South African Computer Journal 22; Special Issue on WOFACS'98, pp47-60, 1999

Problem Analysis Using Small Problem Frames

M. A. Jackson

Independent Consultant
101 Hamilton Terrace, London, England, jacksonma@acm.org

Abstract

The notion of a problem frame is introduced and explained, and its use in analysing and structuring problems is illustrated. A problem frame characterises a class of simple problem. Realistic problems are seen as compositions of simple problems of recognised classes corresponding to known frames.

Keywords: *Software Engineering, Problems, Frames, Frameworks, Patterns*
CR Categories: *D2, H1*

1 Introduction

Software developers have aspired to the status of an engineering profession since the NATO conferences of thirty years ago [Naur 69, Buxton 70]. But few believe that the aspiration has been fulfilled. A leader in the field recently claimed [Parnas 97] that Software Engineering is an 'unconsummated marriage': "The majority of engineers understand very little about the science of programming or the mathematics that one uses to analyse a program, and most computer scientists don't understand what it is to be an engineer." He continued: "Chemists are scientists; chemical engineers are engineers. Software engineering and computer science have the same relationship."

The underlying assumption is that software engineers are practising a single discipline, properly aspiring to become a member of the same set as chemical, aeronautical, electrical, civil or electronic engineers. Like them, we aim to build useful products to serve practical purposes in the physical world. And indeed we do build physical products. The product of successful software development is a machine that interacts with its human users and with other parts of the world. The machine is almost always physically embodied in a general-purpose computer built by hardware engineers; but the software describes the particular machine needed for the purpose in hand, and transforms the computer into that machine.

1.1. The General SE Problem

We may characterise the general form of the software development problem as presented in [Jackson 95] and shown in Figure 1.

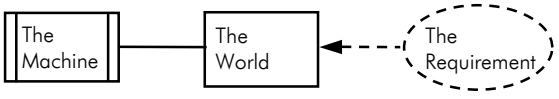

Figure 1
The General SE Problem

The striped rectangle represents the physical *machine* we must build by specialising a general-purpose computer. The plain rectangle represents the part of the *world* that interacts with the machine. The solid line connecting the two rectangles represents an interface of shared phenomena — for example, shared events and shared state. The dotted ellipse represents the intangible *requirement*, the dotted arrow indicating that the requirement is a description — we might say, a predicate — over the phenomena of the world. In the terminology of Polya [Polya 57], the machine, the world and the requirement are the *principal parts* of the software development problem; the *solution task* is to construct a machine such that its interactions with the world will ensure satisfaction of the requirement.

1.2. Descriptions

Although the end product is a description of the machine, a successful result can rarely be achieved by describing the machine alone. In general we need to make the following descriptions:

- The requirement $\mathcal{R}$ is an explicit description of the behaviour and properties that we want the world to have as a result of its interaction with the machine. $\mathcal{R}$ is a description in the optative mood — that is, it expresses what we would like to be true. It is a description over the phenomena of the world that are of interest to the customer of the development: it captures the purpose for which the machine is to be built and installed.
- The unconditional behaviour and properties of the world that do not depend on the machine are expressed in a world description $\mathcal{W}$. $\mathcal{W}$ is an indicative description — that is, it expresses what is true of the world regardless of its interaction with the machine. It is a description over any phenomena of the world whose relationships are significant for the purpose in hand; in particular, it is not restricted to phenomena shared with the machine.
- The *specification* S describes the behaviour and properties that we want the machine to have at its interface with the world. S is an optative description. It is a description over the shared phenomena at the

interface, consistent with the properties of the world and satisfiable by appropriate action of the machine.

- The *program* $\mathcal{P}$ describes the behaviour and properties that we want the machine to have, without restriction to its interface with the world. Again, $\mathcal{P}$ is an optative description. It is a description of the phenomena of the machine, including those private machine phenomena that are in the scope of the programming language.
- The unconditional behaviour and properties of the machine that do not depend on the program are expressed in a *machine semantics* C. C is an indicative description: it expresses what it true of the general-purpose computer independently of the specialisation imposed by the program. It can be thought of as a description of the programming language semantics in terms of the behaviour of the concrete machine.

Two relationships among these descriptions constitute a description of the original problem and a demonstration that it has been solved. First, we must have

$$S, \mathcal{W} \vdash \mathcal{R}$$

If the machine achieves the behaviour S at its interface with the world, then, given the known properties $\mathcal{W}$ of the world, the requirement $\mathcal{R}$ will be satisfied. Second, we must have

$$\mathcal{P}, C \vdash S$$

If the machine is as described in C, then execution of the program $\mathcal{P}$ will ensure the behaviour S at the machine's interface with the world.

Strictly, none of these descriptions may be omitted. $\mathcal{R}$ captures the purpose of the system: many systems have failed because their requirements were not correctly articulated. $\mathcal{W}$ captures the properties of the environment: these properties both constrain what the machine can do and permit it to affect and be affected by world phenomena that it does not directly share. S provides a convenient staging post in the often long chain of reasoning from the system's purpose to the program text; it also allows a practical separation between those developers whose interests and knowledge focus on the application domain world and those who focus instead on the machine and its programs. $\mathcal{P}$ is the indispensable end product of the development. C provides the underlying justification for refinement steps in the path from the specification S to the program $\mathcal{P}$.

In practice, of course, C is already available in the manuals for programming language and its implementation in the particular computer to be used. And the sometimes limited attention paid to S, $\mathcal{W}$ and $\mathcal{R}$ is a measure partly of the development team's competence and partly of their assessment of the risk of system failure and its consequences.

1.3. Specialisation

Unfortunately, this tidy characterisation of software development problems is a grossly insufficient basis for software engineering practice. Projects to develop systems for program compilation, for telephone switching, for banking, for controlling the brakes of a car, for word processing, and for searching the web are radically different and demand different engineering techniques. They differ hugely in their principal parts: their requirements, their worlds and their machines are so different that they need different languages for their description, different description structures, and different reasoning over those descriptions. The worlds of telephone switching software and car braking software are as different as the worlds of communications engineering and automobile engineering.

Established engineers specialise for exactly this reason. Communications engineers and automobile engineers build completely different kinds of product to meet completely different kinds of purpose. Each established engineering field deals with a narrowly defined problem class, applies narrowly defined design methods, and produces equally narrowly constrained solutions. From a perspective that embraces every established branch of engineering, this year's cars are indistinguishable from last year's, and next year's will be no different. Each branch of engineering adjusts its problem definitions and refines its products only gradually, by small perturbations from an established standard. This specialisation allows product function and quality to improve by small steps over many generations of products and engineers: today's cars are recognisably solving the same problem as the cars of 1930, but they are much better in almost every way.

Against this background one can see that the notion of software engineering as a single discipline is misconceived. Software engineering can no more be a single discipline than can 'physical engineering' — an imaginary discipline that embraces all the established branches from aeronautical to telecommunications engineering. Like the established branches, it relies on a core body of fundamental mathematics and science, but these fundamentals lie some distance below the level at which the practising engineer works. At this level, of practical technique, engineering varies enormously from one specialised branch to another.

