Some Principles and Ideas of PFA: the Problem Frames Approach

(Draft of 30th January 2009)

Michael Jackson, The Open University
jacksonma@acm.org

Abstract The problem frames approach to software engineering is primarily—but not exclusively—concerned with computer-based systems. The broad content and nature of the approach are explained, and the underlying ideas and principles are reviewed and discussed. Some familiarity with the approach is assumed, and much detail is omitted. The paper does not aim to offer a tutorial: it presents only enough of the approach to illustrate the ideas.

1. INTRODUCTION

The goal of the problem frames approach—PFA—is to contribute to improving the dependability of computer-based systems. It is concerned in particular with designing the behaviour of a system to satisfy functional requirements—that is, those that concern the observable behaviour of the system in operation. PFA is not concerned with managerial problems, although its adoption could impinge on the technical content—if any—of any chosen managerial style; nor does it directly address such important tasks as eliciting system requirements from stakeholders or negotiating compromises where requirements are in conflict.

1.1 Computer-Based Systems

In a computer-based system the computer—or some assemblage of computers—interacts with some other parts of the physical and human world in order to produce some desired effects there. In this sense, a heart pacemaker is a computer-based system, and so are a telephone switch and a system to administer a lending library: but a program to factorise large integers is not, nor is a program to find cycles in a graph, because they are concerned purely with mathematical abstractions and interact only trivially, if at all, with the human and physical world. Because PFA regards software development as a problem to be solved, the relevant parts of the world outside the computer are referred to as the *problem world*.

For a realistic computer-based system the problem world is likely to comprise many heterogeneous parts or *problem domains*. These may include: human beings (for example, as operators and users, or as the subject of information that the system must use or produce); inert physical structures (for example, the track layout of a railway); mechanical devices (for example, lift cars, doors, and winding gear in a lift control system); actuators and sensors (for example, motor relays); and concrete realisations of lexical structures (for example, credit cards).

The functionality of the system is likely to support many features, and many local and global modes of operation: for example, a modern phone may combine the functions of a phone, an address book, a diary, an alarm clock, a camera, a GPS device, a web browser and an email client. The interactions among these features are likely to give rise to large complexity in the whole system, and to impinge on every aspect of the development task. The customer or end-user *requirements* of different features may conflict, or they may interact in a way that makes them hard to consider separately and even harder to understand in combination.

The versatility and power of the computer encourages this proliferation of system features. It also encourages complexity in the functionality of the individual features. A central-locking system for a family car must deal with four doors, each with an exterior handle, an interior handle and a button, one or two of the doors also having an external key, a tailgate with an external key and an internal release lever, buttons for locking all four doors or the two rear passenger doors, and an interlock with the ignition control. This environment has a large number of states and transitions, and this complexity must be related to several requirements—child safety, guarding against carjackers, preventing locking of the car while the key is inside, convenience of use in shopping, preventing theft of the car or its contents, and accessibility in the event of a crash.

The computer in a computer-based system is installed in the problem world, and interacts with neighbouring domains through a narrow interface of shared phenomena: for example, through control lines by which the computer can set the state of a motor or detect whether a sensor is on or off, or at data ports through which it can read the data encoded on a swipe card or entered on a keyboard. The scope of the system requirements is not restricted to these neighbouring domains, but typically stipulates desired properties and behaviours for distant domains that interact with the computer only indirectly, through each other and through the computer's neighbouring domains.

For all but the most critical systems the computer itself may be effectively regarded as a formal system, faithfully exhibiting the behaviour described by the software it executes and interacting with the world through its narrow—and effectively formal—predefined interface. But the problem world, unlike the computer, is not a formal system: it can readily falsify any formal assumption about its properties and behaviour. There is, therefore, a mismatch between the formal behaviour described by the software and the effects it can evoke in the non-formal problem world. Many of the system failures, large and small, regularly reported among the copious material published in the Risks Digest [Risks08] and elsewhere, are attributable to this mismatch.

1.2 The Nature of the Problem Frames Approach

PFA aims to address some of the challenges posed by the development of computer-based systems. It is not yet another development method or process. It rests on the presumption that the system to be developed is a computer-based system; but beyond that it prescribes no software or system architecture, object-oriented or otherwise. It can be regarded as an intellectual structure within which elements of appropriate development techniques—for example, notations and calculi, modelling languages, and repertoires of program transformations—can be applied to different parts and facets of the problem in hand. It is not itself a method, because it neither provides nor mandates any particular language or process. It offers no particular notations for describing problems or their solutions, beyond its very simple diagrammatic notation for representing the principal parts of a problem and their relationships, and some accompanying annotations of the diagrams. It is above all a structure of principles and ideas for thinking about problems and solutions in software engineering: for helping to master complexity, for directing the developer's attention to concerns that should not be neglected, and for ensuring clarity in the application of descriptive and analytical techniques, whether formal or informal. Its central goal is human understanding: its hope is that increased understanding will prove a powerful tool for increased dependability of the developed system.

The purpose of this paper is to present and motivate the most basic concepts of PFA. The paper is not a tutorial, and it is not in any way comprehensive. In particular, for brevity it includes no detailed treatment of interface phenomena in problem diagrams. Section 2 provides and illustrates a very broad outline of PFA, explaining and illustrating its underlying

principles and motivations. A more detailed discussion of some of the ideas and practices that realise or support these principles is given in Section 3. The paper ends with some general reflections on the role of PFA in the context of a system development.

2. PRINCIPLES AND MOTIVATIONS

The immediate focus of the problem frames approach, as its name suggests, is on software development as a *problem* to be solved. The PFA notion of a problem was originally inspired by Polya's exposition [Polya57] of the work of Pappus and other ancient mathematicians on problems in what today—thanks to their work—we can regard as elementary mathematics. They classified problems in two classes: problems *to find*, and problems *to prove*. An example of a problem to find is: "Find a prime number strictly between 1,000,000 and 1,100,000." An example of a problem to prove is: "Prove that the sum of the angles of a triangle is equal to two right angles."

The key idea in the classification is that problems of different classes have different *principal parts*, and invite different solution techniques specifically adapted to those parts. A problem to find has an *unknown*, a *given* and a *condition* relating them. In the example given earlier: the required prime is the unknown; the given is the specified range; and the condition is that the unknown lies in the range. Polya offers many heuristics for solving such problems—for example: "Could you change the unknown, the given, or both if necessary, so that the new unknown and the new data are nearer to each other?" The heuristics, necessarily, are expressed in terms of the principal parts of the problem. The broad theme of the work is that a firmer grip on a problem structure and elements provides more specific—and therefore stronger—intellectual tools for its solution.

2.1 Software Development as a Problem

Polya's understanding of problems can be applied—in spirit, at least—to problems of developing computer-based systems. These problems can be characterised—with some adjustment—as problems to find. The unknown is the software to be developed, for execution by computer; the given is the problem world; and the condition is the functional requirement that the whole system must satisfy. Figure 1 is a *problem diagram*, showing these principal parts for a problem of controlling traffic at a road junction:

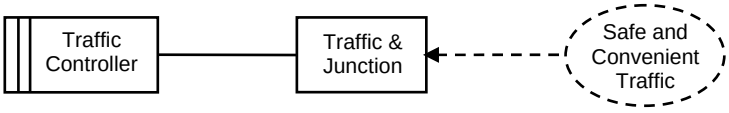

Figure 1: A Problem: Controlling Traffic at a Road Junction

The diagram shows the principal parts of the problem. They are:

- The *problem world*, represented by the plain rectangle. This is the portion of the human and physical world with which the system is concerned. In this problem the problem world has been named *Traffic & Junction*.
- The *requirement*, represented by the dashed ellipse. The requirement is a condition on the problem world. In this problem the requirement has been named *Safe and Convenient Traffic*. Establishing and maintaining that condition is the purpose of the system. The dashed line shows what the requirement refers to. The arrowhead shows that the requirement does not merely refer to the problem world, but also constrains it: for safe and convenient traffic we must constrain the movement of the vehicles using the junction.

- The *machine*, represented by the doubly striped rectangle. This is the computer, executing the software to be developed. In this problem the machine has been named *Traffic Controller*.

The solution task is to find the machine: that is, to develop the software that will be executed. The idea of a machine is quite general—even abstract—here. For example, the eventual implementation may use several computers, one computer, or only a part of a computer shared with another system. These possibilities do not affect the representation of the machine in the problem diagram: it always appears as a single box. The machine interacts with the problem world at the interface of *shared phenomena* represented by the solid line joining them. For example, a shared phenomenon might be an event in which the machine sets one of its output lines high, the line being shared with a traffic light unit for which the high state of the line means 'red light on'.

