Requirement Progression in Problem Frames Applied to a Proton Therapy System

Robert Seater, Daniel Jackson
Massachusetts Institute of Technology
{rseater,dnj}@mit.edu

Abstract

A technique is presented for obtaining a specification from a requirement through a series of incremental steps. The starting point is a Problem Frame description involving a requirement on the phenomena of the problem domain, and a decomposition of the environment into domains, connected to one another and to the machine being implemented by shared phenomena. In each step, the requirement is moved towards the machine, leaving behind a trail of 'breadcrumbs' in the form of domain assumptions. Eventually, the transformed requirement references only phenomena at the interface of the machine and can therefore serve as a specification. Each step is justified by an implication that can be mechanically checked, ensuring that, if the machine obeys the derived specification and the domain assumptions are valid, the requirement will hold. The technique is applied to the logging subproblem of a radiotherapy system.

1 Introduction

Many system failures have resulted from implicit (but incorrect) assumptions about the system's environment which, when made explicit, are easily recognized and corrected [2, 6, 20]. As software is increasingly deployed in contexts in which it controls multiple, complex physical devices, this issue is likely to grow in importance.

The Problem Frames approach offers a framework for describing the interactions amongst software and other system components [14, 15]. 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 [8, 13, 18]. 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 decomposed 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).

Our work aims to create a strategy by which such a decomposition can be performed systematically by a series of incremental transformations. Such a process is alluded to in the Problem Frames book and is termed 'problem progression' or 'requirement progression'.

1.1 Context

Our research group has been involved in an ongoing collaboration with the Burr Proton Therapy Center (BPTC), a radiation therapy facility associated with the Massachusetts General Hospital in Boston, investigating improved methods for ensuring software dependability.

We are currently investigating the use of Problem Frames for constructing dependability cases for the BPTC control software. This work grew out of the difficulty we encountered at keeping track of a large number of domain properties and relating them appropriately to the requirements. We illustrate our approach with a simplified view of the BPTC system which, although redolent of some of the characteristics of the actual system, is perhaps not sufficiently complex to properly demonstrate the need for the approach. It does, however, illustrate the key elements of our approach in the context in which we are working, and the problem addressed – logging of dose delivered – is a real and important one

The various constraints are formalized in the Alloy modeling language, and the Alloy Analyzer [7, 12] is used to check that the resulting specification and domain assumptions do indeed establish the desired system-level logging property. The decomposition and analysis offer insight into both the particular subject system and the transformation strategy in general.

2 Problem Frames

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 analyze 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 follows:

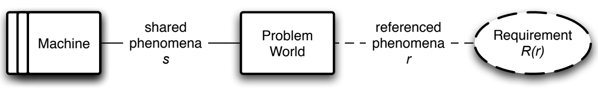


Figure 1. A generic Problem Frames description.

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 *shared phenomena* (left arc).

For example, in a traffic light system, the problem world might consist of the physical apparatus (lights and sensors) and the cars and drivers; the phenomena it shares with the machine might be the control and monitoring signals; the requirement might be that cars do not collide, referencing quite different phenomena (namely, the movement of cars); and the specification would be the protocol by which the machine generates control signals in response to the monitoring signals it receives.

In general, 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 signals), and another for the physical apparatus (whose assumptions describe the reaction of the lights to control signals received, and the relationship between car behaviour and monitoring signals generated).

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. In general, given a requirement R over phenomena r, a specification S over phenomena s, and domain assumptions D_i over both phenomena r and s, this implication takes the form

$$S(s) \wedge D_i(s, r) \Rightarrow R(r)$$

A problem frame is an archetypal pattern that gives a particular structure to 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 [15]. The traffic light system, for example, matches the Required Behavior Frame whose frame concern is shown in Figure 2.

