In Proceedings of NATO Summer School, Marktoberdorf, August 2000 (in publication)

Problem Analysis and Structure

Michael JACKSON
101 Hamilton Terrace, London NW8 9QY, England
and
AT&T Research, 180 Park Avenue, Florham Park NJ 07932, USA

Abstract. An approach to problem analysis is presented in which problems are decomposed into subproblems of recognised classes. These classes can be captured by *problem frames*, which identify domain structures and interfaces in the problem world; domains themselves can be classified into lexical, causal and biddable domains. Each problem frame has a basic associated *frame concern* that must be addressed in analysing the problem. Additional concerns arise from more general considerations, including the characteristics of the problem domains and the composition relationships among the subproblems.

1. Introduction

Traditionally, thinking and research in software development has focused on solutions: on programs and on various abstractions that may be useful in designing and writing program texts. We have paid little or no attention to the problems that those programs are intended to solve. Even methods and approaches that claim the title of 'problem analysis' usually prove, on closer inspection, to deal entirely with putative or outline solutions. The problem to be solved is neither stated in full detail nor explicitly analysed; the reader must infer the problem from its solution.

This solution-oriented approach may work well in a field where the problems are all well known and have been thoroughly described, classified and investigated — where innovation lies chiefly in devising new solutions to old problems. But software development is not such a field. The versatility of computers and their rapid pace of evolution present us with a constantly changing repertoire of problems in whose solution software has a central role. As a result, our field is underdeveloped in crucial respects. In particular, the repeated calls for professionalisation and for the establishment of a corpus of core software engineering knowledge are symptoms of a broad failure to identify what practising software developers should know if they are to be competent to tackle the problems of the many different application areas.

The central part of this paper sketches an approach to problem analysis and structuring that aims to avoid the magnetic attraction of solution-orientation. The approach is based on the idea of a *problem frame* [1-3]. A problem frame characterises a class of elementary simplified problems that commonly occur as subproblems of larger, realistic, problems. The intention is to analyse realistic problems by decomposing them into constituent subproblems that correspond to known problem frames. This analysis guides the decomposition, gives warning of the concerns and difficulties that are likely to arise, and provides a context in which previously captured experience can be effectively exploited.

2. The Problem and Its World

Some problems, including most of the standard examples found in computer science texts, are abstract in a mathematical sense: they do not partake of the physical nature of the world. Factorising large integers, finding strongly connected components of graphs, and playing chess are examples of such problems. But most problems are located in the physical world. Such problems include controlling lifts, switching telephone calls, controlling the brakes of a car, bank accounting, managing theatre seat reservations, controlling a VCR and administering a library. In all these cases the effectiveness of a solution is to be evaluated in the physical world outside the computer. The problem is located in the world; the computer, executing our program text, is the core of the solution.

2.1 Phenomena

Because problems are located in the world, problem analysis must be concerned with the world and its phenomena. We need a phenomenology that has nothing to do with programming languages or object interaction, but everything to do with the physical world. An appropriate phenomenology may include:

- **entities**, which are mutable individuals such as cars and people;
- **events**, which are recognised as individuals;
- **values**, which are immutable individuals such as integers and strings;
- **states**, which are time-changing relations over non-event individuals;
- **truths**, which are unchanging relations over non-event individuals; and
- **roles**, which are the participation of individuals in events.

In particular, it is useful to treat roles as distinct phenomena. One reason is that the alternative — encoding roles as positional arguments of event expressions — leads to difficult and premature choices. In a *DepositCash* event, the account credited and the amount of cash deposited are obviously necessary arguments. But it's hard to say whether the person making the deposit and the ATM at which the deposit is made are also necessary. Roles allow these choices to be deferred, and to be made incrementally and independently.

Among the six kinds of phenomena it is useful to distinguish the class of *controllable* phenomena — events, state changes and roles — that occur on the initiative of one part of the world rather than another. For example, a keystroke is an event in which the user and the keyboard both participate, but it is controlled by the user: the user controls both the event and the role that is the participation of a particular key. In a disk read operation, the reader controls the event occurrence and the participation of an individual input buffer; but the disk controls the participation of the particular record value that is returned. This separation of the control of events and their roles is another reason for recognising roles as distinct phenomena.

Values and truths form the class of *symbolic* phenomena. Symbolic phenomena are carried by controllable phenomena. For example, the data content of a disk file is carried by magnetic encoding on sectors of the disk, and accessed by read and write events in which the data values play roles.

2.2 Domains

For purposes of problem analysis it is natural to recognise distinct parts of the world; we will call them *domains*. A domain can be thought of as a collection of related phenomena; the kinds of phenomena in a domain and the relationships among them constitute the domain properties. Domains may share phenomena: indeed, the only way two domains can interact is by an interface of shared phenomena. For example, a lift and its passengers interact because both the entry of a passenger into the lift car and the pressing of a floor request button are events shared by the lift domain and the passenger domain. The control

computer interacts with the lift because switching on the lift motor is an event shared by the computer and the lift domain (and controlled by the computer). If interaction between two domains is mediated by a channel, and the channel introduces delays or errors or noise that preclude the interaction from being treated as one of shared phenomena, then the channel itself must be treated as another domain, interacting by shared phenomena with each of the original two domains.

The most fundamental distinction for problem analysis is between the machine domain — the computer and its software — and the problem domain — the world where the problem is located and the quality of its solution will be evaluated. These two domains must share phenomena if the problem is to be soluble.

2.3 Descriptions

In very general terms, the process of problem analysis is concerned with these descriptions of relationships among the phenomena of the problem domain:

- **The requirement.** This is a description of properties that the domain does not possess intrinsically but are desired by the sponsor of the development. It will be the machine's task to endow the problem domain with those properties. For example: the property that the lift comes when a button is pressed; that the lift does not stop at a floor for which no request is outstanding; that the lift doors never open except when the lift car is stationary at a floor; and so on.
- **The domain properties.** This is a description of the properties that the domain possesses intrinsically, regardless of the behaviour of the machine. For example: the property that from floor *n* the lift can go only to floor *n+1* or *n-1*; that the sensor at floor *n* is on when the lift car is within 6 inches of the home position at that floor; that if the motor is set *on* and *up* the lift will start to rise; and so on.
- **The machine specification.** This is a description of the desired behaviour of the machine at its interface with the problem domain. For example: when button *n* is pressed in certain circumstances, the machine must set the lift motor polarity to up and set motor power to on. Although this is a description of machine behaviour, it is expressed entirely in terms of problem domain phenomena: the shared phenomena at its interface with the machine belong, of course, both to the problem domain and to the machine domain.