Identifying the requirement with the condition of a problem to find demands some conceptual licence. This is unsurprising, given that mathematical problems are about an abstract world without time or causality, while computer-based systems are about a changing physical problem world. In particular, the condition is not about a relationship between the unknown and the given. It is a condition that the unknown—the machine—must impose on the given—the problem world.

In PFA the problem world plays the central role in the problem for two reasons. First, because the whole purpose of the system is located there, and it is there that the success of the software development will be judged. The requirement, expressed entirely in terms of the problem world, is not restricted to interaction with the machine: on the contrary, it is usually concerned with phenomena located deeper into the problem world, sometimes far from the machine. In this problem the requirement is entirely about vehicles and pedestrians. The machine is only an instrumental means of affecting the problem world—for example, by setting the traffic lighs. Second, because the machine can ensure satisfaction of the requirement only by respecting and exploiting causal properties which the problem world exhibits independently of any possible behaviour of the machine—for example, the propensity of car drivers to stop at a red light. These properties, therefore, play a vital role in the system, providing the necessary causal links that can allow the machine indirectly to monitor and control phenomena to which it is not directly connected.

2.2 The Problem World as a Given

In the perspective adopted in this paper, the problem world is understood to be *given* in the sense that the developer is not free to replace or modify the problem world, or any part of it, by introducing a new or substitute domain that will make the problem easier. For example, in the traffic control system, the developer is not free to introduce vehicle-detecting sensors if they are not already a part of the given problem world. Of course, in practice the software developer might well make such a suggestion, and it might well be adopted by the engineer responsible for the electromechanical equipment at the road junction. The result, from the standard problem frames perspective, would be regarded as a substantially modified problem, not as a step, taken strictly within the scope of software engineering, towards solving the original problem.

This assumption of a fixed problem world should not be misunderstood as an attempt to restrict the choices available to the software developer over a whole career or even over a whole development project. It is, rather, an example of the style of separation of concerns that permeates PFA: the assumption is purely local, adopted for purposes of the problem in hand. Like the problem world properties, the requirement is also assumed to be given, and to bound the developer's concerns accordingly.

If we describe the given properties of the problem world in a description $\mathcal{W}$, and the requirement in a description $\mathcal{R}$, then we can say that the problem is to find a machine whose properties, described in a description $\mathcal{M}$, are such that the developer can argue as follows:

> "The machine behaves as described by description $\mathcal{M}$. The problem world has the properties described by description $\mathcal{W}$. When they act together through their interface of shared phenomena, the requirement, described by description $\mathcal{R}$ in terms of problem world phenomena, will be satisfied."

We are assuming, then, that the problem world $\mathcal{W}$ and the requirement $\mathcal{R}$ are given, and that the problem is to devise the machine $\mathcal{M}$. In the present paper, we will assume this perspective throughout.

However, at least two other perspectives are possible and may sometimes be useful:

- If $\mathcal{M}$ and $\mathcal{W}$ are given the question may be: What will be the effect of the interaction between them? That is, what requirement $\mathcal{R}$'—if any—will they satisfy? If it is different from the proposed requirement $\mathcal{R}$, is it an acceptably close substitute? This question arises, for example, when a software system ($\mathcal{M}$) that has been successfully installed in one branch of an organisation is proposed for installation in another branch having different properties ($\mathcal{W}$).
- If $\mathcal{M}$ and $\mathcal{R}$ are given the question may be: In what problem worlds $\mathcal{W}$ will $\mathcal{M}$ achieve satisfaction of $\mathcal{R}$? More specifically: Will $\mathcal{M}$ achieve satisfaction of $\mathcal{R}$ in the particular problem world $\mathcal{W}$ of interest, and, if not, is there a modified problem world $\mathcal{W}'$ in which $\mathcal{R}$ will be satisfied? This question arises when a COTS package ($\mathcal{M}$) is considered for use as a system component, and can achieve its desired purpose ($\mathcal{R}$) only if the context is modified ($\mathcal{W}'$) to accommodate it.

2.3 Bounding the Problem World

The choice of names for the problem world, requirement and machine in the traffic problem is suggestive but no more than that. It is necessary, at the earliest stage, to be more explicit about the structure and content of the problem world. Figure 2 shows a fuller version of the traffic control problem diagram.

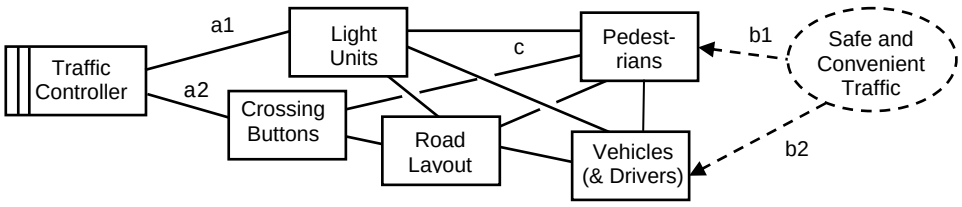

Figure 2: Controlling Traffic at a Road Junction: the Problem World

The problem world is structured as a collection of *problem domains*, interacting with each other and with the machine at interfaces of shared phenomena. The domains have been named to convey a good idea of what they are. The Road Layout domain must be included: the control of traffic cannot be achieved without knowledge of where the Light Units and Crossing Buttons are located. The interfaces also can be named: in a fully detailed problem diagram each interface would be named, and an annotation provided to identify the shared phenomena in each interface. For example, interface *c* might be annotated

c: Pedestrians! {pressButton(j)}

specifying that the shared phenomena are pressButton events controlled by the Pedestrians. For brevity we will say little about the interfaces in this paper. This omission should not be

taken to suggest that they are unimportant: paying proper attention to the phenomena of the problem world is a fundamental tenet of PFA.

A problem diagram with named problem domains and annotated interfaces bounds the problem world for the developer. What is relevant to the problem—and only what is relevant—is included. From the diagram in Figure 2, for example, it can be inferred that:

- there is to be no provision for manual override of the regime imposed by the Traffic Controller: the diagram shows no problem domain that could command an override, and no interface at which the machine could detect such a command; and
- the machine will not take account of varying weather conditions: again, there is no problem domain that can embody the weather, and no interface at which the machine could monitor it; and
- the requirement takes account of pedestrians, and the Traffic Controller machine must accordingly take account of their crossing requests conveyed in pressButton events.

2.4 Understanding and Describing the Problem World

Because all the problem domains shown are relevant to the problem, they will all demand investigation—and, eventually, description—of their relevant properties: each has its own $\mathcal{W}$. For example:

- if the road layout is more complex and irregular than a simple rectangular crossing of two routes, it must be described by some kind of topographical map. The map must show the roads and pedestrian crossings with the positions of all lights and buttons, and dimensions from which the developer can estimate traversal times for vehicles and pedestrians and determine the space available for vehicles waiting on intermediate road segments.
- The light units may be simple reactive devices, changing state by illuminating those lamps that correspond to the control lines currently set *high* by the Traffic Controller machine: they can then be described by a mapping from control line states to visible light states.
- The pedestrians, being human participants in the system, can be expected to behave according to instructions, but not reliably so. There is therefore a stochastic element in their behaviour, demanding estimates of the probability that a pedestrian will start to cross at various times after the ‘walk’ light has been superseded by the ‘wait’ light. There are also physical and other constraints on the speed with which a pedestrians can traverse a crossing of a given size (in [Goh04] this is related to the individual pedestrian’s age, sex and trip purpose, and to the density of pedestrian traffic crossing in each direction).
- The vehicles, with their drivers, can be regarded similarly to the pedestrians. They can be expected to obey the traffic lights, but their obedience is not reliable, and may vary according to their position on the road: for example, a bend in the road approaching the crossing from one particular direction may make approaching drivers slower to see and respond to an amber light.
- The crossing buttons may be simple devices whose state—*pressed* or *not pressed*—is shared with the Traffic Controller machine.

To investigate and describe the problem world adequately, the developer must maintain a resolute focus on the real physical problem world domains and their phenomena. The machine as eventually designed and implemented may—or may not—include software objects corresponding to some or all of these domains; but software objects are not the subject of the description here.

