Problem Frame Transformations: Deriving Specifications from Requirements

Robert Seater, Daniel Jackson
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
Cambridge, Massachusetts, USA
{rseater,dnj}@mit.edu

ABSTRACT

The Problem Frames approach provides a framework for understanding the interaction between software and other system components. It emphasizes decomposing an end-to-end system requirement into a machine specification plus a set of assumptions about domains in the problem world.

However, the standard approach does not provide the designer with a means for performing such a decomposition, apart from consulting a catalog of 'frame concern' patterns. We suggest a more systematic method for transforming an end-to-end system requirement into a machine specification plus a set of domain properties.

Categories and Subject Descriptors

Categories and Subject Descriptors: D.2.1 Software: Software Enigneering: Requirements/Specifications: Languages, Methodologies, Tools

General Terms: Design, Documentation

Keywords: Constraint, Domain Assumption, Goal Refinement, Phenomena, Problem Frames, Problem Progression, Requirements, Specification

1. INTRODUCTION

Many system failures have resulted from implicit assumptions about component boundaries which, when made explicit, are easily recognized and corrected [2, 3, 6, 21, 24]. With the increasing deployment of software, component interactions have become even more complex and subtle, increasing the difficulty of avoiding overlooked assumptions. Having all components satisfy their specifications is insufficient if those specifications do not, in combination, establish the desired end-to-end requirement.

The Problem Frames approach offers a framework for describing the interactions amongst software and other system components [15, 16]. 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, 14, 19].

Problem Frames emphasize decomposing an end-to-end system requirement into a machine specification plus a set of domain properties – assumptions about domains in the problem world. However, Problem Frames do not provide the designer with a systematic means for performing such a decomposition. The designer can try to match the subproblems against a catalog of patterns, called *frame concerns.* Each frame concern has an associated template for an informal argument of correctness. If the designer's catalog does not have an appropriate frame concern, there is no guidance for correctly developing a new one.

We extend the Problem Frames approach to include a systematic way to transform an end-to-end system requirement into a machine specification. Given a Problem Frame description and an end-to-end requirement, a series of transformations turn the requirement into a specification and produce a set of *breadcrumb assumptions* about the problem world. The specification and breadcrumbs form a frame concern correctness argument for why the machine enforces the requirement.

Our work is very similar to the sample transformation of a requirement into a specification performed by Jackson and Zave [17]. Whereas their paper focuses more on the individual steps of the transformation, our paper addresses the overall structure, and we believe it suggests a promising strategy for oranizing transformations in large problem frames with more complex topologies.

2. PROBLEM FRAMES

A system designer has an end-to-end *requirement* on the world that some machine is to enforce. For example, a traffic-light control unit should enforce the requirement that cars do not collide, and the control unit of a radiation therapy machine should enforce the requirement that the radiation delivered to a patient never exceeds the prescribed dose.

In order to implement or analyze the machine, one needs a *specification* on the machine's interface. Since the phenomena referenced in the requirement are typically not shared by the machine, the requirement cannot serve as a specification. How, then, does one ensure that the specification is sufficient to enforce the desired requirement? Problem Frames express this disconnect diagramatically [16] (Figure 1).

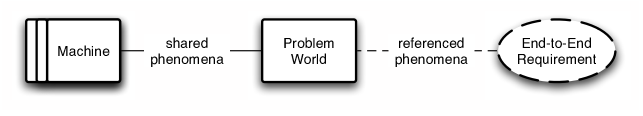


Figure 1: A generic Problem Frames description.

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

Consider the case where the problem world is represented by a single domain with two phenomena: the machine shares `p1` with the problem domain, and the requirement references `p2` (Figure 2).

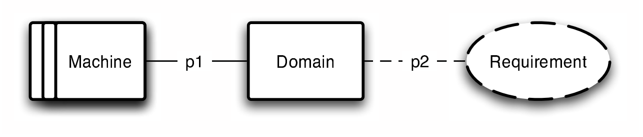


Figure 2: An archetypal problem frame.

Given a requirement $R(p2)$, the designer needs to develop a specification $S(p1)$ and a set of domain assumptions $\{A_1(p1, p2), ...A_n(p1, p2)\}$ such that

$$S(p1) \wedge A_1(p1, p2) \wedge ... \wedge A_n(p1, p2) \Rightarrow R(p2)$$

In the conventional Problem Frames approach, the designer examines a catalog of patterns to find a *frame concern* that matches the problem at hand. The frame concern comes with an informal template of a correctness argument. A diagramatic view of a simple template is shown in Figure 3. When multiple domains are involved, a more elaborate correctness argument is required.

Unfortunately, the designer's catalog may not include a relevant frame concern, or, worse, may include one with an incomplete or inadequate correctness argument. A conventional Problem Frames analysis does not offer guidance on how to produce and verify a specification in such a situation.

