Requirement Progression in Problem Frames: Deriving Specifications from Requirements

Robert Seater, Daniel Jackson
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
Software Design Group
32 Vassar Street, 32-G707
Cambridge, Massachusetts, 02148
{rseater, dnj}@mit.edu

Rohit Gheyi
Massachusetts Institute of Technology, and
Universidade Federal de Pernambuco
Cidade Universitaria, Recife PE
CEP: 50670-901
gheyi@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 decomposition of the environment into interconnected domains and a formal requirement on phenomena of those domains. In each step, the requirement is moved towards the machine, leaving behind a trail of 'breadcrumbs' – partial domain descriptions representing assumptions about the behaviors of those domains. 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 a mechanically checkable implication, ensuring that, if the machine obeys the derived specification and the domain assumptions are valid, the requirement will hold. The technique is formalized in Alloy and demonstrated on two examples.

1 Introduction

Many system failures stem 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. Building an argument is not enough; one must also explicitly expose all the assumptions the argument depends on. This not only gives us greater confidence that the argument is valid now, but also helps us know if the argument will still apply if the system is changed. If the changes to the system do not violate the assumptions used in the argument, then the conclusion of that argument still holds.

1.1 Our Approach

The problem frames approach offers a framework for describing the interactions amongst software and other system components [12, 14]. 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, 17, 18]. 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).

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 embelished with the argument.

In the problem frames book [14], 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.

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 [17], 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. [30]), 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 [14]. 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 [9, 7, 11] 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.

1.2 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. The work described in this paper 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 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 this paper 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.

1.3 Paper Organization

The rest of the paper is organized as follows:

- Section 2 introduces problem frames.
- Section 3 outlines of our technique for requirement progression.
- Section 4 demonstrates the technique on a simple traffic light example.
- Section 5 describes an Alloy model of problem diagrams.
- Section 6 extends that model to describe our technique for requirement progression.
- Section 7 demonstrates the technique on a second, more elaborate, logging example.
- Section 8 reflects on the strengths and weakness of our technique.
- Section 10 discusses future work.
- Section 9 discusses related work.
- The Appendix gives the full text of the Alloy model described in Section 5 and 6.

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 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 1[1].

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)

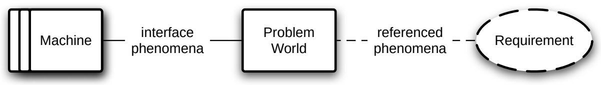


Figure 1. A generic problem frames description showing the disconnect between the phenomena controlled by the machine (the interface phenomena) and those constrained by the requirement (the referenced phenomena).

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 involves in a particular situation. One possible problem diagram for the traffic light system is shown in Figure 2.

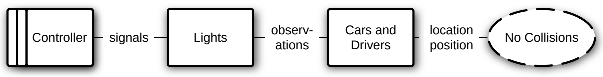


Figure 2. A problem diagram describing the domains and phenomena for a two-way traffic light. The arc connecting two domains is labeled by the phenomena shared by those domains – those phenomena that both domains involve. The arc connecting the requirement to a domain is labeled by the phenomen referenced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

[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 all together, as it is not currently relevant to our approach.

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.

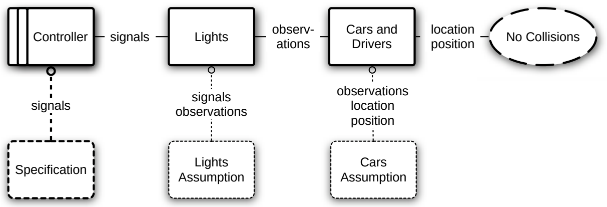


Figure 3. Assumptions about the intervening domains are expressed as partial domain descriptions in the form of constraints on their behaviors. These assumptions help us relate the machine specification to the system requirement. As with a requirement, the arc connecting an assumption or specification 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 4 [14].

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 5.

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 papers describes such an approach.

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.

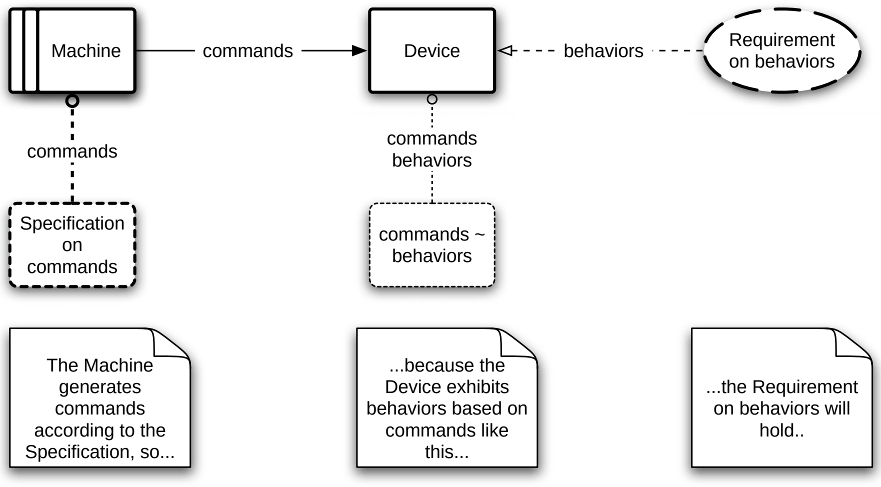
3.1 Available Transformations

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 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`).[2] 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.

