The Fragility of System Organisation

Barry Dwyer

Department of Computer Science
University of Adelaide
Adelaide, Australia
Email: barry.dwyer@adelaide.edu.au

Abstract

It is usually assumed that the gross topology of transaction processing systems depends on the nature of the business, is stable, and determines rather than is determined by detailed procedures. Recent research in automated systems design suggests that the opposite is true: small changes to procedures, or even to data representations, can invalidate a system structure. It is to be supposed that many existing systems either fail to process transactions correctly, or succeed only because of informal communications that were not part of system design.

Keywords

Batch IS, Determines Relation, Determines Graph, DFD, Information Flows, IS, IS Design, Organizational Change, Order-Processing IS, Organizational Design, Process Design, Task Structure, Transaction Processing Systems

INTRODUCTION

Metaphor

Transaction processing systems (Gray and Reuter 1992) are usually divided into processes connected by data flows. The purpose of these systems is to record business data and keep it up-to-date. It does not matter whether we are talking about departments within a business, programs within a computer system, or a PC network using work-flow technology (Flores *et al.* 1993), the principles are the same. In what follows, we use the metaphor of departments made up of clerks working with written records. In computer terms, translate 'department' into 'program', 'clerk' into 'process instance' and substitute a database for the paper work. Several clerks within a department can be seen as parallel invocations of a process, and when they 'contend' for records, this implies the need for record locking, perhaps leading to deadlock (Haerder and Reuter 1983).

Separability and Independence

The Serv-U-Rite Warehouse is a distributor of white goods, obtaining its supplies from several nation-wide manufacturers and importers. Sales orders from its customers pass through four departments:

- *Pricing* extends an order by looking up the current price of each item ordered.
- *Credit Control* checks and updates the customer's current credit standing.
- *Stock Control* determines whether each order item can be delivered, updating the stock on hand, and raising a back-order if necessary.
- *Accounts* updates the customer's balance, and issues an invoice.

We ask four questions: How was this arrangement arrived at? Why does it work? Is it stable? Can it be improved?

Before considering this relatively difficult case, let us consider the evolution of a system at the (fictitious) Macrotopian Reference Library as it grows to meet increasing demands.

The library lends only to its branches. Loans are requested by electronic mail, by telephone, or by post. The loans system records the numbers of copies of books remaining on the shelves, and the number borrowed by each branch. Because there is an inevitable delay in delivering the books themselves, the system does not need to respond in real time.

Each book and each branch is associated with an index card. When a branch library borrows a book, its book record is adjusted to reflect the number of copies on the shelves, and the branch record is adjusted to reflect the number of books borrowed. The library has a rule that the last copy of a book must always remain on the shelves, so that it can be consulted by visitors to the library. However, such visitors cannot themselves borrow books.

In the simplest possible implementation of this requirement, one library clerk receives the orders, and adjusts both sets of index cards *as each loan is made*. The clerk must first check that the book being borrowed is not the last, then increment the number of books borrowed by the branch and decrement the number of copies of the book remaining on the shelves. *If we want to know if a more complex implementation is correct, we need to ask if it would get the same results as this.*

Suppose that, as the demand for the library's services increases, the clerk cannot cope with the work-load. During busy periods, the clerk may then merely record the identifiers of the books and branches, and later, during quiet periods, update the file cards from the batches of loan records (*batch processing*). Rather than search the book and branch library card files at random, the clerk will soon find that it is more efficient to sort the loans into card file order, first by title to update the book card index, then by branch to update the branch library cards (*sequential processing*).

Suppose the library becomes still busier, and it becomes necessary to employ *two* loan clerks. They may return to the direct method of recording loans, but since they share the use of the same index cards, they will need to cooperate in using them, and will sometimes even contend for the same book or branch record. Given enough work, they will again find it easier to record loans and update the files separately. They can do this in several ways. One option is for one clerk to control the book cards, and the second clerk to control the branch library cards. This division of work ensures that they will never have to contend for access to record cards.

At this stage, the library system has the structure,

- Check and update the number of books on the shelves.
- Update the branch record.

This decomposition depends on a property we call *separability* (Conway 1963). The two steps can be separated because, although the updating of the branch records depends on the state of the book records, the issue of a book does *not* depend on how many the branch currently holds.

Suppose that the library becomes so busy that several clerks are needed. How can they be used?

One option is for the new clerks to record loans, while the existing clerks update the book file and branch library file. What if the clerk who updates the books file cannot cope? If two

clerks are allocated to updating it, they will need to share its use, and it may prove that two clerks can work no more quickly than one (*parallel processing, contention*). Suppose that the books file is split, A-N, O-Z. Then the two clerks can work independently, at double the speed (*a partitioned database*). The same trick can be used to speed up access to the branch library file, and by splitting the files into more and more parts, the speed-up can be increased as much as needed.

