Separating Concerns in Requirements Analysis: An Example

Daniel Jackson[1] and Michael Jackson[2]

[1] Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA
[2] Independent Consultant
London, England

Abstract. Often, a requirements document is structured as a long list of individual "requirements", each describing an anticipated function or user interaction. An alternative approach is to identify a collection of subproblems, each representing an aspect of the larger problem, and to describe each subproblem in isolation, deferring their composition to a later stage. This paper illustrates the approach by applying it to the requirements of the positioning functions of a proton therapy installation. It explains how a flaw in the design of the system can be isolated to a single subproblem, which can be formalized and subjected to automatic analysis.

1 Introduction

Many approaches to requirements analysis focus on the anticipated interactions between users and the system to be built. These interactions may be structured as a collection of representative scenarios or 'use cases'. Often the requirements document is just an elaborate informal narrative describing in detail the sessions of each class of user. By drawing attention to the experience of users, these approaches can be a useful kind of paper prototype.

A major flaw of such approaches is that, for many systems, they focus in the wrong place. The problem to be solved by the system usually exists not at the interface with the machine, but deeper in the environment [9]. The purpose of a traffic light system, for example, is not to control the lights but to ensure steady and safe flow of traffic. Its requirements analysis should therefore start with traffic and the expected and desired behaviours of drivers, rather than with the question of how the lights should be sequenced.

This paper addresses a different but related flaw of approaches based on user interaction: that enumerating and elaborating scenarios tends to conflate different concerns. A system must usually satisfy multiple properties, perform multiple functions simultaneously, and satisfy multiple purposes. The eventual design of its user interface brings these multiple concerns together. But to describe the interface be-

M. Butler et al. (Eds.): REFT 2005, LNCS 4157, pp. 210 – 225, 2006.

fore the concerns have been identified and explored puts the cart before the horse. It can easily result in a development in which the individual concerns are never properly grasped, and are therefore inadequately addressed or made unnecessarily complicated.

This problem has special significance for systems that must be highly dependable. An inability to separate concerns makes it hard to pay more attention to the concerns that are more critical, and the resulting system may fail to satisfy its most critical requirements because their implementation is interwoven with the implementation of less critical requirements. In an earlier study, we found that the software control of the emergency stop feature of a radiotherapy machine was dependent on far less important features of the system; a signal to stop could be rejected, for example, if the disk were full so that a log record could not be written [12]. (Fortunately a redundant hardware interlock was in place.)

An alternative approach identifies the concerns at the outset. Instead of attempting to describe an interface that integrates the various concerns, each concern is considered independently, and only later is the composition of the concerns addressed [9]. This paper illustrates the approach with an example of a problem that arose in the development of the software for a proton therapy machine. The work is part of an ongoing collaboration between the Software Design Group at MIT and the Burr Proton Therapy Center (BPTC) at Massachusetts General Hospital whose aim is to find ways to improve the dependability of critical software.

The problem was known to the developers of the therapy system, and had been resolved before the writing of this paper, and it never posed a safety risk. But it is worth studying because it illustrates the pitfalls of the traditional approach to requirements analysis, some potential benefits of an approach based on problem decomposition, and is characteristic of problems that arise in many similar systems.

2 The Proton Therapy System

Proton therapy involves exposing a patient's tumour to a focused beam of protons. The positioning of the patient and the device issuing the beam is an intricate matter. At the BPTC, the positioning is carried out in two distinct phases. In the first phase, the patient and device are put in a "setup position" that is suitable for imaging. An X-ray image is taken to determine the exact position of the tumour, and a "delta" is obtained that captures the difference between the setup position and the position that would be required for the beam to be appropriately aligned. In the second phase, the patient and device are oriented in the "treatment position"; the delta obtained during setup is applied as a correction to the initial treatment position so that the proton beam will be aligned correctly.

Patient and beam position are adjusted in a number of ways. The beam follows a path along a fixed beamline from the cyclotron to the treatment room, and is bent by electromagnets to align with a snout mounted on a gantry that surrounds the patient couch and can rotate around one axis. The snout itself moves in and out

(towards and away from the patient), and can also rotate. The patient is positioned, often on a firm cushion, on a robotic couch that has six degrees of freedom (lateral, longitudinal, vertical, roll, pitch and rotation). When the rotation of the couch is at 0 degrees, adjusting the roll of the couch and the angle of the gantry have the same effect, although the couch can only move plus or minus 3 degrees, so it tends to be used for making small adjustments only.