2.5 Decomposing the Problem—1

When a problem is complex—and realistic systems are complex—it must be decomposed into *simpler subproblems*. Both words are important. The decomposed parts must be simpler than the complex whole: otherwise the task may have been made harder, not easier, by the decomposition. They must also retain the form of problems: otherwise the decomposition will have too much of the flavour of solving, rather than structuring and analysing, the original complex problem.

Simplicity in a subproblem has many facets. □□□ ideas of simplicity in an individual subproblem or problem are discussed and illustrated in Section 3 of this paper. □□□ also adopts a radical principle of simplicity in decomposition: the eventual recombination of the decomposed parts should be ignored in their initial definition and analysis. By separating the analysis of each part from the analysis of its composition with other parts the developer can see its essence more clearly; and analysing the composition task itself becomes easier when the parts to be combined are already understood.

The problem shown in Figure 3, derived from a problem discussed in [Jackson01, Hayes03], is small but rich enough to provide a simple illustration of decomposition.

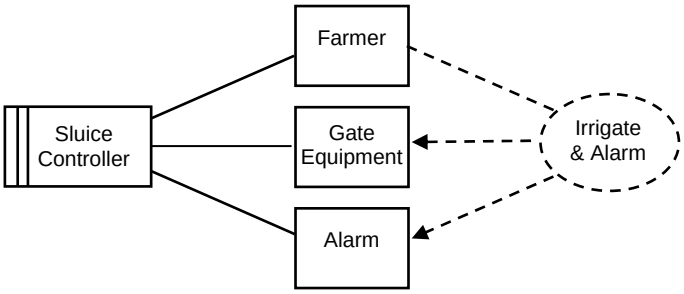

Figure 3: Controlling an Unreliable Sluice Gate

The problem is to control the raising and lowering of a sluice gate in a farm irrigation system according to a schedule specified from time to time by the farmer. The operation of the mechanical and electrical equipment of the gate is unreliable, because it is subject to deterioration or failure due to rust and other factors, and because debris from the irrigation channel may clog the mechanism, damage the sensors, or obstruct the gate's downward travel. The requirement includes the stipulation that when the equipment is faulty, or its operation is obstructed, the machine must sound an alarm to alert the farm engineer to the situation.

The problem frames approach strongly suggests a decomposition into two subproblems: one to raise and lower the gate to satisfy the irrigation requirement as specified by the farmer, and one to monitor the behaviour of the gate equipment and sound the alarm if necessary. The two subproblems are shown in Figure 4.

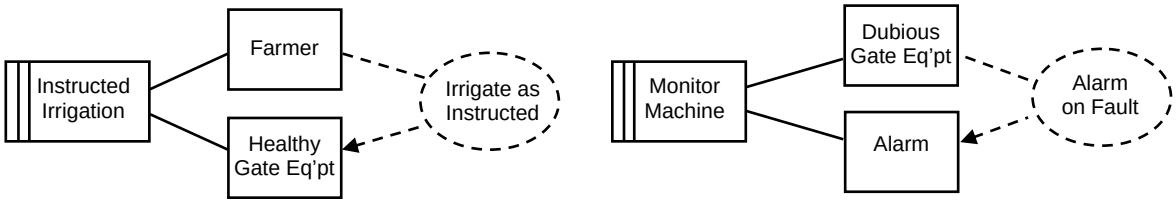

Figure 4: Controlling a Sluice Gate: Decomposition into Two Subproblems

Each subproblem is a well-formed problem. Each has one machine and one requirement, and a problem world. In the subproblem shown on the left the only requirement is to operate the gate in accordance with the farmer's specified irrigation schedule. The requirement refers to

the farmer's instructions, and to the states *open* and *closed* of the gate. It stipulates appropriate openings and closings of the gate: this part of the requirement can be satisfied only if the equipment is functioning correctly. The motor must respond to *up*, *down*, *on* and *off* from the Irrigation Machine, raising and lowering the gate; and the sensors that indicate when the gate reaches its upper and lower travel limits must go *on* and *off* as they are designed to, allowing the machine to stop the motor at the desired point. In this subproblem, this correct functioning is *assumed* as the given property of the Healthy Gate Equipment domain. That is: in the development of the Irrigation Machine no attention is paid to the possibility of equipment faults.

In the subproblem shown on the right, the requirement is not to control the gate, but only to detect possible faults in the equipment and to sound the alarm if a fault is detected. This requirement refers to the Alarm, and it defines *faulty* and *healthy* states of the gate. In this subproblem the Dubious Gate Equipment is regarded as autonomous: the motor states *upward*, *downward*, *on* and *off* are assumed to change spontaneously in the domain. The significant given properties of the domain assumed here are those that allow faults to be detected—that is, they relate the *faulty* and *healthy* states to the motor and sensor states shared with the Monitor Machine. For example, if the motor state has been *on* and *upward* for more than some specified time, and the upper sensor is not *on*, then the Dubious Gate Equipment is in a *faulty* state; similarly, there is a fault if both upper and lower sensor are *on* simultaneously; and so on. If the alarm domain allows for the transmission of a message along with the alarm signal, the requirement may also stipulate some diagnosis of the fault. This would necessitate a more detailed investigation and description of the possible faulty states and their symptoms.

In this decomposition, as in all PFA problem decompositions, each subproblem is fully independent of the others. The developer considering one subproblem is not merely permitted to ignore the other problem, but is positively enjoined to ignore it. The left subproblem is not concerned with possible faults; the right subproblem is not concerned with the irrigation schedule. Nor, in principle, is any consideration given to the eventual need to combine the solved subproblems—for example, to ensure that when a fault has been detected the Irrigation Machine does not continue to make fruitless attempts to control the equipment. In the eyes of the developer considering it, each subproblem is located in an independent universe of its own.

2.6 Decomposing the Problem—2

The decomposition described in the preceding section can be seen as a decomposition of the requirement: the functional requirement falls naturally into two parts. Another, different, motivation for decomposition can be seen, at least partly, as a decomposition of the undecomposed machine: essentially, the decomposition exposes a local variable of the machine, assigning its writing and reading to separate subproblems. Treating this decomposition as a tool for problem analysis is justified when the local variable is a complex data structure and the relationship between the writer and the reader subproblems can be more easily understood by making the variable visible as a part of the problem.

The two subproblems shown in Figure 4 provide two simple examples of this form of decomposition. The first concerns the farmer's specification of the irrigation schedule, and is discussed in this subsection. The second introduces a *model domain*—a notion of enough general importance to justify separate discussion in the following subsection.

The first example decomposes the problem shown on the left of Figure 4. The problem is to operate the Sluice Gate in accordance with the farmer's instructions. If these instructions are no more than the commands *start_irrigating* and *stop_irrigating*, there is no motive for

decomposition. If, at the other extreme, the farmer's instructions can specify an elaborate schedule hour by hour and day by day, there is good reason to decompose the problem.

The appropriate decomposition separates the subproblem in which the farmer specifies the desired schedule from the subproblem in which the gate is operated accordingly. Figure 5 shows the decomposition.

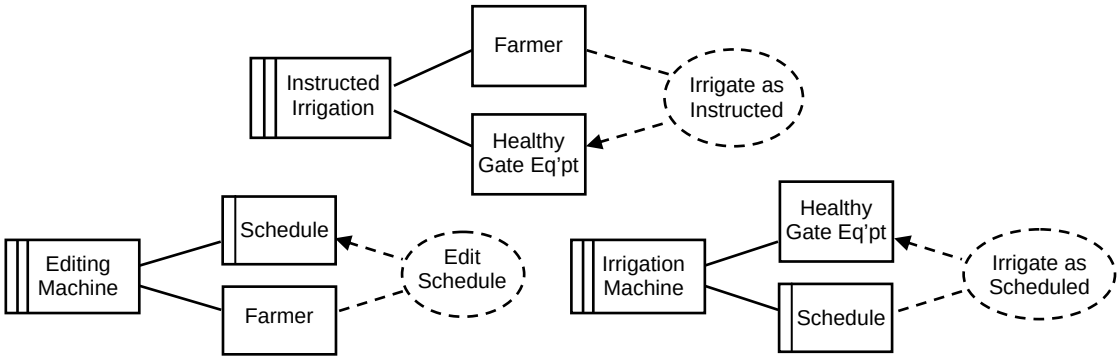

Figure 5: Irrigation as Instructed: Decomposition into Two Subproblems

In the subproblem on the lower left the farmer edits the schedule; in the subproblem on the lower right, the Irrigation Machine operates the gate in accordance with the schedule. The single stripe on the boxes representing the schedule indicates that it is a *designed domain*. That is, within the scope of the irrigation problem it must be, or has been, designed by the software developer to satisfy the needs of both subproblems. A designed domain may be realised as a collection of disk records forming a database, an assemblage of program objects, a USB key, a magnetic-stripe card, or any other concrete representation that the machine can read and write. Its concrete properties allow it to be written and read, and they furnish the substrate for representing its lexical content.