So too must it be for software engineering. We must deal in specialities, not generalities. Already some established specialities have emerged. Compiler construction, operating systems, GUIs and expert systems are notable examples. As each specialised branch emerges and begins to establish a substantial body of practical technique of its own, it breaks away from the parent tree. Compiler construction and operating system design cease to provide illustrative problems for courses on software engineering, and become the rich subject matter of their own separate courses.

2 SE Problems for Generalists

As its specialised branches emerge and take their leave, software engineering finds itself concerned with those particular problems and solutions that are not yet well enough understood to furnish the subject matter for additional specialities. This problem set is very rich indeed: the computer is so versatile that its applications are virtually unbounded. Today most realistic development problems fall into this set: only a few fall squarely into an established speciality. We are not at liberty to disdain these problems on the grounds that specialised knowledge is absent or inadequate. We must do what we can, approaching them as competent generalists.

In the best case this means recognising that the problem is a composition of simpler problems that we do know how to solve, and that a solution can be constructed from the known solutions to those simpler problems. An alarm clock satisfies a certain requirement; a radio satisfies another. A

clock radio satisfies both requirements. Similarly, a breakdown truck satisfies the requirements of a crane and a vehicle. The combination may satisfy additional requirements — for example, the alarm can be set to turn on the radio; and it may exploit common solution parts — for example, the breakdown truck powers uses the same engine to power both the crane and the vehicle.

The work on object-oriented patterns [Gamma 94, Buschmann 96, Pree 97] addresses the need to identify a repertoire of software components and to consider their properties and the ways in which they can interact. Work on software architecture [Shaw 96, Bass 98] addresses larger structures of interaction. Both patterns and architecture are primarily concerned with the space of solutions. Inevitably they pay some attention to the problems being solved, but this attention is focused chiefly on the impact of the problem on the solution. There is a need to address problem structure and classification in a more sharply focused and explicit way. That is the theme of this paper.

2.1. Problem Structures and Problem Frames

We regard particular problem classes as characterised by *problem frames*. Each frame is an elaboration of the general form shown above in Figure 1. Each frame is either *elementary* or *composite*. A problem of the class characterised by an elementary frame is to be captured by building descriptions appropriate to the frame. A problem of a composite class is first decomposed into subproblems characterised by elementary frames.

A particular problem frame elaborates and specialises the general form of Figure 1 in the following ways:

- The *world* is decomposed into *domains*. For example, if the problem concerns the production of an output text stream from an input text stream, we may decompose the world into the domains InputText and OutputText. (The term *domain* is considered to include the machine. The machine is not decomposed within one elementary problem frame, but there is a *submachine* — a projection of the machine — for each subproblem of a composite problem.)
- Different types of domain are distinguished according to the role they play in the problem. For example, one domain may embody a tangible *description* of another domain; or one domain may be *given*, while another is *created* by the action of the system.
- Interfaces of phenomena shared between domains are shown by connecting lines as in Figure 1. Not all domains need be connected to the machine. For example, in the well-known Patient Monitoring problem [Stevens 74] the Patients and the Analogue Devices are two domains: the Analogue Devices are connected to the machine, but the Patients are connected only to the Analogue Devices.
- The connections among the parts of the problem frame are more closely characterised in terms of the types of the connecting phenomena. For example, an interface connecting two domains may have only shared event phenomena, while another interface between two other domains may have both shared events and shared states. The control of phenomena is also indicated, as explained in the following section.
- The phenomena related by the requirement are similarly characterised according to their types. Also, they are identified, where appropriate, with phenomena at interfaces in the frame.
- The characteristics of domains at their interfaces with other domains are classified. For example, an *Inert Reactive* domain initiates no events; it responds to each shared event initiated by the connected domain by changing the shared state, and returns to an inert state until a fresh shared event occurs. In general, the characteristics of a domain are different at its interfaces with different sharing domains: in the Patient Monitoring problem, the Analogue Devices are Inert Reactive at their interface with the Patients, but Active at their interface with the machine.

These elaborations are illustrated and discussed in subsequent sections. They support a repertoire of elementary frames, each very simple.

Composite frames, also illustrated in subsequent sections, are essentially parallel compositions of elementary frames. For some composite frames it is necessary to introduce additional created domains that mediate between subproblems, somewhat after the fashion of local program variables. In general the creation of such an additional domain becomes a subproblem in its own right, with its own elementary problem frame. Such an elementary frame is called a *partial elementary* frame, because the problems it characterises — like the creation of a local variable — can never be independent problems in their own right but occur only as subproblems. Other partial elementary frames, as discussed in subsequent sections, arise from explicating implicit assumptions about the environment of a problem.

2.2. Phenomena and Control

Domains and interfaces, and hence problems, differ in their phenomenological characteristics. For example, a static domain, in which there are no events and no state changes, is different from a dynamic domain in which events and state changes occur over time. They will raise different considerations in the treatment of problems in which they appear, and will demand different kinds of description.

We must also consider the control of events and state changes at the domain interfaces of shared phenomena. For example, in a problem to control a lift, the pressing of buttons is controlled by the users, the polarity of the winding motor is controlled by the computer, and the closing and opening of sensors in the lift shaft is controlled by the lift mechanism itself through the movement of the lift car. (We are, of course, concerned here with proximate control by one of the sharing domains. The sensors are controlled by the lift mechanism, not by the computer, because the proximate cause of the closing of a sensor is the arrival of the lift car; the fact that the computer can indirectly cause the lift car to travel in the shaft is not relevant here.)

In characterising interface properties we use a simple classification of phenomena. We recognise three kinds of individual:

- *Values* (‘V’) are timeless individuals: for example, integers and characters.

- *Entities* ('N') are individuals that change over time: for example, people and bicycles and bank accounts.
- *Events* ('E') are atomic events occurring in time: for example, the pressing of a lift button or the starting of the winding motor.

We recognise three kinds of relation over individuals:

- *Truths* ('U') are timeless relations: for example, 'x>y' over integers.
- *States* ('S') are relations that change over time: for example, 'IsChild(x)' over human beings. Changes of an entity over time are changes in states in which it participates.
- *Roles* ('R') are relations indicating participation of an individual in an event: for example, 'IsButtonIn(b,p)' over buttons and button-press events.

We must also express the temporal ordering of events and capture the relationship between events and states. In [Jackson 93], [Zave 93] and [Jackson 95] this relationship was captured by introducing intervals between events as explicit individuals. Here, as in [Bhargavan 98], we use instead a convention in which state symbols are decorated with the prefix or suffix 'Then' and the argument list augmented by an event identifier. For example, 'IsChildThen(x,e)' means 'IsChild(x) is true immediately before event e occurs', and 'ThenIsChild(x,e)' may mean 'IsChild(x) is true immediately after event e occurs'. These relationships will not play a part in the discussion of problem frames, and we will not pursue them further here.

Larger classes of phenomenon that will be useful are:

- *All phenomena* ('H') is the class containing all the phenomena.
- *Controllable phenomena* ('C') is the class containing those phenomena that can be controlled by a sharing domain. They are roles, states and events.

Controllable phenomena at a domain interface may be *initiated* ('+') by one of the sharing domains. For example, a state change of the lift shaft sensors is initiated by the lift mechanism, and the pressing of a button is initiated by the intending passenger. Truths at a domain interface are *determined* ('=') by one of the sharing domains. For example, an ordering over strings may be shared by the machine and a strings domain, and is determined by the strings domain. In some cases a domain may exercise *inhibition* ('-') over a controllable phenomenon. For example, the user of a personal computer may initiate a key depression, and the computer may inhibit it by locking the keyboard.

