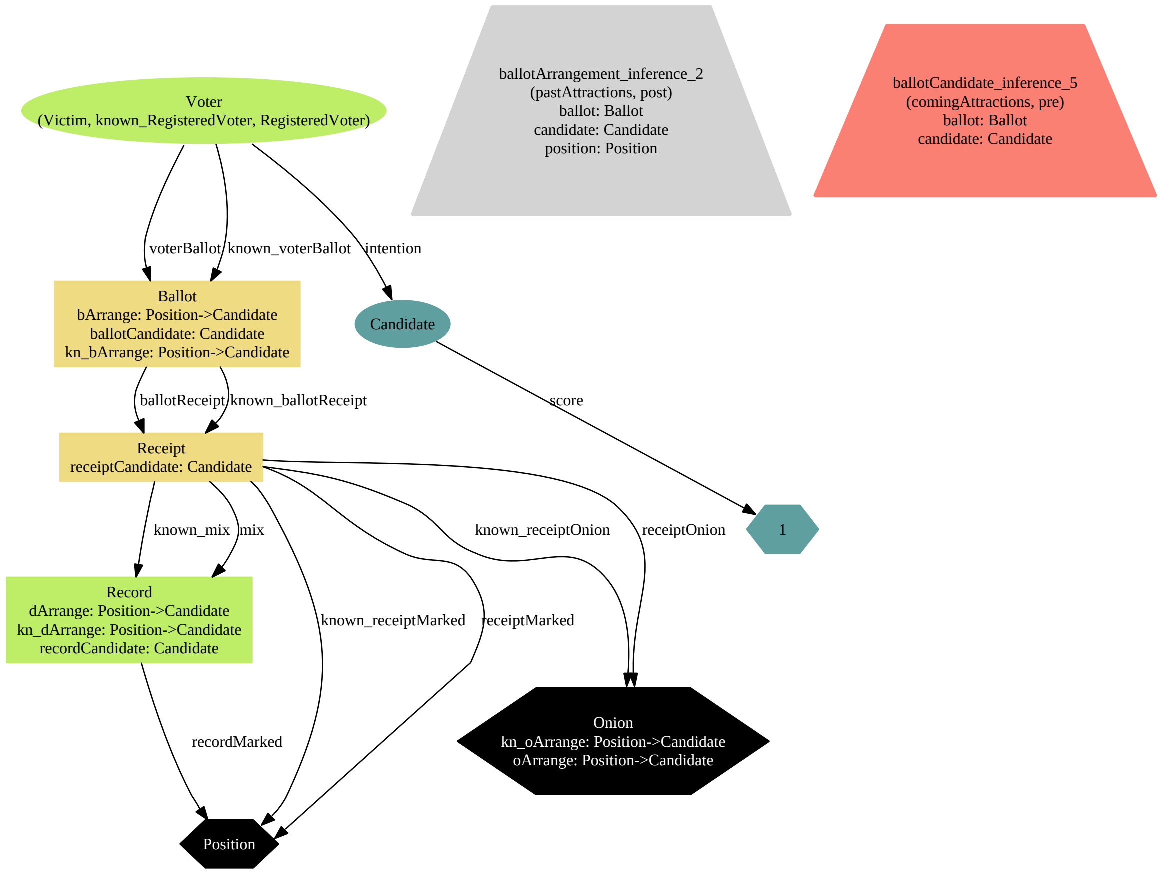

Figure 5-28: The third time step (time2) in a successful attack on the voting system's voter records. The adversary has just inferred the candidate ordering on the ballot by noting the ordering represented by that ballot's receipt's onion.

onion.

From the third to fourth time steps, the adversary uses `ballotCandidate_inference_5` to deduce the candidate marked on the ballot from a combination of that ballot's arrangement (just deduced) and how that ballot's receipt was marked (already known). This inference concludes a successfully attack for the adversary, since the voter's voting behavior has been revealed.

Alloy guarantees sound counterexamples, so we know that this attack is a valid one (given the inference rules provided). Our analysis does not have solutions for a smaller time bound (line 38 of Figure 5-23), so we know that no shorter attack exists (in the given scope).

If we add an assumption that the adversary cannot know the voting boards scrambling pattern (Line 32 of Figure 5-23), then Alloy is unable to find an attack, even if permitted larger bounds. This gives us confidence (but not a guarantee) that no attack against the system can succeed with this arsenal of inferences. It is still possible that an attack exists in a larger scope, or that one is possible with additional (unstated) inference rules. However, with the given inferences and given bounds, we are guaranteed that no attack exists.

Small Elections

An earlier version of our model included another class of inferences pertaining to attacks on small elections. For example, if there is only one voter, then one can determine his or her vote simply by looking at the final candidate scores. Similarly, if you know how all but one voter voted, then you can use the candidate scores to deduce the last voter's vote. More subtle attacks include the fact that an adversary might himself be a voter, and thus might know how one of the voter voted, and attempt to leverage that information. These inferences were not derived directly from the fidelity model (as described in Section 5.6), but rather were added directly to the model according to outside knowledge.

The final version of the model omits these inferences, partly to show the power available from using just the derived inferences, and partly because attacks on small systems are usually considered uninteresting from a security perspective. With the small-attack inferences in place, we had to include a set of assumptions in the attack simulations assuming that there were at least 3 elements of each domain, and at least 3 votes were cast for each candidate. Those assumptions inflated the size of the resulting counterexamples, but did not show more interesting attacks. Analysis time for the slightly larger solutions was not noticeably slower.

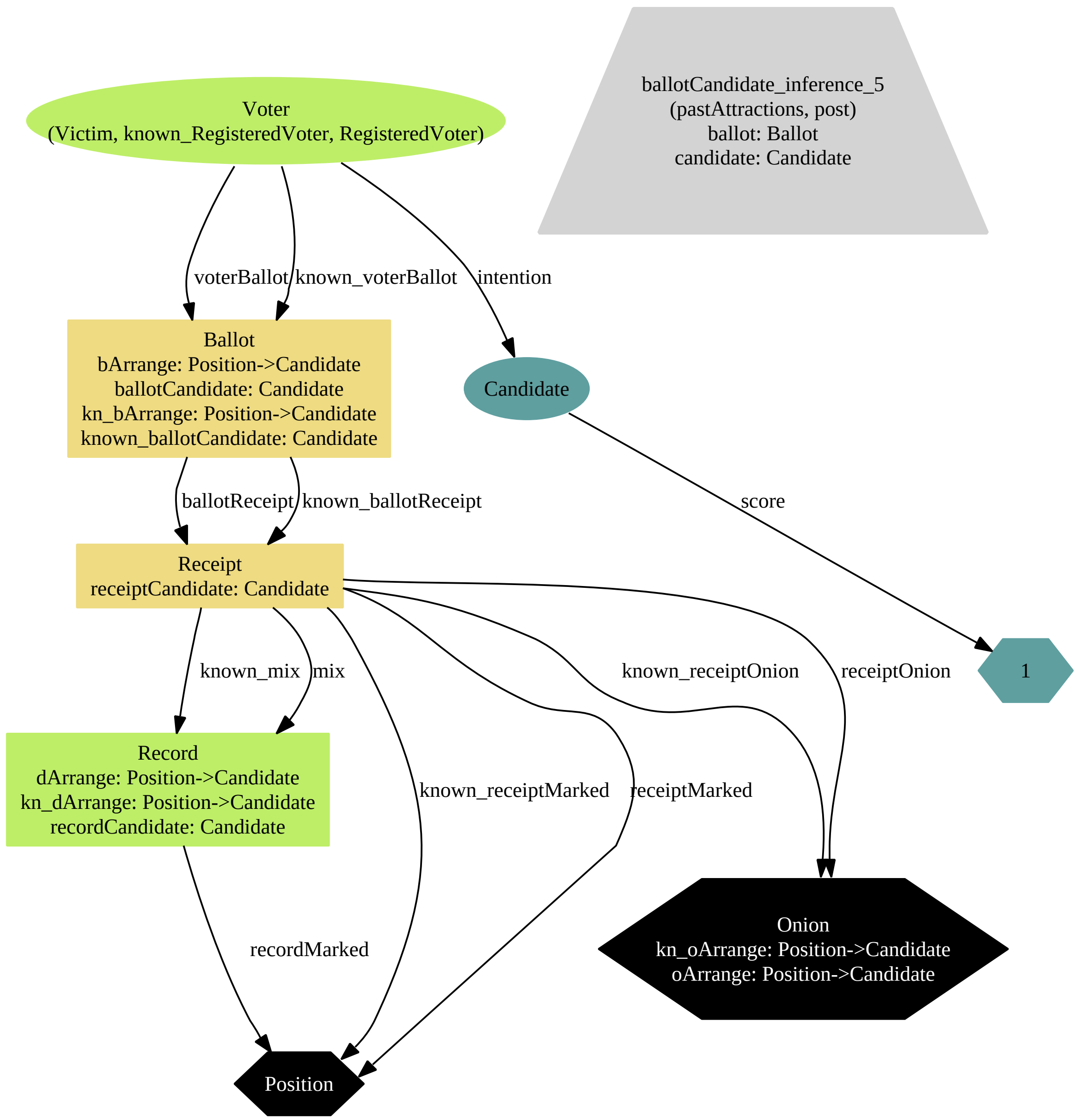

Figure 5-29: The final time step (time3) in a successful attack on the voting system's voter records. The adversary has just inferred what candidate is indicated by the voter's ballot, be examining its ordering (just inferred) and its receipt's marking (initially known). This constitutes a successful attack on the systems secrecy, since the adversary now knows how this voter voted.

5.5 Auditability Goal

It is not enough for the engineers who built the voting system to be sure that it works; public observers must also be able to check that the operating implementation is functioning correctly. In the absence of security and secrecy requirements, auditability can be as simple as publishing the fidelity argument associated with the system – the argument that the designers used to get the system right. However, the fidelity argument is likely to make reference to phenomena that are forbidden from being made public by the secrecy argument, and the censored fidelity argument is not complete enough to provide auditors with confidence.

The typical solution to this problem is to introduce statistical audits of the secure portions of the system [5]. The details of how such a statistical sample is gathered is a domain-dependent question – the question relevant to our analysis is that of *which* phenomena should be audited. By building the secrecy and fidelity arguments with the same lexicon, we can see which phenomena need to be audited by comparing the phenomena used in the two models (as described below, in Section 5.5.3).

The view of auditability that we use in this section is based on Rivest and Wack's [74] notion of *software independence* – the notion that a voting (or other high profile) system should not rely upon the correct (and honest) implementation of its software components. It should be possible to know if the system has failed, no matter what malicious component has been substituted in. In other words, one should not merely rely upon a good system *design* but one should also provide a way to directly audit particular *implementations* of that design as they are running.

5.5.1 Types of Audits

There are three forms of auditability to consider. Applied to a voting system, they are as follows:

- A *design audit* is when the system design is assessed to determine if it will satisfy its requirements (accurately tally votes, protect voter anonymity). Publishing the system's fidelity argument is sufficient to provide design auditability, as it gives observers access to the same information used by the engineers to design the system.

- A *system audit* is when an installation is evaluated by an independent authority, such as a government agency, a newspaper, or a university. It might involve a review of the hardware and software used to implement the design, and it confirms that the machines installed in actual election booths implement the validated design [12, 5].

- A *personal audit* is when individual voters can confirm that their votes were tallied. Perhaps the voter is given a receipt with an ID number, and the final tally lists the ID numbers it included. This process confirms that the vote was counted without revealing how the voter voted [77, 73, 1].

The system audit is at the core of all three styles. The design audit establishes a system-level property like the ones we establish using requirement progression – that the assumptions made about the components are sufficient to enforce a given requirement for the system as a whole. However, it does not actually validate that the components obey their assumptions; that is the job of a system audit. Similarly, the merit of a personal audit relies on the merit of the system audit. The fact that an individual voter's ID appears in the final tally is only meaningful if a system audit has confirmed that IDs only appear in the final tally if they were counted. For these reasons, we will focus on providing system auditability.

5.5.2 A Precise Formulation

We begin with the informal auditability goal, then rewrite it in more precise language.

informal

> Outside public observers need to be confident that the system has not been tampered with.

precise

> If the election does not choose the most popular candidate, then there must be a high probability that this fact is apparent to public observers.

This phrasing has two important implications on how we perform the necessary audits, which enable statistical audits of any phenomena in question.

- There is no need for error recovery, just error detection – one can re-run an election that is deemed to be fraudulent. Thus an audit does not need to catch *all* fraudulent elements of a corrupted system (as would be needed to fix the problem). Rather, it is sufficient to identify just *one* fraudulent element and denounce the entire system.
- There is no need to catch all tamperings, just those that alter the outcome of the election. One can tolerate both a small chance that many votes were changed or a high chance that very few votes were changed. In both cases, there is only a small chance that the election result will be altered.

5.5.3 Identifying Necessary Audits

As discussed earlier, auditability would be easy if it were not for secret data – one could simply publish the fidelity argument and reveal all information used in the operation of the system. Even in the presence of secrecy requirements, some parts of the system can be audited in this direct fashion. To determine which parts require special treatment, note the set of designated phenomena that are kept secret from the public. In our case, as given by lines 29-32 of Figure 5-23, they are `intention`, `ballotArrangement`, `onionArrangement`, and `mix`.

Now consider each domain assumption in the fidelity argument. If it makes no reference to the hidden phenomena, then it can be audited directly (by observing the values of non-secret phenomena). If it references one or more hidden phenomena, then it cannot be audited directly and calls for a statistical audit. The assumptions we must audit are given in figure 5-30 and interpreted below.

Lines 2-3: We must provide an audit of voter *intentions* to assure the public that they are well formed without revealing enough data to an adversary to attack the system's secrecy.

The system designers did not record this assumption or provide a mechanism for ensuring it. The audit might involve a user study showing that voters do

```
1  //from Voter breadcrumb
2  all v: RegisteredVoter | let b = v.voterBallot |
3    b.ballotReceipt.receiptMarked.(b.ballotArrangement) = v.intention
4
5  //from Ballot breadcrumb
6  all b: Ballot |
7    b.ballotArrangment = b.ballotReceipt.receiptOnion.onionArrangement
8
9  //from Board breadcrumb
10 all output: Receipt.mix | one mix.output
11 all b: Ballot | b.ballotReceipt in mix.Record
12 all r: mix.Record | r in Ballot.ballotReceipt
13 all input: mix.Record | let output = input.mix {
14   input.receiptOnion.arrangement = output.recordArrangement
15   input.receiptMarked = output.recordMarked
16 }
```

Figure 5-30: Domain assumptions that cannot be audited directly (without violating secrecy) and thus call for statistical audits.

indeed intend to vote for one candidate – they are not confused about the rules of the election, the implications of their vote, or the meaning of marking the ballot. Put another way, we are assuming that voters know whom they want to win and understand how to mark a ballot to make that happen.

Lines 6-7: Since we are hiding the content of the *onions* (the arrangements they represent), we must provide an audit of them. Specifically, we must check that the arrangements on the ballots matches the arrangements represented by the onions on the receipts of those ballots.

To address the corrupt ballot concern, the system designers suggest a random audit of ballots – randomly select a subset of the unused ballots to be audited and discarded. Each selected ballot's onion is decrypted (using the private key) to confirm that the onion reflects the printed list of candidates. Outside observers can confirm that the decryption is accurate (using the public key) even though they cannot themselves perform the decryption step. The audited ballots are invalidated and discarded. If any ballots are invalid, they must all be re-generated. For example, if one percent of the ballots are corrupt, and one audits 1000 random ballots, then the chance of all bad ballots going undetected is around 1 in $25,000$.

Lines 10-15: We must also randomly audit that the *receipt scrambling* process is working. Since we have hidden the mix relation from directly observation, we need to audit the properties we assume about it (given in the voting board breadcrumb).

To address the corrupt voting board concern, the system designers provide a cryptographic mechanism for auditing individual re-encryption steps. Each receipt is re-encrypted several times before being outputted as a record (and decrypted). However, for secrecy to be maintained, there only needs to be one re-encryption step. As illustrated in Figure 5-31, consider all of the receipts after the first re-encryption step. Randomly select half of them and reveal what receipts they came from. For the other half, reveal what receipts they went to in the next re-encryption step. Reveal no other re-encryption steps. Note that no receipt can be fully tracked through the re-encryption process, and thus secrecy has been preserved. However, if the machine is performing bad re-encryptions, there is a high chance of detecting it. If 1 percent of the ballots are corrupt, and there are 12 re-encryption steps, then the chance of all bad re-encryptions going undetected is about 1 in $25,000$.

5.6 Deriving Inferences from Breadcrumbs

While the primary benefit of building compatible fidelity and secrecy arguments is to aid auditability, there are other benefits. They will share a common set of signature declarations and can both make use of a common set of domain assumptions (the fidelity argument checks their consistency and the secrecy argument enforces them as invariants). Furthermore, as we discuss in this section, the domain assumptions developed in the fidelity argument can be leveraged to derive a core set of inferences for the secrecy model.