The decomposition promotes the schedule from its role in the solution, as a hidden local variable of the Instructed Irrigation machine, to a role in the problem, as a problem domain shared by the two subproblems. It is not given, but designed. The decomposition exposes both the design of the schedule domain and the development task—here postponed—of designing the recombination of the separated subproblems.

2.7 Decomposing the Problem—3

The second example of a decomposition exposing and promoting a local variable of the machine is a decomposition that introduces an *analogical model domain*.

The word 'model' has at least two senses that are relevant to software development. An *analytical* model of a problem domain is a *description* of the domain: a description $\mathcal{W}$ of the problem world's given properties is an analytical model. Such an analytical model is created and used by the developer, during the development. An *analogical* model of a problem domain, by contrast, is a designed lexical domain. If domain A is an analogical model of domain D, then D is the *subject* domain of A, and for some purpose A is intended to act as a surrogate for D. So, for example, a bank account database may be an analogical model of the bank's customers and their accounts: queries about past transactions and balances are answered by inspecting the database, not by asking the customer or the bank manager. The schedule domain in Figure 5 is not an analytical model, because it is not useful to regard it as a surrogate for any physical problem domain.

The decomposition is shown in Figure 6. The upper part shows the Alarm problem of Figure 4. The Monitor machine is required to monitor the behaviour of the gate equipment and

sound the alarm when a fault is detected. The decomposition will introduce a model domain whose subject domain is the Dubious Gate Equipment. As in the preceding example, the decomposition is justified only by the degree of complexity of the local variable to be exposed. If the only faults that can arise are simple and of a kind that can be detected without reference to historical data—for example, top and bottom sensors are both *on* simultaneously—then the decomposition is probably unwarranted. If, by contrast, faults can be detected only by checking the present state of the equipment against a record of past behaviour, then the decomposition is likely to be justified.

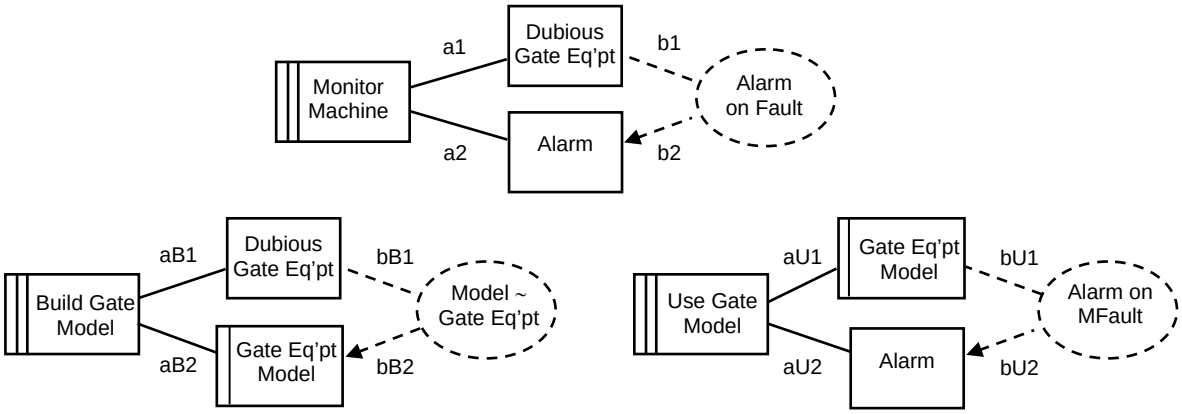

Figure 6: Decomposition Introducing a Model Domain

The lower part of Figure 6 shows the decomposition into two subproblems. On the left is a subproblem that builds the model domain. It creates and maintains the model data structure, continually recording the events and state changes in the subject domain—that is, in the Dubious Gate Equipment—in whatever raw or summarised forms the model design demands. On the right is a subproblem that uses the model, continually scanning the data structure for evidence of the faults that are to be detected. For example: a gradual increase of the upward travel time of the gate implies deterioration of the mechanism; the lower sensor's remaining in the *on* state when the motor has been *on* and *upward* for more than two seconds suggests a failed motor or a stuck sensor; and so on.

The requirement for the Build Gate Model subproblem specifies the relationship to be maintained between the Dubious Gate Equipment domain and the model. This is a correspondence between their states. If this correspondence is correctly maintained, then a fault in the real gate equipment will be represented by a computed state of the model: the requirement for the Use Gate Model subproblem specifies those model state values for which the alarm must be sounded.

The explicit representation of model and subject domains in the problem diagram for building the model shows clearly that the model and its subject matter are two distinct domains having no common shared phenomena. They are connected only by the behaviour of the Model Builder machine: at interface *aB1* the machine detects the state changes of the Gate Equipment; at *aB2* it causes corresponding updates of the model. When the Model User machine examines the state of the model domain at *aU1*, it is not, of course, examining the Gate Equipment itself, but a surrogate. These trivial truths can sometimes be overlooked in developments in which there is no explicit recognition of the nature of analogical models. In particular, when developers describe some of the given properties $\mathcal{W}$ of a problem domain in object or class diagrams—notations designed for describing software, not the physical world—they may be subtly encouraged to forget that the subject matter and its model are distinct realities. It is a conscious intent in PFA to keep this distinction always clearly present in the developer's mind.

2.8 Recombining Subproblems

Recombining the subproblems identified in a PFA decomposition is itself a significant development task. The subproblems are not like the pieces of a well-designed jigsaw puzzle, whose carefully shaped boundaries fit together perfectly to give the whole picture. Instead, because of the simplifications introduced in the decomposition, their boundaries have been shaped with a view to simplicity, not to the eventual need for recombination. Some rework, and even the inroduction of additional subproblems, may be necessary to enable the subproblems to fit together without interfering with each other or frustrating each other's purpose. Further, they must be composed in a way that satisfies the requirements of the problem from which they were derived.

The recombination process has two facets: reconstructing the whole problem, and implementing its solution. From the problem perspective, composition is bottom-up development: the original whole problem is reconstructed upwards from the leaves of the decomposition tree. The task here is to reconcile and combine the problem domains and requirements, both between sibling subproblems and between a parent and its children. Subproblems are the available parts from which the whole is to be assembled; but some additional parts—that is, additional subproblems—may be needed for the assembly, and some of the available parts may need modification. The implementation perspective, by contrast, focuses on combining subproblem machines—that is, transforming and connecting their program texts to give the desired executable machine. PFA has less to say about implementation than about problem analysis; but the two are not disjoint, and PFA makes a contribution here too. In this subsection the problem view is discussed; the implementation view is discussed in the following section.

2.9 Readers and Writers

A frequent task in subproblem composition arises when two subproblems have a problem domain in common. The resulting considerations are roughly those of shared data in programming. The subproblems shown in Figure 6 have the Gate Equipment Model in common: the model building subproblem is both a writer and a reader; the model using subproblem is a reader only. It is necessary to ensure mutual exclusion, with appropriate atomicity. The reader must always be presented with a version of the model that is sufficiently up to date, and is syntactically and semantically consistent with the given properties of the model domain that have been assumed in the analysis of the reader subproblem. The necessary constraints can be enforced, for example, by a transaction mechanism.

The irrigation and alarm subproblems shown in Figure 4 have the physical gate equipment as a common problem domain. The alarm subproblem monitors the gate equipment, which the irrigation subproblem controls, so they can be regarded respectively as reader and writer. In this case the composition will present no difficulty, provided that all access to each element of the gate equipment state—for example, setting the motor state to *on*, or testing the state of the top or bottom sensor—is atomic. The alarm subproblem makes no assumptions about relationships among the elements of the equipment state: atomicity of access to each element individually is therefore enough to guarantee a valid composite state.

Combining the writer and reader subproblems of the farmer’s irrigation schedule, shown in Figure 5, is a little more interesting. By hypothesis the schedule is sufficiently complex to warrant the decomposition: for example, it may be an elaborate program day by day and month by month to reflect the changing seasons and variations in water availability. Clearly at least mutual exclusion is required: the irrigation machine must always see a schedule that satisfies the designed domain’s assumed properties. However, because the schedule is

complex, the farmer may take a long time to complete the editing of a new version (perhaps even leaving it in a partially edited state for many hours). Depending on whether the partially edited schedule satisfies the domain properties, mutual exclusion may demand that execution of the irrigation machine is suspended during this long time. The obvious solution to this difficulty is to allow the irrigation machine to proceed using the current version while the farmer is editing the new version: when the new version is complete the irrigation machine switches from the old schedule to the new.