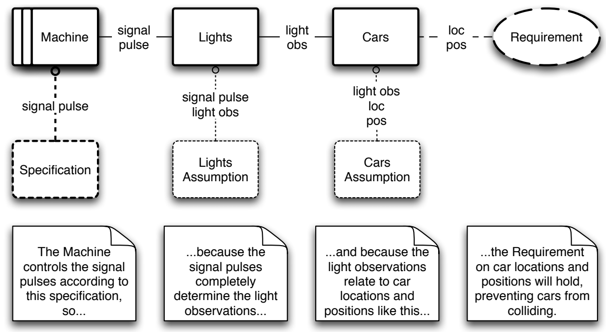


Figure 2. The frame concern for the two-way traffic light problem.

In practice, the frame concern only gives an outline of the argument and is too general to give much detail about the structure of the particular problem at hand. There may be many interconnected domains, with complex properties whose relevance is unclear. A systematic approach is needed for determining which properties of the domains are relevant, and for obtaining a specification for the machine that will, in concert with these properties, establish the requirement.

2.1 Problem Frame Transformations

We introduce a process for systematically performing requirement progression – that is, incrementally transforming a requirement into a specification. A byproduct of the transformations is a trail of domain assumptions, called *breadcrumbs*, that justify the progression.

Requirements, specifications, and breadcrumbs are three instances of *domain constraints*. Requirements connect only to non-machine domains, specifications connect only to the machine domain, and each breadcrumb connects only to a single non-machine domain.

The only thing barring the requirement from serving as a specification is that it references the wrong set of phenomena. Of course, altering it to reference the right set of phenomena (those at the interface of the machine domain) is no easy matter. The transformation process we describe is an incremental method for achieving such an alteration.

2.2 Transformation Process

During the course of the transformation, the specification-to-be may end up simultaneously connecting to both the machine domain and one or more non-machine domains. It is thus neither a requirement nor a specification and will be called the *goal* throughout the process. At any point in the process is exactly one goal constraint; the goal begins as the requirement and will eventually become a specification.

There are three types of steps in the transformation process: *adding* a breadcrumb permits the goal to be *rephrased*, which in turn enables a *push* to change which domains it connects to. Figure 3 shows an archetype of how these steps take a goal from being a requirement to being a specification.

(a) *Add* a breadcrumb constraint, representing an assumption about a domain in the problem world. The breadcrumb can only mention phenomena from a single domain connected to the goal (e.g. `p1` and `p2`). It is chosen so as to enable a useful rephrasing (step b). The breadcrumb is validated by a domain expert.

(b) *Rephrase* the goal so that it references different phenomena than before (e.g. `p1` instead of `p2`), although it still only references phenomena from domains to which it connects. The rephrasing is chosen so as to enable a useful push (step c). The analyst should verify that the breadcrumb is sufficiently strong to permit the rephrasing by establishing the following property:

$$(\text{breadcrumb} \wedge \text{rephrased goal}) \Rightarrow \text{prior goal}$$

(c) *Push* the requirement so that it connects to some domain d' (e.g. the machine) instead of some domain d (e.g. the intervening domain). A push from d to d' is only enabled if d' contains all phenomena from d that are referenced by the goal. The constraint itself is unchanged.

The analyst continues to perform transformations until the goal connects only to the machine domain. At that point, it only references phenomena at the interface of the machine and is a valid specification.