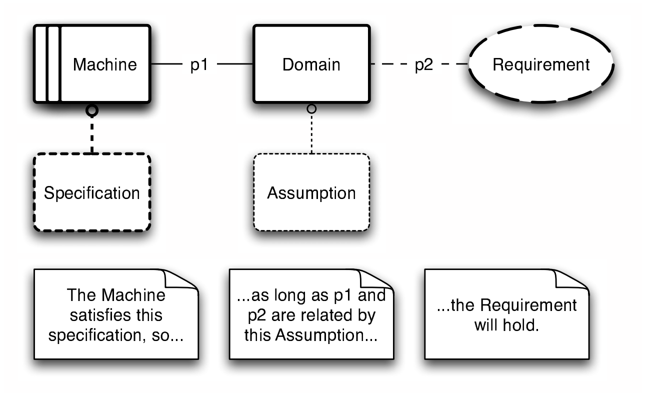

Machine
p1
Domain
p2
Requirement
Specification
Assumption
The Machine satisfies this specification, so...
...as long as p1 and p2 are related by this Assumption...
...the Requirement will hold.

Figure 3: A frame concern.

2.1 Problem Frame Transformations

We introduce a process for systematically transforming a requirement into a specification. A by-product of the transformation is a trail of domain assumptions, called *breadcrumbs*, that justify the progression.

Requirements, specifications, and breadcrumbs are three instances of *domain constraints*. Requirements connect only to non-machine domains[1], specifications connect only to the machine domain, and each breadcrumb connects only to a single non-machine domain. Specifications and requirements differ only in what sorts of domains they connect to; deriving a specification from a requirement is, in principle, just a matter of replacing references to one set of phenomena (from non-machine domains) with another set of phenomena (from the machine domain). The transformation process we propose serves to structure the task of making that replacement.

2.2 Transformation Process

During the course of the transformation, the specification-to-be may end up simultaneously connecting to both the machine domain and one or more non-machine domains. It is thus neither a requirement nor a specification and will be called the *goal*.

There are three types of steps in the transformation process: *Adding* a breadcrumb permits the goal to be *rephrased*, which in turn enables a *push* to change which domains it connects to. Figure 4 shows a diagramatic archetype of how these steps take a goal from being a requirement to being a specification.

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

(b) *Rephrase* the goal so that it references different phe-

[1]Some users of Problem Frames permit requirements to mention machine phenomena, but for simplicity we assume that they do not. Relaxing this restriction would not affect the applicability of our technique; it would just mean that the requirement is that much closer to becoming a specification.

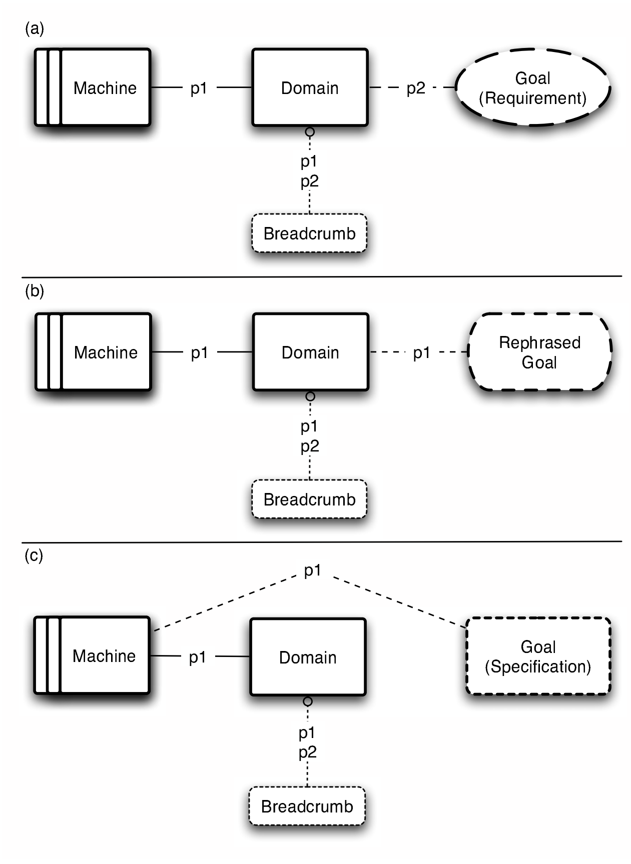


Figure 4: An archetypical transformation: (a) A *breadcrumb* constraint is added, representing an assumption about how the domain relates `p1` and `p2`. (b) That breadcrumb permits the requirement to be *rephrased* to reference `p1` instead of `p2`. (c) The rephrasing enables us to *push* the requirement from the domain onto the machine.

nomena than before (e.g. `p1` instead of `p2`), although it still only references phenomena from domains to which it connects. This is done to enable a useful push (part c). The designer formally verifies that the breadcrumb is sufficiently strong to permit the rephrasing:

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

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

The designer continues to perform transformations until the goal connects only to the machine domain. At that point, it only references phenomena at the interface of the machine and is a valid specification. The decisions of what breadcrumbs to add, how to rephrase the goal, and which enabled pushes to enact are qualitative assessments by the designer based on experience or a catalog of patterns and heuristics.