2.1 The Problem: Gantry Creep

In the initial design of the software, the therapist issued the command "gotoSetup" to move the patient into the recorded setup position. She then took X-rays, and adjusted the position of the gantry and couch until alignment was achieved. A single command "saveSetup" was then executed, whose effect was two-fold: to obtain the delta used to offset the treatment position, and to record a new setup position for subsequent treatment sessions.

The therapists observed that sometimes the gantry angle had deviated over the course of several treatment sessions quite considerably from its initial position, despite the fact that the therapist had made no adjustments to the gantry itself. This was not in itself a safety concern, since the unexpected movement of the gantry had been compensated by a corresponding adjustment of the couch. Eventually, however, the gantry had moved so far that it was no longer possible to compensate because of the limited freedom of movement of the couch.

The problem, it turned out, was that the "saveSetup" command would overwrite the gantry angle setting even when it had not been adjusted. Since the "gotoSetup" command only moved the gantry to within the recorded position by some tolerance, the effect of "saveSetup" was to change the gantry angle setting even when the therapist had not intended any change. In some cases, it seems that these small errors accumulated, resulting eventually in a significant change.

The solution that was implemented was simply to eliminate the ability to adjust the gantry angle during setup. The code of the "saveSetup" command was changed accordingly so that it never overwrites the gantry angle setting. This approach is acceptable because the adjustments that are typically needed are small, and can be achieved by adjustments to the couch position alone.

The solution suggested by the analysis based on subproblems is different. It distinguishes those "saveSetup" commands that follow "gotoSetup" commands, and insists that they make no change to the recorded setup position.

3 Decomposing into Subproblems

A decomposition into subproblems starts with an attempt to uncover the purpose behind the functions to be implemented. In this case, a discussion with the developers revealed two distinct purposes: (1) to save the setup position so that in a subsequent session the need for setup adjustment is eliminated or reduced; and (2) to