The set of derived inferences is not complete – indeed no set of inferences is ever

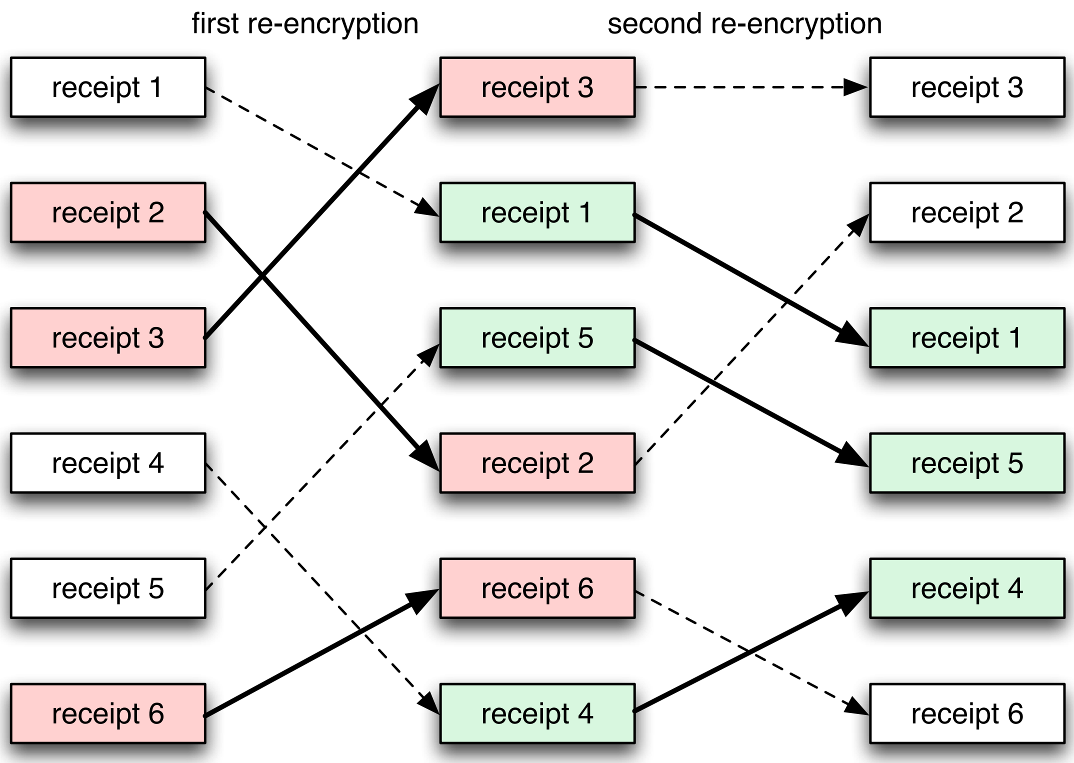

Figure 5-31: Auditing the receipt-scrambling re-encryption process of the voting board. Each column represents the set of receipts after a re-encryption step. We have selected half of the receipts (red / dark gray) and revealed the prior re-encryption step for them (dark arrows) and kept the other re-encryptions hidden (dashed arrows). For the other half of the receipts (green / light gray), we reveal the next re-encryption step. An observer cannot track a receipt through the entire encryption process, but we have had two chances to catch a bad re-encryption step.

complete, as one can never predict the full computational and inferential power of an adversary. These inferences should be augmented by security experts – standard practice is for all inferences to be provided by experts.[4] However, the set of inferences derived in this way are sufficient to mount basic attacks against the system's secret data, even without additional inferences provided by an expert (see Section 5.4.5). The derived inferences not only make developing the secrecy model easier, but they also make the set of inferences more thorough and thus increase the confidence gained

[4]As described in section 5.4, the secrecy model is written in a modular fashion, so an expert can add inferences to the model without making global changes or requiring a global understanding.

when the model reports that no attacks are possible.

5.6.1 Derivation Process

The key insight to deriving inferences from breadcrumbs is that any property the system relies upon is also a property the adversary can rely upon. For example, if the system relies upon the fact that voters mark their ballots according to their intentions, then the adversary can also make that assumption. The technical challenge is to translate declarative assumptions about valid states of the world (breadcrumbs) into operational inferences about what information can be deduced from what other information (inferences).

Currently, our approach can only derive inferences from a narrow (but common) type of assumption: a universal quantifier surrounding an equality between two sequences of relational joins. This pattern is sufficient to subsume most of the breadcrumbs in our fidelity model (Section 5.3), and was used to derive all the numbered inferences given in the secrecy model (Section 5.4).

To derive an inference, follow these steps:

(1) Identify a legal target constraint and relation.

  (1a) Confirm that the constraint is in the required form – a universal quantifier surrounding an equality of two strings of relational joins.

  (1b) Identify one of the relations in the constraint, for which we will attempt to learn a new tuple. That relation cannot have more than one column with the same type.

  (1c) Determine if the target constraint is strong enough to permit an inference of the target relation. This evaluation examines the multiplicities of the other relations in the constraint, aided by an object-model diagram of the constraint.

  (1d) Create a place-holder tuple of the target relation, representing the new tuple that will be learned. Provide fresh variables for the entries of that tuple.

(2) Transform the target constraint to become a precondition for learning a new tuple of the target relation.

- (2a) Drop the universal quantifier if it over a domain that match the types of any of the variables in the send-in tuple.
- (2b) If the universal quantifier does not match any of those domains, replace it with an existential quantifier.
- (2c) Replace all instances of the target relation in the constraint with the place-holder tuple.
- (2d) Replace the equality operator in the constraint with a non-empty equality operator. Non-empty equality is the same as equality, except that it resolves to false if either side of the equation is empty.
- (2e) Manipulate the equation as necessary to get rid of the place-holder tuple. The resulting equation, or set of equations, will still contain the variables used in that tuple.
- (2f) Replace all relations with their matching knowledge relations, joined with a time variable T.

(3) Piece together the inference. The resulting inference states that, if the resulting constraint holds on the knowledge relations in the pre-state, then the place-holder tuple can be added to the target relation's knowledge relation in the post state.

5.6.2 Sample Derivation

For example, from the Voting Board Breadcrumb, we spot the following constraint in the required form:

```
all input: mix.Record |
  input.receiptMarked = input.mix.recordMarked
```

We decide to build an inference for the `receiptMarked` relation, which maps each receipt to the position (if any) that is marked on that receipt. Examining the object-model diagram, we confirm that this constraint is strong enough to permit an inference of the target relation. The details of that process are described in Section 5.6.4. We build a place-holder tuple of the target relation, with fresh variables for each entry of that tuple. The relation maps receipts to positions, so we create the tuple $rm = (r \rightarrow p)$.

Next, we manipulate the target constraint to transform it into the precondition for adding rm to our knowledge of `receiptMarked`. The quantified variable is over receipts; that is the type of one of the variables in the tuple, so we drop the quantifier. We then replace all instances of `receiptMarked` with $(r \rightarrow p)$, producing the following:

```
r.(r → p) = r.mix.recordMarked
```

We manipulate that equation to get rid of the tuples. The left-hand side reduces to p by the definition of relational join. The right-hand side does not reduce. The equation is now the following, taking care to note that "=" now represents non-empty equality.

```
p = r.mix.recordMarked
```

Replacing the relations with the corresponding knowledge relations gives us the following:

```
p = r.(known_mix.pre).(known_recordMarked.pre)
```

Putting this into the Alloy model, we get the following inference:

```
sig receiptMarked_inference_4 extends receiptMarked_Inference {}{
  //what you learn
  (used_receipt_from_receiptMarked → used_position_from_receiptMarked)
    in known_receiptMarked.post
  (used_receipt_from_receiptMarked → used_position_from_receiptMarked)
    not in known_receiptMarked.pre

  //when you can learn it
   used_position_from_receiptMarked =
     used_receipt_from_receiptMarked.(known_mix.pre).(known_recordMarked.pre)
}
```

Intuitively, we have taken a constraint that says "The positions marked on receipts going into the re-encryption process is the same as the positions marked on the corresponding records coming out." and transformed it into an inference that says "The adversary can infer that receipt R is marked at position P if he knows that R was mapped to a record D and that D is marked at position P.".

5.6.3 Another Example

Now consider the same constraint but instead target the `recordMarked` relation, which maps records to positions. The tuple we wish to infer is now $dm = (d \rightarrow p)$. This time, the universal quantification is over a domain not used in the tuple, so the it is replaced by an existential quantifier. We splice the tuple into the constraint in place of the target relation, producing the following:

```
some input: mix.Record |
  input.receiptMarked = input.mix.(d → p)
```

Manipulating that expression (using the fact that the "=" represents non-empty equality), we get the following:

```
some input: mix.Record {
  input.receiptMarked = d
  input.mix = p
}
```

If we had instead targeted `mix`, we could go through all the steps to generate an inference. However, the resulting inference would be invalid, and would be preemptively caught by step 1c, as described next.

5.6.4 Validation via Multiplicities

To determine if a given constraint can produce an inference for a given relation, consider the object-model representation of the relations used in the constraint, annotated with multiplicity marking on all relations.[5] Note the target relation and which node represents the type of the equality – the type of both the left and right hand sides of the equation. Consider all paths along relations from the nodes used in the relation to the equality node. If more than one of them has a * on the origin of the arc or if any of them have a * on the destination of the arc, then the inference is rejected. Otherwise, the derived inference will be valid.

Figure 5-32 shows the diagram that describes the target constraint from the voting board breadcrumb used in the preceding examples.

Looking at the multiplicities, we see that we can derive valid inferences for the `receiptMarked` and `recordMarked` relations, but not for the `mix` relation.

All of the numbered inferences in the secrecy model (described in Section 5.4 and given in full in Appendix 12) were derived in this fashion. The target constraints and

[5]Ternary relations are shown as two merging arrows. Multiplicity markings on ternary relations should be interpreted as "given values for the other two slots of a tuple of this relation, how many possible values are there for the remaining slot in that tuple?".

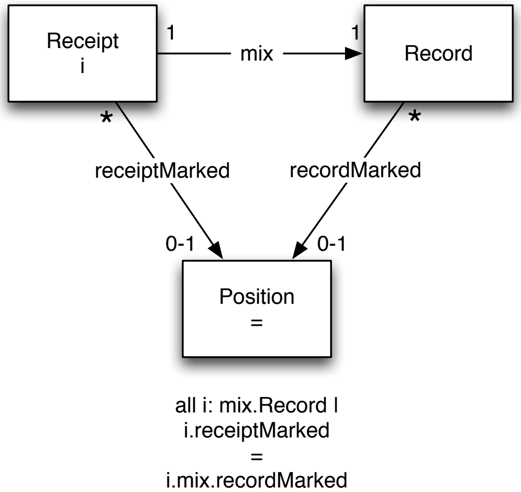

Figure 5-32: The second part of the voting board breadcrumb viewed graphically in preparation for deriving inferences from it.

corresponding multiplicity diagrams are shown in Figures 5-33 through 5-38.

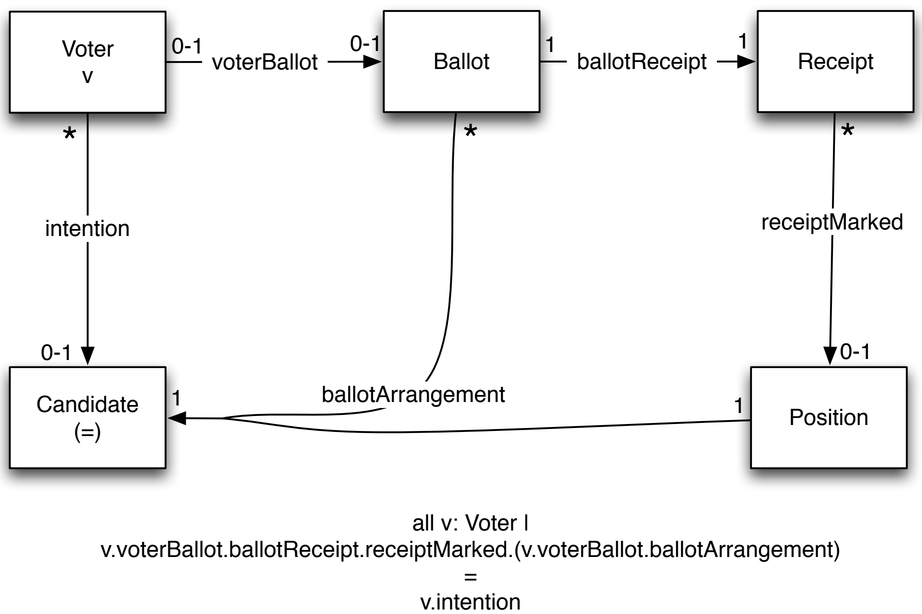

Figure 5-33: The voter breadcrumb viewed graphically in preparation for deriving inferences from it.

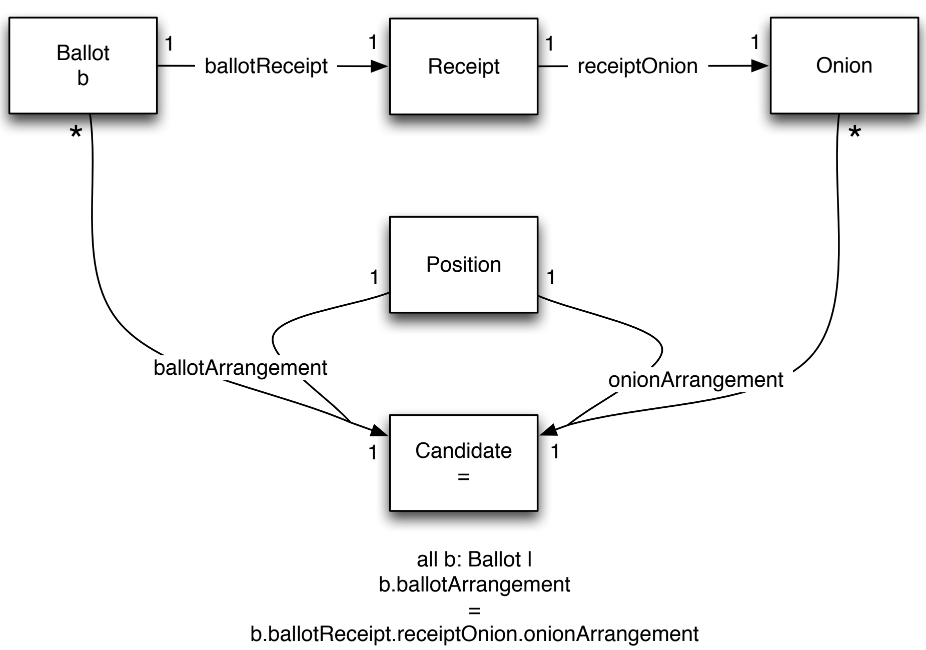

Figure 5-34: The ballot breadcrumb viewed graphically in preparation for deriving inferences from it.

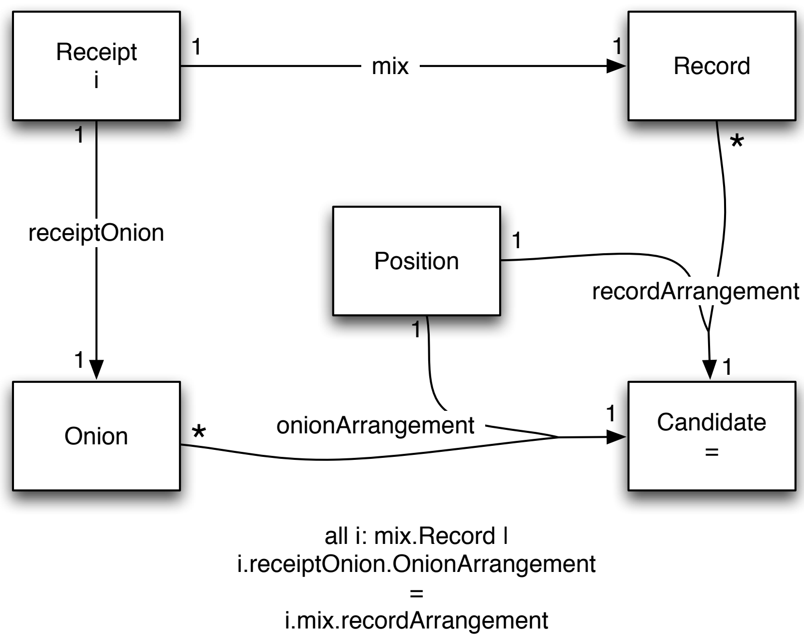

Figure 5-35: The first part of the voting board breadcrumb viewed graphically in preparation for deriving inferences from it.

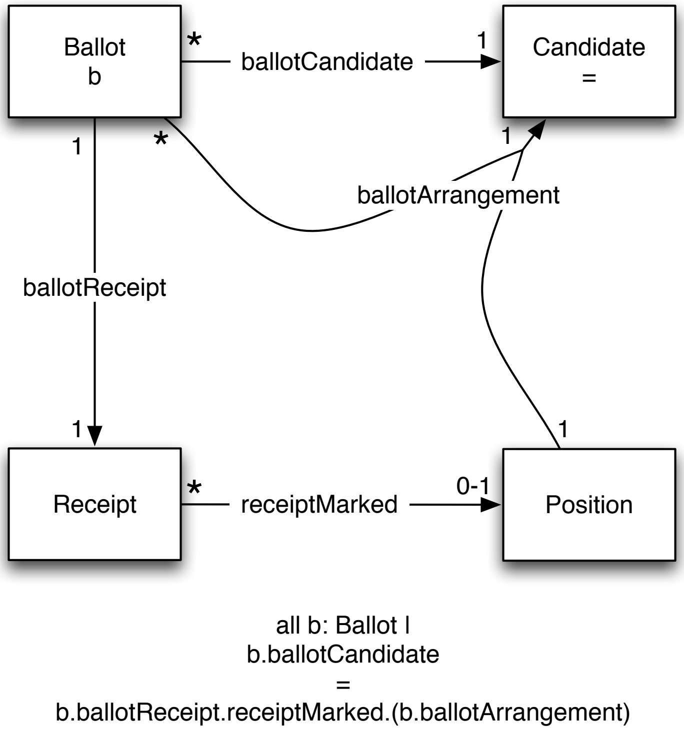

Figure 5-36: The ballot appended fact viewed graphically in preparation for deriving inferences from it.