The use of these classifications and control indications is illustrated in subsequent sections.

2.3. Frames, Subproblems and Methods

The use of tightly constrained problem frames can offer two important advantages. The first advantage is that it underpins a repertoire of known and recognised subproblem classes into which realistic problems can be decomposed. The difficulties of unguided problem decomposition are now widely accepted. The traditional top-down process involves decomposing a problem of no recognised class into a number of subproblems also of no recognised class, and continuing recursively until — if the process succeeds — elementary subproblems are recognised at the lowest level. This process can not be expected to produce a good result. Fred Brooks [Brooks 75] sums up his experience in the aphorism: "Plan to throw one away; you will anyhow." The outcome of the process is not a good decomposition; it is a degree of insight into the difficulties of the problem, so that a second complete attempt can then be based — at least in part — on recognised problem characteristics. A sufficient repertoire of problem frames would allow the first decomposition to be guided by a more systematic problem taxonomy.

The second advantage is that a problem frame is, ideally, associated with one or more methods for capturing the problem in full detail and developing a solution. Software development method is chiefly concerned with stipulating the descriptions to be made, the languages to be used, and the large structures within which the descriptions are related. The decomposition of a problem into subproblems of recognised classes allows the appropriate method to be used for each subproblem. Within each frame the method stipulates descriptions of the problem's principal parts, and the particular way in which their large structures specialise the general structure outlined in Section 1.2 above.

A method associated with a tightly constrained problem frame can take advantage of the known characteristics of the problem in several ways. In particular, it can stipulate a less expressive language than might be needed in a more unconstrained problem. For example, a method may stipulate the use of a regular expression language. A problem whose relevant part can not be described by a regular expression may be deemed to fall outside the frame. Or, in some cases, the method may provide a technique for overcoming the difficulty. For example, the description may consist of two or more regular expressions over intersecting alphabets, perhaps with a corresponding problem decomposition.

More fundamental difficulties may demand a further decomposition of the problem. The use of a model within the machine, simulating a part of the world outside it, may be the result of such a decomposition. This difficulty, and others, are illustrated in subsequent sections.

3 Elementary Problem Frames

In this section some elementary problem frames, including partial frames, are outlined and briefly discussed. Composite frames are discussed in the following section.

3.1. Simple Control Frame

The Simple Control frame characterises problems in which the machine is required to control a simple device. An example of such a problem is the control of a simple pair of traffic lights used to ensure one-way traffic in alternating directions over a stretch of road undergoing repair. The problem frame diagram is shown in Figure 2.

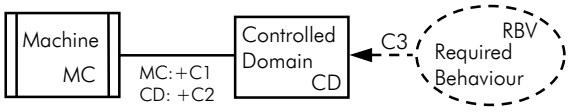

Figure 2
Elementary Frame: Simple Control

In this frame the general form of a software development problem has been elaborated only by more specialised names for the principal parts and by markings on the connecting lines indicating the types and control of the phenomena concerned. The markings indicate that:

- The Required Behaviour RBV is a condition over controllable phenomena (C3).
- The interface between the Machine MC and the Controlled Domain CD consists of two sets of shared controllable phenomena: C1, controlled by MC, and C2, controlled by CD.

In the traffic light control problem, the Machine is the control computer and the Controlled Domain is the pair of traffic lights. The Required Behaviour is a specified sequence of displayed light states: for example (Stop1+Stop2; Stop1+Go2; Stop1+Stop2; Go1+Stop2;)*, each state being required to persist for 50 seconds. The phenomena C3 are the states Stop1, Go1 etc. The phenomena C1 controlled by the Machine are events Red1, Green1, On1, Off1, Red2, Green2, On2 and Off2. The set of phenomena C2 is empty, since the traffic light devices control no phenomena that they share with the control computer.

The relationship between the traffic light states {Stop, Go} and the events {Red, Green, On, Off} is obscure. This obscurity illustrates the need, in capturing any Simple Control problem, to describe the internal properties of the Controlled Domain CD explicitly, even when they are not obscure. In the traffic lights problem is it necessary to describe explicitly how the states Stop and Go are determined by sequences of the Machine-controlled events Red1 etc. This description, of course, is the indicative description $\mathcal{W}$ of the real world discussed in Section 1.2 above.

3.2. Simple Enquiry Frame

The Simple Enquiry frame characterises problems in which the machine is required to answer enquiries about a connected part of the world. An example of such a problem is an experimental laboratory set-up in which voltages are measured at sixteen points and communicated to a computer by A/D devices. The experimenter can enter enquiries asking for the current value at any of the sixteen points, the current highest value, the current average value, and so on.

The problem frame diagram is shown in Figure 3. In this frame the general form of a software development problem has been elaborated not only by more specialised names for the principal parts and by markings on the connecting lines indicating the phenomena concerned, but also by a decomposition of the World into distinct domains. The part of the world about which information is sought is the Real World RW; the source of the enquiries is the Enquirer ENQ; and the responses produced by the Machine are the Responses domain RSP. The single vertical stripe on the RSP domain indicates that it is not given, but is created when the system runs.

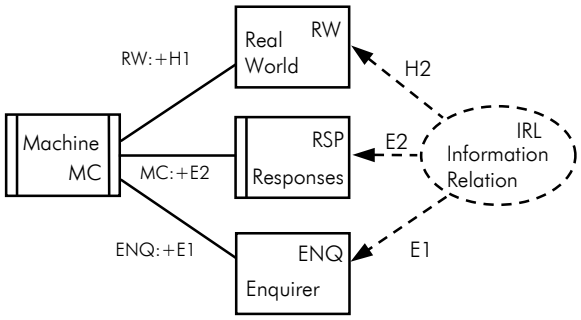

Figure 3
Elementary Frame: Simple Enquiry

The markings indicate that:

- The requirement IRL is a condition over phenomena of RW (H2), events of RSP (E2) and events of ENQ (E1). For example, for the E1 enquiry 'V5', occurring when the voltage at point 5 is 3.2 volts (a state phenomenon in H2), the response event in E2 must be '3.2'.
- The interface between the Machine MC and the Real World RW consists of a set of shared phenomena of any type (H1), controlled by RW. Since RW has no interface at which it shares phenomena controlled by another domain, it is *autonomous*.
- The interface between the Machine MC and the RSP domain consists of a set of shared events (E2), controlled by the Machine. RSP controls no phenomena at any interface, but shares phenomena controlled by MC: it is *passive*.
- The interface between the Machine MC and the ENQ domain consists of a set of shared events (E1), controlled by ENQ. ENQ is *active*.

One aspect of the simplicity of this frame is that the requirement is over the phenomena of RSP and ENQ that those domains share with MC. Further, each response event E2 is a function of the Real World phenomena and of the enquiry event E1 to which it responds, not of preceding or otherwise related enquiry events. More precisely, RSP has no structure more complex than E2*, and ENQ no structure more complex than E1*. There is therefore no need to consider the internal behaviours of those domains.