There is a second important principle at work here: as far as the information system is concerned: The actions on each branch card are independent of those on every other branch card, and the actions on each book card are independent of those on every other book card. It is this property of *independence* that allows the card files to be split into parts that can be processed in *parallel*—the same condition that allowed the files to be processed *sequentially* in A to Z order (Dwyer 1995, 1998a, 1998b). However, the independence property could only be exploited after the system was decomposed into two steps, which relied on the property of *separability*. This example illustrates the importance of both properties in system design.

Before moving on, it is important to say what we mean by a design being *correct* when it is subject to internal delays (Dwyer 1999). The difficulty is that such a system may never be in a consistent state. For example, the clerks who update the branch records may be a day behind the clerks who update the book records. A snapshot audit of the system might find that the records of the numbers of books on the shelves plus those on loan disagreed with the number of books actually owned by the library.

We establish correctness by reference to a system that processes one transaction at a time. We may imagine a series of transactions entering the system, and then allowing the system to reach equilibrium. If the state of system we are considering is always the same as that of the reference system, we say it is correct. In general, we can guarantee this only if each sub-system processes its transactions in their original arrival order. Order is obviously important in this example. If two branches want to borrow the last available copy of a book, the first one to make the request should succeed, and not the second.

DATA DEPENDENCES

The *Determines* Relation

As the Macrotopian reference library grew, its possible system structures were determined by the fact that the number of books available on the shelves determined the number of books borrowed by branches. Moreover, the number of books held by branches did *not* determine the number available on the shelves, the number of available copies of one book did *not* determine the number of another book, and the number borrowed by one branch did *not* determine the number borrowed by another branch. We can summarise this situation in a simple diagram: the graph of its *determines* relation. (For background on graphs, see, eg, Schmidt & Ströhlein (1993).)

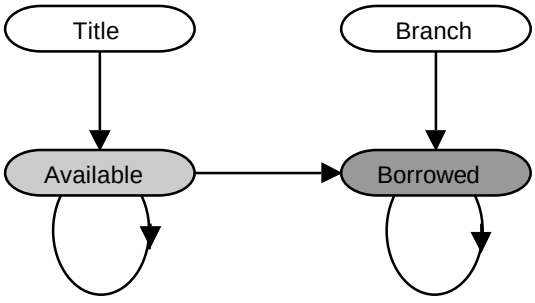

Figure 1: The *Determines* Graph for Macrotopian Reference Library Loans

- The shading of the vertices distinguishes book attributes, branch attributes and inputs.
- The arrow from *Available* to *Borrowed* shows that *Available* determines *Borrowed*.
- The arrow from *Title* to *Available* shows that *Title* determines *Available*.
- The arrow from *Branch* to *Borrowed* shows that *Branch* determines *Borrowed*.
- The (undecorated) loop on *Available* shows that the existing value of *Available* determines the new value for the same book.
- The (undecorated) loop on *Borrowed* shows that the existing value of *Borrowed* determines the new value for the same branch.
- The absence of a reverse arrow shows that *Borrowed* does *not* determine *Available*.

There are three ways one variable can determine another:

- It can be part of some expression that is evaluated to determine a new value of the other.
- It can determine whether (or how often) some expression is used to determine the new value of the other.
- It can determine which record of a file is chosen for inspection or updating.

Correctness of a Data Flow Diagram

In this example the *determines* relation defines a partial order on the variables the system must inspect or update. We can show (Dwyer 1999) that any Data Flow Diagram (DFD) (DeMarco 1978, Gane and Sarson 1979) of a system that can implement this problem correctly (e.g., Figure 2) must preserve this partial order.

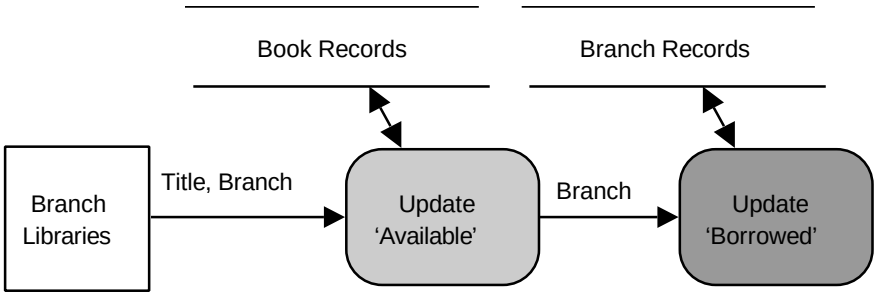

Figure 2: A possible Data Flow Diagram for Macrotopian Library Loans

A DFD preserves the ordering of a *determines* relation only if,

- No updated variable is accessed by more than one process.
- If the process inspecting or updating variable *V* precedes that for variable *U* in the DFD, there is *no* directed path from *U* to *V* in the *determines* relation.

We assume that each process deals with transactions in arrival order, but transactions can queue between processes. The first rule is needed because of delays between processes. A transaction inspecting a variable *downstream* of the process that updates it could reveal a *future* value determined by *later* arrivals. A transaction inspecting a variable *upstream* of where it is updated could reveal a *past* value not yet affected by *earlier* arrivals. The second rule amounts to saying that an upstream process cannot guess what will happen later in a downstream process.