2.3 Definitions

A Problem Frame description consists of the following elements:

- A set of *Phenomena* $P = \{p_1, p_2, ...\}$.
- A set of *Domains* $D = \{d_1, d_2, ...\}$. There is a single *machine* domain, $m \in D$. A domain d *involves* a set of phenomena $I(d) \subseteq P$.
- A set of *Constraints* $C = \{c_1, ..., c_n\}$. A constraint c *references* some set of phenomena $R(c) \subseteq P$ and *touches* a set of domains $T(c) = \{d_1, ..., d_n\} \subseteq D$. (Diagramatically, a dashed arc would be drawn to connect c to each domain in $T(c)$.)

The touches relation must obey a well-formedness property: For any constraint c and phenomenon $p \in R(c)$, there must be some domain $d \in T(c)$ such that $p \in I(d)$. Furthermore, for each $d' \in T(c)$ there must be some $p' \in R(c) \cap I(d')$. That is, if a constraint references a phenomenon then it must touch some domain that involves that phenomenon, and if it touches a domain then it much involve some phenomenon included in that domain.

Using this terminology, the 3 special types of constraints are defined as follows:

- A constraint c is a *requirement* iff $m \notin T(c)$.
- A constraint c is a *specification* iff $M = T(c)$.
- A constraint c is a *breadcrumb* iff there is a single domain $d \neq m$ such that $T(c) = d$.

The three components of a transformation can now be more precisely defined as follows:

Preprocessing:
At all points in the transformation process, there will be exactly one constraint designated as the *goal*. Initially there are no breadcrumbs or specifications, and there is exactly one requirement, which is the initial goal. The goal constraint will eventually be the derived specification.

Breadcrumb Addition:
Let g be the goal, and let d be some domain such that $g \in T(d)$. Create a new breadcrumb constraint b for which $R(b) \subseteq I(d)$ and $T(b) = d$.

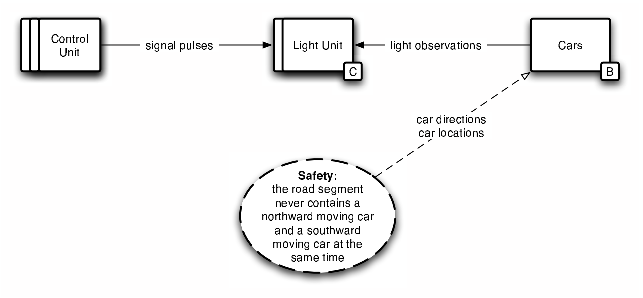
Requirement Rephrasing:
Let g be the goal constraint. Add a new constraint, g', such that $T(g') = T(g)$. The sets of phenomena referenced by g' and g may (and are expected to) differ. There must be some set of breadcrumb constraints $B = \{b_1, ..., b_m\}$ such that $b_1 \wedge ... \wedge b_n \wedge g' \Rightarrow g$. Change g' to be the goal, and remove g.

Requirement Push:
Let g be the goal. Change T to a new T' such that $T'(g) \neq T(g)$ but $\forall c \in C - g | T'(c) = T(c)$. T' must still satisfy the well-formedness property.

Termination:
When the goal constraint is a specification, halt.

3. TWO-WAY TRAFFIC-LIGHT

Consider an informal Problem Frame description of a traffic-light controlling two-way traffic used for road construction [16] (Figure 5)[2].



Control Unit
signal pulses
Light Unit
C
light observations
Cars
B
car directions
car locations
Safety: the road segment never contains a northward moving car and a southward moving car at the same time

Figure 5: A Problem Frame description of a two-way traffic light.

The unit has four physical lights: a red and a green in each direction. The control unit sends signal pulses to the light unit to 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 end-to-end requirement is that cars do not collide, which we will interpret to mean that no two cars are ever in the intersection 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. So what specification should the control unit be expected to uphold?

In Figure 6, the Problem Frame description has been formalized; the phenomena have been concretized and the requirement specified in formal language. For example, `NRpulse(t)` designates that a signal pulse was sent to the control unit at time `t` to toggle whether the northward facing red light is lit or not. `NRobserve(c, t)` designates that a car `c` observes the northward red light to be lit at time `t`. `CarOnSegment(c, t)` designates that a car `c` is on the road segment at time `t`. `CarDirection(c, t)` designates the direction car `c` is moving, and can evaluate to either `north` or `south`.