A source of potential difficulty in this problem frame is that the information to be provided is over phenomena H2 of RW, while the Machine MC has access only to phenomena H1. If the sets H1 and H2 are identical, or if H1 contains H2, there is no difficulty. Otherwise it will be necessary to examine and describe the relationship between H1 and H2. This relationship is embodied in the internal properties and behaviour of RW. In the experimental voltages problem, a trivial relationship presenting no difficulty is that a voltage *v* at point *n* (an element of H2) corresponds to a value *v* in AD register *n* (an element of H1). A more complex relationship would be one in which an enquiry may refer to voltages at past times. This kind of difficulty and its consequences are discussed in subsequent sections.

3.3. Information Display Frame

The Information Display frame characterises problems in which the machine is required to maintain a display about a connected part of the world. An example of such a problem is the provision of a display in a hotel lobby showing the current positions of the hotel lifts. The problem frame diagram is shown in Figure 4.

The Display Machine MC is connected to the Real World RW about which information is to be displayed by an interface of shared controllable phenomena controlled by RW. In the lift problem these phenomena may be the states of sensors in the lift shafts and button-press events.

The Information Display domain DIS is controlled by the Machine through the shared phenomena C2. The requirement DRL stipulates the states S2 of the Display domain that must correspond to states S1 of the Real World. For example, in the lift problem DRL may stipulate that whenever a lift is ascending an Up arrow should be illuminated in the column corresponding to that lift.

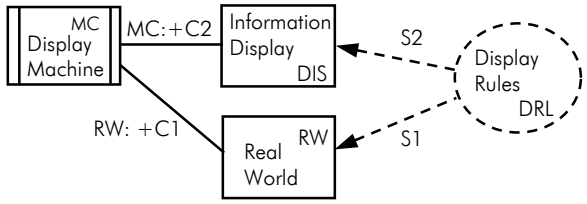

Figure 4
Elementary Frame: Information Display

In an Information Display problem it is necessary to describe the internal properties of both DIS and RW. The relationship between the phenomena C2 and their effects on the states S2 of the Display must be examined and described. The properties of RW are likely to be more complex. For example, in the lift problem it may be that the Display must show outstanding requests: an outstanding request is a member of S1. The phenomena C1 shared by RW with the Machine may be only button-press events and lift-shaft sensor states. It will then be a non-trivial task to determine how the Machine should calculate S1 from C1. We will return to this topic in a later section.

3.4. Simple Workpieces Frame

A Simple Workpieces problem requires the provision of a tool for constructing and editing intangible artifacts such as texts or graphics. The artifacts are restricted to extremely simple objects that can be edited 'blind' as shown in the problem frame in Figure 5.

The machine to be built is the Tool TL. The Workpieces WP are constructed by the Tool and are entirely *contained* within it, the containment being shown by the heavy dot on the connecting line representing the interface. Containment means that all the phenomena of the contained domain are shared with the containing domain; in the Workpieces frame shown above we can infer that WP has no phenomena other than E2 and S1.

The Operation Requests domain is a stream of events E1, each requesting an operation on a workpiece. Because some requests — for example, a request to delete an

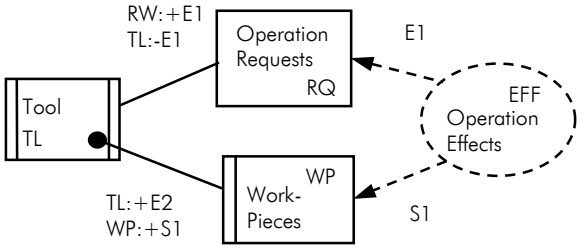

Figure 5
Elementary Frame: Simple Workpieces

element from a non-existent workpiece — are unacceptable, the Tool can inhibit E1 events. Inhibition by the Tool is, of course, quite different from returning a null result to an accepted request — for example, a request to change all occurrences of '1' to '2' in a text containing no occurrence of '1'. Inhibition might be implemented by ignoring the E1 input except to respond with a bleep.

In this simple frame, requests are mutually independent: the meaning of a request does not depend on any other request. Similarly, the workpieces are mutually independent: no operation involves more than one workpiece. The requirement stipulates the effect of each operation in terms of the preceding and resulting states of the affected workpiece.

The Workpieces domain WP is *Inert Reactive* at its interface with the Tool. That is to say, it responds to each event in E2 by a (possibly null) state change in S1, and immediately returns to quiescence. WP never initiates state changes in S1 except in response to events in E2.

3.5. Methods and Descriptions

The characteristics of the principal parts of a problem frame and of their interfaces govern the choice of method, both for capturing the problem in full detail and for developing a solution.

For example, the inert reactive nature of the Workpieces domain WP in the Simple Workpieces frame allows WP to be described as a set of instances of an abstract data type: the events E2 are the operations of the type, and the states S1 define the data representation. In developing a solution this description of WP may be refined into the definition of an object class. Because WP is inert reactive, and not active, the object class definition needs no 'run' or 'live' method: its methods are all externally invoked and executed sequentially. Both the problem statement and the solution therefore avoid the potential complexities of concurrency.

The domain RQ, of requests for operations on Workpieces, is active and autonomous. The requirement EFF therefore can not constrain RQ in any way although it constrains the relationship between RQ events E1 and WP states S1: EFF must be satisfied solely by constraints on the behaviour of TL and WP. Further, the domain RQ has the trivial structure *request**. Hence RQ needs only an indicative description of a simple kind.

In the Simple Control frame discussed in Section 3.1 above, the properties of the Controlled Domain CD and of its interface with the Machine MC will govern the kinds of description needed to capture the problem and to develop a solution. In the very simple case of the traffic lights problem the following descriptions will be appropriate:

- The requirement $\mathcal{R}$ is described as a finite-state machine in which the states denote the pairs of lights showing in the two sets of lights and the transitions denote timeouts for the delays.
- The world description $\mathcal{W}$ is a finite-state machine in which the states denote the single light showing in one set of lights and the transitions denote the events shared with the control computer. Additional states may be needed if more than one event must occur for the light to change. The same finite-state machine describes the properties of both sets of lights.
- The specification S is given as a Mealy machine in which transitions are timeouts for the delays and outputs on the transitions are events shared with the lights. The states are not significant: in a diagrammatic representation of the Mealy machine they need not be named; in a transition-list representation their names are bound variables.

More complex variants of the Simple Control frame may demand more expressive languages. But, as a rule, a problem that demands a more elaborate descriptive structure is not a problem of the class characterised by the frame.

4 Difficulties and Problem Frames

Possession of a set of close-fitting frames and associated methods arms the developer to recognise and deal with problems of the corresponding classes. A method should not be used if the problem does not fit its frame. Construction of a compiler, for example, can not be treated as a Simple Control problem: the compiler world must be structured into at least two domains — the input source program and the output object program; further, the output object program is not given but must be created when the system runs. Similarly, developing a controller for a chemical plant is not a Simple Workpieces problem. The plant evidently does not have the characteristics stipulated for the Operation Requests domain, and it is certainly not inert: even in the absence of shared events initiated by the control computer liquids and gases will continue to flow, to condense or evaporate, and to rise or fall in temperature. Development of an avionics system is not an Information Display problem: the aircraft is neither autonomous like the Real World domain nor inert reactive like the Information Display domain. Developers of an avionics system who mistakenly try to use the Information Display frame will find that the associated methods are quite unable to handle many of their most important concerns. (Regrettably, such misfits between problem and method are often ignored by proponents of development methods that claim very wide, or even universal, validity.)