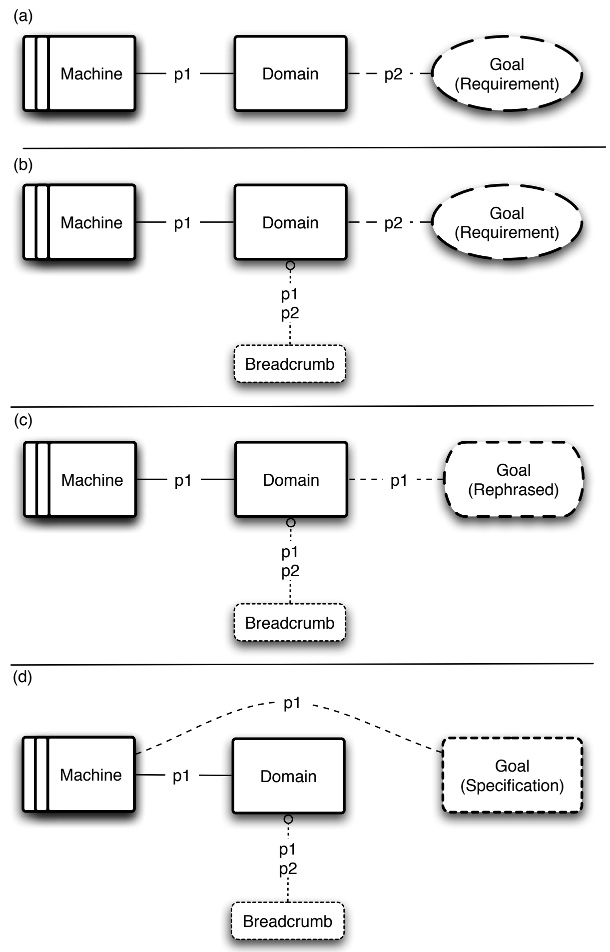
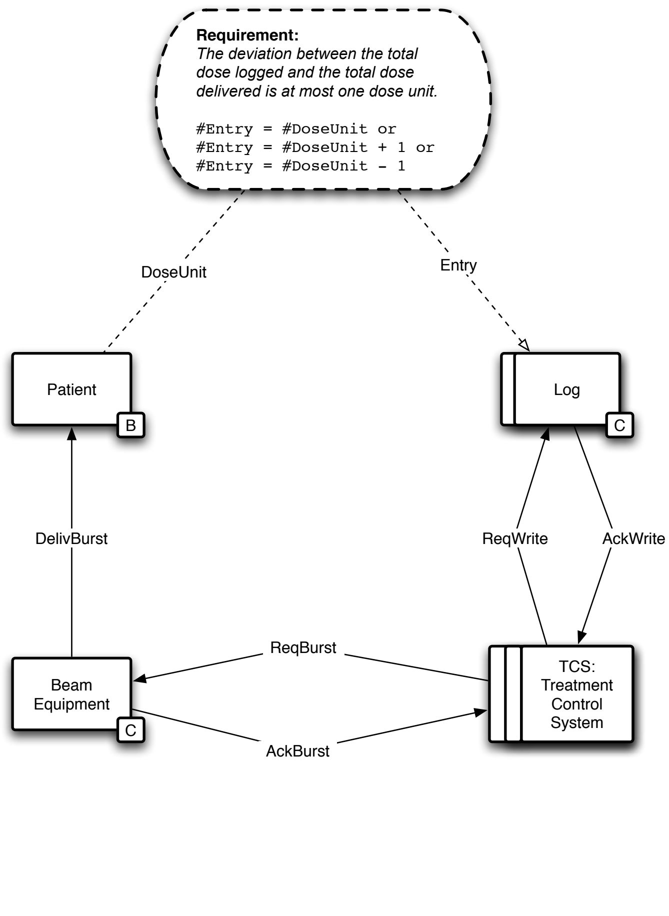
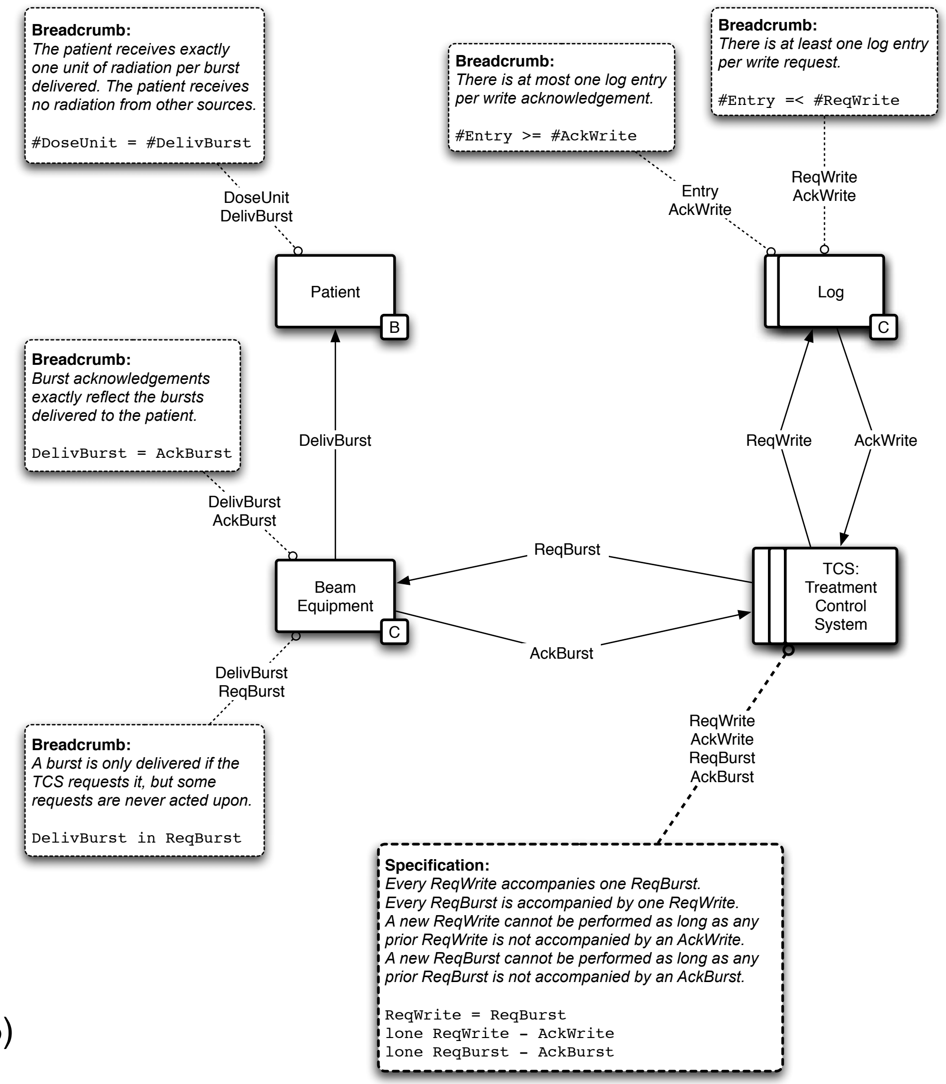