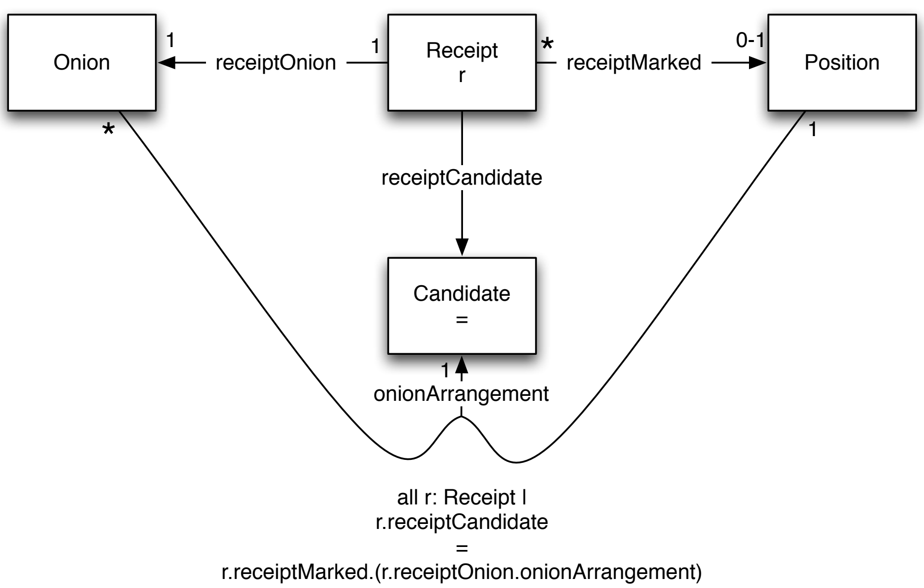

Figure 5-37: The receipt appended fact viewed graphically in preparation for deriving inferences from it.

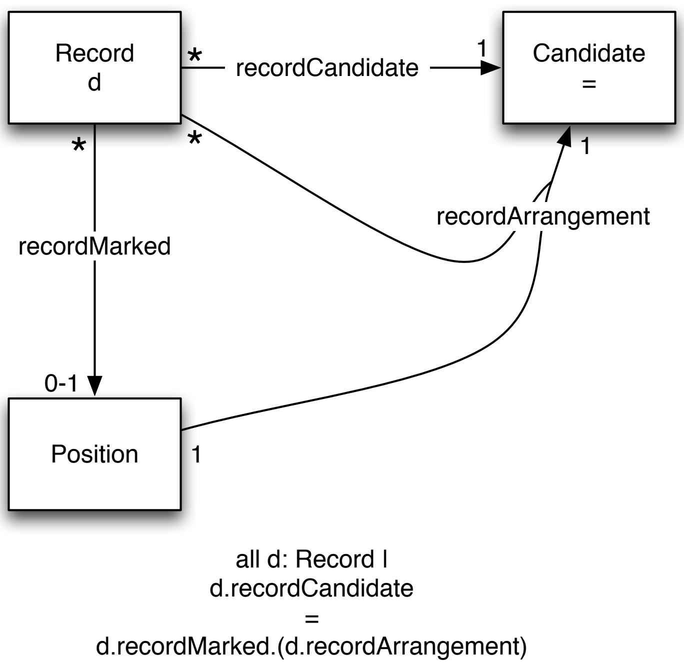

Figure 5-38: The record appended fact viewed graphically in preparation for deriving inferences from it.

5.7 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 a structured, traceable, and readable argument describing *why* the system satisfies its requirements.

5.7.1 Clean Division

In building the argument, we separated the system-level arguments from low-level cryptographic 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.

5.7.2 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.

5.7.3 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.

5.7.4 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 5.7.3. The secrecy argument took four

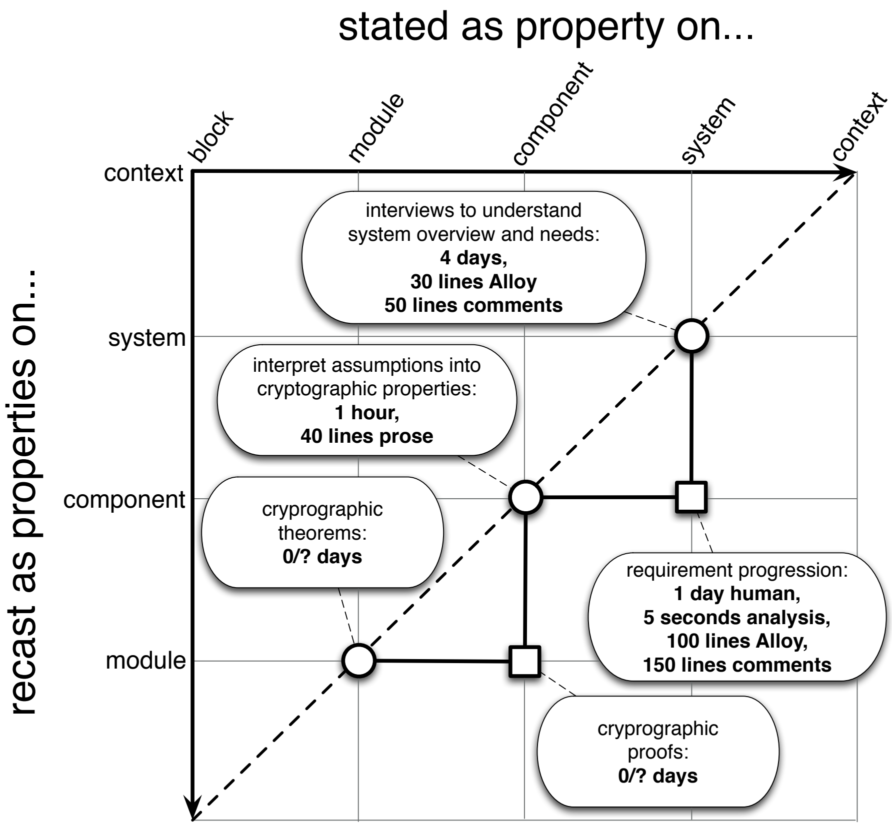

Figure 5-39: Time spent building the fidelity argument.

days, two of which were spent building the model framework and two of which were spent deriving inferences for the adversary. See Figure 5.7.3. 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.

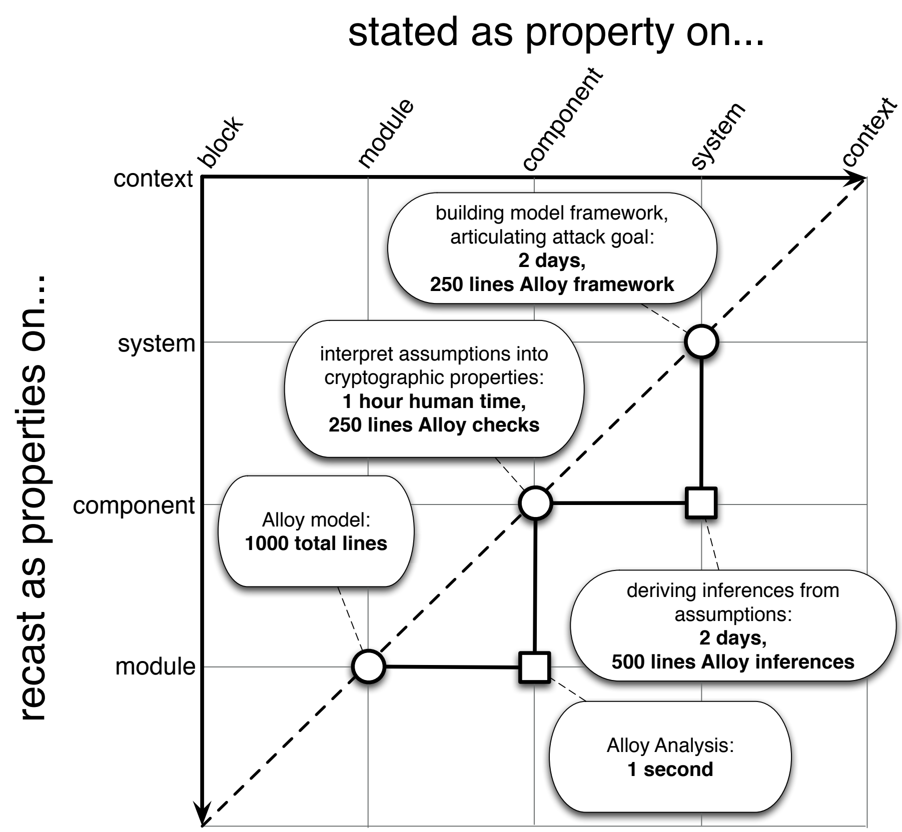

Figure 5-40: Time spent building the secrecy argument.

Chapter 6

Related Work

6.1 Related Work

6.1.1 Requirement Decomposition

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

Assurance and Safety Cases

Assurance and safety cases [4, 49], for example, decompose a critical safety property. They tend to operate at a larger granularity than problem frames, in which the elements represent arguments or large groupings of evidence, rather than constraints. Another class of analyses focus on failures rather than requirements (such as HAZOP [65]), in which decomposition is used to identify the root causes of failures. Our work, like that of assurance cases, provides confidence that a given requirement will hold, rather than establishing that a particular type of error will not occur.

Leveson's STAMP approach involves decomposing design constriants, with a focus on managerial control over the operation of a system [51, 52].

I*, Tropos, KAOS

More similar to our approach are frameworks, such as `i*` [87] and KAOS [21, 22, 18, 8], that decompose system-level properties by assigning properties to agents that work together to achieve the goal. For KAOS, patterns have been developed for refining a requirement into subgoals [22]. In our approach, we have not given a constructive method for obtaining the new constraint systematically, and the refinement strategies of KAOS may fill this gap.

Similar to i*, Tropos [15, 28, 67] is based on actors with different goals for the system and different measures of success. It is focused on early design stages, and is mostly for human-human communication plus some simulation/evaluation support for making sense of larger models.

KAOS refinements has been applied to agent-oriented policy decomposition and applied to Systems of Systems (SoS) [32]. It is used as a means for combatting emergent behaviors that result from independently designed systems combined into a single system.

Four Variable Model

The four-variable model [66, 86] makes a distinction, like Problem Frames, between the requirements, the specification, and domain assumptions. However, in Problem Frame terminology, it assumes that a particular frame always applies, in which there is a machine, an input device domain, an output device domain, and a domain of controlled and monitored phenomena.

Requirement Elicitation

Letier and Lamsweerde show how a goal (requirement) produced from requirement elicitation can be transformed into a specification that is formal and precise enough to guide implementation [48]. That approach is centered around producing operational specifications from requirements expressed in temporal logic, and focuses on proving the correctness of a set of inference patterns. Such inference patterns are correct regardless of context, in contrast to our approach in which transformations are only justified by context-specific domain assumptions.

Refinement

Johnson made an early use of the phrase "deriving specifications from requirements" in 1988 when he showed how requirements written in the relational logic language *Gist* can be transformed into specifications through iterative refinement [43]. Each refinement step places limits on what domains may know and on their ability to control the world, and exceptions are added to global constraints. A specification is not guaranteed to logically imply the requirement it grew out of, and the two descriptions may even be logically inconsistent with each other. In contrast, as we refine (transform) a requirement, the breadcrumbs we add expand our assumptions about the domains rather than restricting them, and a specification will always be consistent with the requirement it enforces.

6.1.2 Problem Frames

Problem Progression

Michael Jackson sketches out a notion of *problem progression* in the Problem Frames book [40]. A problem progression is a sequence of Problem Frame descriptions, beginning with the full description (including the original requirement) and ending with a description containing only the machine and its specification. In each successive description, the domains connected to the requirement are eliminated and the requirement is reconnected and altered as needed. He does not work out the details of how one would derive the successive descriptions, but it seems that he had a similar vision to our own. However, rather than eliminating elements (domains) from the diagram at each step, our approach adds elements (domain assumptions), providing a trace of the analyst's reasoning in a single diagram.

Jackson and Zave use a coin-operated turnstyle to demonstrate how to turn a requirement into a specification by adding appropriate environmental properties (domain assumptions) [41]. Their approach is quite similar to our own, and uses a logical constraint language to express domain assumptions. Our work strives to generalize the process to be applicable in broader and more complex circumstances, and to help guide the analyst through the process with the visual notion of pushing the requirement towards the machine.

Problem Reduction

Rapanotti, Hall, and Li recently introduced *problem reduction*, a technique that uses causal logic to formalize problem progression in Problem Frames [71]. Like our own work, they seek to formalize and generalize problem progression in a way that provides traceability as well as a guarantee of sufficiency. Problem reduction follows the style of problem progression described in the Problem Frames book [40], in which the requirement is moved closer to the machine by eliminating intervening domains.

Calculus of Requirements Engineering

Hall, Rapanotti, Li, and M. Jackson are developing a calculus of requirements engineering based on the Problem Frames approach [44, 54, 55, 70]. They examine how problems and solutions can be restructured to fit known patterns. Part of their technique involves transformation rules for problem progression, in which a requirement (expressed in CSP) is replaced by an equivalent requirement in an alternate form. In contrast, our technique is a form of requirement progression, in which the transformations only change the constraints, not the underlying domain structure. Furthermore, our transformations are not semantics-preserving; they are justified by a set of explicit assumptions rather than proofs of equivalence.

6.1.3 Analysis of the BPTC

Jackson and Jackson have examined the *gantry creep* in the BPTC, in which the angle of delivery slowly shifts over the course of many treatments of the same patient [20].

Rae et al have used lightweight code analysis to determine conditions under which the BPTC emergency stop button would not operate correctly [68].

Dennis et al have shown how commutitivity analysis can be used to detect race conditions between operators of a system, even when that system uses atomic single threaded operations. They apply the technique to the automatic beam scheduler currently employed in the BPTC [24]

In earlier work, we have used the BPTC to motivate the development of a technique for performing requirement progression [78, 79, 80, 81].

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 [40]. The analyst can automatically check the steps of the progression (using Alloy [30, 34]), 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 [76].

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), we saw 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 4.3.7). 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 [60] 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 perform up-front specialist interviews for each component, to build a rough understanding of 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 specialist's (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 it becomes a weak link in the argument and 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* [27]

(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 [23], 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 database 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, we 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 Frames 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 [68] and corrupt network messages could result in patients receiving someone else's treatment (Chapter 4). Similarly, in earlier work [24], 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 [24] 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 [24] 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 [17]).

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 [23], 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 [52] 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

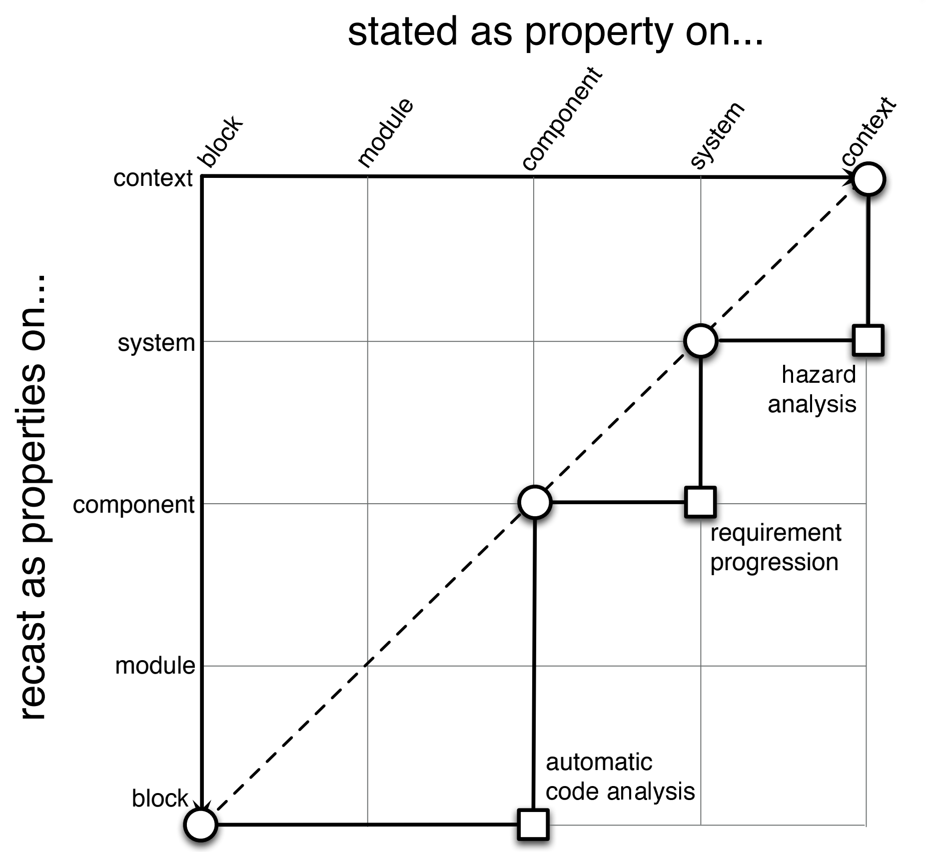

Figure 7-1: Techniques linked together to form an end-to-end dependability argument.

gaining given your current investment in the technique, and the diameter of the glass 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.

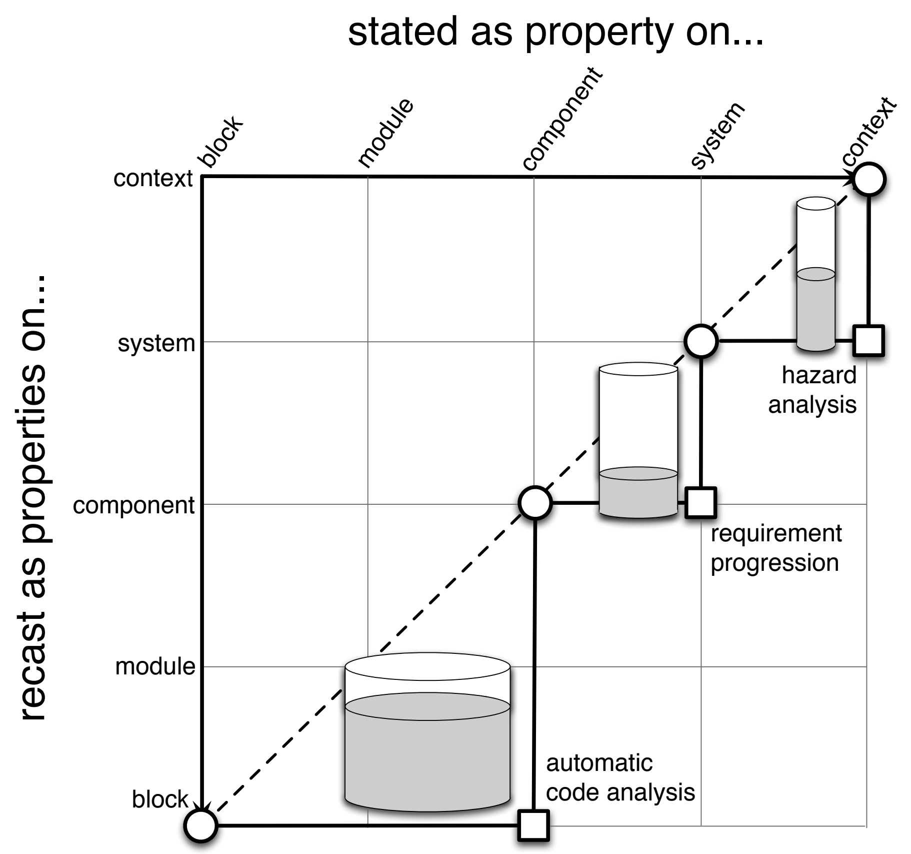

Figure 7-2: Each technique is represented by a glass of water. The height of the glass shows potential confidence gained, the water level shows the current confidence being provided, and the diameter represents return on investment.