Recognition of a frame misfit may simply lead to the selection of another available frame. But sometimes it will lead to recognition of a difficulty of a known kind to which the solution may be a standard elaboration of the misfitting frame or a standard decomposition into two or more frames. In this section some simple illustrations are given.

4.1. Flexible Requirements Difficulty

A common difficulty that can occur in almost any problem frame is a need for flexibility, when a fixed requirement is inappropriate. In the traffic lights problem, for example, the sequencing of light states and the delay for each transition are fixed in the requirement; they will then become fixed in the control computer's program. It may be necessary to arrange for the sequences and delays to be conveniently specified by insertion of a floppy disk or setting switches on a console attached to the computer. The disk or console then becomes a distinct domain, playing the part of a description in the problem frame as shown in Figure 6.

The oval outline of the Sequence Description domain SQD indicates that it plays the part of a description in the problem; the outline is solid — unlike the outline of the requirement — because it has a tangible embodiment in the problem. The requirement SQI is no longer a requirement to evoke a particular sequence of lights, but rather to produce a sequence corresponding to the interpretation of the SQD domain. The phenomena marked as "G1", "50s" &c in the diagram are the syntactic elements of the description SQD. They are shared with the Control Computer CTL, which has access to the Sequence Description — as it must if it is to satisfy the requirement. The marking 'SQD: = "G1", "50s", &c' on the interface between CTL and SQI indicates that these non-controllable phenomena are *determined* by SQD, not by the machine CTL.

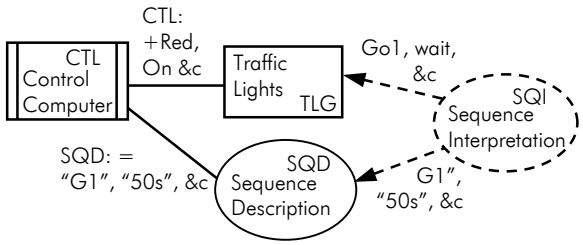

Figure 6
Flexible Traffic Lights Problem

The difficulty recognised here, and its solution, are absolutely standard in software development: part of what might have been treated as compiled program is instead treated as data, and the compiled program then becomes responsible for interpreting that data.

We are assuming here that the Sequence Description is a given part of the problem, fixed for any instance of the system. That is, we regard its construction as falling outside the problem context. The Controller CTL is capable of interpreting any instance of SQD that satisfies the syntactic and semantic rules of that problem part, but it is not responsible for constructing that instance. If instead the Sequence Description is to be created by the operator of the system, the problem fits a more complex, composite, frame. That frame, Simple Control Under Operator's Regime, is discussed in a later section. The frame shown in Figure 6 is then only a partial frame: it addresses only one subproblem in the original problem.

4.2. Identities Difficulty

An important and common class of difficulty is the *identities* difficulty. Whenever a domain contains multiple instances of entities of the same type that must be distinguished by the machine, the mechanism by which each instance is distinguished and identified becomes of interest. It can also become a source of major difficulty. The well-known British Midland 737 crash at Kegworth [Neumann 95] occurred because the engine safety-control system was 'cross-wired', causing the pilot to shut off the starboard engine in response to smoke and vibration in the port engine. Subsequent inspection showed that many 737s were cross-wired in this way.

A tiny manifestation of this difficulty appears in the traffic lights problem. There are two sets of lights to be distinguished by the machine. In the simplest version of the problem the distinction is actually not necessary, because both sets of lights are treated identically after system startup. But as soon as the sequences for the two sets must differ, because traffic in one direction is heavier or slower than in the other direction, the distinction may become important.

A substantial — and life-threatening — version of the difficulty occurs in the well-known Patient Monitoring problem. Patients have names; they are in hospital beds; they are attached to analogue devices; the devices are plugged into ports on the monitoring computer; medical staff specify periods and ranges for monitoring each patient individually, referring to them by their names. The difficulty is evident: to satisfy the requirement, the machine's reading of a patient's pulse rate or temperature or blood pressure at a port must be associated with the correct patient. This association is mediated by mappings between patients' names and patients, patients and devices, and devices and ports. Dealing correctly with these mappings is a vital and substantial aspect of the whole problem.

Generally, solution of an identities difficulty requires the introduction of one or more explicit Mapping domains into the problem frame. In some cases a mapping may be degenerate: the two sets of traffic lights, for example, may be visibly labelled '1' and '2', and the operator instructed to plug them into ports 1 and 2 of the Control Computer. It is then unnecessary to treat the mapping as a distinct domain; it is enough to distinguish it as an element in other descriptions in the development.

In the worst cases — and the Patient Monitoring problem is such a case — a Mapping domain is dynamic. The developers might reasonably ignore the case in which a patient changes her name while under monitoring, leaving the correct handling of this kind of change to the hospital staff. But certainly patients leave the hospital and new patients arrive; additional monitoring, and hence additional devices, may become necessary for an existing patient; and analogue devices could become unplugged from the computer and plugged into the same — or different — ports. The developers must build a system that deals correctly with all of these events.

Introduction of an explicit Mapping domain raises similar issues to the introduction of a flexible requirement domain such as SQD in the traffic lights problem. If the Mapping domain is not a given domain in the problem context it must be created by the action of the system; the creation task is then a separate subproblem in its own right.

4.3. Connection Difficulty & Model Domains

In many problems the available connection between the machine and a domain of the world is not immediately adequate: the shared phenomena are deficient, or are displaced in time, in relation to the requirement.

This is, of course, one fundamental reason why it is necessary to describe the world at all. We must describe the properties of the traffic light sets because the requirement is over the sequence of lights showing and the machine can control this only indirectly by causing signal events. The indicative properties of the traffic light sets guarantee that appropriate sequences of signal events will evoke the required sequences of lights. The specification S can then be written in terms of the signal events accessible to, and controllable by, the machine. Similarly, in the problem of responding to queries about the experimental voltages, the properties of the experimental set-up and the AD devices allow questions about real voltage values to be answered by inspection of integer registers accessible to the machine.

A connection difficulty arises when even the most careful description and exploitation of the indicative properties of a domain in the world are not enough. It is then necessary to find an implementation of some of the 'data freedom' facilities discussed in [Balzer 82]. The standard technique is to create a *model* of the domain inside — or readily accessible to — the machine. The machine can then derive from the model information that it can not obtain directly from the modelled domain. For example, in a more demanding version of the experimental voltages problem in which a required response is the highest average voltage achieved at a specified point over any previous period of ten consecutive seconds, the machine must create, and continually maintain, a dynamic model of the domain of voltages. The required response can then be calculated from data available in the model.

Creation of such a model becomes a problem in its own right, characterised by a partial problem frame such as that shown in Figure 7.