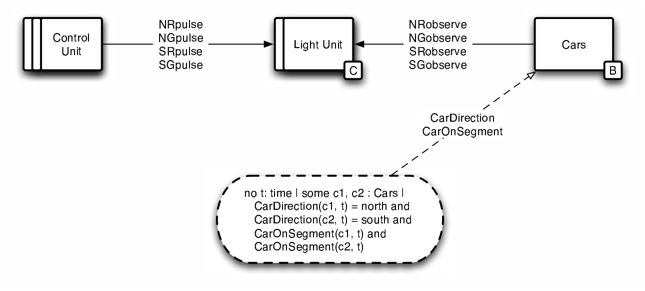
We have chosen to write constraints (both requirements and breadcrumbs) in a first-order relational logic – such logics are amenable to analysis and are often a natural way to express requirements [13].

The requirement is connected to the `Cars` domain via observation phenomena (e.g. `NGobserve`), whereas the specification will have to connect to the `Control Unit` via signal phenomena (e.g. `NGpulse`). In order to turn the requirement into a specification, we will have to reconcile those two sets of phenomena.

3.1 First Transformation

The first thing we would like to do is to push the requirement from the `Cars` domain onto the `Light Units` domain,

[2]We use a slightly non-standard notation in our Problem Frame diagrams for the arc indicating that domain `D` controls phenomenon `p`. Rather than labeling the arc `D!p`, we label it `p` and place an arrow head pointing away from `D`. When not all phenomena shared by two domains are controlled by the same domain, separate arcs are used.



Control Unit
NRpulse
NGpulse
SRpulse
SGpulse
Light Unit
C
NRobserve
NGobserve
SRobserve
SGobserve
Cars
B
CarDirection
CarOnSegment
no t: time | some c1, c2 : Cars | CarDirection(c1, t) = north and CarDirection(c2, t) = south and CarOnSegment(c1, t) and CarOnSegment(c2, t)