Arranging for the switchover of schedules is the requirement for a new subproblem. Clearly, it may be desirable to increase quite radically the separation between creating a new irrigation schedule and applying it: perhaps the farmer should be able to create a whole library of schedules, and to apply any schedule from the library at will. The requirement for this new subproblem will describe the structure and desired management and use of the schedule library.

2.10 Switching a Control Regime

Switching the irrigation from one schedule to another gives rise to an example of a concern—the *switching concern*—that is important in many compositions. The essence of a switching concern is that a problem domain is passed from being controlled buy one regime to being controlled by another. In this case the problem domain is the gate equipment, and the regime is the irrigation schedule. When the irrigation machine switches from one version of the schedule to another, it is necessary to ensure that the switchover is properly managed with respect to two concerns. First, the operation of the gate equipment must continue to satisfy the constraints of the breakage concern discussed in the following section: a certain timed protocol must be observed in motor operations. The switchover is therefore feasible only in certain equipment states: in effect, each *raise* and *lower* operation must be executed within a critical region that includes the periods necessary for the motor to come to rest. Second, the concatenation of acual behaviour under the old schedule and any prefix of the new schedule must satisfy the general scheduling constraints of the undecomposed problem. Suppose, for example, a stipulation that there must be no continuous water-flow of more than four hours: then if the end of the old schedule and the beginninng of the new are both water-flow periods, their sum must be no more than four hours.

Switching concerns arise in many diverse situations. Consider, for example, a system that manages loans to householders. It may be appropriate to decompose the problem of managing each loan into one subproblem for managing loans in good standing, and another for managing delinquent loans in which the borrower has infringed the terms of the loan. Each particular loan is then under control of either the good or the delinquent regime, and may switch—possibly more than once—between them. Depending on the complexity of the good and delinquent regimes there may be severe constraints on the states in which the switchover is feasible.

2.11 Checking a Joint Requirement

In the first decomposition, shown in Figure 4, the whole problem was decomposed into the irrigation and alarm subproblems. The requirement of the original problem, shown in Figure 3, no doubt stipulated some relationship between its irrigation and alarm aspects—perhaps that the alarm must sound at least for any fault that makes continued operation of the gate either impossible or very likely to damage the equipment severely, and perhaps for others. After the decomposition has been made, and the two subproblems have been analysed, the equipment assumptions necessary for irrigation, and those necessary for fault detection and diagnosis, are known. Now it is a part of the composition task to check that the stipulated relationship will be satisfied.

If we name *Faulty* those prefixes of possible equipment behaviours in which the alarm will be sounded, and *Workable* those that satisfy the equipment properties assumed in the irrigation subproblem, then we must show that

$$not\ Faulty \Rightarrow Workable.$$

We should not try to show that

$$Faulty \Rightarrow not\ Workable$$

because some detected faults may indicate impending rather than actual failure; in spite of the fault, *Workable* may still hold. Nonetheless it is probably required to stop operating the gate when any fault is found. It would not be good enough simply to halt the Irrigation machine: there is no guarantee that it would not halt with the motor in the *on* state, which may damage the equipment severely. The combination design must ensure that this does not happen: perhaps by modifying the Irrigation machine to enclose each *raise* and *lower* operation in a crictical region, or perhaps by introducing an additional subproblem whose requirement is to put the equipment unconditionally into a stable and safe state.

3. IDEAS AND PRACTICES

The general conceptual framework described in the preceding section must be fleshed out by specific ideas and practices suited to the particular nature of computer-based systems. In this section, some of these ideas and practices are identified and explained.

3.1 Problem World Phenomena

The problem world is central to a computer-based system, and so the task of describing it is central to software development. A description that can be used for reasoning about the world must use ground terms that denote recognisable *phenomena* in the world: events, states, entities and so on. Two underpinnings are needed. First, a *phenomenology*: that is, a taxonomy of the kinds of phenomena that can be recognised in the problem world and denoted by ground terms in descriptions. Second, a set of *designations*: that is, a mapping of the ground terms to the phenomena they denote, with a *recognition rule* for each term showing how individual instances of the phenomena can be identified in the problem world and distinguished from each other and from phenomena of other kinds.

Sometimes, especially for a problem domain that is itself a engineered artifact of high technical quality, the phenomenology and the designations are provided ready to hand in an instruction manual. But most of the world does not come accompanied by an instruction manual, and there is work to be done. This work demands abstraction and approximation, appropriately judged for the purpose in hand. For example, the proposed designation

$$LightOn \quad \approx \quad event:\ the\ light\ comes\ on$$

may abstract from a causal chain starting at the movement of a mechanical or electronic switch and progressing through a flow of electricity through wires to the heating of the element of an incandescent bulb. Identifying each traversal instance of this causal chain as a single event is evidently an abstraction: the steps in the chain are ignored. The abstraction is justified when the individual steps are not of interest for the description to be made. The whole chain is considered sufficiently reliable, and the small but inevitable delay in its traversal is considered negligible: so *LightOn* can be properly designated as an event—which in the chosen phenomenology is atomic and instantaneous. Similar issues arise in the designation of phenomena in problem domains of largely human activity. For example, in an e-commerce system it may be appropriate to designate

$$Deliver(p,c) \quad \approx \quad \text{event: product p is delivered to customer c.}$$

The designated event is again an abstraction of a causal chain starting perhaps from an instruction to a delivery company and ending at the physical receipt of the product at the customer's location.

The need for designations—whether explicitly considered and recorded, or merely decided tacitly and imperfectly communicated and remembered—cannot be bypassed by basing problem world descriptions solely on the phenomena shared between the machine and its immediately neighbouring problem domains. A developer who decides that *LightOn* means only that the machine has set the appropriate line to high, or that *Deliver(p,c)* means only that the machine has sent an electronic message to the machine of the delivery company's system, is thereby deciding to abandon the task of describing the problem domains. There will be no recorded description $\mathcal{W}$ of the given properties of the problem world, and therefore no explicit basis for arguing that the developed system satisfies its requirement.

3.2 Problem Locality

The designations of the phenomena of a problem domain provide the ground terms for the descriptions of its given properties. In general, the same domain will exhibit different given properties when it appears in different subproblems; these differences may even demand different choices in the set of designations. The irrigation subproblem shown in Figure 4 relies on the healthy operating characteristics of the physical sluice gate equipment. It is therefore appropriate to designate

$$GateOpen \quad \approx \quad \text{state: the gate is open and the top sensor is on}$$

as a simple state value. In the Alarm subproblem, however, the same equipment is assumed to be potentially faulty, and interest centres on the given properties that allow fault diagnosis. For this purpose it is absolutely necessary to distinguish two distinct state values

$$GateAtTop \quad \approx \quad \text{state: the gate is at the top of its travel}$$

$$TopSensor \quad \approx \quad \text{state: the top sensor is on}$$

This is just one small illustration of what may be called *problem locality*. The view of the problem world taken in a problem is strictly local, and may differ—sometimes radically—from the view of the same parts of the world in another problem. Another small illustration is the treatment of the *control* of phenomena in the two subproblems. In the irrigation subproblem the event *MotorOn* is shared by the machine and the gate equipment, and controlled by the machine; in the alarm subproblem it is regarded as an unshared phenomenon controlled by the gate equipment itself. The larger illustration of problem locality is that the domain properties are quite different for the two subproblems.

The assumptions of each subproblem are local, and fully independent of those of the other subproblem. This view of the meaning of problem world properties is strongly influenced by the rely/guarantee structure due to Jones [Jones83]: in *guaranteeing* to satisfy its own requirement, each subproblem machine *relies* on its own problem world assumptions. If the rely condition does not hold—that is, if the problem world's given properties do not satisfy the assumptions—then the guarantee is withdrawn.

3.3 Problem Simplicity

The treatment of the given properties of the problem world as assumptions, rather than truths that hold globally for the whole system, is the basis for simplifying the subproblems in a decomposition. Where the assumptions do not hold for the whole system there will be a need to resolve any inconsistency in the design of the recombination of the subproblems. The

claim implicit in PFA is that buying subproblem simplicity at the price of deferring the recombination concerns is, in general, a profitable exchange. In developments that have a substantial ingredient of *radical design*, in which the problem and its subproblems are largely or entirely novel, the recombination concerns reveal inherent difficulties in the overall problem: addressing them can profitably be separated from the concerns of the individual simple subproblems.