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

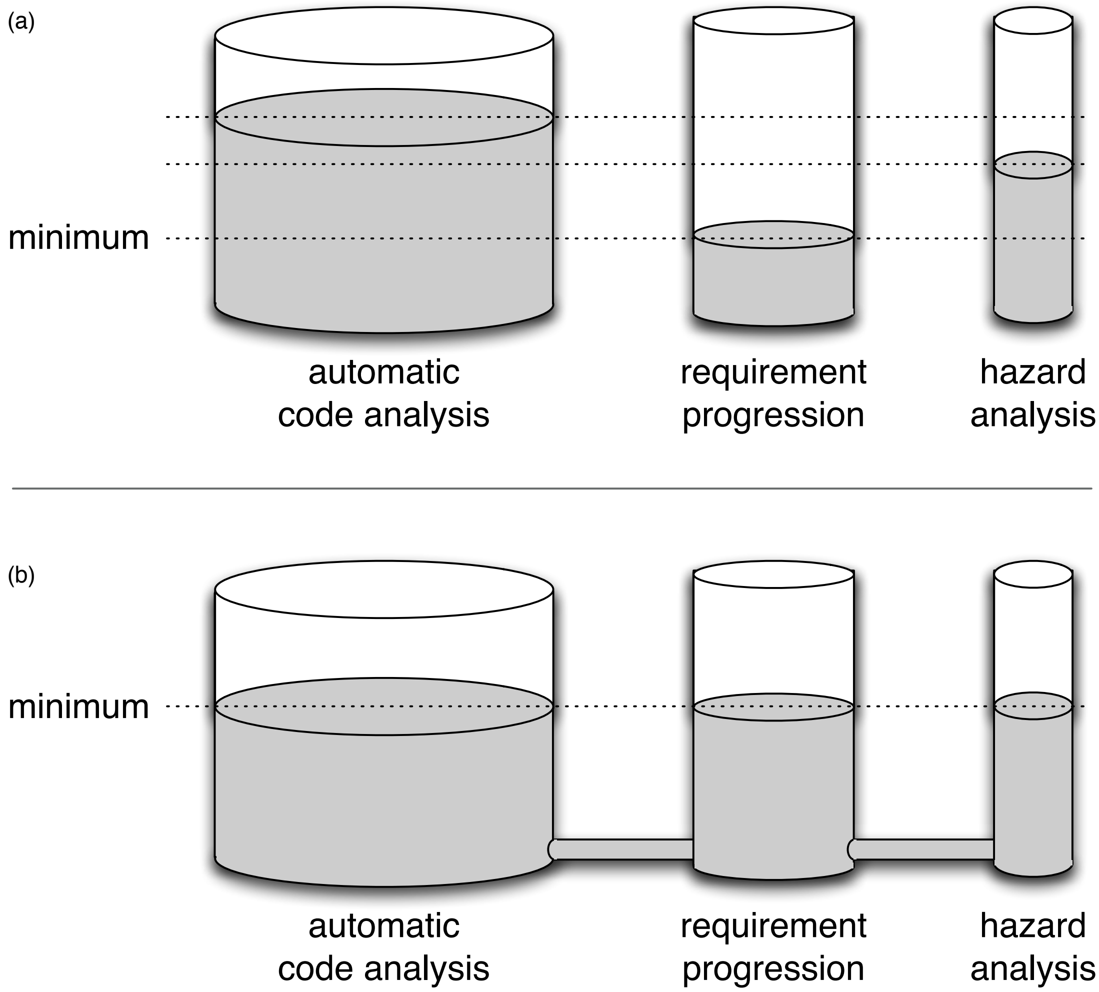

Figure 7-3: Overall confidence in the dependability argument is the minimum confidence of the component techniques.

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 there is 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 represents a technique with a high overhead, such as a custom-build analysis. It takes a lot of work to setup, but then has a 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.

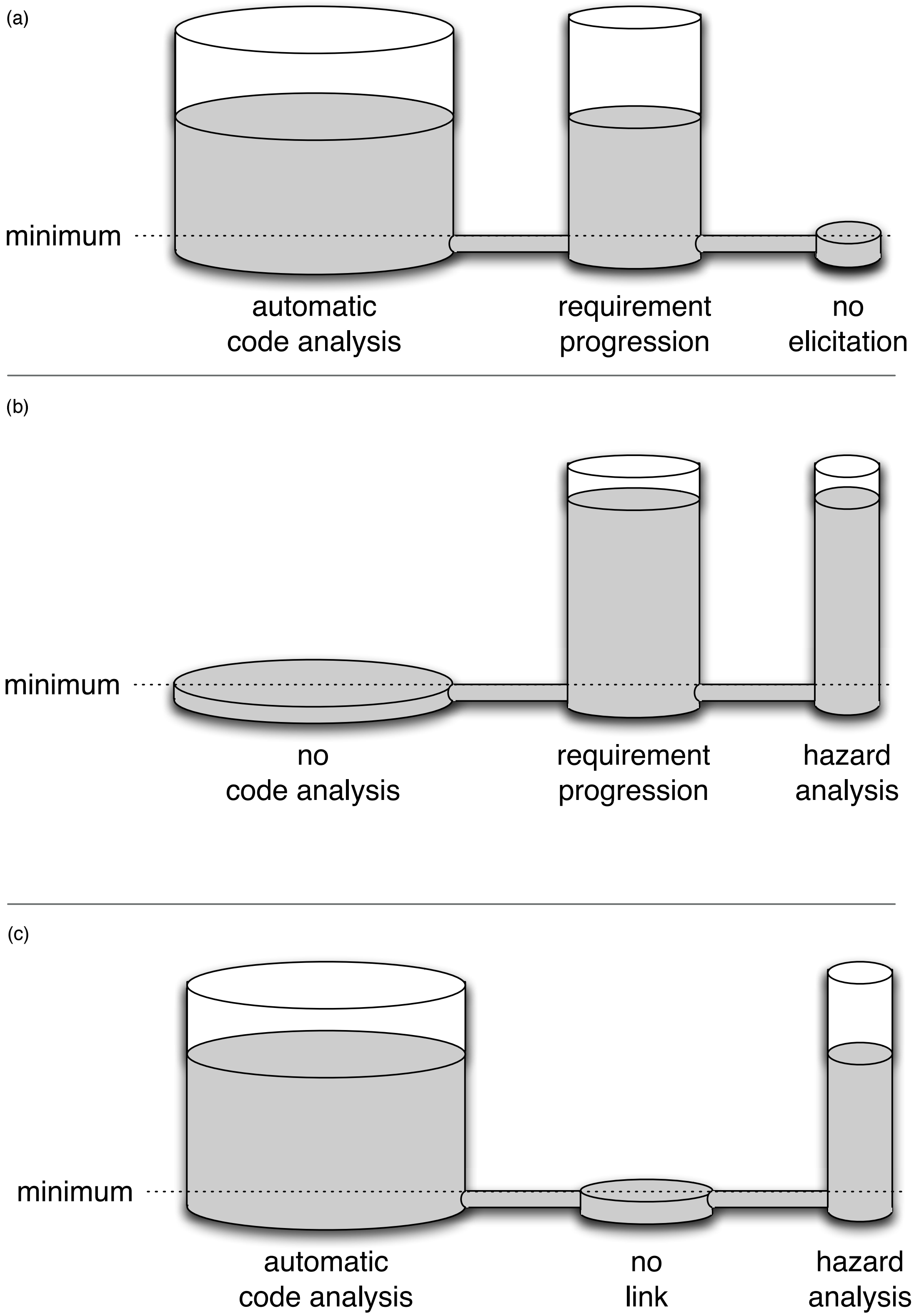

Figure 7-4: Representing errors of omission. Building an incomplete argument greatly harms confidence.

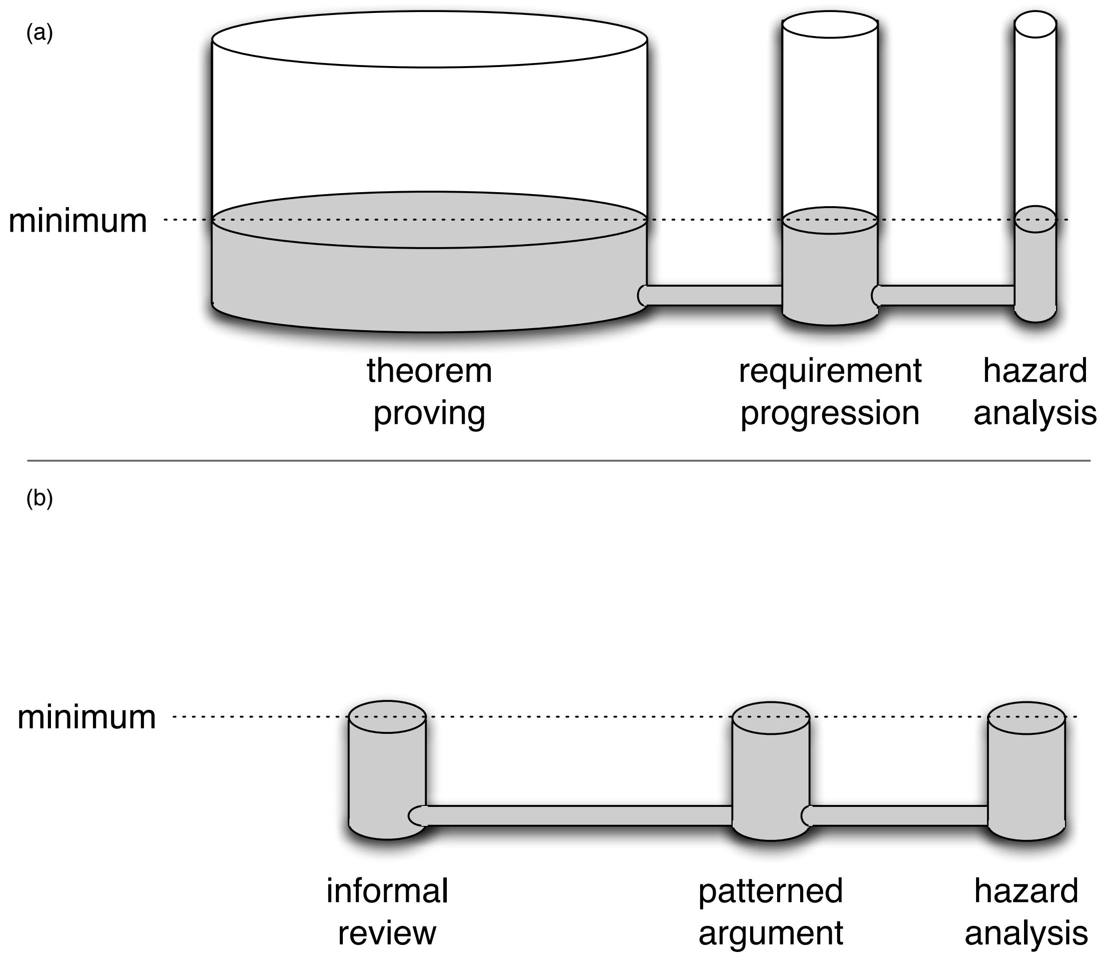

Figure 7-5: Representing errors of technique selection. In the first case, the budget is low so the heavyweight technique sucks all the water out of the other glasses, lowering overall confidence. In the second case, the techniques are lightweight but the budget is high, resulting in wasted budget and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

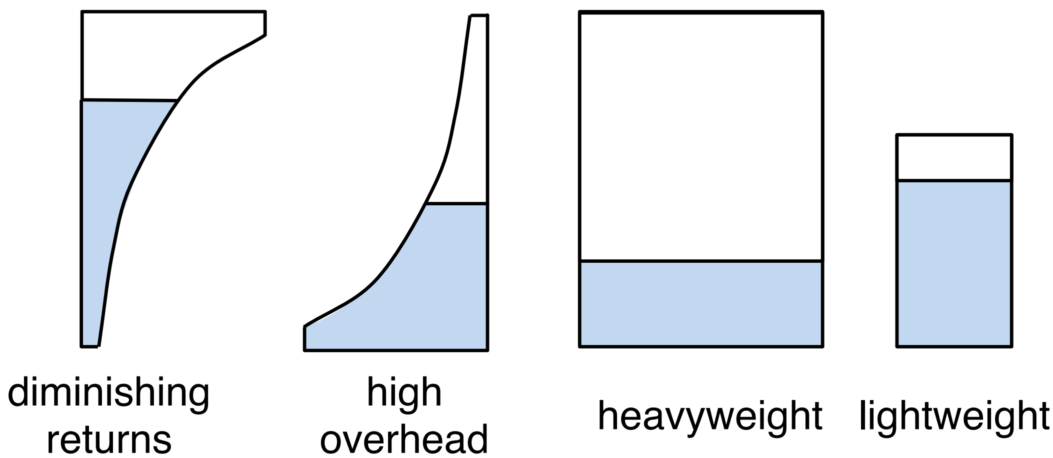

Figure 7-6: Waterglasses have different shapes depending on how their return on investment changes as investment increases.

Chapter 8

Appendix: Automatic Door Model

An Alloy model that checks the requirement progression and resulting argument diagram described in Section 3.6. It permits auotmatic analyses to confirm that the generated domain assumptions are indeed strong enough to enforce the system requirements.

```
/* A model of an automatic door controller, as part of its dependability argument.
 * problem proposed by Nick Ourusoff
 * model created by Robert Seater June 2008
 * last updated August 2008
 */

open util/ordering[Time]
sig Time {
        DistanceSensor: one Int,
        DistanceDoor: one Int,
        DoorGap: one Int,
        MotorSpeed: one Int,
        WalkingSpeed: one Int,
        MotionDetected: one MotionDetectedOption,
        MotorPolarity: one MotorPolarityOption,
        MotorPower: one MotorPowerOption,
        DoorGapMeasure: one DoorGapMeasureOption,
        AppliedForce: one AppliedForceOption,
}

abstract sig MotionDetectedOption {}
one sig Motion, NoMotion extends MotionDetectedOption {}

abstract sig MotorPolarityOption {}
one sig Opening, Closing extends MotorPolarityOption {}

abstract sig MotorPowerOption {}
one sig MotorOn, MotorOff extends MotorPowerOption {}
```

```

abstract sig DoorGapMeasureOption {}
one sig AlmostOpen, AlmostClosed, UnknownGap extends DoorGapMeasureOption {}

abstract sig AppliedForceOption {}
one sig OpeningForce, ClosingForce, NoForce extends AppliedForceOption {}

fact sanity {
        all t: Time | t.DoorGap =< 10
}

/***********************************/
pred ServiceGoal [t: Time] {
        (DistanceDoor[t] =< 1) => (DoorGap[t] >= 9)
        (DistanceDoor[t] >= 11) => (DoorGap[t] =< 1)
}

pred MotorDamage [t: Time] {
        (
                MotorPower[t] = MotorOn
                and MotorPolarity[t] = Opening
                and DoorGap[t] >= 10
        ) or (
                MotorPower[t] = MotorOn
                and MotorPolarity[t] = Closing
                and DoorGap[t] =< 0
        )
}

pred DoorDamage [t: Time] {
        (
                AppliedForce[t] = OpeningForce
                and DoorGap[t] >= 10
        ) or (
                AppliedForce[t] = ClosingForce
                and DoorGap[t] =< 0
        )
}

pred goals [t: Time] {
        ServiceGoal[t]
        !MotorDamage[t]
        !DoorDamage[t]
}

/***********************************/
pred PeopleBC [] {
        //the sensor is located on top of the door
        all t: Time | DistanceDoor[t] = DistanceSensor[t]

        //max walking speed is a constant between 0 and 2 feet per second
        all t,t': Time | WalkingSpeed[t] = WalkingSpeed[t']
        WalkingSpeed[first] >= 0
        WalkingSpeed[first] =< 2
```

```
        //people move up to their max walking speed
        all t: Time, t': t.next {
                (DistanceSensor[t'] >= DistanceSensor[t] - WalkingSpeed[t]
                and
                DistanceSensor[t'] =< DistanceSensor[t] + WalkingSpeed[t])
        }

}

pred MotionSensorBC [] {
        //the sensor has a detection range of 6 feet
        all t: Time | MotionDetected[t] = Motion ⇔ DistanceSensor[t] =< 6
}

pred ControllerBC [] {
        all t: Time {
                (MotionDetected[t] = Motion and DoorGapMeasure[t] != AlmostOpen)
                ⇒
                (MotorPower[t] = MotorOn and MotorPolarity[t] = Opening)
        }
        all t: Time {
                (MotionDetected[t] = NoMotion and DoorGapMeasure[t] != AlmostClosed)
                ⇒
                (MotorPower[t] = MotorOn and MotorPolarity[t] = Closing)
        }
        all t: Time |
                (MotionDetected[t] = Motion and DoorGapMeasure[t] = AlmostOpen)
                        ⇒ MotorPower[t] = MotorOff
        all t: Time |
                (MotionDetected[t] = NoMotion and DoorGapMeasure[t] = AlmostClosed)
                        ⇒ MotorPower[t] = MotorOff
}

pred MotorBC [] {
        //motor speed is 50% per second, and remains constant over time
        all t,t': Time | MotorSpeed[t] = MotorSpeed[t']
        MotorSpeed[first] = 5

        // The motor's power and polarity determine the force applied to the door
        all t: Time |
                MotorPower[t] = MotorOff ⇔ AppliedForce[t] = NoForce
        all t: Time |
                (MotorPower[t] = MotorOn and MotorPolarity[t] = Opening)
                        ⇔ AppliedForce[t] = OpeningForce
        all t: Time |
                (MotorPower[t] = MotorOn and MotorPolarity[t] = Closing)
                        ⇔ AppliedForce[t] = ClosingForce
}

pred DoorBC [] {
        //The applied force on the door bounds how the gap can change, as limited by mo
        all t: Time, t': t.next {
                AppliedForce[t] = OpeningForce ⇒
```

```
                    (DoorGap[t'] = int[DoorGap[t]] + int[MotorSpeed[t]])
        }

        all t: Time, t': t.next {
                (AppliedForce[t] = ClosingForce) ⇒
                        (DoorGap[t'] = int[DoorGap[t]] − int[MotorSpeed[t]])
        }

        all t: Time, t': t.next {
                (AppliedForce[t] = NoForce) ⇒ (DoorGap[t'] = DoorGap[t])
        }
}

pred PositionSensorBC [] {
        all t: Time |
                DoorGapMeasure[t] = AlmostOpen ⇔ DoorGap[t] >=9
        all t: Time |
                DoorGapMeasure[t] = AlmostClosed ⇔ DoorGap[t] =< 1
}

pred breadcrumbs [] {
        PeopleBC
        MotionSensorBC
        ControllerBC
        MotorBC
        DoorBC
        PositionSensorBC
}

/************************************/
pred initialConditions [] {
        goals[first]
        goals[first.next]
        DoorGap[first] >= 0
        DoorGap[first] =< 10
}

assert enforcement {
        breadcrumbs and initialConditions
                ⇒ all t: Time | goals[t]
}

check enforcement for 3 but 6 int, 5 Time
// 5 int gives the bitwidth, thus we are allowed integers in the range [−15,16]
// we use bitwidth 6 to reduce the problems from overflow

pred nice [] {
        breadcrumbs
        initialConditions
        all t: Time | goals[t]
}
run nice for 3 but 6 int, 5 Time
```