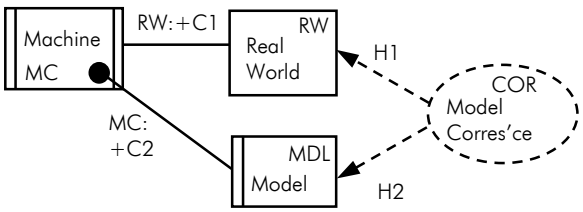

Figure 7
Partial Frame: Dynamic Model

The frame shown is for creation of a dynamic model. The Real World RW is dynamic and autonomously active, and controls phenomena C1 shared with the Machine MC. The Model MDL is passive; it is contained in the Machine MC and is created by the Machine when the system runs. The requirement is that the Model MDL should correspond to the Real World in respects specified by COR. Essentially the correspondence is an isomorphism between individuals

and relations in the two domains, possibly augmented by additional phenomena in the Model.

The central concerns in the problem are selecting the Machine phenomena to be used in the Model — that is, choosing representation and abstraction functions — and capturing and exploiting knowledge of the properties of the Real World to overcome any mismatch between C1 and H1. Suppose, for example, that the Real World is wheeled traffic on a road segment, the purpose of the model being to allow traffic density to be monitored. Suppose also that C1 are states of sensor tubes laid across the road to detect the passage of traffic, and that the set H1 consists of events such as 'motor car passes', 'motor cycle passes', 'light lorry passes' and 'heavy lorry passes'. Then the Real World properties to be captured in the indicative world description W are the relationships between the distinct kinds of traffic event in H1 and the accompanying distinct patterns of the sensor states in C1.

The problem frame for constructing a static model is sometimes similar, *mutatis mutandis*, to the Dynamic Model frame. However, it is usually more elaborate, because static domains are often isolated from the Machine. It is then necessary to introduce a human *Informant* to convey the Real World phenomena to the Machine, and the Informant becomes an additional domain in the problem frame. In a problem concerned with the scheduling and control of a railway, for example, the system must have access to the details of the track layout. This will usually be achieved by manual entry of the layout information, and the manual entry process may be seen as execution of a problem fitting the partial frame Static Model with Informant.

5 Composite Frames

If we were to restrict our repertoire of problem frames to elementary and partial elementary frames, it would be necessary to decompose each realistic problem into a structure of subproblems, each small and simple enough to fit one such elementary frame. This in itself would be disadvantageous: we would be restricting ourselves, in the problem sphere, to the equivalent of a rather low-level programming language in the solution sphere.

More important, we would be forgoing the opportunity to build a repository of experience about problem and solution composition. A substantial part of the knowledge and experience of established engineering branches is concerned with putting parts together to make a complete product. Automobile engineering is not just about engines, gearboxes, steering, differentials and other components of a motor car: it is also, crucially, about their composition into a well-designed whole. One of the most important advances was the recognition that the availability of large powerful presses for sheet steel permitted the integration of the chassis with the body, two components that had previously been regarded as separate. This kind of advance is a nourishing fruit of specialisation.

Being concerned with the residue of non-specialised problems that is the subject matter of Software Engineering, we can go only a short way towards identifying composite problem frames. If we go very far we will give birth to new specialities that will immediately leave their parents' house. Unselfishly, we must go as far as we can. In this section we identify and discuss some small composite frames and some characteristic difficulties they can raise.

5.1. Simple Information System Frames

We use the term *Simple Information System* for a system in which the primary decomposition is into the construction of a model of a Real World domain and the use of the model to provide information about the Real World.

Figure 8 shows the undecomposed problem frame for a static information system with a human informant.

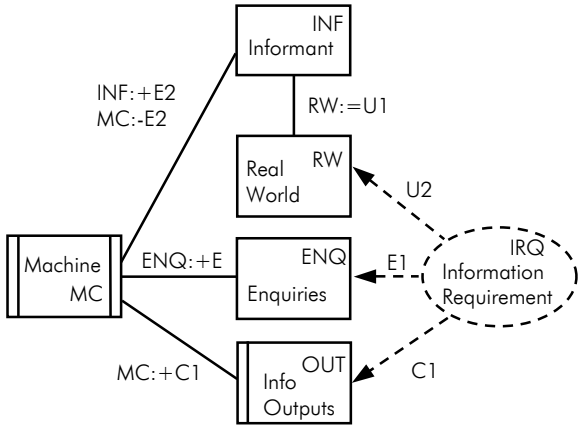

Figure 8
Composite Frame: Static IS with Informant

An example of such a problem is the answering of queries about a text such as Tyndale's Bible. The decomposition into the two constituent subproblems is shown in Figure 9.

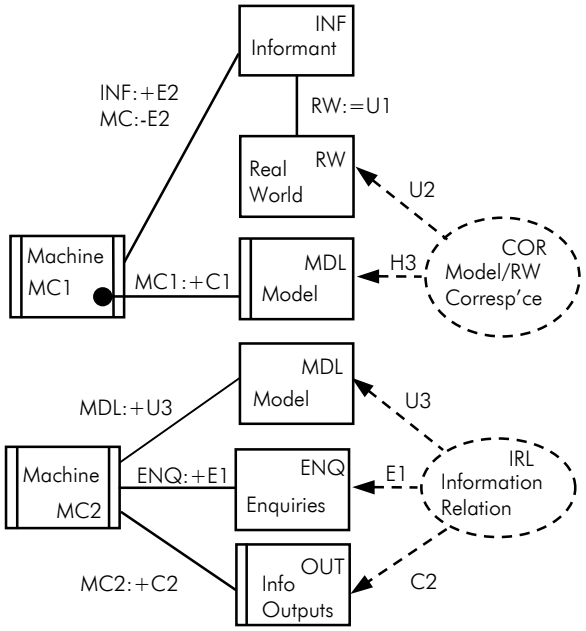

Figure 9
Decomposition of
Static IS Frame with Informant

The original requirement IRQ, over RW, ENQ and OUT, is satisfied by the obvious composition of the subrequirements COR, over RW and MDL, and IRL, over MDL, ENQ and

OUT. The two machine domains, MC1 and MC2, both contain the same Model domain MDL. They must therefore share at least the phenomena of MDL, and in practice this means that they must both be implemented in the same computer. If we choose to make MDL a domain external to the machine, such as a removable disk, the two machines can then be implemented in different computers.

The obvious scheduling of the two machines is to run MC1 to completion before running MC2. However, it may be possible and useful to overlap their executions. In the Bible problem, for example, if the transcription of the text takes several months it will be useful to provide responses to queries of certain kinds on the basis of a model restricted to the books already transcribed.

5.2. Simple Control Under Operator Regime

In Section 4.1 the Traffic Lights problem was elaborated to provide flexibility in the required sequencing of lights. A common composite frame in control programs is shown, decomposed, in Figure 10.

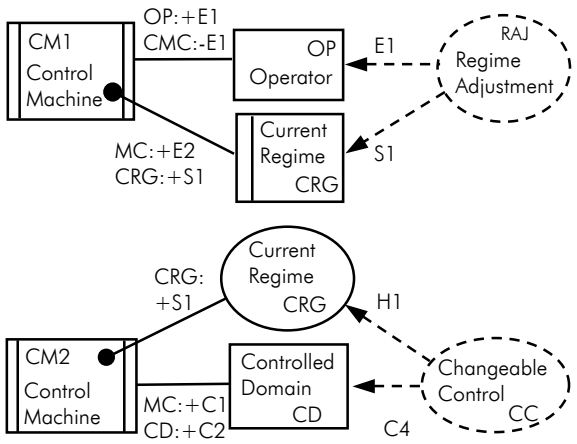

Figure 10
Decomposition of Simple Control
Frame Under Operator's Regime

The first subproblem, in which the operator constructs the Current Regime, fits the Workpieces frame; the second fits the Simple Control frame with the Current Regime as an explicit description of all or part of the requirement. The Current Regime domain is, as will often be the case, of different types in the different subproblems. The world is not typed — let alone strongly typed.