The formal criterion for success in a development is the demonstration of an implication among these descriptions:

$$\textit{machine specification} \wedge \textit{domain properties} \Rightarrow \textit{requirement}$$

If the machine behaves as specified and the domain has the described intrinsic properties, then the requirement will be satisfied. (A more rigorous account of the relationship among the three descriptions is given in [4].)

In the lift control problem this means: if the machine detects button presses and sensor states and operates the lift and door motors, all in accordance with the specification, and if the lift position and behaviour are related to the sensor states and motor settings as described in the domain properties description, then the lift will come when the button is pressed. Essentially, the intrinsic domain properties bridge the gap between the *requirement phenomena* — those mentioned in the requirement — and the *specification phenomena* — those directly accessible to the machine at its external interface to the world. This gap exists in almost every realistic problem: the desired behaviour in the problem domain is not — or not solely — about phenomena shared by the machine; only indirectly, by exploiting the domain properties, can the machine ensure satisfaction of the requirement.

3. Domain Types

It is useful to distinguish some broad types of domain. Machines; other causal domains; lexical domains; and biddable domains. All are physical domains, but demand different kinds of description and raise different development concerns. The distinctions we make among the domain types are subjective: the view we take of a domain, and often the boundary by which we circumscribe it, depend on our purpose in the context of the problem in hand.

3.1 Machines

The machine in a problem is a specialisation of a general-purpose computer, the specialisation being achieved by programming. The general-purpose computer is programmable because it is a Universal Turing Machine: it reads a description of the particular specialised machine needed for the problem in hand, and behaves as that machine. The end-product of the software development task is that description.

The machine domain has familiar characteristics. It is a complex electro-mechanical product: the arithmetic and logical unit and the cache and first level of main storage are purely electronic; disk storage and some input-output devices such as floppy disk and CD and DVD drives, keyboard, screen and mouse, are partly mechanical. These electronic and mechanical components constitute a *causal domain*. The event and state phenomena of the domain are connected by causal chains: the mechanical event of a keystroke causes an electronic signal; depending on the program, this in turn may cause further signals leading to alteration of the screen state, to movement of a disk head, and to alteration of the state of some main storage cells to reflect the state of a sector of the disk.

A crucial characteristic of the machine domain is its very high reliability. In most applications the correct functioning of the machine is taken for granted. This reliability is closely associated with the *engineered formality* of the machine. Formal discrete phenomena — typically, binary states — are constructed on the basis of continuous and imperfect physical phenomena such as magnetisation, charge and voltage. The machine is designed to ensure that the continuous voltage representing a binary state is examined only when it is at a level that can be unambiguously interpreted as 0 or 1.

3.2 Causal Domains

Causal domains other than the machine are common in many applications. The lift equipment — the buttons and lights, the doors, sensors, motors and winding gear — constitutes one or more causal domains. An ATM and the cards that it reads are interacting causal domains. The wheels and brakes of a car, and the road surface it runs on, are causal domains.

The domain properties of a causal domain are, essentially, causal relationships. Setting the lift motor to *on* and *up* causes it to turn; the turning winds the lift cable on a drum; winding the cable causes to lift car to rise; reaching a floor sets the sensor to *on*. These causal chains allow the machine to cause and constrain events and state changes far from its interface with the domain. Descriptions of the domain properties are descriptions of the physical world of these causal chains. For example, a state machine description may assert that the event *motor-on* when *motor-stopped* holds causes a transition to a state in which *motor-running* holds.

However, causal domains in general are often much less reliable than the machine. Brakes fail, magnetic stripes on cards become unreadable, motors burn out. The *motor-on* event may, after all, fail to produce its expected change of state. Depending on the nature and setting of the problem it may be necessary to deal explicitly with this unreliability in a separate subproblem.

3.3 Lexical Domains

A lexical domain is, physically, a causal domain. For example, it may be a database held on one or more disk drives, or an object structure inside a machine, or a file held on a tape or CD. The physical causal domain provides the infrastructure, but the significance of the lexical domain is in its data. That is, in the *values* represented by the physical state phenomena, and the *truths* and other relationships among them.

There is therefore a kind of duality in a lexical domain. At the physical level we can not avoid being concerned with states and events. Data is written and read by executing physical operations whose effects are state changes in the lexical domain or in the machine that causes the operation events. State machines may provide appropriate descriptive techniques here. But at the lexical level we are concerned with the more abstract data structures; descriptions are more appropriately expressed in grammars, functions and other recursive structures. The relationships here are not causal but definitional. It is not appropriate to say "this integer pair is broken because it has only one integer" in the way we might say "this bicycle is broken because it has only one wheel". We must rather say that it is not an integer pair.

3.4 Biddable Domains

Many problems involve human beings as users or operators, or as originators or recipients of information. Human beings are subject to physical laws that limit their behaviour in a negative sense, by making certain behaviours impossible. The operator of the sheet steel pressing tool in the production cell can not insert his hand between the hammer and the anvil and simultaneously hold his finger on the *motor-on* button located six feet away. Development of a safe production-cell controller can rely on this domain property.

There are also some physical laws compelling behaviour in the positive sense of reaction to stimulus. In a healthy human being, the doctor's tap on the knee results in an involuntary jerk. But these reactions are useful only in rare and very specialised applications. More commonly, if positive behaviour is required of a system operator or user the correct behaviour is described in online or offline instructions and the operator or user is *bidden*, or enjoined, to follow the instructions. For this reason we classify the operator or user as a *biddable* domain.