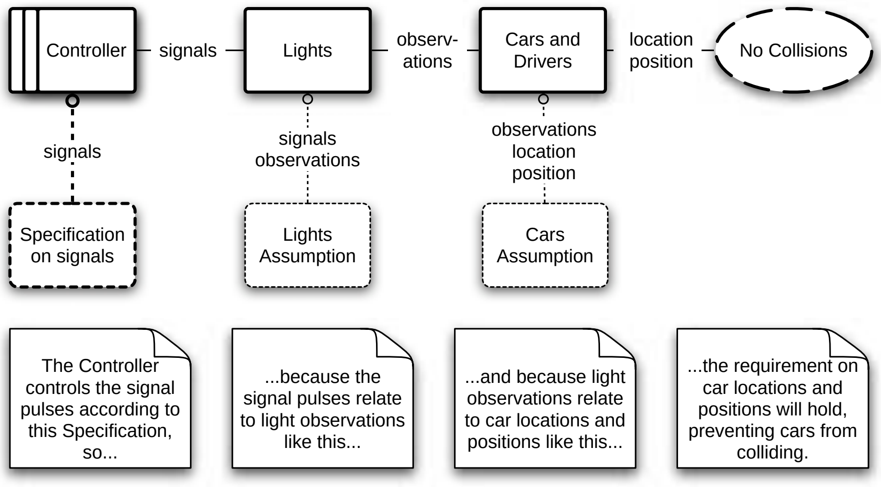
(b) *Rephrase* the requirement so that it represents a different constraint. The new version of the requirement must touch the same domains, but it may mention (and thereby constraint) a different subset of the phenomena of those domains (e.g. mention `p1` instead of `p2`). The rephrasing is chosen so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

[2]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.



Machine
commands
Device
behaviors
Requirement on behaviors
commands
commands behaviors
Specification on commands
commands ~ behaviors
The Machine generates commands according to the Specification, so...
...because the Device exhibits behaviors based on commands like this...
...the Requirement on behaviors will hold..

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



Controller
signals
Lights
observ-ations
Cars and Drivers
location position
No Collisions
signals
signals observations
observations location position
Specification on signals
Lights Assumption
Cars Assumption
The Controller controls the signal pulses according to this Specification, so...
...because the signal pulses relate to light observations like this...
...and because light observations relate to car locations and positions like this...
...the requirement on car locations and positions will hold, preventing cars from colliding.

Figure 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.

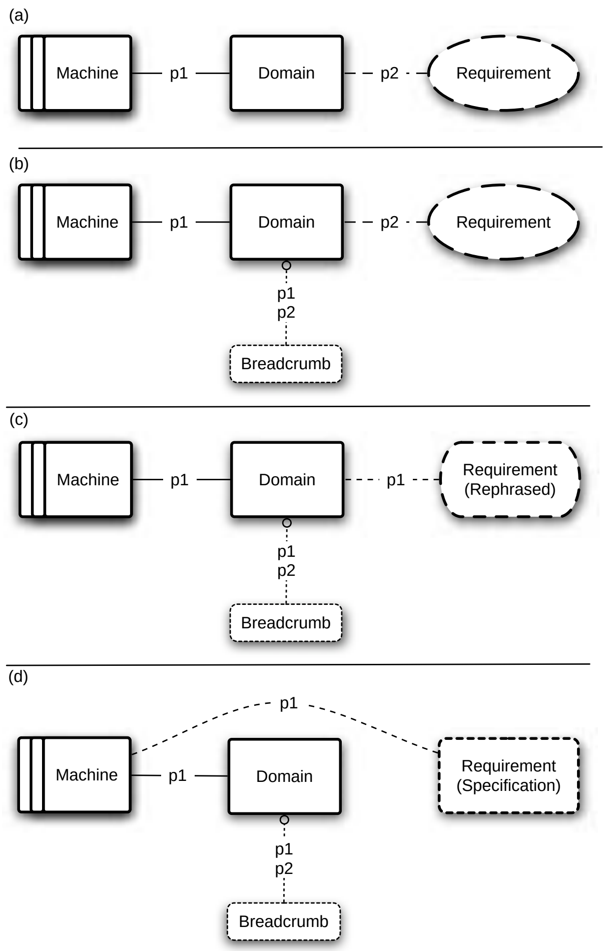


Figure 6. An archetypal requirement progression: (a) Prior to the transformation (b) A *breadcrumb* constraint is added, representing an assumption about how the domain relates phenomena p1 and p2. (c) That breadcrumb permits the requirement to be *rephrased* to reference p1 instead of p2. (d) The rephrasing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

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

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

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 perserve 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'. [3]

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).

[3]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 diagam well formed. Note that the problem frames notation, as given in the problem frames book [14], is ambiguous about this issue.

3.2 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 sourcess 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.

Our experience with the BPTC has been with the fourth case, and that is how we will present the examples in this paper. We have considered building formal models (the third case) from informal descriptions (the fourth case) as preprocessing for requirement progression. We discuss this idea further in Section 9 as future work.

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 [14]. It 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 7, we will see how requirement progression works on a *branching topology*. Requirement progression on larger problem frames involves a combination of both kinds of approaches.

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 [14]. 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 7, along with the requirement we will focus on in this example.

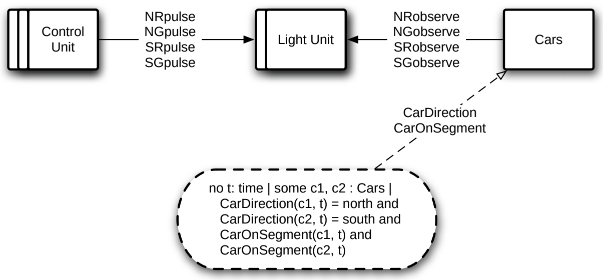


Figure 7. A more detailed problem diagram for the two-way traffic light problem. The constraint has been formalized and expressed using the Alloy language, a relational first-order logic.

The Light Unit has four physical lights: a red light and a green lights 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 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.