Subproblem simplicity has many facets: they are all facets of a kind of *uniformity* in the view that the developer is required to take of the problem and of its principal parts. This means that the terms used in the adequacy argument to be made by the programmer—that the proposed machine, installed in the given problem world, will satisfy the requirement—must themselves be sufficiently simple for the argument to go through with one consistent view of the problem world, the requirement, and the machine.

In effect, this is an insistence on simplicity in what the chemist and philosopher, Michael Polanyi, calls the *operational principle* [Polanyi58] of a working system, whether natural or engineered. Drawing on Polanyi's work, the aeronautical engineer Walter Vincenti writes [Vincenti93]:

> "Designers must first of all know what Michael Polanyi calls the 'operational principle' of their device. By this one means, in Polanyi's words, 'how its characteristic parts ... fulfil their special function in combining to an overall operation which achieves the purpose' of the device—in brief, how the device works. Every device, whether a mobile machine such as an aircraft or a static structure such as a bridge, embodies a principle of this kind."

Polanyi stresses that the operational principle of a device is not deducible from scientific or mathematical knowledge, however detailed and exact, of its possible behaviours. It is parallel to, and distinct from, such knowledge. The operational principle of a subproblem must be expressible at its largest granularity. If the problem is simple its operational principle is both concrete enough to embrace the substance of the problem and abstract enough to be easily understood.

In a simple problem, the 'purpose of the device', the 'characteristic parts', and their 'special function' are all easily grasped because each plays a uniform role. Consider, for example, the undecomposed sluice gate problem shown in Figure 3. The requirement, at the largest granularity is something like "irrigate the field, but if that proves impossible or dangerous to the equipment, then don't irrigate but sound the alarm." This is not a simple problem: the complexity in the requirement lies in the two levels of desired behaviour, corresponding to two levels of given properties in the problem domain. The decomposition shown in Figure 4 separates the two levels, each with a uniform requirement and a problem world with uniform given properties.

In a simple problem each problem domain has a clear and uniform role. Consider, for example, an e-commerce support system in which agents are engaged in responding to customers' email queries. The system must provide an efficient editing tool for the preparation of outgoing emails; it must also provide information to help in managing the relationships between agents and customers—for example, directing a customer's query to an agent who is familiar with that customer. For the editing requirement the agent is an active user of the editing tool; for the relationship requirement the agents and customers are part of a problem domain about which information is needed. The agent is therefore playing different roles in these two requirements: each role should therefore be handled in a distinct subproblem.

Another aspect of uniformity is the need for a simple *tempo* of the problem world and the requirement. In a simple problem, the interacting given and desired behaviours of the problem domains are all accommodated in one tempo—one synchronous temporal structure. The absence of this kind of uniformity creates what in the JSP program design method [Jackson75] is called a *boundary clash*. The classic illustration is the conflict between months and weeks in the Gregorian calendar: their boundaries are not synchronised, and this lack of uniformity causes serious difficulties in accounting. Essentially, the difficulties arise when it becomes necessary to consider explicitly all the cases of interaction between weeks and months: it is a small-scale instance of the combinatorial explosion. A system function that runs at a weekly tempo should therefore be separated in the decomposition from a function that runs at an annual tempo.

A different, but related, temporal uniformity may be achieved by assuming that some phenomenon is constant that does, in reality, vary over time. Consider, for example, a system to administer a lending library, for which there are requirements about the borrowing and return of books, and requirements about people joining the library as members, and renewing or terminating their memberships. The temporal cycle of the membership requirement is roughly annual, membership being renewable each year, while the temporal cycle of borrowing and returning may be one, two or three weeks. There is boundary clash here. It is therefore desirable to separate two subproblems: a membership subproblem, in which book borrowing and returning occur only as mutually unrelated atomic events; and a borrowing subproblem, in which library membership is constant, in the sense that only a person who is a member may borrow a book, and it is assumed that their membership will remain intact during the whole of the borrowing episode. The easily predictable complication of dealing with the subproblems' interactions—for example, handling (or preventing) cases in which membership expires during the currency of a loan—is deferred to the task of subproblem recombination.

This notion of simplicity as uniformity may seem imprecise, and indeed it is. That is not surprising: we are concerned here, as everywhere in PFA, with human capacity for understanding, for which there are no precise criteria.

3.4 Dependability and Simplicity

Other things being equal, a simple subproblem is more understandable than a more complex subproblem. In an important sense, simplicity is a fundamental kind of strength in software: a simpler program is stronger because it is less likely to be faulty. This notion of simplicity as software strength implies a decomposition criterion that is an important contributor to system dependability.

In an imaginary ideal world a system designed for dependability would be dependable in every part and every respect. In practice this cannot be, because human developers are fallible and resources are limited. A more realistic ambition is to try to ensure that the most critical system functions are the most dependable. They must therefore be the most simple. An extremely critical function—for example, the shut-down function activated by an emergency button—must be allocated to a subproblem of its own, and must be purged of every complication that is not vital to the function.

The treatment of a critical subproblem in composition is clear. From the problem perspective, in which subproblems are combined, a critical subproblem should be exempt from invasive composition. If an invasive modification is necessary to reconcile it in some way with a less critical subproblem, only the less critical subproblem should suffer the modification. The same principle applies in the implementation perspective, briefly discussed in a later subsection, in which subproblem machines are combined.

3.5 Subproblem Concerns

Even the simple subproblems aimed at by PFA decomposition may present significant difficulties. Depending on the natures of the problem domains, of their interactions, and of the requirement, a simple subproblem will still present concerns that must be addressed by the developer. Every problem, of course, presents the basic concern of satisfying its operational principle—of ensuring that the parts can, indeed, combine successfully to fulfil the overall purpose. This basic concern may be called the *frame concern*. Other subproblem concerns address possible failures that may interfere with the designed success. It is useful to name these concerns, and to address them separately and explicitly.

One subproblem concern is *initialisation*. The programming danger of uninitialised variables is well known. In a computer-based system the danger is greater. Consider, for example, the subproblem shown at the lower left of Figure 6, where the machine is required to build and maintain the model of the gate equipment so that it can act as a surrogate for the equipment itself. The developer might assume that when the machine begins its execution the gate will be in some suitable initial state: for example, with the gate in its lowest position, the lower sensor *on*, the upper sensor *off*, and the motor it its *off* and *upward* states. This assumption may not hold: the machine may be started in some other state and failure is then a likely result. In general, there are many ways of addressing the concern. The machine can be designed to accommodate any possible initial state of the modelled domain; the machine can execute a prologue phase in which it detects the current state of the modelled domain, or brings the domain into a known state; a manual initialisation procedure can be stipulated to be performed when the machine is started; and so on.

Another concern arising when a physical domain is to be controlled is the *breakage* concern: the system may fail because the machine has damaged the domain. In the irrigation subproblem, the gate is raised and lowered by the machine. The physical gate equipment may have been designed on certain assumptions about the operating protocol: for example, that the motor direction would not be switched between *upward* and *downward* except when the motor is at rest; or that the motor will not be held *on* for long enough to drive the gate hard against its travel limits. The developer must identify such restrictions, and ensure that the machine observes them correctly.

In the Traffic Control problem shown in Figure 2, the machine has an interface of phenomena shared with the Light Units domain. This is a *physically multiple* domain, in the sense that it is populated by multiple instances of a physical type—the individual units. The machine will be designed to set a particular control line *high* in some particular situation because the physical light sharing that phenomenon is, for example, the north-south red light in the unit at a certain place in the road layout. But what if the actual physical pairing of units with control lines has not been set up exactly as the developer supposes? This is an *identities* concern, and arises only for physically multiple domains. Like initialisation, it concerns the set-up conditions of the implemented system, but it is nonetheless a development concern. The developer must consider how to avoid—or tolerate—faulty set-up conditions that might otherwise cause failure.

One final example of a subproblem concern is what we may call the *anomaly* concern. It can arise with any domain of human behaviours, states, and attributes: for such a domain almost every assumed property must be carefully questioned, and measures taken to guard against surprises. It might, for example, be reasonably supposed that a person's date of birth is a constant: but there are many reasons why people might misreport their date of birth, especially immigrants from countries with oppressive regimes, and correct this misreporting some time after they have become known to the system. Similarly, changes of name or gender are easily overlooked.