The extent to which correct behaviour, in this sense, can be relied on varies over a wide spectrum. The pilot of a plane or the driver of a train can be relied on to behave correctly almost always. Failure to follow instructions, especially if it leads to an accident, will be intensively investigated and may even be regarded as criminally culpable. As a result, it is useful in development to treat correct behaviour in a biddable domain in much the same way as in a causal domain: in one subproblem correct behaviour is described and assumed; the possibility of incorrect behaviour is dealt with explicitly in another subproblem. By contrast, the user of an ATM can not be expected to adhere to an instruction manual. The appropriate domain properties description must accommodate every behaviour that is physically possible.

4. Elementary Problem Frames

The account given earlier of problem analysis in terms of *requirement*, *domain properties* and *machine specification*, is too general. Real problems are more specific. A problem frame captures the characteristics of a specific tightly constrained class of idealised problems. These problem classes correspond to intuitive notions of different kinds of problem, but make the intuition more precise. They stipulate the structure and characteristics of the requirement, of the problem domain — possibly structuring it as two or more domains

— and of the interfaces among domains. To illustrate the idea, we will look briefly at four elementary problem frames.

4.1 Simple Behaviour

The first is the *Simple Behaviour* Frame. The problem frame diagram is shown in Figure 1.

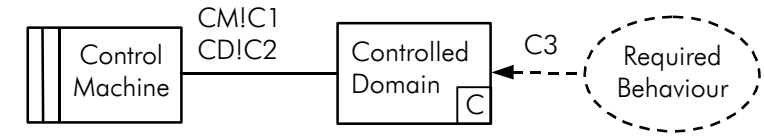

Figure 1. Problem Frame Diagram: Simple Behaviour

The rectangle with two stripes represents the machine; the plain rectangle represents the problem domain; and the dashed oval represents the requirement. This is an idealised form of a simple control problem. The requirement (Required Behaviour) is to impose a certain behaviour on the problem domain (Controlled Domain). The requirement is expressed in terms of controllable phenomena C3. The interface between the machine (Control Machine) and the controlled domain consists of shared controllable phenomena C1 and C2. The exclamation marks indicate that the C1 phenomena are controlled by the machine, and C2 by the controlled domain. The controlled domain is always causal; typically, it is partly autonomous and partly responsive to the phenomena C1. The C in the corner of the box indicates that this is a causal domain.

The basic frame concern in a simple behaviour problem is to devise a control law by which the machine can satisfy the requirement in terms of the phenomena C3. Directly, it must do so by controlling the phenomena C1 and using the feedback information conveyed by the phenomena C2. An automatic braking system for a car is an example of a simple behaviour problem.

4.2 Simple Information Answers

The second problem frame is *Simple Information Answers*. Figure 2 shows its problem frame diagram.

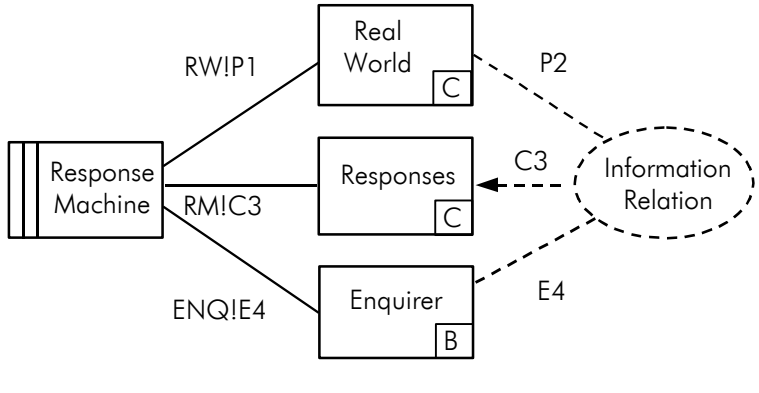

Figure 2. Problem Frame Diagram: Simple Information Answers

The arrowhead on the dashed line indicates that the requirement constrains the Responses; the absence of arrowheads on the other dashed lines indicates that does not constrain either the Real World or the Enquirer. The problem is to construct a simple information system (Response Machine) that answers enquiries. Enquiries in the form of an unstructured stream of events E4 come from an autonomous Enquirer; the Enquirer is a biddable domain, as

indicated by the B in the corner of the box. The machine creates its answers in the causal domain Responses by causing phenomena C3. The subject of the enquiries is the Real World. The Real World is shown as a causal domain, and is autonomous: none of its phenomena are controlled by the Response Machine. In a variant of this frame, the Real World may be static, having no controllable phenomena. In either case, it controls or determines all the phenomena P1 at its interface with the machine. The requirement (Information Relation) stipulates a relationship between the answers C3, the enquiry events E4, and the phenomena P2 of the real world.

The basic frame concern is the use of the specification phenomena P1 to make inferences about the requirement phenomena P2. Answering questions about the weather is an example of a simple enquiry problem.

4.3 Simple Information Display

The third problem frame is *Simple Information Display*. Its problem frame diagram is shown in Figure 3.

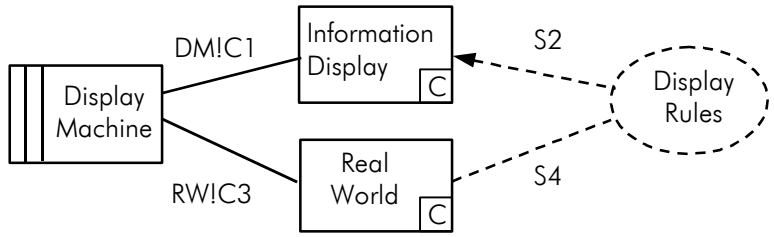

Figure 3. Problem Frame Diagram: Simple Information Display

This is an idealised form of a simple information system (Display Machine) that maintains a continuous display of information (Information Display) about an autonomous causal Real World. The requirement (Display Rules) stipulates the state S4 of the display for each state S2 of the real world. The display is a reactive causal domain, changing its states S4 in response to the machine-controlled phenomena C3.