4.1 Basic Declarations

For completeness, we shall include, in addition to the constraints, the Alloy [9, 7, 11] 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 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}
```

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 7.

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 8 and narrated below.

(A) Add a Breadcrumb

The frame, shown in Figure 5, 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. [4] The result of this addition is shown in Figure 8a.

(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.

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

The result of this rephrasing is shown in Figure 8b.

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 [11, 7]. 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[5].

[4]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.

[5]Each execution of the Alloy model was solved instantaneously on a 133MHz G4 PowerMac with 800Mb of RAM, using the freely available version of Alloy 4 [7]

(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 8c.

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 9 and narrated below.

(A) Add a Breadcrumb

Once again, we appeal to the frame (Figure 5) 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 9a.

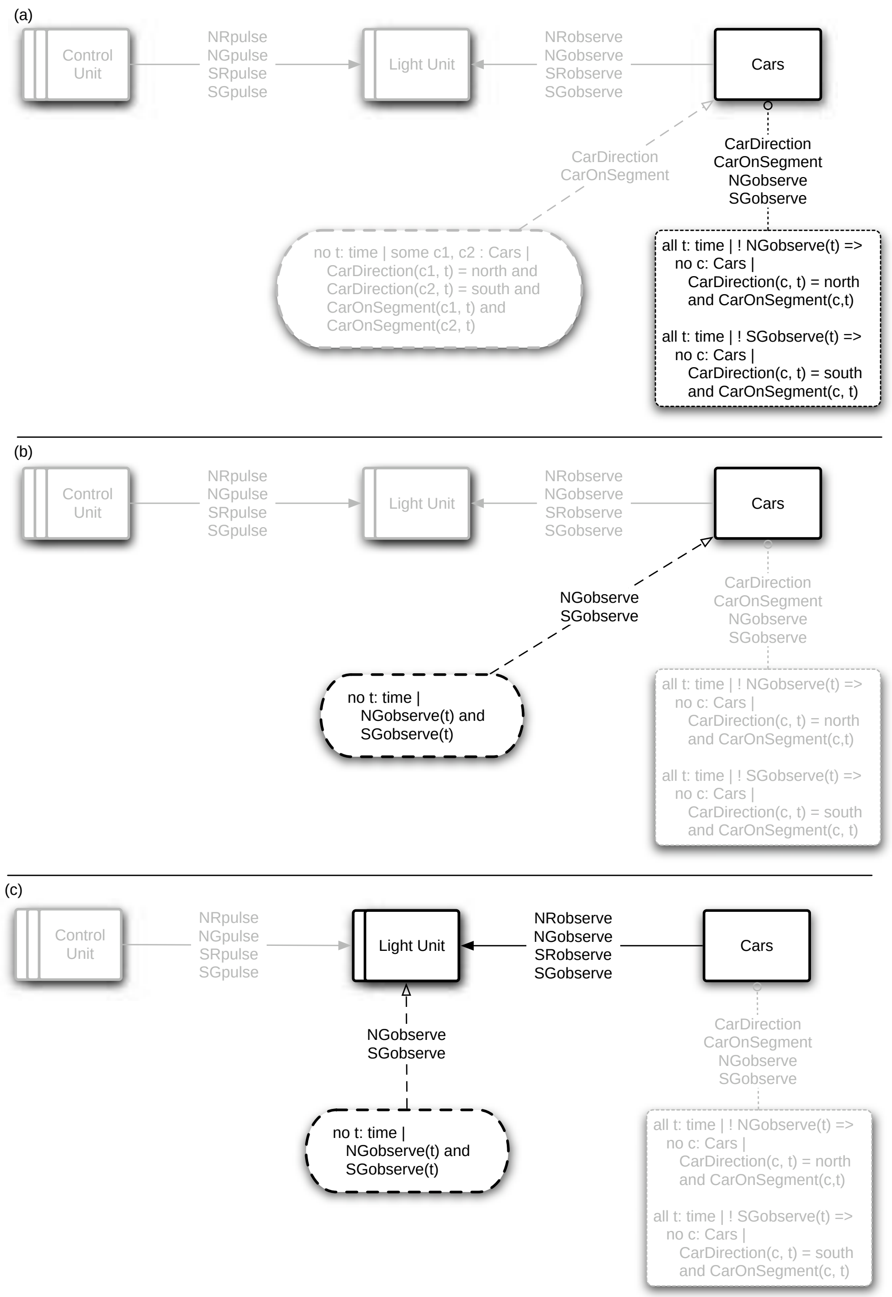

(a)
Control Unit
NRpulse
NGpulse
SRpulse
SGpulse
Light Unit
NRobserve
NGobserve
SRobserve
SGobserve
Cars
CarDirection
CarOnSegment
CarDirection
CarOnSegment
NGobserve
SGobserve
no t: time | some c1, c2 : Cars | CarDirection(c1, t) = north and CarDirection(c2, t) = south and CarOnSegment(c1, t) and CarOnSegment(c2, t)
all t: time | ! NGobserve(t) => no c: Cars | CarDirection(c, t) = north and CarOnSegment(c,t)
all t: time | ! SGobserve(t) => no c: Cars | CarDirection(c, t) = south and CarOnSegment(c, t)
(b)
Control Unit
NRpulse
NGpulse
SRpulse
SGpulse
Light Unit
NRobserve
NGobserve
SRobserve
SGobserve
Cars
NGobserve
SGobserve
CarDirection
CarOnSegment
NGobserve
SGobserve
no t: time | NGobserve(t) and SGobserve(t)
all t: time | ! NGobserve(t) => no c: Cars | CarDirection(c, t) = north and CarOnSegment(c,t)
all t: time | ! SGobserve(t) => no c: Cars | CarDirection(c, t) = south and CarOnSegment(c, t)
(c)
Control Unit
NRpulse
NGpulse
SRpulse
SGpulse
Light Unit
NRobserve
NGobserve
SRobserve
SGobserve
Cars
NGobserve
SGobserve
CarDirection
CarOnSegment
NGobserve
SGobserve
no t: time | NGobserve(t) and SGobserve(t)
all t: time | ! NGobserve(t) => no c: Cars | CarDirection(c, t) = north and CarOnSegment(c,t)
all t: time | ! SGobserve(t) => no c: Cars | CarDirection(c, t) = south and CarOnSegment(c, t)