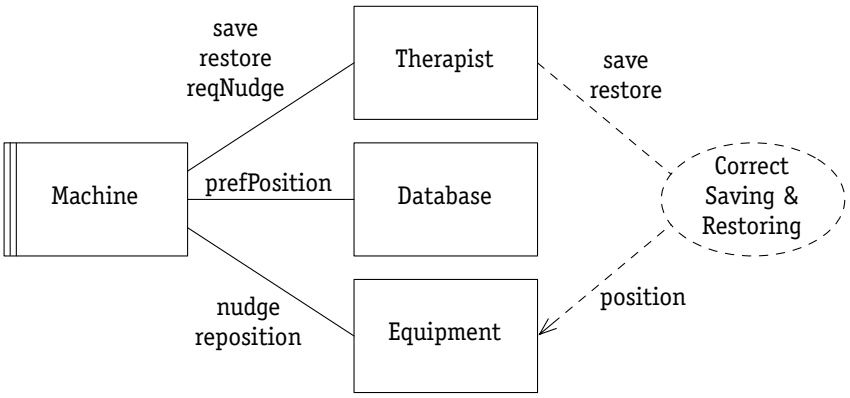

Fig. 1. Set/Restore subproblem

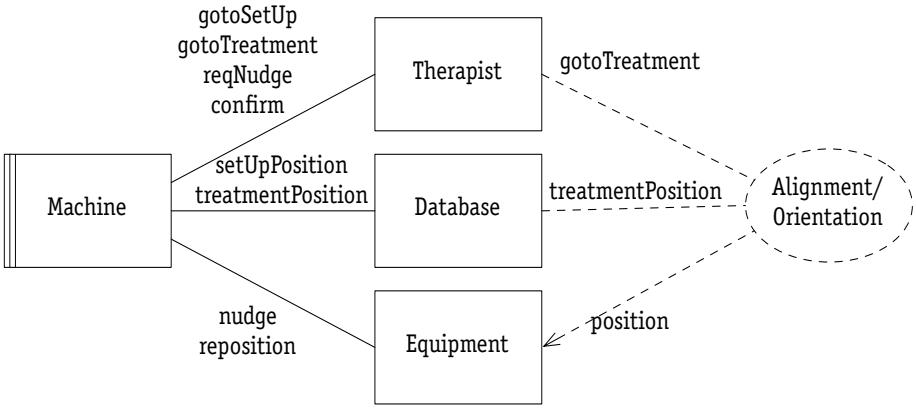

Fig. 2. Alignment subproblem

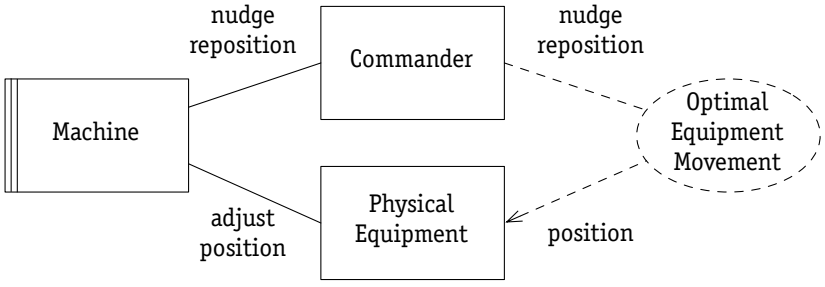

Fig. 3. Positioning subproblem

determine, with the help of an X-ray or some other imaging device, an adjustment to the relative positions of the patient and the beam that will ensure proper alignment during treatment.

An important clue that these purposes should be regarded as distinct subproblems is that they have different *spans*. The first, which we call the *Set/Restore subproblem*, has a span that encompasses multiple treatment sessions. The second, which we call the *Alignment subproblem*, involves only a single session. The two subproblems are shown in figs. 1 and 2 in Problem Frame notation [9]. In each figure the striped rectangle represents the *machine* to be developed for the corresponding subproblem. The other rectangles represent *problem domains* interacting with the machine at interfaces of shared phenomena; the dashed ellipse represents the *requirement*, which is a condition on the problem domains, expressed in terms of phenomena that may or may not be shared with the machine. An arrowhead indicates that the requirement expresses a constraint on the domain to which it points.

Before delving further into either subproblem, we notice that they share a common feature: the positioning of the equipment by the machine. The physical devices that perform this positioning cannot be perfectly controlled; a position is set using a control loop that makes repeated adjustments and measurements. The control loop's design involves tradeoffs between the accuracy of the final position and how quickly it is reached.

Recognizing this, it becomes clear that the positioning of the device in accordance with a desired position is itself a non- trivial, third, subproblem that should be separated from the two subproblems already identified. We shall call this the *Positioning subproblem* (fig. 3). The domains marked Equipment in the subproblems of figs. 1 and 2 now correspond to the PhysicalEquipment domain and Machine of fig. 3. The machines in figs. 1 and 2, issuing nudge and reposition commands to Equipment, correspond to the Commander domain in fig. 3. The domain marked PhysicalEquipment in fig. 3 is less abstract, and represents the actual physical plant and its monitoring and controlling devices. In implementation terms, interactions with the Equipment domain represent indirect interactions with the PhysicalEquipment domain mediated by the Machine in the Positioning subproblem.

Let's now examine each of the three subproblems in more detail.

3.1 Positioning Subproblem

The Positioning subproblem (fig. 3) has a domain Commander that issues two kinds of command: nudge, to request a relative adjustment, and reposition, to request an absolute position. The domain PhysicalEquipment, as mentioned, represents the physical plant and its devices; it is controlled by adjustment commands represented by the operation adjust, and is monitored by the reading of a variable position that is shared with the machine. The requirement is that, after a reposition(x), position is within some epsilon of x, and, after a nudge(d), position is within some epsilon of d applied to the previous value of position.

The details of the tolerance and the time taken to achieve the final position need not concern us here, and are standard issues in the design of a control loop for a physical device (such as a robot arm). A primary benefit of identifying such a sub-

problem is factoring out parts of the development that are complex and tricky when faced for the first time, but are conventional and easily handled by a specialist.

3.2 Set/Restore Subproblem

The Set/Restore subproblem is also an instance of a wider class. The setup protocol in our proton therapy setting is essentially the same as the protocol for adjusting the seat position in a fancy car. The car stores a preferred position for each driver, and has three principal commands: to adjust the seat position; to save a preference; and to restore the position to the last position saved for that driver.

In this subproblem, the domain Equipment has a shared variable position that reveals the current position of the equipment. Unlike the domain PhysicalEquipment in the Positioning subproblem, however, its phenomena include the more powerful commands nudge and reposition rather than just adjust. This subproblem therefore need not be concerned with how a particular positioning command is handled; it assumes that the equipment responds appropriately.

The Therapist issues three kinds of command: save(p) to save the current position as the preferred saved position for patient p, restore(p) to move to the position previously saved for patient p, and reqNudge(d) to request an adjustment by an amount d. The preferred positions are stored in a database represented by the Database domain, which offers a relation prefPosition mapping each patient to a preferred position.