The basic frame concern is the use of the interface phenomena C1 to provide inferences about the phenomena S2 that are the subject of the requirement. Controlling the display in a hotel lobby that shows the current positions of the lifts is an example of a simple information display problem.

4.4 Simple Workpieces

Finally, Figure 4 shows the problem frame diagram for the Simple Workpieces frame.

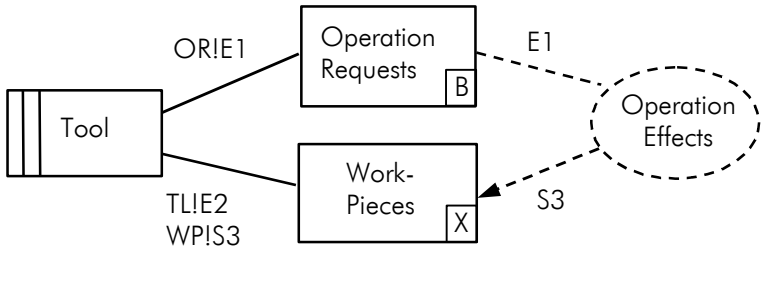

Figure 4. Problem Frame Diagram: Simple Workpieces

This is an idealised form of a problem in which the machine (Tool) acts as a simple tool for the creation and manipulation of text or graphic objects in the Workpieces domain.

Workpieces is a lexical domain, as indicated by the X in the corner of the box. The user of the tool autonomously issues an unstructured stream of commands Operation Requests E1, that constitute a biddable domain. The machine may sometimes ignore a command: for example, if it makes no sense in the current context. The workpieces are regarded as *given* — that is, although they are designed software artifacts, their design is not considered to be a part of the workpieces problem itself. They are *state-reactive*: that is, their behaviour consists only of changing their states S3 in response to events E2 caused by the machine.

The basic frame concern is that the machine must ignore senseless operation requests E1 (such as a deletion request for a non-existent workpiece), and must convert valid requests E1 into appropriate combinations of invocations E2 at the interface with the workpieces. The control of setting a VCR memo to record a TV program is an example of a simple workpieces problem.

5. Problem Decomposition

The elementary problem frames deal only with simplified idealised problems. Even when extended by a number of common variants and composites, the corpus of frames does not encompass many — perhaps any — problems of realistic size and complexity. Dealing with a realistic problem means decomposition into subproblems. A sufficient corpus of frames is one in which we can always find a set of subproblems to give an appropriate decomposition of any realistic problem. There are several possible approaches to the decomposition task. In this section we mention three of them.

5.1 Outside-In Decomposition

Sometimes the problem in hand seems to fit no known frame even approximately. It may then be helpful to decompose the problem by working from the outside towards the inside, as it were. The approach here is to try to find recognisable parts or aspects of the problem that correspond to known frames, and analyse them in the context of those frames. Then they may be regarded as solved problems, and the parts and aspects of the original problem that remain to be solved can be considered without the added complication of the already solved subproblems.

This approach is essentially an iterative application of the often-quoted heuristic “find a piece of the problem that you can solve”. If the approach succeeds, the original problem is eventually whittled down to a simple nucleus that can now be recognised as fitting a known frame.

5.2 Inside-Out Decomposition

Sometimes the problem in hand seems to fit a known frame approximately, but exhibits difficulties that frustrate the pure application of the frame. These difficulties themselves give rise to subproblems that may be recognisable as fitting other frames in their own right. For example, one form of difficulty is a *connection* difficulty: it may be that some information needed by the machine in a behaviour problem is not available directly when it is needed. It may then be possible to cast the difficulty as an information answers subproblem in which the original machine plays the part of the enquirer. Another kind of difficulty is an *identities* difficulty, in which the machine shares a set of event or state phenomena with the problem domain but does not share the associated roles that identify the participating domain entities.

This approach can be thought of as working from the inside towards the outside, where the inside is the frame that seems to fit approximately and the outside is the surrounding set of difficulties. The core problem can be analysed on the assumption that the difficulties will be overcome in the solutions to the subproblems that capture them. This approach, too, is an

application of a well-known heuristic: “ignore complications and solve the simpler problem”.

5.3 Decomposition by Subproblem Tempo

Each elementary problem frame captures a very simple problem class. It demands only a simple view of each domain and interface. The requirement is conceptually simple enough to be expressible as a single responsibility: “Make the Controlled Domain behave in such-and-such a way;” “Answer the Enquirer’s questions about the Real World;” “Maintain a continuous Display of information about the Real World;” “Perform Requested Operations on the Workpieces;” and so on.

Another dimension of the frames’ simplicity is a consistent temporal granularity. The Workpiece operations proceed at the tempo of the Operation Requests; the Display changes at the tempo of Real World changes. When a problem exhibits two or more clearly different tempi, that is a strong indication of a contour along which distinct subproblems can be recognised. For example, in a system to administer a lending library we may recognise at least two tempi. At the slower tempo the subscribing members join, pay their subscriptions, renew membership and resign; the tempo here may be linked to annual cycles of membership validity. At the faster tempo, measured in days, books are borrowed, renewed and returned. The different tempi characterise two distinct subproblems.

5.4 Recognising a Standard Composite Frame

Although the elementary frames form the basis of the technique of problem analysis and structuring advocated here, a developed form of the technique will have a rich set of composite frames. It may be expected that a substantial part of a realistic problem, or even, occasionally, the whole of it, will fit a known composite frame. To recognise and exploit this fit is to apply the heuristic “reduce to a previously solved problem”, or “the best method is to have solved the same problem before”.

One example of a composite frame is the Interactive Workpieces frame which, unlike the Simple Workpieces frame, includes an Interactive Screen domain at which the user can interact with the Tool by viewing the workpiece states and entering operation requests by a mouse or similar device. This composite frame, of course, has a solution in the form of the MVC (Model-View-Controller) [5] framework, well-known in object-oriented design.

Another example of a composite frame is an Information System with a Model Domain. Where the shared phenomena are inadequate or untimely in an information problem, the difficulty can often be overcome by introducing a model domain and decomposing into two subproblems. In one subproblem the machine builds the model from the real world; in the other it uses the model to maintain the display or to answer the enquiries. Here, the world ‘model’ is used in a quite specific sense. The model domain is an *analogic model* [6] of the real world. The phenomena of the model domain are regarded as surrogates for the phenomena of the real world. Because they satisfy analogous relationships, they can be used to provide information about the real world.