Figure 8. 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.

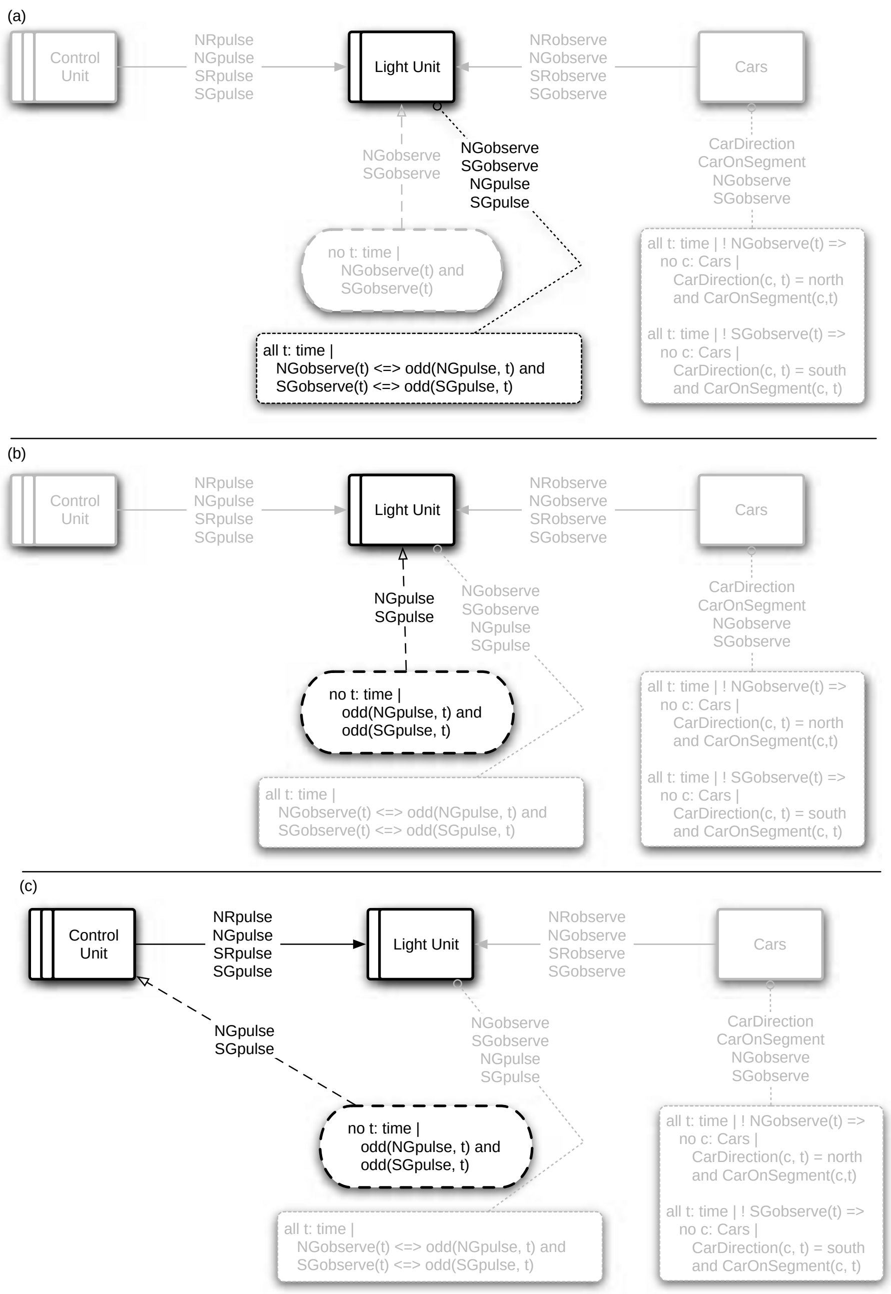

(a)
Control Unit
NRpulse
NGpulse
SRpulse
SGpulse
Light Unit
NRobserve
NGobserve
SRobserve
SGobserve
Cars
NGobserve
SGobserve
NGobserve
SGobserve
NGpulse
SGpulse
CarDirection
CarOnSegment
NGobserve
SGobserve
no t: time | NGobserve(t) and SGobserve(t)
all t: time | ! NGobserve(t) => no c: Cars | CarDirection(c, t) = north and CarOnSegment(c,t)
all t: time | ! SGobserve(t) => no c: Cars | CarDirection(c, t) = south and CarOnSegment(c, t)
all t: time | NGobserve(t) <=> odd(NGpulse, t) and SGobserve(t) <=> odd(SGpulse, t)
(b)
NGpulse
SGpulse
no t: time | odd(NGpulse, t) and odd(SGpulse, t)
(c)
no t: time | odd(NGpulse, t) and odd(SGpulse, t)

Figure 9. 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* that 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.

(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 9b.

(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 9c.

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.

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 10.

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.

5 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 6, 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. [6]

5.1 Sets and Relations

The key sets and relations that define a problem diagram are shown in object model notation [33] in Figure 11. 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.

[6] We use Alloy to formalize problem diagrams and the effect of our transformations on them (Sections 5 and 6) and also to express the constraints in particular examples (Sections 4 and 7). 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. Connecting the two kinds of models is future work (Section 10).

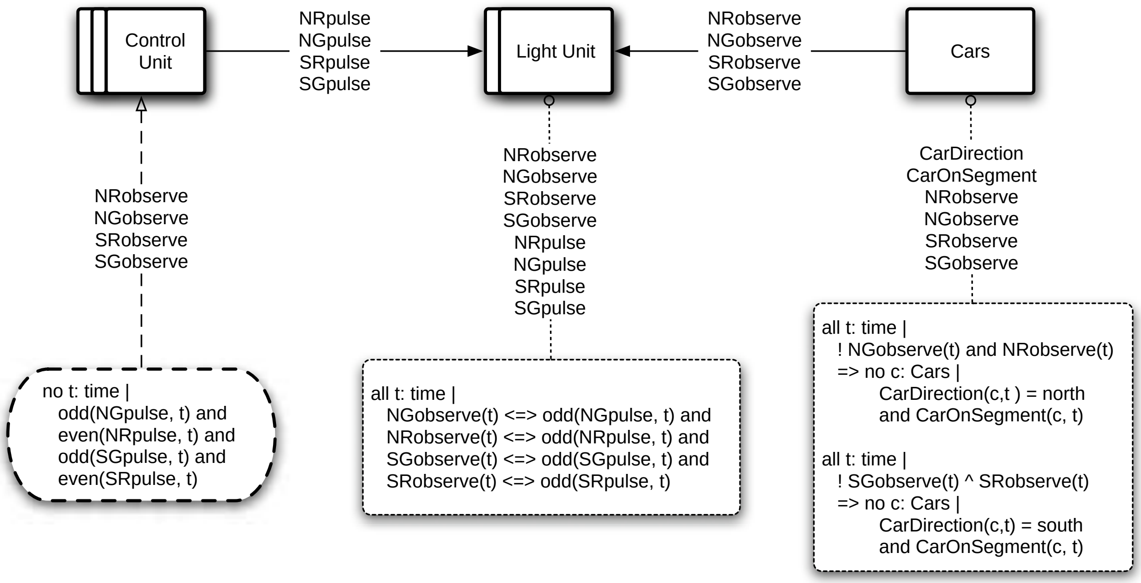

Control Unit
NRpulse
NGpulse
SRpulse
SGpulse
Light Unit
NRobserve
NGobserve
SRobserve
SGobserve
Cars
NRobserve
NGobserve
SRobserve
SGobserve
NRobserve
NGobserve
SRobserve
SGobserve
NRpulse
NGpulse
SRpulse
SGpulse
CarDirection
CarOnSegment
NRobserve
NGobserve
SRobserve
SGobserve
no t: time | odd(NGpulse, t) and even(NRpulse, t) and odd(SGpulse, t) and even(SRpulse, t)
all t: time | NGobserve(t) <=> odd(NGpulse, t) and NRobserve(t) <=> odd(NRpulse, t) and SGobserve(t) <=> odd(SGpulse, t) and SRobserve(t) <=> odd(SRpulse, t)
all t: time | ! NGobserve(t) and NRobserve(t) => no c: Cars | CarDirection(c,t) = north and CarOnSegment(c, t)
all t: time | ! SGobserve(t) ^ SRobserve(t) => no c: Cars | CarDirection(c,t) = south and CarOnSegment(c, t)

Figure 10. 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.