A careful consideration of this subproblem in isolation reveals the creep problem. Since reposition only achieves an approximation to the desired position, issuing the command reposition (position) repeatedly can cause arbitrary changes in position; each request to set the position to the current recorded position may actually result in a change in position. A naive design in which every save(p) writes the current value of position to prefPosition[p] will exhibit this anomaly if a sequence of save/restore pairs is executed.

To avoid the problem, we can make save(p) have no effect if the preceding event was a restore(p). A full formalization of this subproblem is discussed below, with a more detailed explanation of this decision.

3.3 Alignment Subproblem

The Alignment subproblem is the hardest to handle, because it is more complicated, and because it seems to be unique to this domain. It can nevertheless be described fairly succinctly. Rather than representing the gantry and couch as distinct components with distinct positions, we regard the system as a whole as occupying a coordinate in some abstract space, just as we did with other subproblems.

In this space, some coordinates can be classified as *oriented*: these correspond to the gantry and couch positions in which the patient is oriented appropriately for treatment. Some coordinates, likewise, can be regarded as *aligned*: these are the coordinates in which the relative positions of the couch and the gantry will ensure that the beam is appropriately directed at the tumour. By viewing alignment and orientation as projections of a coordinate, we can define planes (isosurfaces) in the

abstract space of coordinates that share a particular alignment or a particular orientation. Correct alignment (or orientation) means that the alignment (or orientation) projection has a particular value.

The Therapist issues four kinds of command: reqNudge to request a position adjustment, gotoSetUp to request the setup position, and gotoTreatment to request that the equipment move to the treatment position stored in the Database, and confirm to confirm that the equipment is well aligned in the current position. The Database holds a setup position and a treatment position for each patient p represented as shared variables setUpPosition[p] (assumed to be almost aligned) and treatmentPosition[p] (assumed to be oriented, and also almost aligned).

The procedure to be followed by the Therapist is first to request the setUpPosition with gotoSetUp; then, if adjustment is necessary to effect it by reqNudge commands; then to issue a confirm followed by gotoTreatment command.

The requirement is roughly that, following gotoTreatment, the equipment is both aligned and oriented. It will be established by a combination of assumed properties of the Therapist, Database and Equipment domains, and of the specification of the Machine, namely that (1) Therapist will issue the confirm command only when the equipment is shown to be aligned by the X-ray or other imaging technique; (2) the value of treatmentPosition[p] in Database is oriented; (3) in the Equipment domain, the command reposition(x), where x is aligned and oriented, results in a value of position that is also aligned and oriented.

4 Set/Restore Formalized

Decomposing into subproblems allows us to analyze each subproblem independently. In this section, we illustrate this by formalizing the Set/Restore subproblem in Alloy [8], and subjecting the formal model to an automatic analysis using the Alloy Analyzer [2].

An Alloy model begins with a module name, and imports for any modules that are used. In this case, we import a library module that imposes a total ordering on the set Event, to be declared later:

```
module saveRestore
open util/ordering [Event]
```

The import makes available functions which will be used later: next(e), nexts(s), prev(e), and prevs(s), which for an element e (or a set s) give respectively the next element, all subsequent elements, the immediately preceding element, and all preceding elements; and first() and last(), which give the first and last events in the ordering.

The set of positions is declared, with a relation near associating each position with the set of positions that are within some epsilon (the tolerance of the Positioning subproblem), along with a fact (a global assumption) that this relation is reflexive and symmetric:

```
sig Position {near: set Position}
```

```
fact {
    Position <: iden in near
    near = ~near
    }
```

It is significant that near is not transitive; its lack of transitivity is the source of the gantry creep problem.

A set of patients is likewise declared:

```
sig Patient {}
```

The states of the system are declared explicitly as a set also; Alloy has no built-in state machine idiom. Two relations are declared on states, one for the state of the Equipment domain that associates each state with a position – the physical position of the equipment – and one for the state of the Database domain that associates each state with a function mapping patients to preferred positions:

```
sig State {
    Equipment_position: Position,
    Database_prefPosition: Patient -> one Position
    }
```

The Database_prefPosition relation is a total function: it maps each patient to exactly one position.