5.5 Standardisation by Composite Frames

The recognition of composite frames is closely analogous to the universal practice in established branches of engineering, where standardised products — such as cars and television sets and bridges — are elaborate composites of standardised components. The value of a repertoire of well-understood composite frames is, of course, that understanding a composite frame means a lot more than understanding its component subproblems. It means also understanding how the subproblems fit together, being aware of the concerns and difficulties that arise from the composition itself, and knowing how to fit the subproblem solutions together into a satisfactory solution to the original composite problem.

6. A More Realistic Problem

To illustrate the problem frame technique we take the problem of controlling a package router. Here is the problem statement, adapted from [7]:

> "Packages with bar-coded destination labels move along a conveyor to a reading station where their package-ids and destinations are read. They then slide by gravity down pipes fitted with sensors at top and bottom.
>
> "The pipes are connected by two-position switches that the computer can flip (when no package is present between the incoming and outgoing pipes). The configuration of pipes therefore forms a tree.
>
> "At the leaves of the tree are destination bins corresponding to the bar-coded destinations. A package can not overtake another either in a pipe or in a switch. However, because the packages are of varying shapes and sizes, they slide at unpredictable speeds and may therefore get too close together to allow a switch to be set correctly. A misrouted package may be routed to any bin, an appropriate message being displayed.
>
> "The system must route packages to their destination bins by setting the switches appropriately for each package as it slides down the pipes of the tree."

6.1 Inside-Out Approach

At first sight, this appears to be a simple behaviour problem. Figure 5 shows the problem diagram with the part names of the simple behaviour frame superimposed.

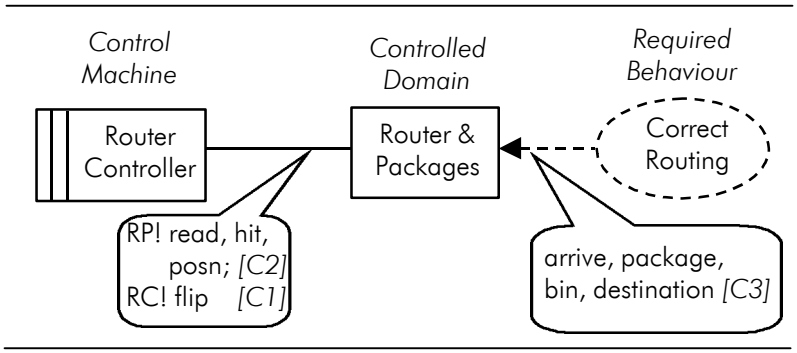

Figure 5. Problem Diagram: Package Router Control as a Simple Behaviour Problem

The interface between the Router Controller machine and the Router & Packages domain, shown in the left callout, consists of these phenomena:

- **read**: Read events in which the barcode of a package entity is read, and the participating barcoded string is transmitted to the Router Controller. These events are controlled by the Router & Packages domain.
- **hit**: Hit events in which a package hits a sensor. The Router & Packages domain controls hit events.
- **posn**: These states are the physical positions — left or right — of the router switches. They are controlled by the Router & Packages domain.
- **flip**: These are the flip events, in which a switch is flipped, changing its position. They are controlled by the Router Controller machine.

The requirement (Correct Routing) is concerned with the following phenomena, indicated in the right callout:

- **arrive, package, bin**: arrive events in which a package arrives at a bin at a leaf of the tree, and the associated roles which are the participation of the package and the bin.

- **package, destination**: destination states relating a package to its barcoded package-id and destination strings.
- **bin, destination**: states relating the barcoded destination strings to the corresponding router bins.

6.2 A Connection Difficulty

The briefest attempt at describing domain properties that can close the gap between the requirement and the machine — if carried out with a properly meticulous attitude to the phenomena — shows at once that there is a connection difficulty. The essence of the difficulty is that the requirement is concerned with roles and truths involving package entities, but packages appear in none of the roles shared by the Router Controller. For example, the Router Controller can detect that a sensor has been hit, but can not detect which package is responsible. The package information — its barcoded package-id and its destination string — were transmitted to the machine when the package first entered the router at the reading station. But the machine has no direct way of determining associating this information with the anonymous package that has just hit the sensor.

The difficulty can be dealt with by a standard composite frame: an information problem with a model domain. This composite frame decomposes an information problem into two subproblems. The machine in the first subproblem has a direct interface with the Real World domain, and builds an analogic model domain that can act as a surrogate for the Real World. The machine in the second subproblem uses the analogic model domain to produce the required information without direct reference to the Real World.

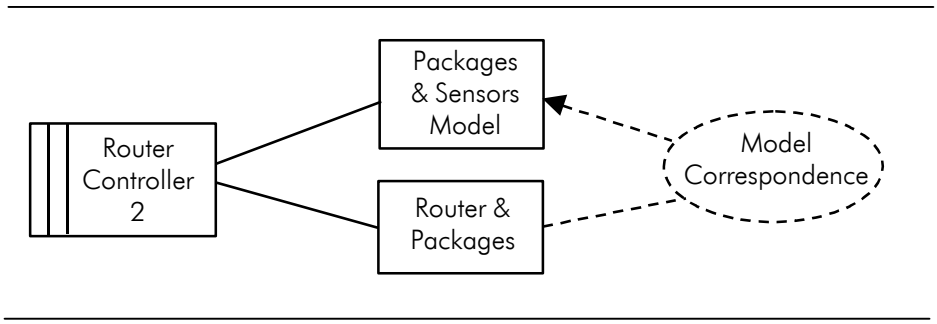

Figure 6. Building a Model Domain for the Packages and Sensors

Figure 6 shows the problem diagram for the first subproblem, in which the model is built. Because packages do not overtake each other in the pipes and switches, it is possible to regard the packages in the router as forming a set of queues: on each read event at the reading station a new package enters the tail of the queue in the topmost pipe, and on hitting a sensor at the bottom of a pipe it moves from the head of one queue to the tail of another. The model domain (Packages & Sensors Model) is a lexical domain containing a representation of these queues and package-id and destination attributes for each queue element. These model attributes are assigned when the barcoded label of the package is read at the reading station. The model domain is maintained by the Router Controller 2 machine: as each sensor hit event occurs, the appropriate model queue element moves from the head of one queue to the tail of another.