A major concern in the composition of the two subroblems here is the scheduling of the machines CM1 and CM2. In the simplest case CM1 can be run to completion before CM2 is run. In the Traffic Lights problem this means that the operator must set up the regime before using the lights to control the traffic, and can not then alter it. However, it may be necessary to alter the regime during a period of traffic control — for example, because the regime must be altered to handle morning and evening rush hours; execution of the machines must then be interleaved in some way.

There are several possibilities for this interleaving. For example:

- Machine CM2 is halted from the start to the finish of the creation of CRG by CM1. In the traffic lights problem this is unlikely to be acceptable because traffic in at least one direction must then wait while the operator changes the regime. This is the coarsest grain of interleaving.
- Machine CM2 is halted while the operator alters CRG from one valid state to another, and machine CM1 is halted while CM2 progesses through one cycle of CRG. This scheme gives a finer granularity of interleaving while maintaining the invariant 'CRG is a valid regime'.
- Two copies of CRG are used. Machine CM2 runs on one copy concurrently with the creation of the other copy by CM1. At the end of the creation process CM2 switches to the new copy, and the other is now available for updating by CM1.

Describing and managing the last of these possibilities is a non-trivial problem in itself, justifying its own problem frame. In this subproblem a third machine treats CM1 and CM2 as parts of the world, controlling their behaviour in relation to controllable phenomena of other domains.

5.3. Visible Workpieces Frame

The Simple Workpieces frame discussed in Section 3.4 above is very unrealistic. It is hard, though not impossible, to think of a practical problem, however small, that could fit it. The lack of realism lies in the absence of any feedback to the source of the Operation Requests: the operator must perform the editing operations 'blind', without seeing any representation of the object being edited. A slightly more realistic version is the Visible Workpieces frame shown in Figure 11.

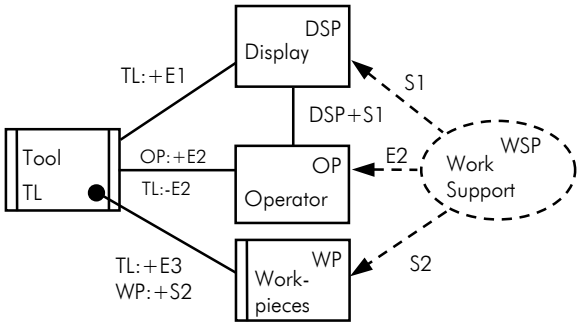

Figure 11
Composite Frame: Visible Workpieces

The feedback is provided by the Display domain DSP, whose state is the set S1 of phenomena shared with the Operator. The Work Support requirement WSP stipulates not only the effects of the operations requested by the operator but also the visible state of the Display in relation to the states of the Workpieces. This visible state of the display is not shared by the Tool TL, although it is indirectly controlled by TL through its control of the events E1. If the relationship between the Display state and the Workpieces state is not simple, it may be necessary for the Tool to create and maintain its own model of the state of the

Display. (This would, of course, be a third subproblem in addition to the two shown in Figure 12.)

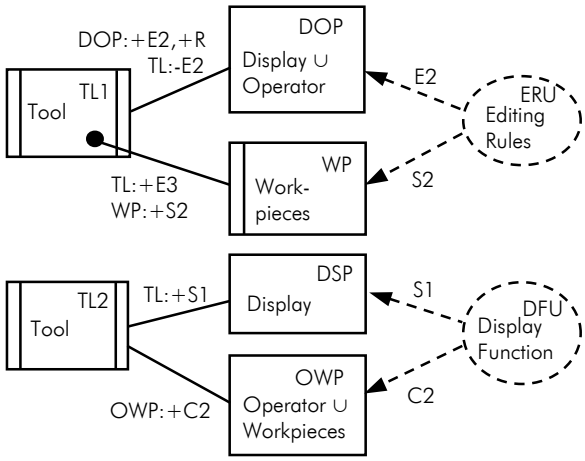

Figure 12
Decomposition of Visible Workpieces Frame

The first subproblem is a Simple Workpieces problem, in which the source of operation requests is the union of the Display and Operator domains: the initiation of a request event (E2) is controlled by the Operator, but the selection of the operands that play roles (R) in the event is controlled by the Display. The second subproblem is an Information Display problem, in which the Real World about which information is to be displayed is the union of the Operator and Workpieces domains.

For the present discussion, the most important point about the Visible Workpieces frame is that it exemplifies the advances that can be made by even a modest degreee of specialisation. The problem class is, of course, the class addressed by the MVC object pattern: in MVC, roughly, the Workpieces are the Model, the Display is the View, and the Tool is the Controller. It is notable that the MVC pattern has received much attention — because it is a common fundamental component of GUI systems — and undergone much discussion, criticism and improvement [Buschmann 96] in the ten years or so since its introduction [Krasner 88]. This process is very similar to the improvement that takes place in products like motor cars; only a specialised focus on a particular problem class allows a sufficient concentration of attention for significant improvement to take place. The problem frame of Figure 12 is only a first crude characterisation of the problem class.

6 Decomposing a More Realistic Problem

A realistic problem in Software Engineering will always demand a fresh decomposition. The most to be hoped for by a developer who commands a good repertoire of elementary and composite frames is that meeting the challenge of decomposition will be eased by the ready recognition of familiar subproblems. Each subproblem class has an associated repertoire of potential characteristic difficulties: checking for each such difficulty can lead readily to the recognition of further subproblems. The task of composing the solution elements, at least above the level of the well-explored composite frames, will be unique to the problem in hand.

6.1. The Package Router Problem

As an example of a nearly realistic problem we take the problem of controlling a Package Router [Swartout 82]. The treatment of this problem here is based on the treatment in [Jackson 96]. Here is the problem as described in [Swartout 82], translated from the original German version of Hommel:

> "The package router is a system for distributing packages into destination bins. The packages arrive at a source station, which is connected to the bins via a series of pipes. A single pipe leaves the source station. The pipes are linked together by two-position switches. A switch enables a package sliding down its input pipe to be directed to either of its two output pipes. There is a unique path from the source station to any particular bin.
>
> "Packages arriving at the source station are scanned by a reading device which determines a destination bin for the package. The package is then allowed to slide down the pipe leaving the source station. The package router must set its switches ahead of each package sliding through the pipes so that each package is routed to the bin determined for it by the source station.
>
> "After a package's destination has been determined, it is delayed for a fixed time before being released into the first pipe. This is done to prevent packages from following one another so closely that a switch cannot be reset between successive packages when necessary. However, if a package's destination is the same as that of the package which preceded it through the source station, it is not delayed, since there will be no need to reset switches between the two packages.
>
> "There will generally be many packages sliding down the pipes at once. The packages slide at different and unpredictable speeds, so it is impossible to calculate when a given package will reach a particular switch. However, the switches contain sensors strategically placed at their entries and exits to detect the packages.
>
> "The sensors are placed in such a way that it is safe to change a switch setting if and only if no packages are present between the entry sensor of a switch and either of its exit sensors. The pipes are bent at the sensor locations in such a way that the sensors are guaranteed to detect a separation between two packages, no matter how closely they follow one another.
>
> "Due to the unpredictable sliding characteristics of the packages, it is possible, in spite of the source station delay, that packages will get so close together that it is not possible to reset a switch in time to properly route a package. Misrouted packages may be routed to any bin, but must not cause the misrouting of other packages. The bins too have sensors located at their entry, and upon arrival of a misrouted package at a wrong bin, the routing machine is to signal that package's intended destination bin and the bin it actually reached".