The various requests and commands are modelled as event objects. We start with a set of events declared to be abstract (indicating that it will be exhausted by the subsets that will be subsequently declared), and with relations associating each event with its pre-state (the state before its occurrence), its post-state (the state after its occurrence), and the patient to which the event applies:

```
abstract sig Event {
    pre, post: State,
    patient: Patient
    }
```

The pre- and post-state relations must be constrained so that for any event e except the last event in a trace, the pre- state of e's successor event is the post-state of e:

```
fact {
    all e: Event - last () | next (e).pre = e.post
    }
```

We declare a partition of the event set into subsets corresponding to the three commands issued by the therapist:

```
sig Therapist_save, Therapist_restore, Therapist_reqNudge extends Event {}
```

The Equipment domain has two event sets of its own; the use of the in keyword in their declarations allows these sets to overlap with the other event sets:

```
sig Equipment_reposition in Event {position: Position}
sig Equipment_nudge in Event {}
fact {no Equipment_reposition & Equipment_nudge}
```

Our plan is to have them overlap with the Therapist events, so that a reqNudge in the Therapist domain can be equated to a nudge in the Equipment domain. They will not overlap with each other, however, so an explicit fact is recorded to this effect.

Note that the reposition event has a position relation declared for it; this is in fact the only event in which a position must be made explicit. The commands of the Therapist domain are interpreted with respect to the current position in the Equipment domain, which is not communicated by the therapist.

It will be convenient to have two functions for describing temporal relationships between events. The function following takes an event e and a set of events s and returns either the first event that follows e that belongs to the set s or the empty set if there is none:

```
fun following (e: Event, s: set Event): lone Event {
  let succs = s & nexts (e) | succs - nexts (succs)
  }
```

This defines succs as the intersection of s and the set of all events occurring after e. The difference between succs and the set of all events occurring after any of its members is then the singleton set containing its first member or the empty set if it has no first member. The lone keyword indicates that the function following may return a singleton or empty set of events; it can be read 'less than or equal to one'.

The function between takes two events and returns the set of events that occur between them:

```
fun between (from, to: Event): set Event {
  nexts (from) & prevs (to)
  }
```

Now we can define the constraints: the requirements and the domain properties. There are two distinct requirements. The first says, roughly speaking, that a restore command returns the equipment to the position prior to the last save. More precisely, for any patient p and save command s associated with p, and for any restore command r following s, if there is no other save for p that intervenes between the two, the position after the restore is 'nearish' to the position before the save:

```
pred Memory_Requirement () {
  all p: Patient, s: Therapist_save & patient.p |
    all r: following (s, Therapist_restore &
    patient.p) |
      no Therapist_save & patient.p & between (s, r) implies
        nearish (r.post.Equipment_position, s.pre.Equipment_position)
  }
```

The expression Therapist_save & patient.p denotes the set of Therapist_save events applying to patient p, and so on.

Two positions are 'nearish' if there is some position they are both near to:

```
pred nearish (p, p': Position) {some p'': Position | p+p' in p''.near}
```

(The need for this notion is explained below). The second requirements says, roughly speaking, that there is no creep. For any patient p, save command s associated with p, and restore commands r and r', also associated with p, that follow s without an intervening reqNudge command, the positions resulting from r and r' are nearish:

```
pred Consistency_Requirement () {
   all p: Patient, s: Therapist_save & patient.p, r: nexts (s), r': nexts (r)  |
      (r + r' in Therapist_restore & patient.p and
      no between (r, r') & Therapist_reqNudge & patient.p) implies
         nearish (r.post.Equipment_position, r'.post.Equipment_position)
   }
```

The two restore commands need not follow immediately, and can have other restore commands occurring between them.

The therapist positions each patient afresh, rather than using the position of the previous patient. We record this assumption as a predicate saying that if an event is associated with a different patient than its predecessor, it must be a restore command:

```
pred Therapist () {
   all e: Event | e.patient != prev(e).patient implies e in Therapist_restore
   }
```

(Note that if e has no predecessor, then the expression prev(e).patient denotes the empty set: there are no undefined expressions or special values in Alloy.)

The specification of the machine links together the commands of the therapist with the reading and updating of the database, and the issuing of commands to the equipment:

```
pred Specification () {
   -- respond to a restore command from the therapist by issuing
   -- a reposition command to the equipment whose position argument
   -- is that position of this patient in the database
   all r: Therapist_restore |
      r in Equipment_reposition and
         r.position = r.pre.Database_prefPosition[r.patient]
   -- a reqNudge command from the therapist is matched to a nudge
   --  command to the equipment and a restore is matched to a reposition
   Therapist_reqNudge = Equipment_nudge
   Therapist_restore = Equipment_reposition
   -- when a save command is received from the therapist, the position of
   -- the associated patient is updated in the database with the current
```

```
    -- equipment position, unless the previous command was a restore
    -- for this patient
    all s: Therapist_save | let p = s.patient |
      s.post.Database_prefPosition = s.pre.Database_prefPosition ++
        if some prev (s) & Therapist_restore & patient.p
          then none -> none else p -> s.pre.Equipment_position
    -- for any event except a save, the database is not written
    all e: Event - Therapist_save |
      e.pre.Database_prefPosition = e.post.Database_prefPosition
    }
```