Chapter 9

Appendix: BPTC Case Study History

This research project has grown largely out of an ongoing collaboration with the Burr Proton Therapy Center (BPTC), a radiation therapy facility associated with the Massachusetts General Hospital in Boston. It has served as both inspiration for new approaches for ensuring software dependability. and as a reality check for what approaches are realistic on a real, working, safety-critical system.

History of the BPTC

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

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

The BPTC installation has at its core a cyclotron that generates a beam of protons. The beam is multiplexed amongst several treatment rooms, each with its own gantry and nozzle for positioning the beam. Technicians in a master control room supervise the cyclotron and allocate the beam to treatment rooms. Each treatment room is paired with a treatment control room, in which clinicians enter and execute treatment prescriptions.

The patient is placed on a couch which is electromechanically positioned by staff within the treatment room. The beam emitter is also positioned, and its aim verified by staff using X-rays and lights attached to the emitter. The staff then leave the room, and the treatment is initiated from the treatment control room. Treatment consists of irradiating a specific location on the patient using a beam of protons with a defined lateral and longitudinal distribution.

The machine is considered safety critical primarily due to the potential for overdose — treating the patient with radiation of excessive strength or duration. The International Atomic Energy Agency lists 80 separate accidents involving radiation therapy in the United States over the past fifty years. The most famous of these accidents are those involving the Therac-25 machine. Faulty software was a primary cause of the Therac-25 failures. More recently, software appears to have been the main factor in similar accidents in Panama in 2001.

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

Our Experience

The work described in Chapter 3 grew out of the difficulty we encountered with keeping track of a large number of domain properties, relating them appropriately to the requirements and specifications.

Initially, we used problem diagrams simply to describe the BPTC system – keeping track of how domains interacted and recording properties about the domains. As we spent more time interacting with the BPTC engineers, we found that the problem diagrams were not only useful recording information they had told us, but also for indicating what questions to ask. The information they initially gave us was not enough to build a safety case, yet it was not clear what additional information would be. There simply was not time to get full descriptions of all the parts of the system, so we needed to narrow our questions and focus our inquiry.

We found that we could use the structure of a problem diagram to (at least start to) build a safety argument for a requirement and, by doing so, explicitly expose the assumptions we were making about the behavior of different parts of the system. Once those assumptions were exposed and articulated, we could ask the BPTC engineers if they were reasonable. This approach was a big improvement over our earlier attempts to build safety arguments out of the information the engineers volunteered on their own or blindly probing their knowledge of the immensely complex system.

The requirement progression technique described in Chapter 3 and applied in Chapter 4 is a more general and systematic way for doing the kind of reasoning that has helped us communicate with the BPTC. This method can either be used as a means of focusing requirements elicitation, or it can be used to build an auditable argument – one in which an outside reviewer can understand why the argument is correct. We originally developed it to help us do the former task, although our current work focuses more on the latter task.

Chapter 10

Appendix: Requirement Progression Model

This Alloy model is analyzable with the current, freely available, version 4 of the Alloy Analyzer [30].

```
module requirementProgression
open util/ordering[Diagram] as ord
-- for effective visualization, project over Diagram

/************************************/
/* defining a problem diagram */
/************************************/

sig Phenomenon, Domain, Constraint {}

-- the anatomy of a problem diagram
sig Diagram {
  phenomena: set Phenomenon,
  domains, machines: set Domain,
  constraints, requirements, specifications: set Constraint,
  connects: Domain -> Domain,
  involves: Domain -> Phenomenon,
  touches: Constraint -> Domain,
  mentions: Constraint -> Phenomenon
  }

pred wellFormedDiagram [x: Diagram] {
  -- relations do not cross between diagrams
  selfContained[x]
  -- there is exactly one machine
  one x.machines
  -- domains connect iff they involve a shared phenomenon
  connectIffShare[x]
  -- diagrams are non-trivial
  nonTrivial[x]
  -- all constraints are well formed
  all c: x.constraints | wellFormedConstraint[c,x]
}

run wellFormedDiagram for 4
run wellFormedDiagram for 35 --2 minutes to solve
```

```
/*************************************/
/* helper functions for well formedness */
/*************************************/

-- domains connect iff they involve a shared phenomenon
-- domains do not connect to themselves
pred connectIffShare [x: Diagram] {
  all d,d': Domain |
    d' in x.connects[d] <=>
      (d != d' and some x.involves[d] & x.involves[d'])
}

pred selfContained [x: Diagram] {
  -- domains don't connect to domains in other diagrams
  (x.domains).(x.connects) in (x.domains)
  -- domains do not involve phenomena from other diagrams
  (x.domains).(x.involves) in (x.phenomena)
  -- requirements and specifications are not from other diagrams
  x.requirements + x.specifications in x.constraints
  -- the machine is not in another diagram
  x.machines in x.domains
}

pred nonTrivial [x: Diagram] {
  -- each constraint mentions some phenomena
  all c: x.constraints | some x.mentions[c]
  -- each domain involves some phenomena
  all d: x.domains | some x.involves[d]
  -- the diagram is connected
  all d,d': x.domains | d' in d.*(x.connects)
  -- there is at least one non-machine domain
  some x.domains - x.machines
}

pred wellFormedConstraint [c: Constraint, x: Diagram] {
  -- constraints can only touch the domains that involve phenomena they mention
  -- constraints must touch the domains that involve the phenomena they mention
  all p: x.mentions[c] | some d: x.touches[c] | p in x.involves[d]
  all d: x.touches[c] | some x.involves[d] & x.mentions[c]
  -- specifications only touch machines
  c in  x.specifications <=> x.touches[c] in x.machines
  -- c is contained entirely within x
  fullyContainedConstraint [c, x]
}

pred fullyContainedConstraint [c: Constraint, x: Diagram] {
  -- c must be one of x's constraints
  c in x.constraints
  -- constraints do not touch domains in other diagrams
  x.touches[c] in x.domains
  -- constraints do not mention constraints in other diagrams
  x.mentions[c] in x.phenomena
}
```

```
/************************************/
/* requirement progression transformations */
/************************************/

pred addBreadcrumb [before, after: Diagram] {
  --nothing changes except for the addition of a single breadcrumb
  structureEquivalent[before, after]
  some bc: Constraint {
    addConstraint[bc, before, after]
    -- bc is a well formed valid breadcrumb
    one after.touches[bc]
    wellFormedConstraint[bc, after]
    -- bc is not a requirement or a spec
    bc !in after.requirements + after.specifications
  }
}

pred rephraseRequirement [before, after: Diagram] {
  --nothing changes except for r' replacing r
  structureEquivalent[before, after]
  some r: before.requirements, r': after.requirements {
    wellFormedConstraint[r', after]
    replace[r,r',before,after]
    onlyChanges[r, r', before, after]

    -- r and r' have different phenomena but same domains
    before.mentions[r] != after.mentions[r']
    before.touches[r] = after.touches[r']

    -- the change is justified by the other constraints
    implication[after.constraints, r, after]
  }
}

pred pushRequirement [before, after: Diagram] {
  structureEquivalent[before, after]
  onlyTouchesChanges[before, after]
  -- one requirement changes what it touches
  some r: before.requirements & after.requirements {
    before.touches[r] != after.touches[r]
    before.touches - (r -> univ) = after.touches - (r -> univ)
    wellFormedConstraint[r, after]
  }
}
```

```
/************************************/
/* simulation and invariant preservation */
/************************************/

pred commonTransformation [x,x': Diagram] {
  some y,z: Diagram {
    addBreadcrumb[x, y]
    rephraseRequirement[y, z]
    pushRequirement[z,x']
  }
}

pred someTransformation [x,y: Diagram] {
  addBreadcrumb[x,y] or
  rephraseRequirement[x,y] or
  pushRequirement[x,y] or
  commonTransformation[x,y]
}

pred simulation [] {
  -- the first diagram is well formed
  wellFormedDiagram[first[]]
  -- it has no spec,
  no first[].specifications
  -- and it has a requirement
  some first[].requirements

  -- a spec is eventually derived via transformations
  some final: Diagram - first[] {
    all x: prevs[final] | someTransformation[x,next[x]]
    final.requirements in final.specifications
  }
}
run simulation for 4

assert wellFormednessPreservation {
  all x,y: Diagram |
    wellFormedDiagram[x] and someTransformation[x,y]
      => wellFormedDiagram[y]
}
check wellFormednessPreservation for 4
-- the check executes faster if commonTransformation
-- is eliminated from someTransformation
```

```
/************************************/
/* helper functions for the transformations */
/************************************/

-- add a non-requirement, non-specification constraint
pred addConstraint [c: Constraint, x,y: Diagram] {
  onlyChange[c, x, y]
  c = y.constraints - x.constraints
  x.requirements = x.requirements
  y.specifications = y.specifications }

-- only constraints, requirements, specifications, touches, mentions vary
pred structureEquivalent [x,y: Diagram] {
  x.domains = y.domains
  x.machines = y.machines
  x.phenomena = y.phenomena
  x.connects = y.connects
  x.involves = y.involves }

-- approximates meaning of the implication (a_0 ^ a_1 ^ ... ^ a_n => b)
pred implication [a: set Constraint, b: Constraint, x: Diagram] {
  x.mentions[b] in x.mentions[a] }

-- r disappears and r' appears to replace it
pred replace [r,r': Constraint, x,y: Diagram] {
    r in x.requirements
    r !in y.requirements
    r' !in x.requirements
    r' in y.requirements }

-- only constraints in c change
pred onlyChange [c: set Constraint, x,y: Diagram] {
  x.specifications - c = y.specifications - c
  x.touches - (c -> univ) = y.touches - (c -> univ)
  x.requirements - c = y.requirements - c
  x.constraints - c = y.constraints - c
  x.specifications - c = y.specifications - c
  x.mentions - (c -> univ) = y.mentions - (c -> univ) }

-- only constraint c changes
pred onlyChange [c: set Constraint, x,y: Diagram] {
  onlyChanges [c,c,x,y] }

-- only changes are c in x and c' in y'
pred onlyChanges [c,c': set Constraint, x,y: Diagram] {
  x.specifications - c = y.specifications - c'
  x.touches - (c -> univ) = y.touches - (c' -> univ)
  x.requirements - c = y.requirements - c'
  x.constraints - c = y.constraints - c'
  x.specifications - c = y.specifications - c'
  x.mentions - (c -> univ) = y.mentions - (c' -> univ) }


-- nothing but the touches relation differs between x and y
```

```
pred onlyTouchesChanges[x,y: Diagram] {
  x.requirements = y.requirements
  x.constraints = y.constraints
  x.mentions = y.mentions }
```