The model domain can then be interrogated by the Router Controller in the behaviour problem to answer the question: “what is the barcoded destination of the package that participated in the most recent hit event in which this particular sensor participated?” The answer to this question solves the connection difficulty: when a package arrives at the sensor guarding a switch or a bin its destination is known.

6.3 A Digression on Model Domains

Model domains play a central part in many computer-based systems. Strictly, they arise only in information problems; but information problems appear as subproblems in many contexts. In particular, a behaviour problem often involves one or more information subproblems to deal with the correct interpretation of the feedback phenomena C2 in the Simple Behaviour frame of Figure 1.

One way of thinking about analogic model domains is to see them as an elaboration of local variables in the machine behaviour specification of the undecomposed problem. In order to respond appropriately to sensor hit events, the Router Controller must maintain an internal data structure from whose current values it can identify which package has hit the sensor and retrieve its bar-coded label information. This internal data structure, along with the regime by which it is to be created, updated, and accessed, is far too elaborate to be treated merely as an aspect of the machine behaviour specification. It must be separated out as an explicit lexical domain in the problem decomposition. Timely execution of its constructor operations then form one subproblem, and of its accessor operations form another.

Ideally, the model domain is a perfect analogue of the real world phenomena and properties that are of interest for the original information problem. But perfection is rarely attainable. Instead, the model is likely to exhibit imperfections arising from many sources. For example:

- Model updating must necessarily lag behind events and state changes of the Real World. Sometimes this lag may be highly variable, leading to different orderings of model and real world events.
- Model updating often depends on inferences which in turn depend on assumed domain properties of a causal or biddable real world domain. If in some case the assumed properties do not hold, the inference will be invalid and the resulting model not correspond correctly to the Real World. If in fact one package does overtake another in a router pipe, the resulting queue model will no longer correctly reflect the contents of the pipes and switches.
- A model domain must inevitably possess phenomena and properties that are not shared by the Real World. For example, tuples in a relational database must be ordered; space must be conserved by deleting records; in the absence of information *null* values may be stored where the Real World value is certainly not null.

When a model domain is introduced into an problem analysis, it must never be forgotten that the model and the Real World are completely distinct domains. They share no phenomena at all, and in principle they demand completely distinct descriptions.

6.4 Two Identities Difficulties

The problem offers two clear examples of the identities difficulty. The first concerns the sensors and switches. Each sensor and switch is connected to a particular port — a register or sense line — of the machine. When the machine detects a hit event it detects it at a particular port, but the identity of the sensor is not explicitly known. A similar difficulty arises for the switches, both for their position states and for their flip events.

The second identities difficulty concerns the barcoded destination strings and the bins. Each bin can be associated with the sensor guarding its entrance, but this does not help: the machine has no access to the mapping between the bin sensors and the strings. Effectively, therefore, it has no way of determining which is the desired bin for a particular barcoded destination string.

Both of these difficulties are solved in the standard way for identities difficulties: the mapping must be made explicit, and put in a form accessible to the machine. Creating the mapping is a simple problem, perhaps fitting the Workpieces frame. The mapping is then

used by the router controller to identify the destination of each package with the bin sensor that is its eventual goal. An identities mapping, is, of course, a particular kind of model domain.

6.5 Another Connection Difficulty

Another connection difficulty still lurks in the problem. When a package arrives at the sensor guarding a switch the machine must flip the switch or not according to the required routing of the package. But the machine has no access to the necessary routing information: that is, it has no way of determining whether a particular bin can be reached from a particular switch, and if so, whether by its left or its right exit.

The solution to this difficulty is another model, but this time of a static domain. As often happens with static models, it appears necessary to create the model with the help of a human informant, as shown in Figure 7.

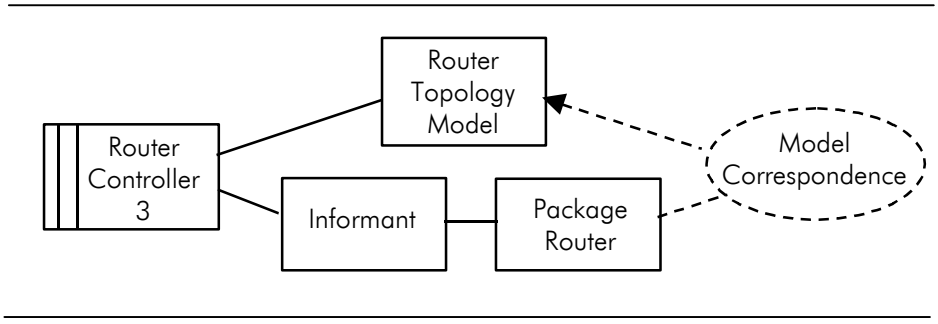

Figure 7. Making a Model Domain for the Router Topology

This static model provides all necessary information about the router topology: Which sensors are on which ends of which pipes? Which pipes feed which switches? Which pipes leave which switches? Which pipe leaves the reading station? Which bins are guarded by which sensors? From this information the router controller can determine the package routes to their destination bins.

6.6 An Information Display Problem

The display of an appropriate message when a misrouted package arrives at a wrong bin is, at first sight, a simple information display problem. When each package arrives at the sensor guarding a bin, the bin identity can be compared with the package's destination and a message produced in the case of a mismatch. Connection and identity difficulties are present, as in the behaviour subproblem; they are solved by the same model and mapping domains.

6.7 Composing the Solution

To compose the solutions of subproblems it is necessary to consider the scheduling of their machines.