The assumptions about the equipment are that a reposition moves the equipment to a position near to the position requested, and that only reposition and nudge events result in a change in position:

```
pred Equipment () {
  all r: Equipment_reposition | r.post.Equipment_position in r.position.near
  all e: Event | e.post.Equipment_position = e.pre.Equipment_position
    or e in Equipment_reposition + Equipment_nudge
  }
```

Finally, we can declare as assertions the key correctness properties, namely that the combination of the specification and domain properties implies each of the requirements:

```
assert CorrectnessM {
  Specification () and Equipment () and Therapist ()
    implies Memory_Requirement ()
  }
assert CorrectnessC {
  Specification () and Equipment () and Therapist ()
    implies Consistency_Requirement ()
  }
```

The Alloy language is undecidable, so an assertion cannot be checked automatically in an unbounded space. So Alloy's checking commands specify a *scope* indicating how many elements each set may have. For example, for an initial analysis, we might execute the command

```
check CorrectnessM for 3
```

which checks the assertion CorrectnessM for all scenarios involving up to 3 events, states, positions and patients. Because there are so many scenarios even within small scopes, they are often sufficient to detect interesting flaws.

For example, if the definition of nearish is replaced by

```
pred nearish (p, p': Position) {p in p'.near}
```

so that two points are nearish only when they are near ŧhe Alloy Analyzer finds a counterexample for this command in about 5 seconds (on a 1.67GHz Powerbook

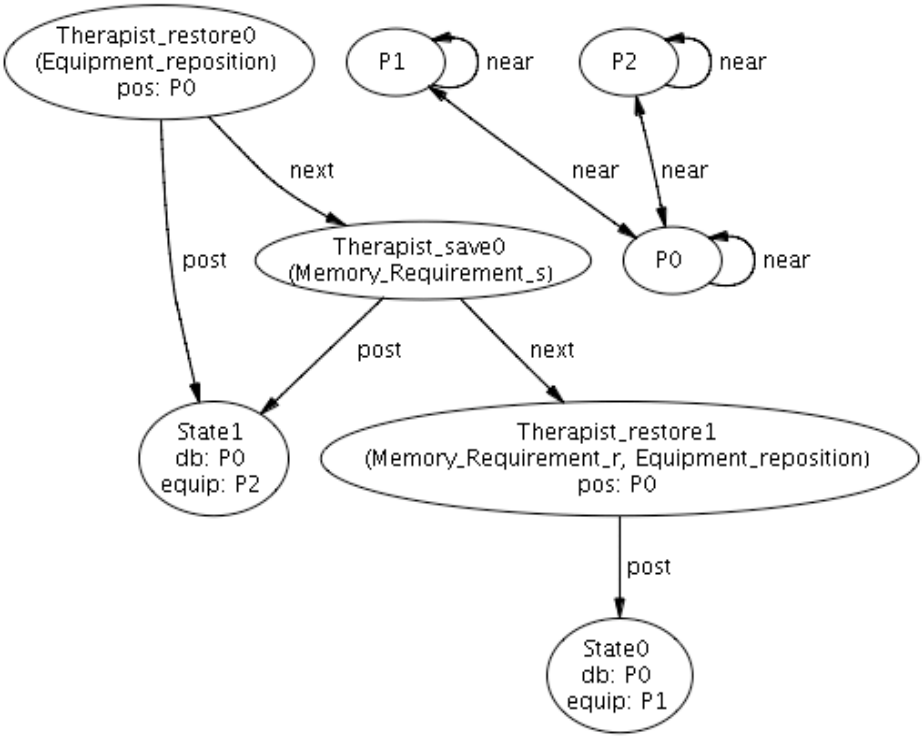

Fig. 4. A counterexample

G4 laptop), as shown in fig. 4. The large ovals linked by next show the chain of events for a particular patient. The first event is a restore; its pre-state, not shown in this particular visualization, associates position P0 with the patient in the database, so P0 is the argument to the reposition command. On receiving this command, the equipment is free to set the position to any that is near P0; it chooses P2. (The near relation amongst positions is shown in the upper right.) Now a save command occurs, which has no effect, since it is preceded by a restore. Then a second restore is performed. The database has not changed, but this time, the equipment chooses a different position near P0, namely P1. So although creep can't happen, since different restore commands can approximate the commanded position differently, the actual error margin is twice the tolerance of the equipment.