Addressing subproblem concerns is about avoiding failures. After a failure has occurred it is relatively easy to identify the causes that have contributed to the failure. Often a superficially reasonable reaction to a software failure is to wonder how the developers could have been so negligent. The answer to this question is usually straightforward: the developers lacked the hindsight available to the investigators and critics. A realistically large system provides a very large conceptual space in which potential failures can hide: exhaustive search is not feasible, and teasing out potential failures depends on knowing where they are most likely to be found.

A discipline of identifying and addressing subproblem concerns is an essential part of any serious approach to system dependability. It can be seen as a part of the normal process of constructing a safety case for a system, but it is different in two ways. First, it is integrated into the basic development activity; and second, it takes place at the subproblem level, where the scope to be considered is smaller and the desired goal—with respect to which failure must be avoided—is clearly expressed in a simple operational principle.

3.6 Software Work and Application Work

The PFA view of developing computer-based systems identifies three large topics for investigation and description at evey level: the stakeholders' purposes $\mathcal{R}$; the context in which those purposes have meaning, $\mathcal{W}$; and the software programs themselves, $\mathcal{M}$. How should the work on these topics be divided? In particular, can a clean division be made between 'application' tasks and 'pure software' tasks?

Certainly no clean division can be made by specifying the machine's external behaviour at its interface with the problem world purely in terms of the interface phenomena without reference to the problem world. For the Traffic Controller problem of Figure 2, the interface may comprise:

- *a1*: twenty seven groups of control lines (for light units), some groups having four control lines and some having five, the lines being labelled *u1.1* .. *u27.5*; and
- *a2*: fourteen control lines (for pedestrian crossing buttons), labelled *p1* .. *p14*.

The specification must describe a machine behaviour that sets the control line states at *a1* depending on the passage of time and on the current and earlier states at *a2*. It must be written entirely in terms of the control lines, making no mention of the problem world at all—the software developer need not even know that the system for which the software is being developed is a traffic control system.

In its extreme form, this imaginary specification is completely impracticable—not because it could not be constructed, but because it would be unintelligible: the programmer receiving such a specification could see it only as arbitrary and meaningless. The difficulty stems from the power and versatility of the computer. Because it is so versatile it has no intrinsic functionality of its own, but—suitably programmed—adopts a functionality somehow derived from the problem in hand; and because it is so powerful, that adopted functionality can be arbitrarily complex. The architecture of the hardware and operating system provide no useful clues: in Polyani's phrase, the computer's *operational principle* is too abstract and general to contribute much to the human understanding of any particular specified external behaviour.

The software developer's work must, therefore, somehow involve the problem world and the requirement expressed in problem world terms. Modifying and limiting the extent of this involvement in the context of a particular system can be seen as *problem reduction*, which is a kind of problem transformation.

3.7 Problem Reduction

A problem diagram is simple, but it captures important decisions that will have a large effect on the subsequent analysis and development. The diagram bounds the problem to be solved, both above and below. Everything that is in the diagram is relevant, and everything relevant is in the diagram. In Figure 2 the developer cannot ignore any part of the requirement or any of the pictured problem domains, but is entitled—indeed, obliged—to ignore everything else.

As discussed in an earlier section, acceptance of the problem diagram therefore obliges the developer to document and reason about the given properties of the Pedestrians and Vehicles and of the Road Layout domain. However, the problem can be *reduced* if this obligation is discharged—probably by someone other than the developer—in a preceding phase of work. A reduced problem is shown in Figure 7.

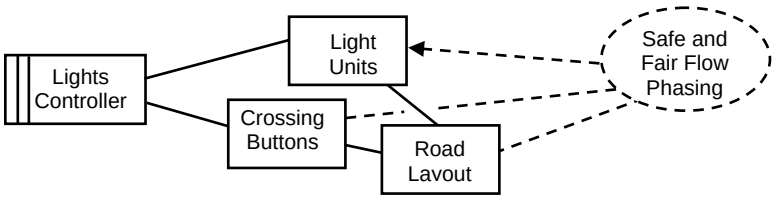

Figure 7: Reduced Problem: Traffic Lights Control at a Road Junction

The Pedestrian and Vehicles domains have been removed in the reduction. Their influence on the problem has been precalculated, and their given domain properties, such as speeds and accelerations, have been taken into account in a modified requirement. The modified requirement says nothing about collisions, about speeds of traversing the pedestrian crossings and road segments of the junction, or about how many vehicles can wait in particlar controlled segments. Instead it stipulates the permissible patterns of combinations of light settings, specifying for each combination which lights on the Road Layout are red and which are green, the patterns specifying the timed sequencing of combinations and of the intermediate settings when the lights change between successive combinations. The developer now need know nothing about the behaviours of cars and pedestrians, but is still required to understand notions of the Road Layout, positions of Light Units and Crossing Buttons, and the operation protocols for setting the traffic lights. In effect, we may say that the problem has been moved closer to the machine—but not so close that the resulting problem is impossible for the developer to understand.

In an important sense, every computer-based system problem presented to a developer has already been to some extent reduced. The problem of controlling the sluice gate was reduced from a larger problem about crops, fields and water—which was itself reduced from an even larger problem of managing the farm, and so on in an endless regression of purposes and means. Drawing the problem diagram bounds the problem at both ends. The requirement cannot be about the meaning of life; neither can it be about the behaviour of the machine.

3.8 POSE: Problem-Oriented Software Engineering

Problem reduction, discussed in the two preceding subsections, is an example of a *problem transformation*: it has been discussed in a formal setting in [Rapanotti06]. Other examples of transformation are problem decomposition and problem recombination, discussed in the preceding section. Even the elaboration of an empty description of a problem domain—consisting only of its name—by discovering and recording relevant given properties can be viewed as a problem transformation. More generally, the whole development activity can be regarded as the execution of problem transformation steps.

One of the respects in which the problem frames approach, as described in the earlier sections of this paper, is incomplete is the lack of a formally structured framework for the whole development activity. Problem-Oriented Software Engineering [Hall07] offers such a structure. The structure captures the step-by-step solution of a system development problem; it also captures the argument that must eventually justify the adequacy of the developed system. The framework is itself formal, defining what is essentially a sequent calculus in which each step is regarded as a problem transformation according to a rule. The rules themselves are formally expressed, but the content of the transformations, and the associated arguments that justify their application, can be either formal or informal.

3.9 Implementation Aspects of Combining Subproblems

Eventually the subproblem machines must be composed to give the complete machine, however implemented, that solves the original problem. The subproblem machines are not a set of programs encoded in a specific programming language, in whose texts many problem-independent implementation choices have already been bound. They are to be regarded, in effect, as more abstract than the given domains of the physical problem world, because they are merely projections of the behaviour of the complete machine—a machine domain that has not yet been constructed. The way is therefore open to a range of behaviour-preserving transformations of subproblem machines, and designed domains offer similar opportunities. For example:

- Two model domains whose subject is the same underlying problem domain—though with different assumptions—may be merged into a single composite model domain.
- Where one subproblem machine takes the initiative in causing certain events or state changes in a problem domain, and another monitors the same phenomena in the same domain, it may be possible to fragment the second machine into a collection of actions that can be inserted into the behaviour of the first machine.

Such transformations embody implementation decisions that are often taken prematurely in an earlier development phase.

Because PFA is concerned to structure and analyse problems, it has in general paid less attention to the implementation task of designing the complete machine. However, some specific proposals have been presented for implementation techniques whose basic starting point is a PFA problem structuring:

- *Architectural Frames* [Rapanotti04] are problem diagrams in which the machine is elaborated into a sructure of component machines. The component machines have interfaces to each other, and also to the neighbouring problem domains. This technique allows advantage to be taken of appropriate known solution patterns—for example, of the Model-View-Controller pattern, or of a Pipe-And-Filter machine architecture—both in forming the implementation design and also, at an earlier stage, in guiding the problem analysis and structuring towards a desired solution.
- *Composition Controllers* [Laney07] are machines introduced into the system to take control of subproblem machines. A composition controller interposes itself between each subproblem and its problem domains, and is therefore capable of managing the interactions at that interface. For example, it can cut a subproblem machine off completely from its problem world, thus terminating its ability to affect the world; or it can selectively suppress individual interaction phenomena to resolve a conflict between one subproblem's requirement and another's. An advantage of the composition controller technique, where applicable, is that the controller manages subproblem interactions at execution time, reducing the need for invasive compositions in which the subproblem design is modified.