Figure 6: A formalization of the requirement and phenomena in the two-way traffic light example.

as that will obey our heuristic of trying to get it closer to the `Control Unit` machine domain. However, no push is currently enabled.

In order to enable such a push, we add a breadcrumb constraint on `Cars` which permits us to rephrase the requirement so that the only phenomena it references are `NRobserve`, `NGobserve`, `SRobserve`, and `SGobserve`. We will then be able to push the requirement from `Cars` onto `Light Unit`.

These three steps are illustrated in Figure 7 and narrated below.

(A) Add a Breadcrumb

We need to make an assumption about the `Cars` domain that will help us reconcile `CarDirection` and `CarOnSegment` with the four observation phenomena. If we assume that cars obey traffic signals that they observe, then we can substitute information about observations of light colors for information about car behavior. We do so by adding the following breadcrumb constraint to `Cars`:

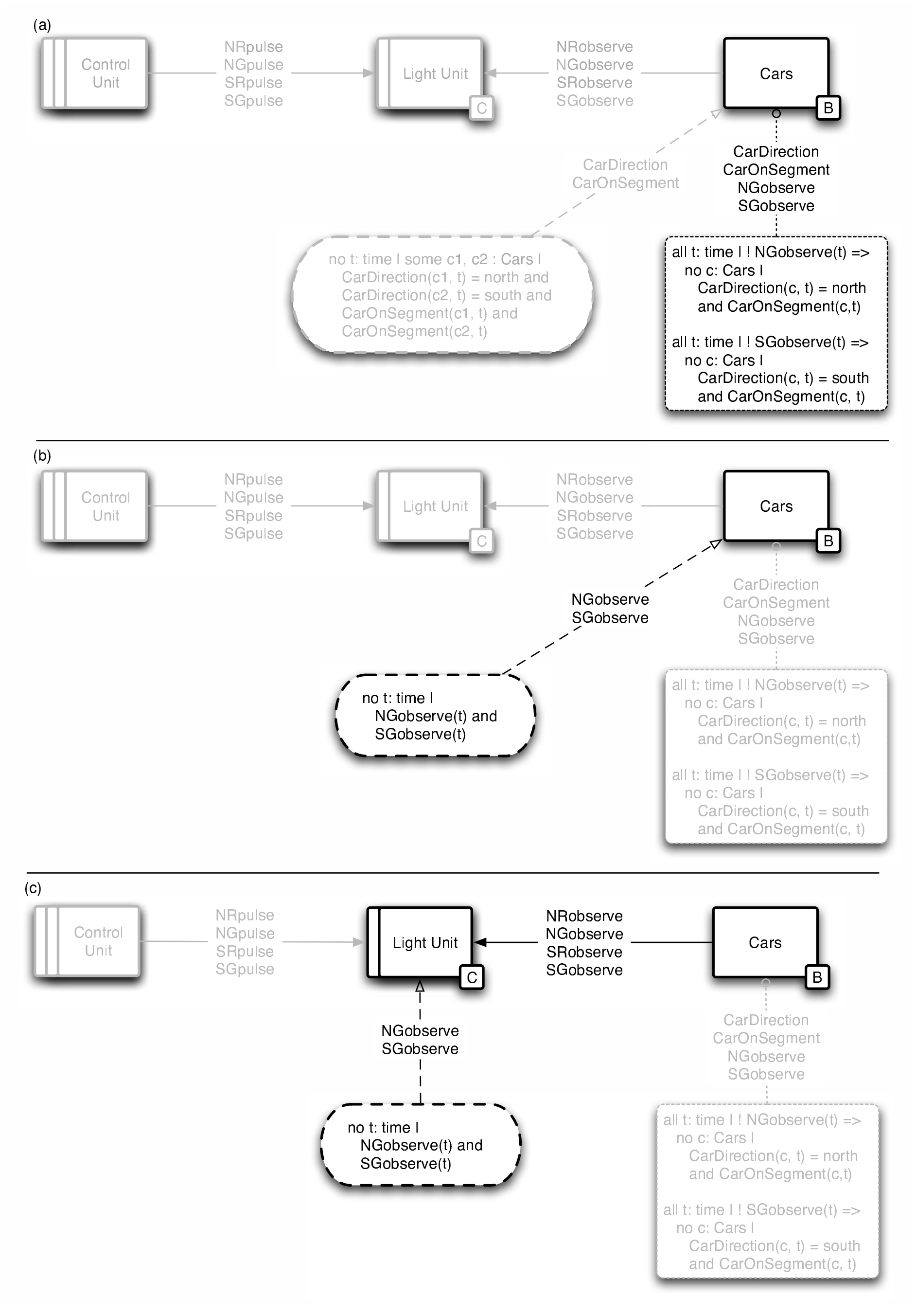
```
pred BreadCrumb_Cars() {
  all t: time |
    ! NGobserve(t) =>
    no c: Cars |
      CarDirection(c, t) = north
      and CarOnSegment(c,t)

  all t: time |
    ! SGobserve(t) =>
    no c: Cars |
      CarDirection(c, t) = south
      and CarOnSegment(c, t)
}
```