6.2. Recognising Some Subproblems

Initially this problem appears to be essentially a control problem. The machine must flip the switches so that the packages arrive at their proper destinations. The switch

must be flipped when a package passes the sensor at the bottom of the pipe leading into the switch.

Brief consideration of the phenomena concerned immediately reveals a connection difficulty. The package destination is read at the source station, but is no longer available when the package passes the sensor. The shared phenomena between the machine and each sensor *s* are no more than the states SensorOpen(*s*) and SensorClosed(*s*): the package causing the state change is anonymous, and its destination bin is unknown. This connection difficulty is soluble by a dynamic model. Since the packages can not overtake one another, the state of the packages and pipes can be regarded as a set of queues. The package arriving at a sensor above a switch is the package at the head of the queue in the pipe to which the sensor is attached. Package destinations, read at the source station, are attached to the package objects in this queue model. When a switch is to be flipped the controlling machine consults the queue model to identify the package destination and hence the required switch setting.

There is a further connection difficulty. The switch to be set is determined by the router topology — the positioning of sensors on pipes, and the pipe and switch layout. Evidently a static model is needed here. Augmented with bins, the same model will allow the required setting of the determined switch to be chosen according to the route from the switch to the bin.

The switches are attached to ports of the controlling machine. This is the standard form of an Identities problem. An exactly similar problem is present for the sensors.

Signalling arrival of a misrouted package at a wrong bin is, of course, a Simple Information Display problem.

6.3. Another Concern

In discussing software engineering problems it is always tempting to look for the sixpence under the light. We naturally asume that the most important problem aspects are those for which we have suitable techniques ready in our toolkit. The list of recognisable subproblems in the previous section illustrates the point. An important concern in operating the package router in practice will undoubtedly be the behaviour of the router and its controller when a package becomes stuck in a switch. So far, this concern has been ignored because our repertoire contains no appropriate problem frame.

In fact, this concern — handling a malfunctioning physical world — is very common in problems of many kinds. How it might be captured in a problem frame is left as an exercise for the reader.

7 Summary

This informal paper has sketched a selection of elementary, partial and composite problem frames. No claim, of course, is made that the selection is complete or canonical, or even that any particular frame presented truly characterises a set of problems that are best considered as a class: other classifications are surely possible. But it is claimed that the approach is valuable in at least these respects:

- It is useful to consider problems largely — though not entirely — independently of their putative solutions.
- Software Engineering problems are located in the world, and their analysis and structuring is primarily an analysis and structuring of the world, not of the machine.
- The classification of phenomena and the consideration of their control is a central ingredient in problem analysis. Mathematical abstractions alone are not enough.
- A repertoire of recognised problem classes, with associated characteristic difficulties and solution methods, provides an important structure for the discipline of Software Engineering. Within this structure specialisations can emerge and achieve incremental advances that can not be achieved by attacks on a more abstract or a broader front.

In short, it is claimed that problem frames are a contribution to making Software Engineering more like the established branches of engineering that it aspires to emulate.

Acknowledgements

This paper has been much improved by discussion with Daniel Jackson.

References

[Balzer 82] Robert M Balzer, Neil M Goldman and David S Wile; Operational Specification as the Basis for Rapid Prototyping; ACM Sigsoft SE Notes 7, 5, December 1982, pages 3-16; reprinted in New Paradigms for Software Development; W W Agresti; IEEE Tutorial Text, IEEE Computer Society Press, 1986.

[Bass 98] Len Bass, Paul Clements and Rick Kazman; Software Architecture in Practice; Addison-Wesley 1998.

[Bhargavan 98] Karthikeyan Bhargavan, Carl A Gunter, Elsa L Gunter, Michael Jackson, Davor Obradovic and Pamela Zave; The Village Telephone System: A Case Study in Formal Software Engineering; in Proc 11th International Conference on Theorem Proving in Higher-Order Logics TPHOLs98.

[Brooks 75] Frederick P Brooks Jr; The Mythical Man-month: Essays on Software Engineering; Addison-Wesley 1975.

[Buschmann 96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stahl; Pattern-Oriented Software Architecture: A System of Patterns; John Wiley 1996.

[Buxton 70] J N Buxton and B Randell eds; Software Engineering Techniques; Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Rome, Italy, 27th to 31st October 1969; NATO April 1970.

[Gamma 94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides; Design Patterns: Elements of Object-Oriented Software; Addison-Wesley 1994.

[Jackson 93] Michael Jackson and Pamela Zave; Domain Descriptions; in Proceedings of the IEEE International Symposium on Requirements Engineering, pages 56-64; IEEE CS Press, 1993.

[Jackson 95] Michael Jackson; Software Requirements & Specifications: a lexicon of practice, principles and prejudices; Addison-Wesley and ACM Press 1995.

[Jackson 96] Daniel Jackson and Michael Jackson; Problem Decomposition for Reuse; Software Engineering Journal 11,1 pages 19-30, January 1996.

[Krasner 88] G E Krasner and S T Pope; A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80; Journal of Object-Oriented Programming 1, 3, August/September 1988, pages 26-49.

[Naur 69] Peter Naur and Brian Randell eds; Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968; NATO January 1969.

[Neumann 95] Peter G Neumann; Computer-Related Risks; Addison-Wesley 1995, pages 44-45.

[Parnas 97] D L Parnas; Software Engineering: An Unconsummated Marriage; CACM 40, 9, September 1997, page 128.

[Polya 57] G Polya; How To Solve It; Princeton University Press, 2nd Edition 1957.

[Pree 97] Wolfgang Pree; Design Patterns for Object-Oriented Software Development; Addison-Wesley 1995.

[Shaw 96] Mary Shaw and David Garlan; Software Architecture: Perspectives on an Emerging Discipline; Prentice-Hall 1996.

[Stevens 74] W P Stevens, G J Myers, and L L Constantine; Structured Design; IBM Systems Journal 13, 2, 1974, pages 115-139; reprinted in Tutorial on Software Design Techniques, Peter Freeman and Anthony I Wasserman eds; IEEE Computer Society Press, 4th edition 1983.

[Swartout 82] William Swartout and Robert Balzer; On the Inevitable Intertwining of Specification and Implementation; CACM 25, 7, July 1982, pages 438-440.

[Zave 93] Pamela Zave and Michael Jackson; Conjunction as Composition; ACM Transactions on Software Engineering and Methodology, October 1993, pages 379-411.