3.10 Implementation of Critical Subproblems

Criticality has been mentioned as a criterion of decomposition and of subproblem recombination: a subproblem providing a critical function must not be complicated by the influence of less critical functions. The same general principle applies in the implementation perspective, when subproblem machines are combined: a more critical subproblem must not be exposed to the consequences of failure of a less critical subproblem.

Here is an example. In a system to control and manage a proton therapy machine, two of the identified subproblems were Emergency Stop and Command Logging. The emergency stop subproblem shuts off the beam and closes down the equipment in response to the operator's pressing of the emergency button. Command logging ensures that every command issued by or through the control system is logged to disk for auditing in the event of any incident demanding investigation. It seemed necessary to the developers that commands issued by the emergency stop function should be routed to the equipment through the logging function, to ensure that they would be logged; that was how the software was originally configured. However, the developer of the logging subproblem had overlooked a subproblem concern: the behaviour of the logging function was unspecified if there was not enough disk space to record the current command. The actual—unintended—behaviour was that the logging function failed to log the command, failed to pass it on to the equipment, and failed to return to the procedure that had invoked it: in the chosen composition the current command would not then be executed. In effect, the emergency button ceases to work if the logging disk is full.

4. THE ROLE OF PFA

The PFA emphasis on problem analysis and structuring is in some respects unusual. Ralph Johnson—who was one of earliest champions of the Design Patterns movement, wrote [Johnson94]:

> "'We have a tendency to focus on the solution, in large part because it is easier to notice a pattern in the systems that we build than it is to see the pattern in the problems we are solving that lead to the patterns in our solutions to them."

As Johnson points out, it is somehow easier to focus on solutions than on problems. Solutions are concrete and specific, and constrained by the programming language and implementation environment, while problems naturally seem to be more abstract and general, and relatively unconstrained. In the development of computer-based systems the role of PFA is first to show that problems too can be be concrete and specific—constrained by the properties of the given problem domains—and second to provide an intellectual framework for analysing and structuring them.

A large part of the original motivation [Jackson94] for PFA sprang from the recognition that inappropriate software development methods were often chosen for the problem in hand. Sometimes the chosen method was too general to provide a good grip on the problem; sometimes it was specific enough, but made implicit assumptions that did not fit the problem. It seemed clear that realistic software development problems could not be usefully classified as wholes: they are complex and heterogeneous asssemblages of smaller problems. It is more useful, then, to identify and name different classes of elementary software development problem—*information display*, *controlled behaviour*, *transformation*, *workpieces* and others. Each class can be described [Jackson01] by a *problem frame*. A problem frame is a formalised problem diagram, with formal annotations for interface and requirement phenomena and with each problem domain marked with its type—*causal*, *lexical*, or *biddable*. A particular small problem fits a class if its actual topology, domain

types and interface phenomena can be exactly mapped to their formal equivalents in the class description, and its requirement fits the purpose associated with the class.

Although problem classification has not been emphasised in this paper, it remains a significant aspect of the problem frames approach. It offers a rudimentary basis for extending the applicability of *normal design* to the development of computer-based systems. A brilliant book [Vincenti93] about engineering points out that normal design is everyday practice in the established enginering branches:

> "[in normal design] ... the engineer knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task."

In *radical* design, by contrast:

> "… how the device should be arranged or even how it works is largely unknown. The designer has never seen such a device before and has no presumption of success. The problem is to design something that will function well enough to warrant further development."

Suitably refined and developed, the classification of elementary problems in PFA could offer a taxonomy of what Vincenti calls 'devices', and a structure within which accumulating experience and knowledge can be codified and made more easily accessible. The notion of subproblem concerns, discussed in an earlier section, is an example of capturing such knowledge and, to that extent, bringing those subproblems within the ambit of normal design. As knowledge increases, problems that had previously been regarded as composite could come to be regarded as elementary—just as multiplication comes to be regarded as an elementary operation when we learn arithmetic.

Few realistic computer-based systems are the object of purely normal or purely radical design. Vincenti, regarding the engineering process and its product as a hierarchy, goes on to say:

> "Whether design at a given location in the hierarchy is normal or radical is a separate matter—normal design can (and usually does) prevail throughout, though radical design can be encountered at any level."

In computer-based systems, where feature proliferation is the rule rather than the exception, radical design is perhaps most notably encountered in designing the composition of subproblems, especially where solutions to some subproblems are already available. Because much of current software development practice is strongly focused on solutions rather than on problems, it may be difficult to determine exactly what problem is solved by an available component, and what assumptions its designer has made about the problem world. In such an environment, effective analysis of composition concerns may demand some reverse engineering to clarify the subproblems that are to be composed. PFA can offer guidance in reverse engineering: the notions of subproblem simplicity are useful in the reverse, as well as in the forward, engineering direction.

ACKNOWLEDGEMENTS

Extensive discussions over several years with Cliff Jones and Ian Hayes have illuminated the Sluice Gate problem and its many lessons. Colleagues and postgraduate students at the Open University have made large contributions to the understanding and extension of the ideas described here. Notable among them are Bashar Nuseibeh, Charles Haley, Jon Hall, Robin Laney, Zhi Li, Armstrong Nhlabatsi, Lucia Rapanotti, Mohamed Salifu, Thein Than Tun and

Yijun Yu. Discussions with Daniel Jackson, and his perspicuous comments on many topics, have been invaluable.

REFERENCES

[Goh04] P K Goh and William H K Lam; *Pedestrian Flows and Walking Speed: A Problem at Signalized Crosswalks*; Institute of Transportation Engineers. ITE Journal, Jan 2004.

[Hall 07] J G Hall, L Rapanotti and M Jackson; *Problem oriented software engineering: A design-theoretic framework for software engineering*; in Proceedings of 5th IEEE International Conference on Software Engineering and Formal Methods, IEEE Computer Society Press, 2007.

[Hayes03] Ian J Hayes, Michael A Jackson and Cliff B Jones; *Determining the specification of a control system from that of its environment*; in Keijiro Araki, Stefani Gnesi and Dino Mandrioli eds, Formal Methods: Proceedings of FME2003, pages 154-169, Springer Verlag, Lecture Notes in Computer Science 2805, 2003.

[Jackson75] M A Jackson; *Principles of Program Design*; Academic Press, 1975.

[Jackson94] M A Jackson; *Software Development Method*; in A Classical Mind: Essays in Honour of C A R Hoare, A W Roscoe ed; Prentice-Hall International, 1994.

[Jackson01] Michael Jackson; *Problem Frames: Analysing and Structuring Software Development Problems;* Addison-Wesley, 2001.

[Johnson94] Ralph E Johnson; *Why a Conference on Pattern Languages?* ACM SE Notes Volume 19 Number 1, pages 50-52, January 1994.

[Jones83] C B Jones; *Specification and design of (parallel) programs*; IFIP'83 Proceedings, pages 321–332, North-Holland, 1983.

[Laney07] Robin Laney, Thein T Tun, Michael Jackson and Bashar Nuseibeh; *Composing Features by Managing Inconsistent Requirements*; in Proceedings of the Ninth International Conference on Feature Interactions in Software and Communications Systems, Grenoble, September 2007.

[Polanyi58] Michael Polanyi; *Personal Knowledge: Towards a Post-Critical Philosophy*; Routledge and Kegan Paul, London, 1958.

[Polya57] G Polya; *How To Solve It*; Princeton University Press, 2nd Edition 1957.

[Rapanotti04] Lucia Rapanotti, Jon G. Hall, Michael Jackson and Bashar Nuseibeh; *Architecture-driven Problem Decomposition*; in Proceedings of the 2004 International Conference on Requirements Engineering (RE'04), Kyoto, IEEE CS Press, 2004.

[Rapanotti06] Lucia Rapanotti, Jon G Hall and Zhi Li; *Deriving specifications from requirements through problem reduction; IEE Proceedings—Software, Volume 153 Number 5, pages 183-198, October* 2006.

[Risks08] The Risks Forum; http://catless.ncl.ac.uk/Risks (accessed 19th November 2008).

[Seater06] Robert Seater and Daniel Jackson; *Requirement Progression in Problem Frames Applied to a Proton Therapy System*; Proceedings of the 14th International Requirements Engineering Conference (RE06), Minneapolis USA, 2006.

[Vincenti93] Walter G Vincenti; *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History;* The Johns Hopkins University Press, Baltimore, paperback edition, 1993.