```
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
}
```

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 `Domains`.

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 `Constraints` 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]`.

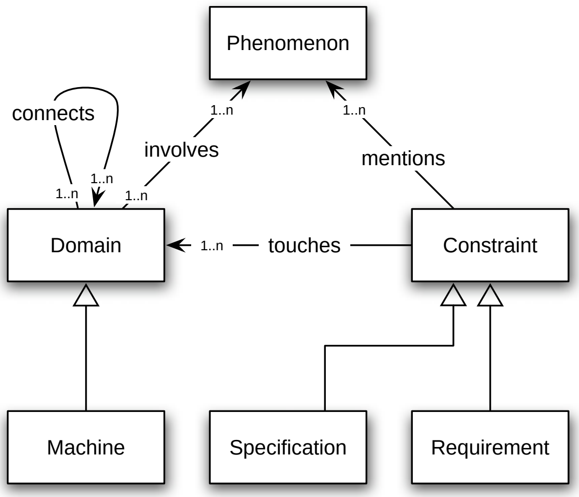


Figure 11. A metamodel of problem diagrams, expressed using standard object model notation.

5.2 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.

- 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.
- There must be exactly one machine.
- Every domain must be reachable from every other domain by following the `connects` relation zero or more times.
- Trivial diagrams are forbidden, such as disconnected diagrams or domains that contain no phenomena. Nontriviality 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.
- 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.

- 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.
- 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.
- A constraint is a specification if it touches only the machine.
- 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.

6 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
```

6.1 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.

6.2 The Transformations

The addition of a breadcrumb to a diagram is modeled as a predicate, given in Figure 12. 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.

The rephrasing of a requirement is modeled as another predicate, given in Figure 13. 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.

A third predicate, given in Figure 14, defines a requirement push. The only change to the diagram is that one requirement changes what it touches but remains well formed.

6.3 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.

7 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 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 [14]. 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.

7.1 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 [32]. The most infamous of these accidents are those involving the Therac-25 machine [20, 23], in whose failures faulty software was a primary cause. More recently, software appears to have been the main factor in similar accidents in Panama in 2001 [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.

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 [10]. We shall consider only the logging subproblem here.

7.2 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, 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.

```
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
  }
}
```

Figure 12. Adding a breadcrumb to a problem diagram.

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

Figure 13. Rephrasing a requirement.

```
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]
  }
}
```

Figure 14. Pushing a requirement.