Figure 3. An archetypal transformation: (a) Prior to the transformation (b) A *breadcrumb* constraint is added, representing an assumption about how the domain relates phenomena `p1` and `p2`. (c) That breadcrumb permits the requirement to be *rephrased* to reference `p1` instead of `p2`. (d) The rephrasing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



(4)



(5)

Figure 4. The logging requirement expressed in Problem Frames notation: 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.

Figure 5. The breadcrumb assumptions that result from transforming the requirement into a specification. The breadcrumbs plus the specification enforce the requirement. The Alloy keyword *lone*, used in the TCS specification, indicates that a set has a cardinality of zero or one.

3 Proton Therapy Logging

3.1 System Requirements

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

> **Overdose:** 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 concern 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 [11]. We shall consider only the logging subproblem here.

3.2 Logging Subproblem

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 that the TCS, however, *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 making stronger assumptions about the reliability of other components, or weakening the requirement.

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, 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.

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 results in a distribution of small but subtle assumptions across the various components of the system.

3.3 The Phenomena

Figure 4 shows a Problem Frame description for the logging sub-problem. [1]

A `Patient` is positioned under the `Beam Equipment` in preparation for treatment. The `Treatment Control System` (TCS) reads the patient's treatment plan and issues an appropriate number of `ReqBurst` requests to the `Beam Equipment`. 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 partially unreliable. The `Beam Equipment` will never perform a `DelivBurst` without first receiving a `ReqBurst`, but it may ignore some `ReqBursts`. Similarly, the `Log` will never write erroneous `Entries`, but it may ignore some `ReqWrite` requests (if, for example, the log has reached its capacity or the disk crashes).

3.4 The Requirement

All constraints, including the initial requirement, are expressed in the Alloy modeling language, a first-order relational logic [12, 10]. The Alloy Analyzer can check the validity of a transformation with a bounded, exhaustive analysis [7]. Our transformation technique is not tied to Alloy; we chose it here because it was familiar to us, because it provides automatic analysis, and because it allows a fairly natural expression of the requirement and assumptions. An analysis that considered timing issues explicitly would probably be better expressed in a different notation.