Chapter 11

Appendix: Voting Fidelity Model

The full fidelity model for the cryptographic voting case study.

```
/*
 * A model describing a secure voting procedure.
 *
 * The voting scheme was developed, in part, by Peter Ryan.
 * This model was developed by Rob Seater and Eunsuk Kang,
 * with help from Emina Torlak.
 *
 * model created 3-24-08
 */
module voting

// Someone who is capable of voting (but not necessary authorized to do so)
sig Voter {
  // Each voter wants zero or one candidates to win.
  intention: lone Candidate,

  // Each voter receives a set of 0 or more ballots.
  voterBallot: set Ballot,
}

// Some voters are registered voters; they allowed to vote
//  and are assumed to show up to vote.
sig RegisteredVoter extends Voter {}
```

```
// The name of a candidate who is running in the election.
sig Candidate {
  // The total number of votes computed by the voting method for this candidate.
  // A candidate can only get one number of votes,
  //  but we don't yet say how they are computed.
  score: one Int
}

// A checkable location on a ballot/receipt;
//  e.g. ``the third box down from the top''
sig Position {}

// A ballot is the piece of paper given to a voter.
// It consists of a list of candidates next to a list of checkboxes (positions).
// Each candidate name is at some (vertical) position on the ballot.
// Different ballots may have different arrangements/orderings of those names.
// The list of checkboxes can be torn off from the list of candidates,
//  forming the ``receipt''.
// At the bottom of the receipt is an onion -- an encrypted representation
//  of the order of the candidate names on the ballot.
sig Ballot {
  // the order in which the candidate names appear on the ballot
  // each position on a ballot lists one candidate
  // each candidate is listed at one position on a ballot
  arrangement: Position one → one Candidate,

  //each ballot has exactly one Receipt stuck to it
  ballotReceipt: one Receipt,

  // Helper relation: makes the model more readable but is really
  //  just sugar (constrained by the appended fact).
  // The candidate indicated by this ballot, based on the ballot's
  //  arrangement and the position marked on the receipt.
  ballotCandidate: lone Candidate,
}{
  //defining constraint for the ballotCandidate helper relation
  ballotCandidate = ballotReceipt.marked.arrangement
}
```

```
// an Onion is an encrypted representation of an arrangement of candidate names
// according to a given onion, each candidiate name is given one position,
//  and each position holds one candidate name
sig Onion {
  // the order in which the candidate names appear, according to the onion
  // each position on a ballot lists one candidate
  // each candidate is listed at one position on a ballot
  arrangement: Position one → one Candidate,
}

// A receipt is attached to a ballot, but can be torn off and separated.
// It has a set of positions (arranged vertically so they can line up
//  with a list of candidates), any of which can be checked off.
// We assume that at most one box is checked -- otherwise the ballot is
//  voided and the voter is given a new one.
sig Receipt {
  // each receipt has exactly 1 onion associated with it
  receiptOnion: one Onion,

  // Zero or one positions on the receipt have been marked
  //  (e.g. checked off by a voter)
  marked: lone Position,

  // Helper relation: makes the model more readable but is really
  //  just sugar (constrained by the appended fact).
  // The candidate indicated by this receipt, based on it's onion's
  //  arrangement and the marking on the receipt.
  receiptCandidate: lone Candidate,
}{
  // the defining constraint for the receiptCandidate helper relation
  receiptCandidate = marked.(receiptOnion.arrangement)
}
```

```
// A record is a decrypted version of a receipt;
// it's arrangement and marking is visible for all to see,
// although it's connect to a voter has hopefully be obscured.
sig Record {
  // the order in which the candidate names appear on the record
  // each position on a ballot lists one candidate
  // each candidate is listed at one position on a ballot
  arrangement: Position one → one Candidate,

  // zero or one positions on the receipt have been marked, indicating
  // a vote for the candidate at that position
  marked: lone Position,

  // Helper relation: makes the model more readable but is really just
  //  sugar (constrained by the appended fact).
  // The candidate indicated by this record, based on it's arrangement
  //  and marking.
  recordCandidate: lone Candidate,
}{
  // the defining constraint for the recordCandidate helper relation
  recordCandidate = marked.arrangement
}

// The voting board takes in the receipts from ballots, then outputs
//  a set of records.
// The input receipts are re-ordered, and their onions re-encrypted.
// The output records have their onions decoded and made public,
//  so that they can be counted.
// There is only one voting board.
one sig Board {
  //the the reordering done by the voting board
  // each input receipt corresponds to 0 or 1 output receipts
  //  (losses are by default allowed)
  // each output receipt corresponds to 0 or 1 input receipts
  //  (spontaneous generation is by default allowed)
  scramble: Receipt lone → lone Record
}

// A helper function that represents the scrambling done by the voting board.
// Because there is only one voting board, we can represent the
//  ternary relation <board, input, output>
// as a binary relation <input, output> without any ambiguity.
fun mix[] : Receipt → Record { Board.scramble }
```

```
/* LIVENESS SIMULATION */
// generates an interesting election
pred liveness [] {
  // at least one voter wants to vote for at least one candidate
  some c: Candidate | #(RegisteredVoter & intention.c) > 0

  // all of our assumptions hold
  VoterBreadcrumb
  BallotBreadcrumb
  BoardBreadcrumb

  // There is at least one ballot that doesn't get counted
  //  (but it could be one that wasn't used)
  #Receipt != #Ballot

  //there is at least one unused record floating around
  some Record - Receipt.mix

  // there are at least 2 onions that represent the same ordering of candidates
  some disj o,o': Onion | o.arrangement = o'.arrangement
}
run liveness expect 1


/* REQUIREMENT PROGRESSION SETUP */
// The system requirement is the initial version of our goal constraint
// The score for a candidate is exactly the number of people who
//  intended to vote for that candidate.
pred Goal0 [] {
  all c: Candidate |
    c.score = #(RegisteredVoter & intention.c)
}
```

```
/* VOTER TO BALLOT */
// The first rewriting of the goal.
// The score for a candidate is the same as the number of ballots that
//  get marked for that candidate
pred Goal1 [] {
  all c: Candidate |
    c.score = #(ballotCandidate.c & Ballot)
}

//assumptions made about the voter domain
pred VoterBreadcrumb [] {
  // ballots that aren't given to voters don't get marked
  all b: Ballot - Voter.voterBallot | no b.ballotReceipt.marked

  // every registered voter gets exactly one ballot
  all v: RegisteredVoter | one v.voterBallot

  // no unregistered voter gets a ballot
  all v: Voter - RegisteredVoter | no v.voterBallot

  // each ballot is given to at most one voter
  all b: Ballot | lone voterBallot.b

  // Voters mark the ballots they are given according to their
  //  intention and the ordering on the ballot.
  // Note that they don't pay any attention to the onion or the
  //  ordering it represents.
  all v: RegisteredVoter | let b = v.voterBallot |
    b.ballotReceipt.marked.(b.arrangement) = v.intention
}

// the voter breadcrumb justifies replacing Goal0 with Goal1
assert partialClaim1 {
  VoterBreadcrumb and Goal1 => Goal0
}
check partialClaim1 expect 0
```

```
/* BALLOT TO BOARD */
// The second rewriting of the goal.
// The score for a candidate is the same as the number of receipts that,
//  according to the marking made on the receipt and according to the
//  ordering encoded in the onions of those receipts, favor that candidate.
pred Goal2 [] {
  all c: Candidate |
  c.score =
  #(Ballot.ballotReceipt & receiptCandidate.c)
}

// assumptions made about the ballot domain
pred BallotBreadcrumb [] {
  // different ballots have different receipts attached to them
  all disj b,b': Ballot | b.ballotReceipt != b'.ballotReceipt

  // A ballot's onion accurately encodes the arrangement of candidates
  //  shown on the ballot.
  // That is the, the order of candidates on the ballot is the same as
  //  the order of candidates encoded in the onion.
  all b: Ballot | b.arrangement = b.ballotReceipt.receiptOnion.arrangement
}

// the ballot breadcrumb justifies replacing Goal1 with Goal2
assert partialClaim2 {
  BallotBreadcrumb and Goal2 => Goal1
}
check partialClaim2 expect 0
```

```
/* BOARD TO RECORD */
// The third rewriting of the goal.
// The score for a candidate is the same as the number of votes that the
//  candidates gets according to the output (records) of the election board.
pred Goal3 [] {
  all c: Candidate |
    c.score = #(Receipt.mix & recordCandidate.c)
}

// assumptions about the voting board
pred BoardBreadcrumb[] {
  // Every ballot receipt gets sent into the board exactly once.
  // Receipts are transformed but not destroyed:
  //  each receipt going into the board corresponds to 1 record coming out
  all input: Ballot.ballotReceipt | one input.mix

  // Receipts are transformed but not created:
  //  each record coming out of the board corresponds to 1 receipt going in
  all output: Receipt.mix | one mix.output

  // all receipts from all ballots go into the voting board for scrambling
  all b: Ballot | b.ballotReceipt in mix.Record

  // only receipts from ballots go into the voting board for scrambling
  all r: mix.Record | r in Ballot.ballotReceipt

  // The scrambling process may change the onions, but it does not change
  //  the candidate orderings they represent or the markings made.
  // That is, for every input receipt, there is a corresponding output record
  //  with the same arrangement of candidates and same marked position
  all input: mix.Record | let output = input.mix {
    input.receiptOnion.arrangement = output.arrangement
    input.marked = output.marked
    }
}

// the board breadcrumb justifies replacing Goal2 with Goal3
assert partialClaim3 {
  BoardBreadcrumb and Goal3 ⇒ Goal2
}
check partialClaim3 expect 0

// Since goal3 connects to only one domain, it has become a breadcrumb
//  (for Public Record) and progression is complete.
```

```
/* EQUIVALENCE CLAIMS */

//all the breadcrumb domain assumptions put together
pred allBreadcrumbs [] {
  VoterBreadcrumb
  BallotBreadcrumb
  BoardBreadcrumb
}

// Checks that it is ok to use a more compact & efficient style for
//  counting votes.
assert equivalence {
  // counting the number of voters who intend to vote for a given candidate
  allBreadcrumbs => all c: Candidate |
    #(RegisteredVoter & intention.c)
    = #{v: RegisteredVoter | v.intention = c}

  // Counting the number of ballot that are marked in favor of a given
  //  candidate.
  allBreadcrumbs => all c: Candidate |
    #{b: Ballot | b.ballotReceipt.marked.(b.arrangement) = c}
    = #(RegisteredVoter & intention.c)

  // Counting the number of receipts that are marked in favor of a given
  // candidate, according to the arrangements given by their onions.
  allBreadcrumbs => all c: Candidate |
    #{b: Ballot | b.ballotReceipt.marked.
      (b.ballotReceipt.receiptOnion.arrangement) = c}
    = #(Ballot.ballotReceipt & receiptCandidate.c)

  // Counting the number of records outputted by the board that are marked
  //  in favor of a given candidate,
  //  according to the arrangements given by their onions.
  allBreadcrumbs => all c: Candidate |
    #{r: Board.scramble[univ] | (r.marked).(r.arrangement) = c}
    = #(Receipt.mix & recordCandidate.c)
}
check equivalence expect 0
```