Replacing the definition of nearish by its original definition results in no counterexample. To gain further confidence, we can increase the scope. Checking the command for all scenarios involving 7 events, 7 states, 7 positions and 3 patients

```
check CorrectnessM for 7 but 3 Patient
```

gives no counterexamples, in a search that takes under 4 minutes.

5 Discussion

Separating concerns. By separating the problem into three subproblems, we were able to see more clearly what the essential difficulties were. The creep problem, for example, is a direct consequence of the interface presented by the Positioning subproblem to the Set/Restore subproblem, and can be solved by ensuring that saves that do not follow nudges have no effect. In the original requirements document, the description of the setup procedure involves reading both setup and treatment positions from a database, and using both to compute the final treatment position. Examination of the Alignment subproblem reveals that this need only depend on the treatment position given in the database and the alignment information obtained from any reasonable setup position. The conflation occurs only because the implemented database incorporates the databases of both subproblems, and because the setup position used to obtain alignment is the same setup position that is saved and restored.

Formal analysis. The decomposition into subproblems simplifies the formal analysis, not only in allowing smaller models, but also by making them more tractable and the results easier to interpret. Simply writing things down more formally reveals misunderstandings; mechanical analysis inevitably reveals additional, more subtle problems. Our experience formalizing the Set/Restore subproblem was typical in this respect.

Span. The span of a subproblem is the set of phenomena it involves. In this case, the span of a subproblem might involve one or many patients, and one or many treatment sessions. Identifying the span is a crucial first step in understand a problem, and the presence of requirements with different spans suggests a decomposition into subproblems.

Abstraction. A subproblem is easier to understand and analyze when the phenomena have been abstracted appropriately. In the original requirements document, for example, the discussion of positioning involves the many components of the gantry and patient couch position. This level of detail is not relevant to these subproblems.

Distinct phenomena. A scenario-based analysis encourages the developer to conflate phenomena, for example to assume that the saving of a preferred setup position and the confirmation of alignment are the same event. They happen to be performed by the same person at the same time, often for the same position, but there is no fundamental reason that they need to be equated. Arguably, a cleaner design would offer two separate commands, allowing the therapist to save a preferred position without confirming alignment, for example. In short, it is better to start with the assumption that phenomena are distinct and merge them than to start with a smaller set and try to split phenomena later.

Composing the Positioning subproblem. Analysis of a problem into distinct subproblems must be followed by recombination of the analysed subproblems to give a solution to the original problem. Recombining the Positioning subproblem with the other two is straightforward and entirely conventional. The span of the Positioning

subproblem is receipt and execution of a single nudge or reposition command: the Machine in the Positioning subproblem has no need to save state from one command to the next, because the only significant state is held in the Physical Equipment. This Machine can therefore be easily implemented as a module that interfaces with the equipment on one side and offers the nudge and reposition commands on the other. This module is made available to the Set/Restore and Alignment subproblems.

Composing the Set/Restore and Alignment subproblems. Recombining the Set/Restore and Alignment subproblems demands more care. The composition task is to combine the subproblems by identifying phenomena that are common to both, and to ensure that the composition preserves the properties of each. The Therapist has, in principle, the full repertoire of both subproblems available, but each subproblem imposes its own restrictions on the acceptable command sequences. As has already been mentioned, it is appropriate to identify the Database field prefPosition in the Set/Restore subproblem with the setUpPosition field in the Alignment subproblem. The reqNudge commands in the two subproblems are evidently identical. The confirm command in the Alignment subproblem can be identified as a save command in the Set/Restore subproblem: responsibility for avoiding the creep problem belongs to the Set/Restore subproblem, where save will have no effect unless there has been a reqNudge since the most recent save. The databases are composed simply by merging their schemas. The two subproblem machines can be combined, in an object-oriented setting, by introducing a control layer that delegates commands issued by the Therapist to lower-level objects implementing the two machines.

Without hindsight? We have, of course, had the benefit of hindsight. The gantry creep problem had already been identified in the existing system, and we took that as our starting point. Would we have identified the problem if we had been doing an original design without the benefit of hindsight? We believe that we would. By our criteria the *Set/Restore* subproblem is clearly distinct from the *Alignment* subproblem, because the two have different spans: many sessions versus one session. Once these subproblems have been separated there is no reason to confuse the save action in the *Set/Restore subproblem* with the confirm action in the *Alignment subproblem*.