In the present case the composition is fairly easy. The machines providing the solutions to the identities difficulties, along with the machine that builds the static model of the router topology, must be run to completion first. The mappings and models produced are then available for use by the other machines. Those other machines can run in parallel, essentially synchronised by the read and hit events controlled by the Router & Packages domain. That is, when one of those events occurs, each machine reacts to the event and returns to a quiescent state. Of course, the machines' reactions must be appropriately ordered: the information on which the answer to a question is based must be established before the question is answered.

7. Subproblem Concerns and Composition Concerns

The treatment of the Package Router Control problem in the preceding section has been brief and necessarily somewhat superficial. Decomposition proceeded both by recognising familiar subproblem classes — controlling the package flow, reporting misroutings — and by adopting familiar solutions to recognised concerns — the connection and identities difficulties. The classification and recognition of familiar difficulties and concerns, arising in particular problem classes, for particular domain types, or for particular combinations of subproblems, is fundamental to any sound development discipline. In this final section some common difficulties and concerns are discussed, and some of their implications for software development technique are briefly sketched.

7.1 The Initialisation Concern

Initialisation is a well known concern in programming: an *uninitialised variable* is a classic error. It is perhaps less well known in problem analysis and structuring, where it is of equal importance and often greater difficulty.

It is easy in a formal approach to software development to introduce a formal treatment of initialisation. For example, in the Z specification notation the *Init* schema is deemed, by convention, to be an operation executed at the beginning of system operation. Conventional representations of state machines have an *initial state*, which is deemed to hold at the beginning. But deeming is not the same as doing; and initialising the machine or the model domain is not the same as initialising the Real World. A desired initial state of the machine or of a model domain, defined without reference to the problem world, is easily achieved by programming. But the initialisation concern is about the initial state both of the machine and of the problem world: when system operation begins, the machine must be in the initial state described in the machine behaviour specification, and the problem world in the initial state described in the domain properties descriptions.

In an information problem the Real World is autonomous. There is therefore no way of forcing it into an initial state; the initial correspondence between the Real World on one side and the machine or a model domain on the other must be achieved by initialising the machine or model domain. In database applications this is sometimes called by the slightly misleading name of ‘populating the database’.

In a behaviour problem the Controlled Domain is not autonomous, and it may be appropriate to initialise it to conform to a fixed initial state of the machine. Sometimes this can be done by the machine itself, sometimes by external human physical help; sometimes a combination is needed. The Package Router pipes and switches must be initially empty, to correspond to empty queues in the model. This can only be achieved by stopping the flow of packages past the reading station and waiting until every package in the router has reached a bin. The necessary operating instructions are as much a part of the software development as the programs themselves.

7.2 The Breakage Concern

In a Behaviour problem the Controlled Domain may be vulnerable to certain occurrences of events controlled by the machine. For example, a router switch must not be flipped when a package is passing through. The direction of the lift motor must not be changed while the motor is running. The gearbox of a car must not be shifted into low gear while the car is travelling at more than 50mph.

Handling the breakage concern means, of course, avoiding the machine behaviours that can cause breakage of the domain. In itself it provides a compelling reason for explicit domain properties descriptions. The machine behaviour specification alone can not express the consequences of particular departures from that specification. Domain property descriptions, therefore, may need language elements to denote a broken state. The behaviour

of a domain that has reached a broken state should be assumed to be entirely unpredictable. It is the CHAOS process of CSP [12].

The breakage concern does not arise in Information problems, because the Real World is autonomous, and unaffected by the Response Machine or Display Machine. In Workpieces problems it arises only if the Workpieces lexical domain has been so designed that it is vulnerable to certain sequences of operations.

7.3 The Reliability Concern

Causal and biddable domains are, in general, unreliable. There is no cast-iron guarantee that they will conform to the domain property descriptions on which we must rely to solve the core of the Information or Behaviour problem in hand. Addressing the reliability concern means dealing appropriately with the possibilities of failure — that is, providing against those failures to the extent that is merited by the cost of provision, and by the likelihood and the consequences of failure.

Almost always, non-trivial provision demands a separate *audit subproblem* to detect, and perhaps diagnose, failure. The separation is valuable because it allows analysis and solution of the core subproblem to proceed on the basis of a clear description of the domain properties that solution must rely on, without becoming entangled in the complications of every possible failure. Similarly, the audit subproblem does not become entangled in the complications of the causal chains associated with correct operation; it needs a very different description of domain properties.

A simple illustration of the point is the separation of compilation from source program error diagnosis. Very early compilers largely ignored error diagnosis, because the problem of compilation was complex enough to tax the designer fully. When error diagnosis is treated as a separate subproblem it becomes practical to use much additional information that is irrelevant to compilation — for example, text indentation, personal style in choosing identifiers, and so on.

7.4 Interference and Synchronisation

Problem structure is most often a parallel composition of subproblems. Hierarchical and embedded structures are also found, but parallel structure is the commonest. The right metaphor for problem structure is not the bill-of-materials assembly structure but the superimposition of CYMK separations in the printing of a four-colour graphic. Within such a problem structure, the subproblem machines behave as concurrent processes; any problem domain common to two or more subproblems is potentially a shared variable.

The concerns arising from variables shared by concurrent processes are well known. There are two distinct concerns: *interference* and *synchronisation*. Interference is the classic concern, to be solved by choosing appropriate critical regions and implementing them by mutual exclusion. For example, mutual exclusion must be enforced between the updating of the Packages & Sensors model and use of the model for routing packages and for reporting misrouted packages. The need for mutual exclusion is easily seen: without it, at least one of the subproblem analyses is invalidated. More precisely, the described domain properties of the model domain, on which the package routing machine relies, can not be guaranteed to hold if the model is being updated while it is being inspected.

Synchronisation is a larger concern, more directly arising from the particular problem in hand. By separating subproblems as we have done, we obtain a vital simplification of each subproblem; but we may also leave an indeterminacy in their scheduling that reaches all the way up to the problem requirement.