Chapter 12

Appendix: Voting Secrecy Model

The full secrecy model for the cryptographic voting case study.

```
/* A model of what information about a voting system can be observed,
 * and how an adversary can use inferences to attack that system.
 * Created 9-12-08 (by Rob Seater), updated 11-11-08 (by Rob Seater).
 * This model uses an extention of the event-based idiom from Daniel Jackson's
 * book Software Abstractions (p. 197).
 *
 * If you need to add a new inference, just add a new signature paragraph in a
 * parallel style to the existing inferences. No other parts of the model
 * need changing.
 *
 * The current inference rules are derived (by hand) from the assumptions,
 * allowing the adversary to attack the system using information that was
 * useful for proving the system correct. As a naming convention, all
 * inferences derived from the same assumptions are given the same number
 * (e.g. a_inf_5 and b_inf_5 would both be based on the same assumption).
 * By convention, a "d" is used to name for Record variables, and
 * "r" is reserved for Receipt variables.
 */
```

```
/*********************/
/* TEMPORAL FRAMEWORK */
/*********************/
open util/ordering[Time]
sig Time {
        comingAttractions: set Inference,
        pastAttractions: set Inference
}
fact history_matches_prophesy {
        all t: Time | t.comingAttractions = t.next.pastAttractions
        no first.pastAttractions
        all t: Time - last | t.comingAttractions.pre = t
        all t: Time - first | t.pastAttractions.post = t
}
```

```
/**************/
/* SIGNATURES */
/**************/
// Each relation in the fidelity model is mirrored by a "known_" version
//  with an extra time column at the end.
sig Voter {
        intention: set Candidate,
        known_intention: Candidate → Time,
        voterBallot: set Ballot,
        known_voterBallot: Ballot → Time,
        known_RegisteredVoter: set Time,
}
sig RegisteredVoter in Voter {}
sig Candidate {
        score: set Int,
        known_score: Int → Time,
}
sig Ballot {
        ballotReceipt: set Receipt,
        known_ballotReceipt: Receipt → Time,
        ballotCandidate: set Candidate,
        known_ballotCandidate: Candidate → Time,
        ballotArrangement: Position → Candidate,
        known_ballotArrangement: Position → Candidate → Time,
}
sig Position {}
sig Receipt {
        receiptMarked: set Position,
        known_receiptMarked: Position → Time,
        receiptOnion: set Onion,
        known_receiptOnion: Onion → Time,
        receiptCandidate: set Candidate,
        known_receiptCandidate: Candidate → Time,
}
sig Onion {
        onionArrangement: Position → Candidate,
        known_onionArrangement: Position → Candidate → Time,
}
sig Record {
        recordArrangement: Position → Candidate,
        known_recordArrangement: Position → Candidate → Time,
        recordMarked: set Position,
        known_recordMarked: Position → Time,
        recordCandidate: set Candidate,
        known_recordCandidate: Candidate → Time,
}
sig Board {
        scramble: Receipt → Record,
        known_scramble: Receipt → Record → Time,
}
fun mix[] : Receipt → Record { Board.scramble }
fun known_mix[] : Receipt → Record → Time { Board.known_scramble }
```

```
/******************/
/* INFERENCE RULES */
/******************/
abstract sig Inference {
  pre, post: one Time
}
sig pause extends Inference {} {} //the trivial inference that learns nothing


abstract sig intention_Inference extends Inference {
  used_voter_from_intention: one Voter,
  used_candidate_from_intention: one Candidate,
}

// VoterBreadcrumb
// all v: RegisteredVoter | let b = v.voterBallot |
//    b.ballotReceipt.receiptMarked.(b.ballotArrangement) = v.intention
sig intention_inference_1 extends intention_Inference {}{
  //what you learn
  (used_voter_from_intention → used_candidate_from_intention)
    in known_intention.post
  (used_voter_from_intention → used_candidate_from_intention)
    not in known_intention.pre

  //when you can learn it
  used_voter_from_intention in known_RegisteredVoter.pre
  let b = used_voter_from_intention.(known_voterBallot.pre) |
    b.(known_ballotReceipt.pre).(known_receiptMarked.pre)
         .(b.(known_ballotArrangement.pre)) = used_candidate_from_intention
}


abstract sig voterBallot_Inference extends Inference {
  used_voter_from_voterBallot: one Voter,
  used_ballot_from_voterBallot: one Ballot,
}

// VoterBreadcrumb
// all v: RegisteredVoter | let b = v.voterBallot |
//    b.ballotReceipt.receiptMarked.(b.ballotArrangement) = v.intention
sig voterBallot_inference_1 extends voterBallot_Inference {}{
  //what you learn
  (used_voter_from_voterBallot → used_ballot_from_voterBallot)
    in known_voterBallot.post
  (used_voter_from_voterBallot → used_ballot_from_voterBallot)
    not in known_voterBallot.pre

  //when you can learn it
  used_voter_from_voterBallot in known_RegisteredVoter.pre
  used_ballot_from_voterBallot.(known_ballotReceipt.pre).(known_receiptMarked.pre)
       .(used_ballot_from_voterBallot.(known_ballotArrangement.pre))
    = used_voter_from_voterBallot.(known_intention.pre)
  some used_voter_from_voterBallot.(known_intention.pre)
}
```

```
abstract sig RegisteredVoter_Inference extends Inference {
  used_voter_from_RegisteredVoter: one Voter,
}

// VoterBreadcrumb
//   all v: Voter - RegisteredVoter | no v.voterBallot
sig RegisteredVoter_inference_A extends RegisteredVoter_Inference {}{
  //what you learn
   (used_voter_from_RegisteredVoter) in known_RegisteredVoter.post
  (used_voter_from_RegisteredVoter) not in known_RegisteredVoter.pre

  //when you can learn it
  some used_voter_from_RegisteredVoter.(known_voterBallot.pre)
}


abstract sig score_Inference extends Inference {
  used_candidate_from_score: one Candidate,
  used_score_from_score: one Int,
}

// System Fidelity Requirement
//   all c: Candidate |
//     c.score = #(RegisteredVoter & intention.c)
sig score_inference_A extends score_Inference {}{
  //what you learn
  (used_candidate_from_score -> used_score_from_score) in known_score.post
  (used_candidate_from_score -> used_score_from_score) not in known_score.pre

  //when you can learn it
  // The first line is technically cheating, since it mentions a non-"known_"
  // variable, while an adversary should never directly access such information.
  // We use it to replicate the effect of adding negative knowledge to the system,
  // without the need to add a lot of additional complexity to the model. Were
  // more inferences to use negative knowledge, it could be added to the model by
  // adding a "known_not_" version of each knowable relation, representing tuples
  // that the adversary knows are not part of that relation. We have omitted
  // those relations since they are not relevant to any inference but this one.
  //
  // This relation says that we know everyone's registration status.
  // In this problem domain, we can get away with this, since voter registration
  // is public knowledge. To reflect this assumption, note the final constraint
  // in the seededKnowledge predicate.
  //
  // The second and third lines are legit according to our style, and mirror the
  // constraint given in the system fidelity requirement.
  RegisteredVoter = known_RegisteredVoter.pre
  all v: known_RegisteredVoter.pre | some v.(known_intention.pre)
  used_score_from_score = #((known_RegisteredVoter.pre)
    & (known_intention.pre).used_candidate_from_score)
}
```

```
abstract sig ballotReceipt_Inference extends Inference {
  used_ballot_from_ballotReceipt: one Ballot,
  used_receipt_from_ballotReceipt: one Receipt,
}

// VoterBreadcrumb
// all v: RegisteredVoter | let b = v.voterBallot |
//   b.ballotReceipt.receiptMarked.(b.ballotArrangement) = v.intention
sig ballotReceipt_inference_1 extends ballotReceipt_Inference {}{
  //what you learn
  (used_ballot_from_ballotReceipt → used_receipt_from_ballotReceipt)
    in known_ballotReceipt.post
  (used_ballot_from_ballotReceipt → used_receipt_from_ballotReceipt)
    not in known_ballotReceipt.pre

  //when you can learn it
  some v: known_RegisteredVoter.pre {
    v.(known_voterBallot.pre) = used_ballot_from_ballotReceipt
    used_receipt_from_ballotReceipt.(known_receiptMarked.pre)
        .(v.(known_voterBallot.pre).(known_ballotArrangement.pre))
      = v.(known_intention.pre)
    some v.(known_intention.pre)
  }
}

// AppendedFacts
//   all b: Ballot |
//   (b.ballotCandidate) = (b.ballotReceipt).receiptMarked.(b.ballotArrangement)
sig ballotReceipt_inference_5 extends ballotReceipt_Inference {}{
  //what you learn
  (used_ballot_from_ballotReceipt → used_receipt_from_ballotReceipt)
    in known_ballotReceipt.post
  (used_ballot_from_ballotReceipt → used_receipt_from_ballotReceipt)
    not in known_ballotReceipt.pre

  //when you can learn it
  used_ballot_from_ballotReceipt.(known_ballotCandidate.pre)
    = used_receipt_from_ballotReceipt.(known_receiptMarked.pre)
      .(used_ballot_from_ballotReceipt.(known_ballotArrangement.pre))
}
```

```
abstract sig ballotCandidate_Inference extends Inference {
  used_ballot_from_ballotCandidate: one Ballot,
  used_candidate_from_ballotCandidate: one Candidate,
}

// AppendedFacts
//    all b: Ballot |
//      (b.ballotCandidate) = (b.ballotReceipt).receiptMarked.(b.ballotArrangement)
sig ballotCandidate_inference_5 extends ballotCandidate_Inference {}{
  //what you learn
  (used_ballot_from_ballotCandidate → used_candidate_from_ballotCandidate)
    in known_ballotCandidate.post
  (used_ballot_from_ballotCandidate → used_candidate_from_ballotCandidate)
    not in known_ballotCandidate.pre

  //when you can learn it
  used_candidate_from_ballotCandidate
    = used_ballot_from_ballotCandidate.(known_ballotReceipt.pre)
      .(known_receiptMarked.pre).(used_ballot_from_ballotCandidate
      .(known_ballotArrangement.pre))
}
```

```
abstract sig ballotArrangement_Inference extends Inference {
  used_ballot_from_ballotArrangement: one Ballot,
  used_position_from_ballotArrangement: one Position,
  used_candidate_from_ballotArrangement: one Candidate,
}

// VoterBreadcrumb
// all v: RegisteredVoter | let b = v.voterBallot |
//    b.ballotReceipt.receiptMarked.(b.ballotArrangement) = v.intention
sig ballotArrangement_inference_1 extends ballotArrangement_Inference {}{
  //what you learn
  (used_ballot_from_ballotArrangement → used_position_from_ballotArrangement
    → used_candidate_from_ballotArrangement) in known_ballotArrangement.post
  (used_ballot_from_ballotArrangement → used_position_from_ballotArrangement
    → used_candidate_from_ballotArrangement) not in known_ballotArrangement.pre

  //when you can learn it
  some v: known_RegisteredVoter.pre {
    v.(known_voterBallot.pre)
      = used_ballot_from_ballotArrangement
    used_ballot_from_ballotArrangement.(known_ballotReceipt.pre)
        .(known_receiptMarked.pre) = used_position_from_ballotArrangement
    used_candidate_from_ballotArrangement
      = v.(known_intention.pre)
  }
}

// BallotBreadcrumb
//    all b: Ballot | b.ballotArrangement = b.ballotReceipt.receiptOnion.onionArrangemen
sig ballotArrangement_inference_2 extends ballotArrangement_Inference {}{
  //what you learn
  (used_ballot_from_ballotArrangement → used_position_from_ballotArrangement
    → used_candidate_from_ballotArrangement) in known_ballotArrangement.post
  (used_ballot_from_ballotArrangement → used_position_from_ballotArrangement
    → used_candidate_from_ballotArrangement) not in known_ballotArrangement.pre

  //when you can learn it
  used_position_from_ballotArrangement → used_candidate_from_ballotArrangement
    = used_ballot_from_ballotArrangement.(known_ballotReceipt.pre)
      .(known_receiptOnion.pre).(known_onionArrangement.pre)
}
```

```
// AppendedFacts
//    all b: Ballot |
//      (b.ballotCandidate) = (b.ballotReceipt).receiptMarked.(b.ballotArrangement)
sig ballotArrangement_inference_5 extends ballotArrangement_Inference {}{
  //what you learn
  (used_ballot_from_ballotArrangement → used_position_from_ballotArrangement
    → used_candidate_from_ballotArrangement) in known_ballotArrangement.post
  (used_ballot_from_ballotArrangement → used_position_from_ballotArrangement
    → used_candidate_from_ballotArrangement) not in known_ballotArrangement.pre

  //when you can learn it
  used_ballot_from_ballotArrangement.(known_ballotCandidate.pre)
    = used_candidate_from_ballotArrangement
  used_ballot_from_ballotArrangement.(known_ballotReceipt.pre)
    .(known_receiptMarked.pre) = used_position_from_ballotArrangement
}




abstract sig receiptMarked_Inference extends Inference {
  used_receipt_from_receiptMarked: one Receipt,
  used_position_from_receiptMarked: one Position,
}

// VoterBreadcrumb
// all v: RegisteredVoter | let b = v.voterBallot |
//   b.ballotReceipt.receiptMarked.(b.ballotArrangement) = v.intention
sig receiptMarked_inference_1 extends receiptMarked_Inference {}{
  //what you learn
  (used_receipt_from_receiptMarked → used_position_from_receiptMarked)
    in known_receiptMarked.post
  (used_receipt_from_receiptMarked → used_position_from_receiptMarked)
    not in known_receiptMarked.pre

  //when you can learn it
  some v: known_RegisteredVoter.pre {
    v.(known_voterBallot.pre).(known_ballotReceipt.pre)
      = used_receipt_from_receiptMarked
    used_position_from_receiptMarked.(v.(known_voterBallot.pre)
      .(known_ballotArrangement.pre)) = v.(known_intention.pre)
    some v.(known_intention.pre)
  }
}
```

```
// BoardBreadcrumb
//    all input: mix.Record | let output = input.mix {
//       input.receiptMarked = output.recordMarked
//    }
sig receiptMarked_inference_4 extends receiptMarked_Inference {}{
  //what you learn
  (used_receipt_from_receiptMarked → used_position_from_receiptMarked)
    in known_receiptMarked.post
  (used_receipt_from_receiptMarked → used_position_from_receiptMarked)
    not in known_receiptMarked.pre

  //when you can learn it
  used_position_from_receiptMarked
    = used_receipt_from_receiptMarked.(known_mix.pre).(known_recordMarked.pre)
}

// AppendedFacts
//    all b: Ballot |
//       (b.ballotCandidate) = (b.ballotReceipt).receiptMarked.(b.ballotArrangement)
sig receiptMarked_inference_5 extends receiptMarked_Inference {}{
  //what you learn
  (used_receipt_from_receiptMarked → used_position_from_receiptMarked)
    in known_receiptMarked.post
  (used_receipt_from_receiptMarked → used_position_from_receiptMarked)
    not in known_receiptMarked.pre

  //when you can learn it
  some b: Ballot {
    b.(known_ballotCandidate.pre)
      = used_position_from_receiptMarked.(b.(known_ballotArrangement.pre))
    b.(known_ballotReceipt.pre) = used_receipt_from_receiptMarked
  }
}

// AppendedFacts
//    all r: Receipt |
//       (r.receiptCandidate) = (r.receiptMarked).((r.receiptOnion).onionArrangement)
sig receiptMarked_inference_6 extends receiptMarked_Inference {}{
  //what you learn
  (used_receipt_from_receiptMarked → used_position_from_receiptMarked)
    in known_receiptMarked.post
  (used_receipt_from_receiptMarked → used_position_from_receiptMarked)
    not in known_receiptMarked.pre

  //when you can learn it
  used_receipt_from_receiptMarked.(known_receiptCandidate.pre)
    = used_position_from_receiptMarked.(used_receipt_from_receiptMarked
       .(known_receiptOnion.pre).(known_onionArrangement.pre))
  some used_receipt_from_receiptMarked.(known_receiptCandidate.pre)
}
```

```
abstract sig receiptOnion_Inference extends Inference {
  used_receipt_from_receiptOnion: one Receipt,
  used_onion_from_receiptOnion: one Onion,
}

// AppendedFacts
//    all r: Receipt |
//    (r.receiptCandidate) = (r.receiptMarked).((r.receiptOnion).onionArrangement)
sig receiptOnion_inference_6 extends receiptOnion_Inference {}{
  //what you learn
  (used_receipt_from_receiptOnion → used_onion_from_receiptOnion)
    in known_receiptOnion.post
  (used_receipt_from_receiptOnion → used_onion_from_receiptOnion)
    not in known_receiptOnion.pre

  //when you can learn it
  used_receipt_from_receiptOnion.(known_receiptCandidate.pre)
    = used_receipt_from_receiptOnion.(known_receiptMarked.pre)
      .(used_onion_from_receiptOnion.(known_onionArrangement.pre))
  some used_receipt_from_receiptOnion.(known_receiptCandidate.pre)
}
```

```
abstract sig receiptCandidate_Inference extends Inference {
  used_receipt_from_receiptCandidate: one Receipt,
  used_candidate_from_receiptCandidate: one Candidate,
}

// AppendedFacts
//    all r: Receipt |
//       (r.receiptCandidate) = (r.receiptMarked).((r.receiptOnion).onionArrangement)
sig receiptCandidate_inference_6 extends receiptCandidate_Inference {}{
  //what you learn
  (used_receipt_from_receiptCandidate → used_candidate_from_receiptCandidate)
    in known_receiptCandidate.post
  (used_receipt_from_receiptCandidate → used_candidate_from_receiptCandidate)
    not in known_receiptCandidate.pre

  //when you can learn it
  used_candidate_from_receiptCandidate
    = used_receipt_from_receiptCandidate.(known_receiptMarked.pre)
      .(used_receipt_from_receiptCandidate.(known_receiptOnion.pre)
      .(known_onionArrangement.pre))
}


abstract sig onionArrangement_Inference extends Inference {
  used_onion_from_onionArrangement: one Onion,
  used_position_from_onionArrangement: one Position,
  used_candidate_from_onionArrangement: one Candidate,
}
```

```
// BallotBreadcrumb
//    all b: Ballot | b.ballotArrangement
//       = b.ballotReceipt.receiptOnion.onionArrangement
sig onionArrangement_inference_2 extends onionArrangement_Inference {}{
  //what you learn
  (used_onion_from_onionArrangement → used_position_from_onionArrangement
    → used_candidate_from_onionArrangement) in known_onionArrangement.post
  (used_onion_from_onionArrangement → used_position_from_onionArrangement
    → used_candidate_from_onionArrangement) not in known_onionArrangement.pre

  //when you can learn it
  some b: Ballot {
    b.(known_ballotArrangement.pre)
      = (used_position_from_onionArrangement → used_candidate_from_onionArrangement)
    b.(known_ballotReceipt.pre).(known_receiptOnion.pre)
      = used_onion_from_onionArrangement
  }
}

// BoardBreadcrumb
//    all input: mix.Record | let output = input.mix {
//       input.receiptOnion.onionArrangement = output.recordArrangement }
sig onionArrangement_inference_3 extends onionArrangement_Inference {}{
  //what you learn
  (used_onion_from_onionArrangement → used_position_from_onionArrangement
    → used_candidate_from_onionArrangement) in known_onionArrangement.post
  (used_onion_from_onionArrangement → used_position_from_onionArrangement
    → used_candidate_from_onionArrangement) not in known_onionArrangement.pre

  //when you can learn it
  some i: (known_mix.pre).Record {
    i.(known_receiptOnion.pre) = used_onion_from_onionArrangement
    (used_position_from_onionArrangement → used_candidate_from_onionArrangement)
      = i.(known_mix.pre).(known_recordArrangement.pre)
  }
}

// AppendedFacts
//    all r: Receipt |
//       (r.receiptCandidate) = (r.receiptMarked).((r.receiptOnion).onionArrangement)
sig onionArrangement_inference_6 extends onionArrangement_Inference {}{
  //what you learn
  (used_onion_from_onionArrangement → used_position_from_onionArrangement
    → used_candidate_from_onionArrangement) in known_onionArrangement.post
  (used_onion_from_onionArrangement → used_position_from_onionArrangement
    → used_candidate_from_onionArrangement) not in known_onionArrangement.pre

  //when you can learn it
  some r: Receipt {
    r.(known_receiptOnion.pre) = used_onion_from_onionArrangement
    r.(known_receiptCandidate.pre) = used_candidate_from_onionArrangement
    r.(known_receiptMarked.pre) = used_position_from_onionArrangement
  }
}
```

```
abstract sig recordArrangement_Inference extends Inference {
  used_record_from_recordArrangement: one Record,
  used_position_from_recordArrangement: one Position,
  used_candidate_from_recordArrangement: one Candidate,
}

// BoardBreadcrumb
//    all input: mix.Record | let output = input.mix {
//      input.receiptOnion.onionArrangement = output.recordArrangement }
sig recordArrangement_inference_3 extends recordArrangement_Inference {}{
  //what you learn
  (used_record_from_recordArrangement → used_position_from_recordArrangement
    → used_candidate_from_recordArrangement) in known_recordArrangement.post
  (used_record_from_recordArrangement → used_position_from_recordArrangement
    → used_candidate_from_recordArrangement) not in known_recordArrangement.pre

  //when you can learn it
  some i: (known_mix.pre).Record {
    i.(known_receiptOnion.pre).(known_onionArrangement.pre)
      = (used_position_from_recordArrangement → used_candidate_from_recordArrangement)
    i.(known_mix.pre) = used_record_from_recordArrangement
  }
}

// AppendedFacts
//    all r: Record |
//      (r.recordCandidate) = (r.recordMarked).(r.recordArrangement)
sig recordArrangement_inference_7 extends recordArrangement_Inference {}{
  //what you learn
  (used_record_from_recordArrangement → used_position_from_recordArrangement
    → used_candidate_from_recordArrangement) in known_recordArrangement.post
  (used_record_from_recordArrangement → used_position_from_recordArrangement
    → used_candidate_from_recordArrangement) not in known_recordArrangement.pre

  //when you can learn it
  used_record_from_recordArrangement.(known_recordCandidate.pre)
    = used_candidate_from_recordArrangement
  used_record_from_recordArrangement.(known_recordMarked.pre)
    = used_position_from_recordArrangement
}
```

```
abstract sig recordMarked_Inference extends Inference {
  used_record_from_recordMarked: one Record,
  used_position_from_recordMarked: one Position,
}

// BoardBreadcrumb
//    all input: mix.Record | let output = input.mix {
//       input.receiptMarked = output.recordMarked
// }
sig recordMarked_inference_4 extends recordMarked_Inference {}{
  //what you learn
  (used_record_from_recordMarked → used_position_from_recordMarked)
    in known_recordMarked.post
  (used_record_from_recordMarked → used_position_from_recordMarked)
    not in known_recordMarked.pre

  //when you can learn it
  some i: (known_mix.pre).Record {
    i.(known_receiptMarked.pre) = used_position_from_recordMarked
    i.(known_mix.pre) = used_record_from_recordMarked
  }
}

// AppendedFacts
//    all r: Record |
//       (r.recordCandidate) = (r.recordMarked).(r.recordArrangement)
sig recordMarked_inference_7 extends recordMarked_Inference {}{
  //what you learn
  (used_record_from_recordMarked → used_position_from_recordMarked)
    in known_recordMarked.post
  (used_record_from_recordMarked → used_position_from_recordMarked)
    not in known_recordMarked.pre

  //when you can learn it
  used_record_from_recordMarked.(known_recordCandidate.pre)
    = used_position_from_recordMarked.(used_record_from_recordMarked
      .(known_recordArrangement.pre))
  some used_record_from_recordMarked.(known_recordCandidate.pre)
}
```

```
abstract sig recordCandidate_Inference extends Inference {
  used_record_from_recordCandidate: one Record,
  used_candidate_from_recordCandidate: one Candidate,
}

// AppendedFacts
//    all r: Record |
//      (r.recordCandidate) = (r.recordMarked).(r.recordArrangement)
sig recordCandidate_inference_7 extends recordCandidate_Inference {}{
  //what you learn
  (used_record_from_recordCandidate → used_candidate_from_recordCandidate)
    in known_recordCandidate.post
  (used_record_from_recordCandidate → used_candidate_from_recordCandidate)
    not in known_recordCandidate.pre

  //when you can learn it
  used_candidate_from_recordCandidate
    = used_record_from_recordCandidate.(known_recordMarked.pre)
      .(used_record_from_recordCandidate.(known_recordArrangement.pre))
}
```

```
// There are no inference rules for these guys, so we have to explicitly ban
// them from existing. Otherwise Alloy will create trivial instantiations,
// in accordance with its language semantics, which will permit bad inferences.
// Extended abstract sigs cannot exist on their own right, but unextended ones can.
fact {no scramble_Inference + mix_Inference}
abstract sig scramble_Inference extends Inference {
  used_board: one Board,
  used_receipt: one Receipt,
  used_record: one Record,
}
abstract sig mix_Inference extends Inference {
  used_receipt: one Receipt,
  used_record: one Record,
}
```

```
/************************/
/* INFERENCE GUIDELINES */
/************************/

//knowledge is not forgotten
pred memory [] {
  all t: Time, t': t.next {
    known_intention.t in known_intention.t'
    known_voterBallot.t in known_voterBallot.t'
    known_RegisteredVoter.t in known_RegisteredVoter.t'
    known_score.t in known_score.t'
    known_ballotReceipt.t in known_ballotReceipt.t'
    known_ballotCandidate.t in known_ballotCandidate.t'
    known_ballotArrangement.t in known_ballotArrangement.t'
    known_receiptMarked.t in known_receiptMarked.t'
    known_receiptOnion.t in known_receiptOnion.t'
    known_receiptCandidate.t in known_receiptCandidate.t'
    known_onionArrangement.t in known_onionArrangement.t'
    known_recordArrangement.t in known_recordArrangement.t'
    known_recordMarked.t in known_recordMarked.t'
    known_recordCandidate.t in known_recordCandidate.t'
    known_scramble.t in known_scramble.t'
    known_mix.t in known_mix.t'
  }
}

//you learn (or forget) something new every day; knowledge can't remain static
pred progress [] {
  all t: Time, t': t.next {
    known_intention.t != known_intention.t'
    or known_voterBallot.t != known_voterBallot.t'
    or known_RegisteredVoter.t != known_RegisteredVoter.t'
    or known_score.t != known_score.t'
    or known_ballotReceipt.t != known_ballotReceipt.t'
    or known_ballotCandidate.t != known_ballotCandidate.t'
    or known_ballotArrangement.t != known_ballotArrangement.t'
    or known_receiptMarked.t != known_receiptMarked.t'
    or known_receiptOnion.t != known_receiptOnion.t'
    or known_receiptCandidate.t != known_receiptCandidate.t'
    or known_onionArrangement.t != known_onionArrangement.t'
    or known_recordArrangement.t != known_recordArrangement.t'
    or known_recordMarked.t != known_recordMarked.t'
    or known_recordCandidate.t != known_recordCandidate.t'
    or known_scramble.t != known_scramble.t'
    or known_mix.t != known_mix.t'
  }
}
```

```

//initial knowledge is correct, but possibly incomplete
pred seededKnowledge [] {
        known_intention.first in intention
        known_voterBallot.first in voterBallot
        known_RegisteredVoter.first in RegisteredVoter
        known_score.first in score
        known_ballotReceipt.first in ballotReceipt
        known_ballotCandidate.first in ballotCandidate
        known_ballotArrangement.first in ballotArrangement
        known_receiptMarked.first in receiptMarked
        known_receiptOnion.first in receiptOnion
        known_receiptCandidate.first in receiptCandidate
        known_onionArrangement.first in onionArrangement
        known_recordArrangement.first in recordArrangement
        known_recordMarked.first in recordMarked
        known_recordCandidate.first in recordCandidate
        known_scramble.first in scramble
        known_mix.first in mix

        RegisteredVoter = known_RegisteredVoter.first //public record
}

//take it slow; only do inference one at a time
pred sequentialInferences [] {
        all t: Time | lone t.comingAttractions
        all t: Time | lone t.pastAttractions
}

// You can only pause after all the work is done, and you can only pause if you do noth
// That is, once you start pausing you must do nothing but pause.
pred onlyPauseAtEnd [] {
        all t: Time |
                some pause & t.comingAttractions ⇒ no t.comingAttractions − pause
        all t: Time − last |
                some pause & t.pastAttractions ⇒ some pause & t.comingAttractions
}
```

```
//additions to current state must be explained
pred explainAdditions [] {
  all t: Time - first , v: Voter , c: Candidate |
    (v -> c) in known_intention.t - known_intention.(t.prev)
      => some inf: intention_Inference & t.pastAttractions |
        inf.used_voter_from_intention = v and inf.used_candidate_from_intention =c

  all t: Time - first , v: Voter , b: Ballot |
    (v -> b) in known_voterBallot.t - known_voterBallot.(t.prev)
      => some inf: voterBallot_Inference & t.pastAttractions |
        inf.used_voter_from_voterBallot = v and inf.used_ballot_from_voterBallot =b

  all t: Time - first , v: Voter |
    (v) in known_RegisteredVoter.t - known_RegisteredVoter.(t.prev)
      => some inf: RegisteredVoter_Inference & t.pastAttractions |
        inf.used_voter_from_RegisteredVoter = v

  all t: Time - first , c: Candidate , s: Int |
    (c -> s) in known_score.t - known_score.(t.prev)
      => some inf: score_Inference & t.pastAttractions |
        inf.used_candidate_from_score = c and inf.used_score_from_score = s

  all t: Time - first , b: Ballot , r: Receipt |
    (b -> r) in known_ballotReceipt.t - known_ballotReceipt.(t.prev)
      => some inf: ballotReceipt_Inference & t.pastAttractions |
        inf.used_ballot_from_ballotReceipt = b
        and inf.used_receipt_from_ballotReceipt = r

  all t: Time - first , b: Ballot , c: Candidate |
    (b -> c) in known_ballotCandidate.t - known_ballotCandidate.(t.prev)
      => some inf: ballotCandidate_Inference & t.pastAttractions |
        inf.used_ballot_from_ballotCandidate = b
        and inf.used_candidate_from_ballotCandidate = c

  all t: Time - first , b: Ballot , p: Position , c: Candidate |
    (b -> p -> c) in known_ballotArrangement.t - known_ballotArrangement.(t.prev)
      => some inf: ballotArrangement_Inference & t.pastAttractions |
        inf.used_ballot_from_ballotArrangement = b and
        inf.used_position_from_ballotArrangement = p and
        inf.used_candidate_from_ballotArrangement = c

  all t: Time - first , r: Receipt , p: Position |
    (r -> p) in known_receiptMarked.t - known_receiptMarked.(t.prev)
      => some inf: receiptMarked_Inference & t.pastAttractions |
        inf.used_receipt_from_receiptMarked = r
        and inf.used_position_from_receiptMarked = p

  all t: Time - first , r: Receipt , o: Onion |
    (r -> o) in known_receiptOnion.t - known_receiptOnion.(t.prev)
      => some inf: receiptOnion_Inference & t.pastAttractions |
        inf.used_receipt_from_receiptOnion = r
        and inf.used_onion_from_receiptOnion = o
```

```
  all t: Time - first, r: Receipt, c: Candidate |
    (r → c) in known_receiptCandidate.t - known_receiptCandidate.(t.prev)
      ⇒ some inf: receiptCandidate_Inference & t.pastAttractions |
        inf.used_receipt_from_receiptCandidate = r
        and inf.used_candidate_from_receiptCandidate = c

  all t: Time - first, o: Onion, p: Position, c: Candidate |
    (o → p → c) in known_onionArrangement.t - known_onionArrangement.(t.prev)
      ⇒ some inf: onionArrangement_Inference & t.pastAttractions |
        inf.used_onion_from_onionArrangement = o and
        inf.used_position_from_onionArrangement = p and
        inf.used_candidate_from_onionArrangement = c

  all t: Time - first, r: Record, p: Position, c: Candidate |
    (r → p → c) in known_recordArrangement.t - known_recordArrangement.(t.prev)
      ⇒ some inf: recordArrangement_Inference & t.pastAttractions |
        inf.used_record_from_recordArrangement = r and
        inf.used_position_from_recordArrangement = p and
        inf.used_candidate_from_recordArrangement = c

  all t: Time - first, r: Record, p: Position |
    (r → p) in known_recordMarked.t - known_recordMarked.(t.prev)
      ⇒ some inf: recordMarked_Inference & t.pastAttractions |
        inf.used_record_from_recordMarked = r
        and inf.used_position_from_recordMarked = p

  all t: Time - first, r: Record, c: Candidate |
    (r → c) in known_recordCandidate.t - known_recordCandidate.(t.prev)
      ⇒ some inf: recordCandidate_Inference & t.pastAttractions |
        inf.used_record_from_recordCandidate = r
        and inf.used_candidate_from_recordCandidate = c

  all t: Time - first, b: Board, rt: Receipt, rd: Record |
    (b → rt → rd) in known_scramble.t - known_scramble.(t.prev)
      ⇒ some inf: scramble_Inference & t.pastAttractions |
         inf.used_board = b and inf.used_receipt = rt and inf.used_record = rd

  all t: Time - first,rt: Receipt, rd: Record |
    (rt → rd) in known_mix.t - known_mix.(t.prev)
      ⇒ some inf: mix_Inference & t.pastAttractions |
         inf.used_receipt = rt and inf.used_record = rd
}
```

```
// A statement that at time t the adversary only knows things that are correct
//    (but may have holes in knowledge).
// This pred is not assumed/enforced, but rather it is explicitly checked in
//    the "error" anaysis paragraph.
pred correctKnowledge [t: Time] {
  no known_intention.last              - intention
  no known_voterBallot.last            - voterBallot
  no known_RegisteredVoter.first       - RegisteredVoter
  no known_score.first                 - score
  no known_ballotReceipt.first         - ballotReceipt
  no known_ballotCandidate.first       - ballotCandidate
  no known_ballotArrangement.first     - ballotArrangement
  no known_receiptMarked.first         - receiptMarked
  no known_receiptOnion.first          - receiptOnion
  no known_receiptCandidate.first      - receiptCandidate
  no known_onionArrangement.first      - onionArrangement
  no known_recordArrangement.first     - recordArrangement
  no known_recordMarked.first          - recordMarked
  no known_recordCandidate.first       - recordCandidate
  no known_scramble.first              - scramble
  no known_mix.first                   - mix
}
```

```
/************/
/* ANALYSES */
/************/

pred sim [] {
  assumptions
  memory
  progress
  sequentialInferences
  seededKnowledge
  onlyPauseAtEnd
  explainAdditions

  some Inference - pause
}
run sim for 1 but 1 Inference, 2 Time, 3 int expect 1

pred easy_attack [] {
  assumptions
  memory
  progress
  sequentialInferences
  seededKnowledge
  onlyPauseAtEnd
  explainAdditions

  some c: Candidate | c.score != 0
  no known_intention.first
  #known_intention.last >= 2
  #Position >= 2
}
run easy_attack for 3 but 4 Inference, 4 Time, 3 int expect 1
```

```
pred hard_attack [] {
//general rules
  assumptions //oddly, this can be removed and the attack still fails
  seededKnowledge
  explainAdditions

//restrictions on knowledge
  //you can't initially know any meta-information, only literal information
  no known_ballotCandidate.first
  no known_receiptCandidate.first
  no known_recordCandidate.first //not necessary to block attacks

  no known_intention.first //no telepathy
  no known_ballotArrangement.first //tear-off receipts hide this
  no known_onionArrangement.first //encryption
  no known_mix.first
  // If you remove this last one, then the attack succeeds!
  // If you leave it in, the unsat core feature of Alloy 4 highlights most
  //   constraints in this predicate, indicating their relevance to the result.
  // It should highlight all of them.

//malicious goal
  some v: Voter | some v.(known_voterBallot.last).(known_ballotCandidate.last)
}
run hard_attack for 3 but 3 Inference, 4 Time, 3 int expect 0
run hard_attack for 4 but 5 Inference, 6 Time, 4 int expect 0
// Ensure #inferences + 1 >= #Time (or else you get an overconstraint)

pred successful_hard_attack [] {
//general rules
  assumptions
  seededKnowledge
  explainAdditions
  memory
  progress
  sequentialInferences
  onlyPauseAtEnd

//restrictions on knowledge
  //you can't initially know any meta-information, only literal information
  no known_ballotCandidate.first
  no known_receiptCandidate.first
  no known_recordCandidate.first

  no known_intention.first //no telepathy
  no known_ballotArrangement.first //tear-off receipts prevent you know knowing this
  no known_onionArrangement.first //encryption
  --no known_mix.first //REMOVED to enable attack!

//malicious goal
  some v: Voter | some v.(known_voterBallot.last).(known_ballotCandidate.last)
}
run successful_hard_attack for 2 but 3 Inference, 4 Time, 3 int, 1 Record expect 1
run successful_hard_attack for 2 but 2 Inference, 3 Time, 3 int, 1 Record expect 0
```

```

//situation in which adversary infers an incorrect fact
pred error [] {
  assumptions
  seededKnowledge
  explainAdditions

  //Add in these guys to make counterexamples easier to understand,
  // but leave them out in the final check.
  --memory
  --progress
  --sequentialInferences
  --onlyPauseAtEnd

  --first.comingAttractions = ballotReceipt_inference_1
  some known_ballotReceipt.first - ballotReceipt

  not correctKnowledge[last]
}
run error for 2 but 2 Time expect 0
--run error for 3 but 3 Time expect 0
```

```
/**********************/
/* ASSUMPTIONS & GOALS */
/**********************/

pred AppendedFacts [] {
  all b: Ballot |
    (b.ballotCandidate) = (b.ballotReceipt).receiptMarked.(b.ballotArrangement)
  all r: Receipt |
    (r.receiptCandidate) = (r.receiptMarked).((r.receiptOnion).onionArrangement)
  all r: Record |
    (r.recordCandidate) = (r.recordMarked).(r.recordArrangement)
}

// Muliplicity markings from the fidelity argument are written explicitly here.
// They could have been left inlined (as they are in the fidelity argument)
//  without disrupting this model.
pred Multiplicities [] {
  all v: Voter | lone v.intention

  all c: Candidate | one c.score

  all b: Ballot, p: Position | one p.(b.ballotArrangement)
  all b: Ballot, c: Candidate | one (b.ballotArrangement).c
  all b: Ballot | one b.ballotReceipt
  all b: Ballot | lone b.ballotCandidate

  all o: Onion, p: Position | one p.(o.onionArrangement)
  all o: Onion, c: Candidate | one (o.onionArrangement).c

  all r: Receipt | one r.receiptOnion
  all r: Receipt | lone r.receiptMarked
  all r: Receipt | lone r.receiptCandidate

  all r: Record, p: Position | one p.(r.recordArrangement)
  all r: Record, c: Candidate | one (r.recordArrangement).c
  all r: Record | lone r.recordMarked
  all r: Record | lone r.recordCandidate

  all r: Receipt | lone r.(Board.scramble)
  all r: Record | lone (Board.scramble).r

  one Board
}
```

```
pred VoterBreadcrumb [] {
  // ballots that aren't given to voters don't get marked
  all b: Ballot - Voter.voterBallot | no b.ballotReceipt.receiptMarked
  // every registered voter gets exactly one ballot
  all v: RegisteredVoter | one v.voterBallot
  // no unregistered voter gets a ballot
  all v: Voter - RegisteredVoter | no v.voterBallot
  // each ballot is given to at most one voter
  all b: Ballot | lone voterBallot.b

  // Voters mark the ballots they are given according to their intention
  // and the ordering on the ballot.  Note that they don't pay any attention
  // to the onion or the ordering it represents (it's encrypted!).
  all v: RegisteredVoter | let b = v.voterBallot |
    b.ballotReceipt.receiptMarked.(b.ballotArrangement) = v.intention
}

pred BallotBreadcrumb [] {
  // different ballots have different receipts attached to them
  all disj b,b': Ballot | b.ballotReceipt != b'.ballotReceipt

  // A ballot's onion accurately encodes the arrangement of candidates shown
  // on the ballot.  That is the, the order of candidates on the ballot is the
  // same as the order of candidates encoded in the onion.
  all b: Ballot | b.ballotArrangement = b.ballotReceipt.receiptOnion.onionArrangement
}

pred BoardBreadcrumb[] {
  // every ballot receipt gets sent into the board exactly once
  // receipts are transformed but not destroyed:
  //  each receipt going into the board corresponds to 1 record coming out of the board
  all input: Ballot.ballotReceipt | one input.mix

  // receipts are transformed but not created:
  // each record coming out of the board corresponds to 1 receipt going into the board
  all output: Receipt.mix | one mix.output

  // all receipts from all ballots are put into the voting board for scrambling
  all b: Ballot | b.ballotReceipt in mix.Record

  // only receipts from ballots are put into the voting board for scrambling
  all r: mix.Record | r in Ballot.ballotReceipt

  // The scrambling process may change the onions, but it does not change the
  // candidate orderings they represent, and it must leave the position of the
  // marking on the receipt the same. That is, for every input receipt, there
  // is one output receipt that both has an onion with the same arrangement of
  // candidates and has the same position marked.
  all input: mix.Record | let output = input.mix {
    input.receiptOnion.onionArrangement = output.recordArrangement
    input.receiptMarked = output.recordMarked
  }
}
```

```

pred RecordBreadcrumb [] {
  all c: Candidate |
    c.score = #(Receipt.mix & recordCandidate.c)
}

pred assumptions [] {
  AppendedFacts
  Multiplicities
  VoterBreadcrumb
  BallotBreadcrumb
  BoardBreadcrumb
  RecordBreadcrumb
}

pred goal [] {
  all c: Candidate |
    c.score = #(RegisteredVoter & intention.c)
}

pred correct_system [] {
  assumptions
  goal
  some c: Candidate | c.score > 0
}
run correct_system for 1 expect 1

assert implication {
  assumptions ⇒ goal
}
check implication for 1 expect 0
check implication for 3 expect 0
check implication for 6 expect 0
```