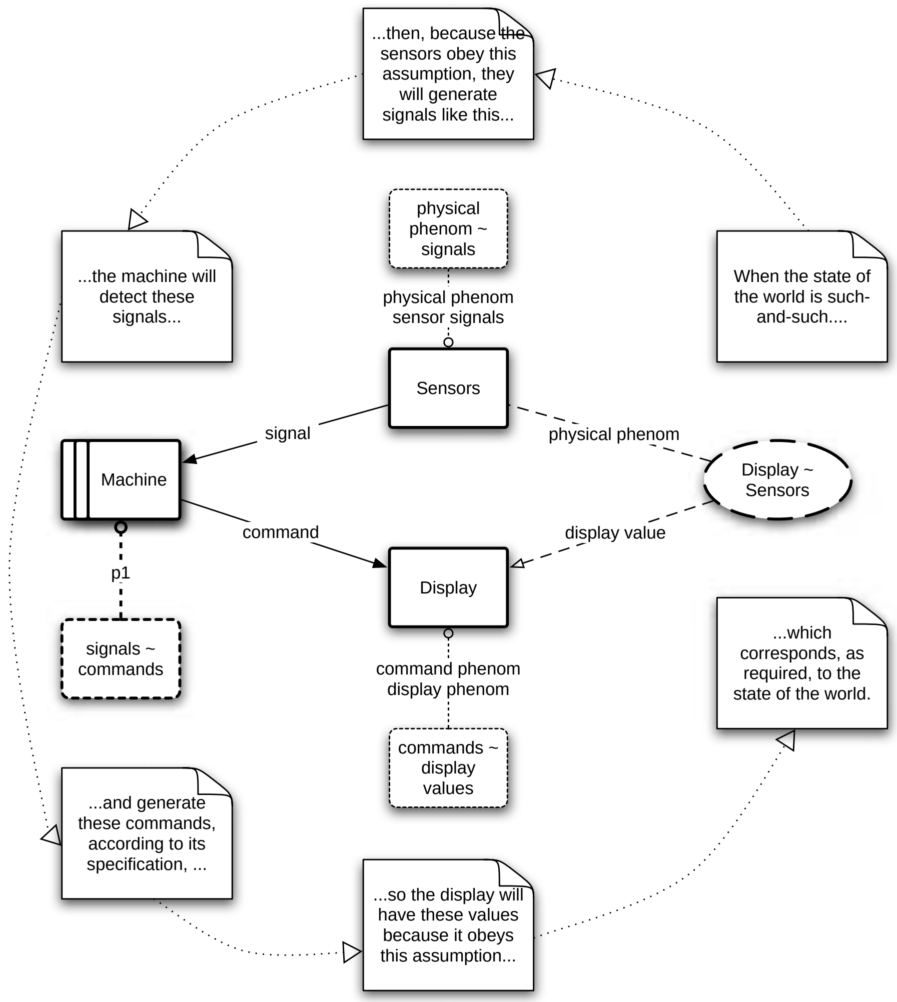

...then, because ~~the~~ sensors obey this assumption, they will generate signals like this...
physical phenom ~ signals
physical phenom sensor signals
...the machine will detect these signals...
When the state of the world is such-and-such....
Sensors
signal
physical phenom
Machine
Display ~ Sensors
command
display value
p1
Display
signals ~ commands
command phenom display phenom
...which corresponds, as required, to the state of the world.
commands ~ display values
...and generate these commands, according to its specification, ...
...so the display will have these values because it obeys this assumption...

Figure 15. An informal argument diagram for the *information display* frame.

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

Figure 16. 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.

7.3 The Phenomena

Figure 17 shows a problem diagram for the logging subproblem. In it, the informal logging requirement has been formalized using the Alloy language [9, 7, 11]. Designations[7] for the phenomena used in that diagram are given in Figure 16.

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 `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 reprsented 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.

7.4 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 15. 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 `Entries` correspond to `DoseUnits`.

The TCS does not just passively watch the patient and react to changes in `DoseUnit`s 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.)

[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. [14]

[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.