which says that, at any given time, if a car does not observe a green light in its direction, then it cannot be on the road segment. For the sake of simplicity, we will ignore the delays between when an observation is made, when a car has reacted and entered the road segment, and when the car has exited the road segment. We therefore omit a yellow light.

(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.



(a)
Control Unit
NRpulse
NGpulse
SRpulse
SGpulse
Light Unit
C
NRobserve
NGobserve
SRobserve
SGobserve
Cars
B
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)
NGobserve
SGobserve
no t: time | NGobserve(t) and SGobserve(t)
(c)

Figure 7: 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.

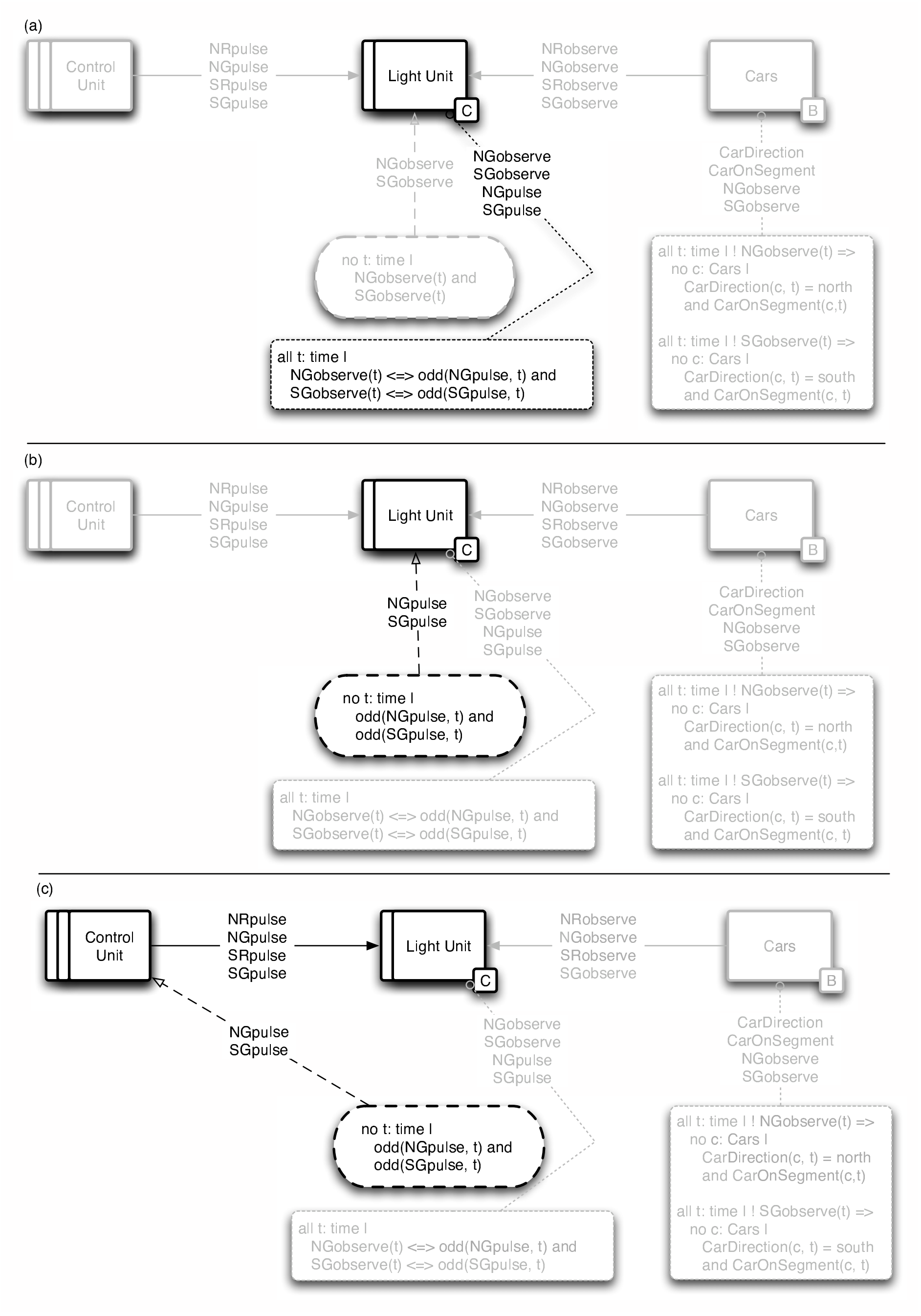

(a)
Control Unit
NRpulse
NGpulse
SRpulse
SGpulse
Light Unit
C
NRobserve
NGobserve
SRobserve
SGobserve
Cars
B
NGobserve
SGobserve
NGobserve
SGobserve
NGpulse
SGpulse
CarDirection
CarOnSegment
NGobserve
SGobserve
no t: time | NGobserve(t) and SGobserve(t)
all t: time | NGobserve(t) <=> odd(NGpulse, t)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

Figure 8: 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.

```
pred Rephrased_Goal() {
  no t: time |
     NGobserve(t) and
     SGobserve(t)
}
```

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

```
assert Transformation_1 {
  Rephrased_Goal() and BreadCrumb_Cars()
   => Original_Requirement()
}
check Transformation_1 for 10
```

In general, how such implications are discharged will depend on the problem domain. Since our constraints are written in first-order relational logic, we used the Alloy Analyzer to perform a bounded, exhaustive check [13, 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[3].

(C) Push the Requirement

The only phenomena referenced by the new requirement are `NGobserve` and `SGobserve`. Since those phenomena are shared by both the `Cars` and `Light Unit` domains, we are now permitted to push the requirement from one to the other.

3.2 Second Transformation

The requirement is now one step away from being a specification. The second transformation is to push the requirement the rest of the way onto the `Control Unit` domain. In order to do so, we will need add another breadcrumb and rephrase the requirement.

These three steps are illustrated in Figure 8 and narrated below.

(A) Add a Breadcrumb

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 references to signal pulses for references to 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 BreadCrumb_LightUnit {
  all t: time |
     NGobserve(t) <=>
        odd(NGpulse, t) and
     SGobserve(t) <=>
        odd(SGpulse, t)
}
```

[3]The Alloy model included the requirements exactly as they are stated in the diagrams, plus a few supporting definitions and data structures to describe the structure of the problem domain. It was solved instantaneously on a 133MHz G4 PowerMac with 800Mb of RAM, using the downloadable version of Alloy 3 [7]. More detail on using Alloy to model Problem Frames and check transformation steps is given in another paper [28].

which 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.

(B) Rephrase the Requirement

Dividing the requirement by our breadcrumb produces a new requirement which refers to signal pulses instead of observations:

```
pred Specification {
  no t: time |
     odd(NGpulse, t) and
     odd(SGpulse, t)
}