Consider, for example, a small traffic light system in which the required regime of traffic control is represented in a lexical domain that can be manually altered by any authorised person possessing the key to the control box. Addressing the interference concern means ensuring that the regime is syntactically correct whenever the control machine is

accessing it. Addressing the synchronisation concern means ensuring that the result of altering one correct regime to give another satisfies the problem requirements: in particular, that the crossover from old to new regime does not allow a dangerous traffic control condition that is not allowed by either regime separately.

To take another example, in the library administration problem it is necessary to check membership validity when a book is borrowed. Access to the membership model domain must therefore subject to mutual exclusion. But it is also necessary to answer some requirements questions: How close to membership expiry can a member take out a fresh loan? Renew an existing loan? What transactions can take place after membership has expired?

7.5 Consistency and Contradiction

The specification, problem domain and requirement descriptions for each subproblem describe *projections* of those problem parts. A significant part of the rigorous treatment of a problem is dealing with the possibility of inconsistency among those projections. Different subproblems may demand incommensurable descriptions of the same phenomena. For example, the subproblem of reporting misrouted packages may be expressed in terms of the phenomena earlier referred to as *hit events*; but the package routing subproblem may need a finer-grained view of the phenomena, in which the arrival of the leading edge of a package at a sensor is distinguished from the departure of the trailing edge.

Requirements in different subproblems may be in direct contradiction in certain circumstances. For example, subproblem requirements must sometimes be placed in a precedence order, to express preferences among behaviours. For example, in a safety-critical system it may be right to distinguish three operational modes: *normal*, *danger*, and *emergency*. The treatment of each mode can be regarded as a distinct subproblem; the precedence among them is not embodied in each separate subproblem, but in the statement of their composition. A similar approach can be used to impose a precedence order on requirements of different degrees of importance in the presence of specification and programming errors. For example, in a radiotherapy machine, restriction of the radiation dose to the maximum safe level is a requirement that must take precedence over all others, and must be safeguarded against errors of specification or progamming in all other subproblems.

The need for appropriate treatment of this kind of contradiction is one reason why problem analysis must form an integral part of the discipline of software development and hence of computer science: it brings into prominence description structures that might otherwise receive too little attention.

8. Summary

The approach presented aims, above all, to be an approach to *problem* analysis and structuring rather than to *solution* design. There are many forces drawing the developer's attention towards devising solutions; to focus attention on the problem requires conscious effort and awareness. The developer must always bear certain fundamental principles in mind:

- Problems are usually located some way into the external world, distant from its interface with the computer. Problem analysis must therefore focus on that world.
- The world is physical and therefore informal. Finding a sound basis for formal descriptions of its properties, and then making and using such descriptions, is a central task that has received less attention that it demands. As Scherlis [13] observed:

  "One of the greatest difficulties in software development is formalization — capturing in symbolic representation a worldly computational problem so that

the statements obtained by following rules of symbolic manipulation are useful statements once translated back into the language of the world."

The careful modesty of the need for 'useful statements' should be noted. In describing the world, formalisation and formal reasoning can show only the presence of errors, not their absence.

- To come to grips with the world it is necessary to pay the most careful attention to its *physical phenomena*. In an abstract, formal treatment, it is misleadingly easy to introduce and use global indices — for example, for the packages in the router — without considering whether the phenomena on which identification must be based are equally global.
- The model domain must not be confused with the Real World. This confusion is endemic in today's software development methodology and practice. It can be largely traced to the unthinking use of the word 'model' where the more modest word 'description' would often be far preferable.
- The ancient principle *divide and conquer* holds good in problem analysis no less than in program and algorithm design. The divided tribes must eventually be reunited under the conqueror's rule — but not yet. Much of the composition of subproblem machines belongs to the solution rather than to the problem analysis. And even that part that indubitably belongs to the problem analysis should be deferred as long as possible. Premature composition is the largest source of combinatorial complexity.

Acknowledgements

Many of the ideas discussed in this paper have been developed during several years' cooperation with Daniel Jackson of MIT. Structuring a development in terms of *requirement*, *domain properties* and *machine specification* descriptions formed the basis of much work with Pamela Zave of AT&T Research. There are earlier discussions of the Package Router problem in [8] and in [9]. Further discussion of problem frames may be found in [2] and [3]. The decomposition ideas have something in common with the goal decomposition approach of KAOS [10,11].

References

[1] Michael Jackson; *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*; Addison-Wesley, 1995.

[2] Michael Jackson; Problem Analysis Using Small Problem Frames; *Proceedings of WOFACS '98, South African Computer Journal 22*, pages 47-60, March 1999.

[3] Michael Jackson; *Problem Frames: Analysing and Structuring Software Development Problems*; Addison-Wesley, 2000.

[4] Carl A Gunter, Elsa L Gunter, Michael Jackson and Pamela Zave; A Reference Model for Requirements and Specifications; *Proceedings of ICRE 2000, Chicago Ill, USA*; reprinted in *IEEE Software* Volume 17 Number 3, pages 37-43, May/June 2000.

[5] 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* Volume 1 Number 3, pages 26-49, August/September 1988.

[6] R L Ackoff; *Scientific Method: Optimizing Applied Research Decisions*; Wiley, 1962.

[7] William Swartout and Robert Balzer; On the Inevitable Intertwining of Specification and Implementation; *Communications of the ACM* Volume 25 Number 7, pages 438-440, July 1982.

[8] Robert M Balzer, Neil M Goldman and David S Wile. Operational Specification as the Basis for Prototyping; *ACM Sigsoft SE Notes* Volume 7 Number 5, pages 3-16, December 1982.

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

[10] A van Lamsweerde, R Darimont and Ph Massonet; Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt; in *Proceedings of RE95, the Second IEEE International Symposium on Requirements Engineering*, pages 194-203, May 1995.

[11] Axel van Lamsweerde and Emmanuel Letier; Integrating Obstacles in Goal-Directed Requirements Engineering; in *Proceedings of the 20th International Conference on Software Engineering*, April 1998.

[12] C A R Hoare; *Communicating Sequential Processes*; Prentice-Hall, 1983.

[13] W L Scherlis; responding to E W Dijkstra “On the Cruelty of Really Teaching Computing Science”; *Communications of the ACM* Volume 32 Number 12, page 1407, December 1989.