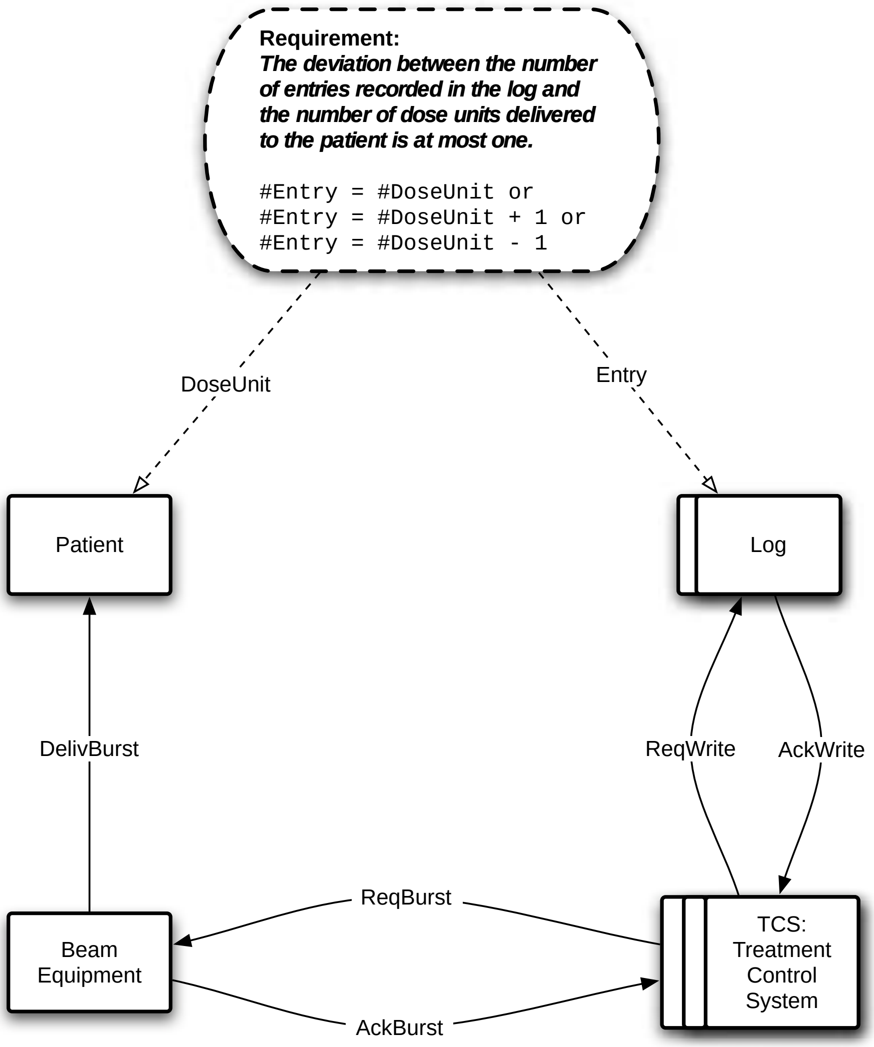
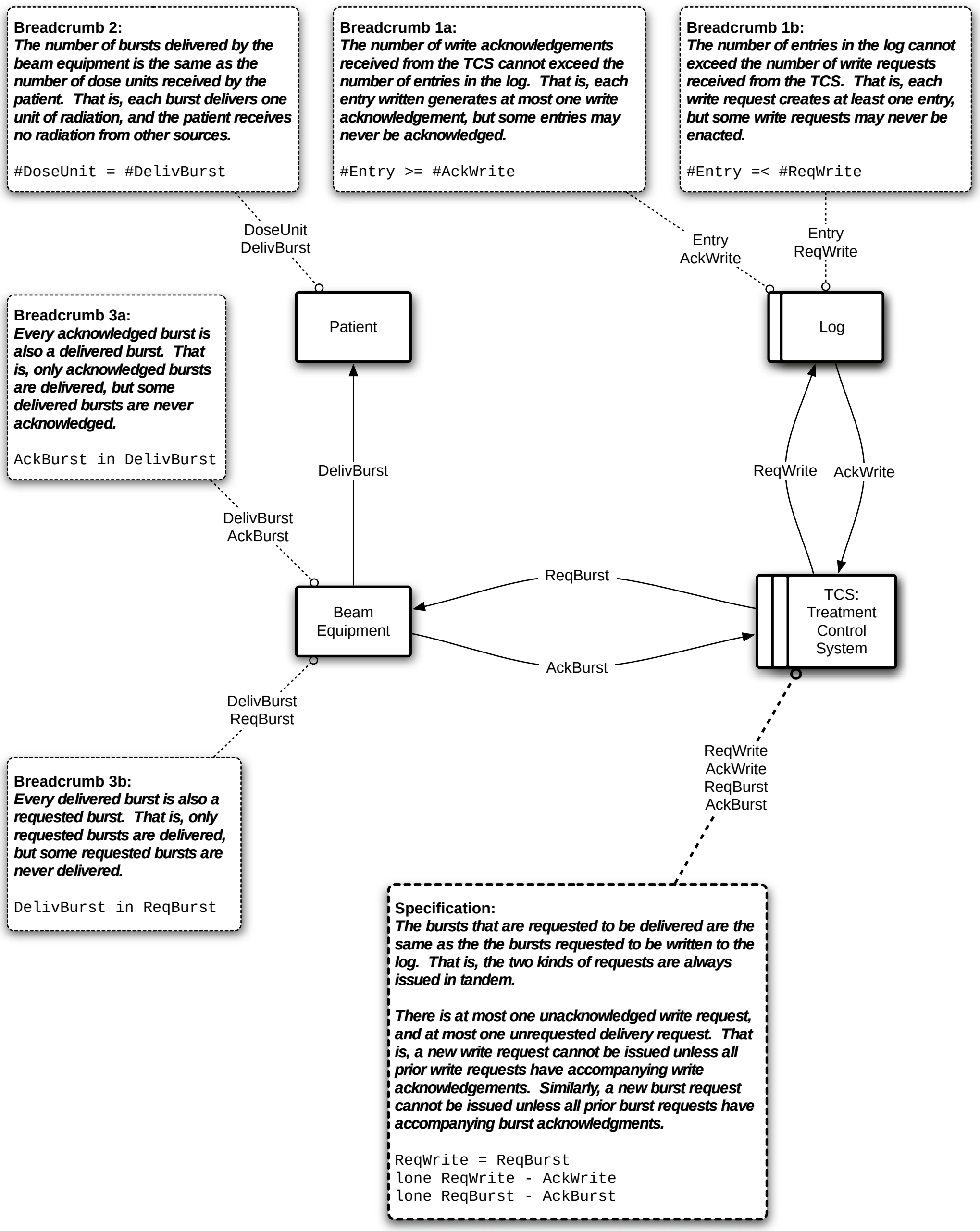

Requirement:
The deviation between the number of entries recorded in the log and the number of dose units delivered to the patient is at most one.
#Entry = #DoseUnit or
#Entry = #DoseUnit + 1 or
#Entry = #DoseUnit - 1
DoseUnit
Entry
Patient
Log
DelivBurst
ReqWrite
AckWrite
ReqBurst
Beam Equipment
TCS: Treatment Control System
AckBurst

Figure 17. 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.