assert Transformation_2 {
  Specification() and BreadCrumb_LightUnit()
   => Rephrased_Goal()
}
check Transformation_1 for 10
```

where `odd` is a function that determines the parity of the occurences of the given phenomenon up to the given time.

We use the Alloy Analyzer to verify that the new requirement plus the breadcrumb imply the prior requirement. It passes, so the breadcrumb is strong enough.

(C) Push the Requirement

The requirement now reference only phenomena shared by both the `Light Unit` and `Control Unit` domains, so we can push it from one to the other.

3.3 Specification

Now that the requirement has been pushed all the way onto the machine domain, it only refers to phenomena known about by the machine and is a legal specification for that machine. We have derived a specification for the control unit, a correctness argument for why it enforces the desired requirement, and a set of assumptions about the world upon which we are relying. 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 requirement.

3.4 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 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 9.

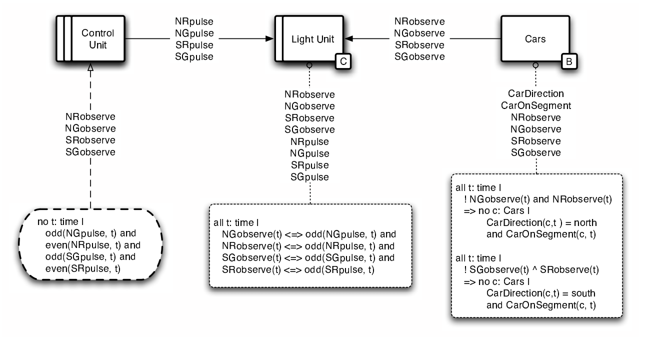

Control Unit
NRpulse
NGpulse
SRpulse
SGpulse
Light Unit
C
NRobserve
NGobserve
SRobserve
SGobserve
Cars
B
NRobserve
NGobserve
SRobserve
SGobserve
NRobserve
NGobserve
SRobserve
SGobserve
NRpulse
NGpulse
SRpulse
SGpulse
CarDirection
CarOnSegment
NRobserve
NGobserve
SRobserve
SGobserve
no t: time | odd(NGpulse, 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 9: The result of correcting the erroneous breadcrumb.

4. LIMITATIONS AND FUTURE WORK

4.1 Decisions

The transformation process we propose serves to structure the task of decomposing a requirement into a specification plus a set of domain assumptions. It offers a guarantee that if each individual step is sound then the resulting specification will be sufficient to enforce the desired requirement. It does *not* automate the task or substitute for a skilled human designer. Which transformations are made (how it is are made) is still up to the designer's intuition and experience. Poor choices by the designer along the way will never produce an incorrect specification, but they may

(1) produce specifications that are excessively complex and useless for guiding an actual implementation, and/or

(2) make later pushes difficult or impossible. For example, a breadcrumb strong enough to enable the next desired push may be too weak to permit a later push.

We hope that this framework can be expanded to include patterns which will help designers avoid and recover from such situations.

4.2 Topologies

The traffic light example we use in this paper has a *sequential* topology; the domains in the problem world are laid out in a chain, with the machine at one end and the requirement at the other. In such a case, the transformations are just a matter of pushing the requirement down the sequence of domains until it rests on the machine.

We have applied the technique in a limited fashion to parts of a Proton Therapy device [28], for which the Problem Frame description had a *parallel* topology – the machine and the requirement both connect to each of the domains in the problem world. The requirement thus has several different dashed arcs connecting it to problem domains. Each of those arcs can be pushed towards the machine, independent of each other, in exactly the same manner as the sequential case.

More generally, a problem frame description might have both sequential and parallel parts, forming diamonds and loops. In such cases, pushing an arc will sometimes involve splitting it into several arcs, each labeled with a subset of the phenomena on the original arc. Similarly, if several arcs are pushed onto a common domain then they can be merged. Apart for split and merge operations on arcs, the technique functions the same as in the sequential and parallel cases.

In both the parallel and diamond cases, there is a risk that choices make on the two paths may end up contradicting each other when they reconverge, preventing progress. Such cases can have the undesirable effect of forcing the designer to reason about the system as a whole, undoing the modularity that our technique provides.

5. RELATED WORK

5.1 Non-Problem Frames

Many approaches to system analysis involve some kind of decomposition of end-to-end requirements into subconstraints, often recursively. Assurance and safety cases [1, 21], for example, decompose a critical safety property. They tend to operate at a larger granularity than problem frames, in which the elements represent arguments or large groupings of evidence, rather than constraints. Analyses that focus on failures rather than requirements (such as HAZOP [25]) are duals of these approaches, in which decomposition is used to identify the possible root causes of failure.

More similar to our approach are frameworks, such as `i*` [29] and KAOS [4], that decompose system-level properties by assigning properties to agents that work together to achieve the goal. For KAOS, patterns have been developed for refining a requirement into subgoals [5]. In our approach, we have not given a constructive method for obtaining the new constraint systematically, and the refinement strategies of KAOS may fill this gap.

The four-variable model [26] makes a distinction, like Problem Frames, between the requirements, the specification, and domain assumptions. However, in Problem Frame terms, 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 [18]. To him, specifications and requirements only differ in how specific they are about what parts of the state elements of the problem domain can know, what their capabilities are to change the state, and to what extend they can violate given constraints. Initially, a requirement permits domain elements to know everything and have unlimited capability, but they must completely obey all given constraints. As a requirement is refined into a specification, limitations are placed on knowledge and capabilities, and exceptions are added to the constraints. Consequently, 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, our use of problem frames means that we take the opposite view. As we transform a requirement into a specification, we add assumptions (breadcrumbs) which expand our assumptions about the domains rather than restricting them. Furthermore, a specification and its requirement are always consistent; in fact, the specification conjoined with the breadcrumbs will logically imply the original requirement.