The confirm action need not, in principle, cause a database update, because the confirmed position will be used immediately to compute the delta and the treatment position. Only a later recognition that it, too, could involve saving a position in the database suggests the possibility that the two saved positions might be represented by the same database field in the patient's record. Such a design choice, in our approach, would be a conscious decision in an explicit composition task, and would demand careful examination of the circumstances in which the two actions could share a part of their implementation.

6 Related Work

Dijkstra coined the term 'separation of concerns'. In an early note [4], he advocated the idea of focusing on one aspect of a problem at a time. Since then, the notion of

'separating concerns' has become standard, although often only lip service is paid to it.

The insight that the requirements of a system to be built should be viewed as a collection of fairly independent subproblems is now also widely understood, although in practice the identification of subproblems is not made explicit in the requirements document, but arises only during design, when the subproblems emerge as design challenges. The idea that the requirements themselves should be structured around subproblems is the premise of the Problem Frames approach [9], which characterizes problems into archetypal classes, in the hope that most subproblems encountered will be instances of subproblems that have already been faced, and for which simple and effective solutions are well known.

Formal methods attempt to uncover the essence of the requirements problem, and to express it precisely and unambiguously in a formal notation. They do not tend, however, to give effective guidance or heuristics for decomposing problems into subproblems, although the presence of conjunction in declarative specification languages makes them well suited to such a decomposition [1, 7, 10].

Viewpoints [5] are a bit like subproblems, but they arise from the interests of different stakeholders, rather than from structure inherent to the problem itself.

Aspect-oriented programming [11] and subject-oriented programming [6] aim to achieve better separation of concerns by new implementation constructs. Work on 'early aspects' seems to focus not so much on separation of concerns in the early phases of development as on the early identification of features that can be implemented using the technology of aspect-oriented programming.

Failure to recognize that a problem is composed of multiple subproblems is likely to result in complicated and obscure implementation. An extreme programming approach [3] may well exacerbate the difficulties, by encouraging the coding of a complex composite machine before simpler submachines have been identified. The effort invested in an early decomposition into subproblems is likely to pay off, and an extreme programming approach in which individual submachines are implemented and evaluated prior to consideration of their composition might work well.

Acknowledgments

Dr. Jay Flanz, director of the Burr Proton Therapy Center, generously explained to us the details of the gantry creep problem; Dr. Hanne Kooy, radiation physicist, and Doug Miller and Nghia Ho Van, developers of the Therapy Control System were also very helpful. Robert Seater is developing a problem-frame- based analysis of the system, and shared his ideas and insights with us. This research was funded in part by grant 0325283 (Safety Mechanisms for Medical Software) from the ITR program of the National Science Foundation.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the National Science Foundation, or the Burr Proton Therapy Center.

References

1. M. Ainsworth, A.H. Cruickshank, L.J. Groves and P.J.L. Wallis. Formal Specification via Viewpoints. *Proc. 13th New Zealand Computer Conference*, New Zealand Computer Society, Auckland, New Zealand, 1993.
2. *The Alloy Language and Analyzer*, http://alloy.mit.edu.
3. Kent Beck. *Extreme Programming Explained*. Boston, Addison Wesley, 1999.
4. Edsger Dijkstra. On the role of scientific thought. EWD 447, 30th August 1974, Neuen, The Netherlands. Appears in: Edsger W. Dijkstra, *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982. ISBN 0–387–90652–5, pp. 60–66. Available at http: www.cs.utexas.edu/users/EWD/.
5. A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein and M. Goedicke. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal on Software Engineering and Knowledge Engineering*, 2(1):31–57, World Scientific Publishing Company, March 1992.
6. William Harrison and Harold Ossher. Subject-Oriented Programming – A Critique of Pure Objects. *Proc. 1993 Conference on Object- Oriented Programming Systems, Languages and Applications*, September 1993.
7. Daniel Jackson. Structuring Z Specifications with Views. *ACM Transactions on Software Engineering and Methodology*, Vol. 4, No. 4, October 1995, pp. 365–389.
8. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge, MA, March 2006.
9. Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Boston, Addison Wesley Professional, 2000.
10. Daniel Jackson and Michael Jackson. Problem Decomposition for Reuse. *Software Engineering Journal*, Vol. 11, No. 1, January 1996, pp. 19–30.
11. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin. Aspect- Oriented Programming. *Proc. European Conference on Object- Oriented Programming*, 1997.
12. Andrew Rae, Daniel Jackson, Prasad Ramanan, Jay Flanz and Didier Leyman. Critical Feature Analysis of a Radiotherapy Machine. *Reliability Engineering and System Safety*, Elsevier Science, 2004.