Bibliography

[1] Ben Adida and Ronald L. Rivest. Scratch & vote: self-contained paper-based cryptographic voting. In *WPES '06: Proceedings of the 5th ACM workshop on Privacy in electronic society*, pages 29–40, New York, NY, USA, 2006. ACM.

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

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

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

[5] Javed A. Aslam, Raluca A. Popa, and Ronald L. Rivest. On estimating the size and confidence of a statistical audit. In *EVT'07: Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology*, page 8, Berkeley, CA, USA, 2007. USENIX Association.

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

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

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

[9] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. *Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software*, pages 85–108. Number 2566 in LNCS. Springer, 2002.

[10] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of ACM SIGPLAN 2003: Programming Language Design and*

Impelementation(PLDI'03), number 7-14, pages 196–207, San Diego, CA, USA, June 2003. ACM Press.

[11] P. D. Bruza and Th.P. van der Weide. The Semantics of Data Flow Diagrams. In N. Prakash, editor, *Proceedings of the International Conference on Management of Data*, Hyderabad, India, 1989.

[12] Jeremy W. Bryans, Maciej Koutny, Laurent Mazaré, and Peter Y. A. Ryan. Opacity generalised to transition systems. *Int. J. Inf. Secur.*, 7(6):421–435, 2008.

[13] NASA California Institute of Technology. Jet propulsion laboratory. website, 2008. http://www.jpl.nasa.gov/index.cfm.

[14] Caltech/MIT. VTP: Voting technology project. website, 2000-2008. http://vote.caltech.edu/drupal/.

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

[16] Richard I. Cook and Michael F. O'Connor. *Medication Safety: A Guide to Health Care Facilities,*, chapter Thinking about accidents and systems, pages 73–87. American Society of Health-System Pharmacists, Bethesda, MD, 2005.

[17] 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.

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

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

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

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

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

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

[24] 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.

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

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

[27] 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.

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

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

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

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

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

[33] Mats P. E. Heimdahl. Safety and software intensive systems: Challenges old and new. In *Future of Software Engineering (FOSE'07)*, pages 137–152, Washington, DC, USA, 2007. IEEE Computer Society.

[34] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge, MA, March 2006.

[35] Daniel Jackson. A case for dependable software, 2008.

[36] Daniel Jackson and Michael Jackson. *Rigorous Development of Complex Fault Tolerant Systems*, chapter Separating Concerns Requirements Analysis: An Example. Springer-Verlag. To appear.

[37] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proceedings of the 8th European Software Engineering Conference / Proceedings of the 9th ACM SIGSOFT Sypmosium on the Foundations of Software Engineering (ESEC/FSE'01)*, pages 62–73, Vienna, Austria, September 2001.

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

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

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

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

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

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

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

[45] Trevor A. Kletz. Human problems with computer control. *Plant/Operations Progress*, 1(4):209–211, October 1982.

[46] Patrick Lam, Viktor Kuncak, and Martin Rinard. Hob: A tool for verifying data structure consistency. In *In 14th International Conference on Compiler Construction (tool demo*, 2005.

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

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

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

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

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

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

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

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

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

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

[57] 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.

[58] 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.

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

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

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

[62] University of Texas at Austin. Software engineering program. website, 2007. http://www.utexas.edu/student/ admissions/ugdegrees.html.

[63] University of Waterloo. Software engineering program. website, 2007. http://www.softeng.uwaterloo.ca/.

[64] Nick Ourusoff. Personal communication, 2006.

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

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

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

[68] 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.

[69] Brian Randell and Peter Y. A. Ryan. Voting technologies and trust. *IEEE Security and Privacy*, 4(5):50–56, 2006.

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

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

[72] Robert C. Ricks, Mary Ellen Berger, Elizabeth C. Holloway, and Ronald E. Goans. *REACTS Radiation Accident Registry: Update of Accidents in the United States*. International Radiation Protection Association, 2000.

[73] Ronald L. Rivest and Warren D. Smith. Three voting protocols: Threeballot, vav, and twin. In *EVT'07: Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology*, pages 16–16, Berkeley, CA, USA, 2007. USENIX Association.

[74] Ronald L. Rivest and John P. Wack. On the notion of software independence in voting systems, 2006.

[75] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented modeling and design.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[76] 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.

[77] Altair O. Santin, Regivaldo G. Costa, and Carlos A. Maziero. A three-ballot-based secure electronic voting system. *IEEE Security and Privacy*, 6(3):14–21, 2008.

[78] 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.

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

[80] 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.

[81] Robert Seater, Daniel Jackson, and Rohit Gheyi. Requirement progression in problem frames: Deriving specifications from requirements. *Requirements Engineering Journal (REJ'07)*, 2007.

[82] Michael Shnayerson. Hack the vote. *Vanity Fair*, page 158, April 2004.

[83] 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.

[84] Mana Taghdiri. Inferring specifications to detect errors in code. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 144–153, Washington, DC, USA, 2004. IEEE Computer Society.

[85] Mana Taghdiri, Robert Seater, and Daniel Jackson. Lightweight extraction of syntactic specifications. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 276–286, New York, NY, USA, 2006. ACM.

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

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

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