For completeness, we shall include, in addition to the constraints, the Alloy declarations needed to complete the model. Alloy keywords are written in boldface. From the user's perspective, there are two fundamental sets – a set of dose units and a set of log entries.

[1] We use a slightly non-standard notation in our Problem Frame diagrams for the 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.

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

The 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 Goal1 ( ) {
  #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 had 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 messages. 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 connection between `Entries` and `DoseUnits`, as per the requirement. We will introduce domain assumptions about the `Patient` and `Beam Equipment` to relate `DoseUnits` 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.

3.5 Transformation and Derivation

Initially, the goal constraint is the requirement we want to enforce. The derivation happens in three stages: First, we push the goal from the `Log` to the `TCS`, and add a breadcrumb and rephrase the goal as needed to permit that push. Second, we push the goal from the `Patient` to the `Equipment`, adding another breadcrumb and performing another rephrasing. Finally, we push the goal from the `Equipment` to the `TCS`, adding a third breadcrumb and performing a third rephrasing. At that point, the goal only connects to (only references phenomena from) the machine domain, and has thus been transformed into a specification. Figure 5 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 goal from the `Log` domain onto the `TCS` domain. We cannot yet do so because the goal references the `Entry` phenomenon, which is not part of the `TCS`. We will thus need to *rephrase* the goal 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 goal constraint currently references 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 goal as follows:

```
pred Goal2 ( ) {
  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.

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

```
assert StepOne ( ) {
  LogBreadcrumb( ) and Goal2( )
   => Goal1( ) }
check StepOne for 100
```

Now that the goal only references 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 goal 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 `DoseUnits` of radiation from other sources. The breadcrumb permits the goal to be rephrased as follows:

```
pred Goal3 ( ) {
   lone ReqWrite - AckWrite and
   (#ReqWrite = #DelivBurst or
   #ReqWrite = #DelivBurst + 1) }
```

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

```
assert StepTwo ( ) {
   PatientBreadcrumb( ) and Goal3( )
    => Goal2( ) }
check StepTwo for 100
```

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

Step 3: from Equipment to TCS

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

```
pred EquipBreadcrumb( ) {
   DelivBurst = AckBurst
   DelivBurst in ReqBurst }
```

which says that an acknowledgement must be sent exactly whenever 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. The goal can now be rephrased like this:

```
pred Goal4 ( ) {
   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 breadcrumb:

```
assert StepThree ( ) {
   EquipBreadcrumb( ) and Goal4( )
    => Goal3( ) }
check StepThree for 100
```

Finally, we push the goal from `Beam Equipment` to `TCS`. At this point, the goal references only phenomena from `TCS` and has become a specification.

4 Discussion

One of the primary benefits of the Problem Frames approach is to force analysts to be explicit about domain assumptions. The strategy we have presented gives some structure to the process by which such assumptions are obtained and used, in the process of deriving a specification strong enough to enforce the desired requirement.

4.1 Significance

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 [27]. The most famous of these accidents are those involving the Therac-25 machine [20, 21], 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 [5].

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.

4.2 Role of the Analyst

The transformation process is systematic but not automatic. The decisions of what breadcrumbs to add, how to rephrase the goal, and which enabled pushes to enact are qualitative assessments by the analyst based on experience or a catalog of patterns and heuristics.

The approach is incremental, and justified by assertions that involve, in any step, at most the assumptions about a single domain.

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.

4.3 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 that necessary, forcing the rephrased goal to be stronger that 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 goal 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.

5 Related Work

5.1 Requirement Decomposition

Many approaches to system analysis involve some kind of decomposition of end-to-end requirements into subconstraints, often recursively. Assurance and safety cases [1, 20], 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. Analyses that focus on failures rather than requirements (such as HAZOP [25]) are duals of these approaches, in which decomposition is used to identify the root causes of failures.

More similar to our approach are frameworks, such as `i*` [30] and KAOS [3], 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 [4]. 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.

The four-variable model [26, 29] 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.

Letier and Lamsweerde show how a goal (requirement) produced from requirement elicitation can be transformed into a specification which is formal and precise enough to guide implementation [19]. 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.