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 [20]. They are interested in producing operational specifications from requirements expressed in temporal logic, and focus on proving the correctness of a set of inference patterns. These inference patterns are correct regardless of context, in contrast to our approach in which transformations are only made through the introduction of domain assumptions.

5.2 Problem Frames

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) [17]. Their work is very similar to our own, and uses a logical constraint language to express domain assumptions. Our work attempts to generalize the process to be applicable in broader circumstances, and to help guide the designer through the process with the visual notion of pushing the requirement towards the machine.

Jackson sketches out a notion of *problem progression* in the Problem Frames book [16]. A progression of problems is a sequence of Problem Frame descriptions, beginning with the full description (including the original requirement) and ending with a description containing only the machine and its specification. In each successive description, the domains connected to the requirement are eliminated and the requirement is reconnected and altered as needed. He does not work out the details of how one would derive the successive descriptions, but it is clear that he had a similar vision to our own. Rather than eliminating elements of the diagram, our approach actually adds to it; as the requirement is shifted towards the machine domain, it leaves a trail of breadcrumbs as a record of the designer's reasoning.

Rapanotti, Hall, and Li show how causal reasoning can be used as one way to formalize problem progression for Problem Frames [23]. They are interested not only in problem progression, but also in *requirements traceability*, the ability to relate requirements written during different phases of a product's lifecycle.

Hall and Rapanotti have developed a notion of a *problem transformation*, which attempts to do for solutions what Problem Frames do for problem contexts [10]. They focus on understanding how one solution can lead to, or be transformed into, another solution for a similar problem. One of the components of a problem transformation is a *requirement progression*, in which a requirement is replaced by a weaker (or equivalent) requirement in a different form. This differs from our notion of a *problem frame transformation* in that our transformations change a requirement into a *different* requirement, and justify the difference with a set of explicit assumptions.

Hall, Jackson, Laney, Nuseibeh, and Rapanotti extend Problem Frames to allow architectural structures and services to be represented as part of the problem domain [9, 27]. One can then identify frame concerns that match parts of the architecture and apply their associated correctness arguments to questions of system correctness and stability. Like our own work, they are concerned with how to relate Problem Frame descriptions to the specifics of a machine, but their focus is on representing the structure of the machine rather than relating a specification to a requirement.

Acknowledgments

This research was supported, in part, by grants 0086154 ('Design Conformant Software') and 6895566 ('Safety Mechanisms for Medical Software') from the ITR program of the National Science Foundation.

We appreciate the feedback we received from Michael Jackson, Jon Hall, and our anonymous reviewers.

6. REFERENCES

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

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

[3] Mars Climate Orbiter Mishap Investigation Board. Phase I report. 1999.

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

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

[6] Sol Greenspan, John Mylopoulos, and Alex Borgida. On formal requirements modeling languages: RML revisited. In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*, pages 135–147. IEEE Computer Society Press, 1994.

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

[8] C. B. Haley, R. Laney, and B. Nuseibeh. Using Problem Frames and projections to analyze requirements for distributed systems. In *Proceedings of the 10th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'04)*, pages 203–217. Essener Informatik Beitrage, 2004. Editors: B. Regnell, E. Kamsties, and V. Gervasi.

[9] J. Hall, M. Jackson, R. Laney, B. Nuseibeh, and L. Rapanotti. Relating software requirements and architectures using Problem Frames. *IEEE Computer Society Press*, pages 137–144, Sept 2002.

[10] John G. Hall and Lucia Rapanotti. A framework for software problem analysis, 2006. Tecnical Report.

[11] Jon G. Hall, Lucia Rapanotti, and Michael Jackson. Problem Frame semantics for software development. *Journal of Software and Systems Modeling*, 4(2):189 – 198, 2005.

[12] C. A. R. Hoare and Jifeng He. The weakest prespecification. *Inf. Process. Lett.*, 24(2):127–132, 1987.

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

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

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

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

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

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

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

[20] E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. 2002.

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

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

[23] Zhi Li Lucia Rapanotti, Jon G. Hall. Problem reduction: a systematic technique for deriving specifications from requirements, 2006. Technical Report 2006/02.

[24] Peter Naur and Brian Randell. Software engineering, January 1969.

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

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

[27] Lucia Rapanotti, Jon G. Hall, Michael Jackson, and Bashar Nuseibeh. Architecture-driven problem decomposition. In *Proceedings of the 2004 International Conference on Requirements Engineering (RE04)*. IEEE Computer Science Press, 2004.

[28] Robert Seater and Daniel Jackson. Problem Frame transformations in the context of a proton therapy system. Unpublished manuscript.

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

[30] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997.