Breadcrumb 2:
The number of bursts delivered by the beam equipment is the same as the number of dose units received by the patient. That is, each burst delivers one unit of radiation, and the patient receives no radiation from other sources.

```
#DoseUnit = #DelivBurst
```

Breadcrumb 1a:
The number of write acknowledgements received from the TCS cannot exceed the number of entries in the log. That is, each entry written generates at most one write acknowledgement, but some entries may never be acknowledged.

```
#Entry >= #AckWrite
```

Breadcrumb 1b:
The number of entries in the log cannot exceed the number of write requests received from the TCS. That is, each write request creates at least one entry, but some write requests may never be enacted.

```
#Entry =< #ReqWrite
```

Breadcrumb 3a:
Every acknowledged burst is also a delivered burst. That is, only acknowledged bursts are delivered, but some delivered bursts are never acknowledged.

```
AckBurst in DelivBurst
```

Breadcrumb 3b:
Every delivered burst is also a requested burst. That is, only requested bursts are delivered, but some requested bursts are never delivered.

```
DelivBurst in ReqBurst
```

Specification:
The bursts that are requested to be delivered are the same as the the bursts requested to be written to the log. That is, the two kinds of requests are always issued in tandem.

There is at most one unacknowledged write request, and at most one unrequested delivery request. That is, a new write request cannot be issued unless all prior write requests have accompanying write acknowledgements. Similarly, a new burst request cannot be issued unless all prior burst requests have accompanying burst acknowledgments.

```
ReqWrite = ReqBurst
lone ReqWrite - AckWrite
lone ReqBurst - AckBurst
```

Figure 18. 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.

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 are should change in order to satisfy he 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 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.

7.5 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 `DoseUnits`, 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 17 shows the problem diagram before requirement pregression begins, and Figure 18 shows the same diagram at upon completion.

7.6 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 18 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 DoseUnits 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 whene 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 18.

8 Discussion

8.1 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.

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

8.2 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.

8.3 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.

9 Related Work

9.1 Requirement Factoring

Many approaches to system analysis involve some kind of factoring of end-to-end requirements into subconstraints, often recursively. Assurance and safety cases [1, 20], for example, factor 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 [28]) are duals of these approaches, in which factoring is used to identify the root causes of failures. Leveson's STAMP approach involves decomposing design constriants, with a focus on managerial control over the operation of a system [21, 22].

More similar to our approach are frameworks, such as `i*` [37] and KAOS [3], that factor 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.

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.

The four-variable model [29, 36] 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.

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 [16]. Each refinement step places limits on what domains may know and on the domains' abilities 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.

9.2 Problem Frames

Michael Jackson sketches out a notion of *problem progression* in the problem frames book [14]. 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. Each step involves dropping the domains touching the requirement, then reconnecting the requirement to other domains and rephrasing it 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 instead 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) [15]. 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 [31]. 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 [14], in which the requirement is moved closer to the machine by eliminating intervening domains.

Hall, Rapanotti, Li, and M. Jackson are developing a calculus of requirements engineering based on the problem frames approach [24, 25, 30]. 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.

10 Future Directions

Our ultimate goal is to provide a structured and systematic way of building code-level arguments about system-level properties. Central to our effort is the use of problem progression to derive checkable specifications from system requirements. In this way, our broader work is similar to work that is being done on synthesizing problem frames with assurance cases [35, 27, 26]. So far, we have found requirement progression (problem progression in which only the constraints are changed) to be the most manageable way of doing problem progression in complex systems.

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. That said, the examples used in this paper are still too small to adequately determine if our technique scales. To better explore the issues that arise in more complex problem diagrams, we are developing several more case studies involving the BPTC software, involving requirements such as overdose avoidance, patient identity consistency, the accuracy of information presented to the therapist, and a more elaborate description of dose logging.

There currently is an incompatability between the formal description of our requirement progression technique in general (Sections 5 and 6) and the models we write to validate goal rephrasings in particular problem diagrams (Sections 4 and 7). We have done some preliminary work on connecting those two kinds of models. That is, Alloy models of particular problem diagrams can be written as extensions to the metamodel of problem diagrams given in Section 5. One result would be that the analyst could check that each proposed progression step is indeed following our requirement progression process. For example, one can ensure that a breadcrumb is not added to a domain that does not involve all of the necessary phenomena. In fact, building such a model has already found a (minor) bug in the arc labels of one of our published requirement progressions. Such a synthesis will also permit us to check the completeness of our transformation set.

The biggest shortcoming of our requirement progression technique 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. We would like to better incorporate and represent existing domain knowledge, so that the analyst is not producing breadcrumbs as blindly. For example, one might have state machine descriptions of each domain (similar to those suggested in the Problem Frame book [14]). Not only would such descriptions suggest possible breadcrumbs to the analyst, but they would also allow the analyst to check that a desired breadcrumb is usable (by checking that it is implied by the relevant domain's state machine). This approach might be especially helpful if the domain experts are not available on short notice. The analyst could build up a body of knowledge at a single meeting by building formal domain models, then gradually draw on that knowledge later on, during the course of the progression.

Another way to provide the analyst with more guidance is through the development of a catalog of transformation heuristics based on our experience with further case studies. 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.

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, and by the Brazilian Research Agency CNPq.

This article extends previous work published at the 2006 Requirement Engineering conference [34].

Appendix: An Alloy Model of Requirement Progression

This Alloy model is analyzable with the current, freely available, version 4 of the Alloy Analyzer [7].

```
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 }
```

References

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

[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 (ICSE'67)*, 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 International Symposium on the Foundations of Software Engineering (FSE'96)*, 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] Software Design Group. The Alloy Analyzer. website, 2007. http://alloy.mit.edu.

[8] 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.

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

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

[11] 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 Symposium on the Foundations of Software Engineering (ESEC/FSE'01)*, pages 62–73, Vienna, Austria, September 2001.

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

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

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

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

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.