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 [17]. 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.

5.2 Problem Frames

Michael Jackson sketches out a notion of *problem progression* in the Problem Frames book [15]. 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) [16]. 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.

Rapanotti, Hall, and Li recently introduced *problem reduction*, a technique that uses causal logic to formalize problem progression in Problem Frames [24]. 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 [15], in which the requirement is moved closer to the machine by eliminating intervening domains (as opposed to our approach in which we instead add domain assumptions).

Hall, Rapanotti, and Li are developing a calculus of requirements engineering based on the Problem Frames approach [9, 23, 22]. They examine how problems and solutions can be restructured to fit known patterns. Part of their technique involves a form of requirement progression, in which a requirement is replaced by an equivalent requirement in an alternate form. In contrast, our form of requirement progression is not semantics-preserving and the transformations are justified by a set of explicit assumptions rather than proofs of equivalence.

Acknowledgments

This work is part of an ongoing collaboration between the Software Design Group at MIT and the Burr Proton Therapy Center (BPTC) of the Massachusetts General Hospital. We appreciate the assistance of Jay Flanz and Doug Miller (BPTC), the advice of Michael Jackson and Jon Hall (Open University), and the feedback of the anonymous reviewers. This research was supported by grants *0086154* ('Design Conformant Software') and *6895566* ('Safety Mechanisms for Medical Software') from the ITR program of the National Science Foundation.

References

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

[2] T. E. Bell and T. A. Thayer. Software requirements: are they really a problem? In *Proceedings of the 2nd International Conference on Software Engineering*, pages 61–68. IEEE Society Press, 1967.

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

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

[5] Food and Drug Administration. FDA Statement on Radiation Overexposures in Panama. http://www.fda.gov/cdrh/ocd/panamaradexp.html.

[6] 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.

[7] MIT Software Design Group. The Alloy Analyzer. http://alloy.lcs.mit.edu.

[8] C. B. Haley, R. Laney, and B. 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)*, pages 203–217. Essener Informatik Beitrage, 2004. Editors: B. Regnell, E. Kamsties, and V. Gervasi.

[9] John G. Hall and Lucia Rapanotti. A framework for software problem analysis, 2005. Tecnical Report.

[10] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge, MA, April 2006.

[11] Daniel Jackson and Michael Jackson. *Rigorous Development of Complex Fault Tolerant Systems*, chapter Separating Concerns Requirements Analysis: An Example. Springer-Verlag. To appear.

[12] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *ACM SIGSOFT Conference on Foundations of Software Engineering / European Software Engineering Conference*, Vienna, September 2001.

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

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

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

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

[17] W. Lewis Johnson. Deriving specifications from requirements. In *Proceedings of the Tenth International Conference on Software Engineering*, pages 428–438. IEEE Computer Society, 1988.

[18] Robin Laney, Leonor Barroca, Michael Jackson, and Bashar Nuseibeh. Composing requirements using Problem Frames. In *Proceedings of the 2004 International Conference on Requirements Engineering (RE04)*. IEEE Computer Science Press.

[19] E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. In *ACM SIGSOFT Software Engineering Notes, 27(6)*, 2002.

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

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

[22] Zhi Li, Jon G. Hall, and Lucia Rapanotti. A constructive approach to Problem Frame semantics. Technical Report 2004/26, compdep, 2005.

[23] Zhi Li, Jon G. Hall, and Lucia Rapanotti. From requirements to specifications: a formal approach. In *Proceedings of the Second International Workshop on Applications and Advances in Problem Frames (IWAAPF'06)*, page 65, Shanghai, May 2006.

[24] Zhi Li Lucia Rapanotti, Jon G. Hall. Problem reduction: a systematic technique for deriving specifications from requirements, Feb 2006. Technical Report, ISSN 1744-1986.

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

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

[27] 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.

[28] Robert Seater and Daniel Jackson. Problem Frame transformations: Deriving specifications from requirements. In *Proceedings of the Second International Workshop on Applications and Advances in Problem Frames (IWAAPF'06)*, page 71, Shanghai, May 2006.

[29] Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.

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