Figure 2 is not the *only* possible DFD. The *determines* relation shows that the *Update 'Available'* process does not really need the *Branch* information—although Figure 2 certainly makes for a more convenient work-flow. Also, combining the two update processes would

preserve the *determines* relation, as in the case where one clerk does all the work. Finally, because no *Title* determines another and no *Branch* determines another, both processes in Figure 2 can access records independently. This permits parallel or sequential processing.

A CASE STUDY

Requirements

Let us return to the more complex Serv-U-Rite sales orders system.

We assume that Serv-U-Rite's sales order system maintains the following variables:

- *Authorised*: an indication whether a customer account number is valid,
- *Balance*: the amount the customer owes for goods sold,
- *Accrued*: the sum of *Balance* and the value of goods on back-order.
- *Credit-Limit*: the maximum allowed value of *Accrued*,
- *Offered*: an indication whether a product code is valid,
- *Price*: the price of the product,
- *Stock*: the quantity of the product available for sale.

Sales order lines carry three attributes:

- *Who*: the customer making the order,
- *What*: the product being ordered,
- *Qty-Ordered*: the amount of product *What* that customer *Who* requires.

There are also two outputs:

- *Invoice-Detail*: a record showing the quantity and value of each item delivered.
- *Back-Order*: a record made when there is insufficient stock to fill an order. (We assume that back-orders are filled by a separate system.)

A real system would also store descriptions of products and names and addresses of customers, and the transactions would include dates, order numbers, etc., which, although they complicate real-life situations, add nothing to this case study. *Authorised* and *Offered* would probably be inferred by the presence or absence of *Credit-Limit* or *Price* records, but we treat them here as true-false variables. This has the advantage that inserting and deleting records can be analysed like ordinary updates.

The *Determines* Relation

Drawing the graph of the *determines* relation is straightforward:

- We draw a vertex for each variable, input, or output.
- We consider each updated variable or output in turn, and
  - decide on which other variables it depends,
  - draw an arrow to it from each such variable.

The result of these steps is shown in Figure 3. Note that *Accrued* of customer *Who* depends on

- itself, because its existing value is incremented,

- the value sold: the product of *Qty-Ordered* and the *Price* of product *What*,
- *Credit-Limit* and itself, which determine whether the sale satisfies credit policy, and
- *Authorised* and *Offered*, which determine whether the sale proceeds at all.

However, we note—and this is important—*Accrued* does *not* depend on *Stock*, because the customer is committed either to pay for the goods now, or to pay for a back-order in the future.

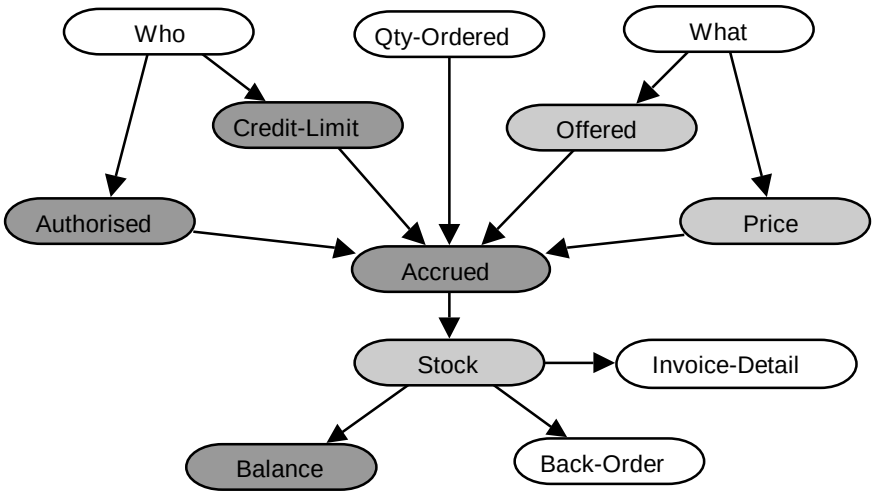

Figure 3: The *Determines* Relation for Sales Orders

A similar reasoning shows that *Stock* depends on itself, *Accrued*, and all the above. We use a shorthand in drawing the *determines* relation, and simply draw an edge from *Accrued* to *Stock*. Because the partial ordering implied by the graph is what matters, we do not need to draw additional edges from any variables that already determine *Accrued*. Similarly, we draw a single edge from *Stock* to *Balance* and from *Stock* to *Back-Order*. For the same reason, we omit the self-loops on *Accrued*, *Stock* and *Balance*.

Choosing The Best Data Flow

One way to implement this system is shown in the DFD of Figure 4. This corresponds to the current organisation of Serv-U-Rite: *Pricing*, *Credit Control*, *Stock Control* and *Accounts*. However, it is one of many possible solutions. For example, a separate process could attach *Authorised* and *Credit-Limit* information to incoming orders before the process that updates *Accrued*, or the *Stock Control* department could issue the invoices.

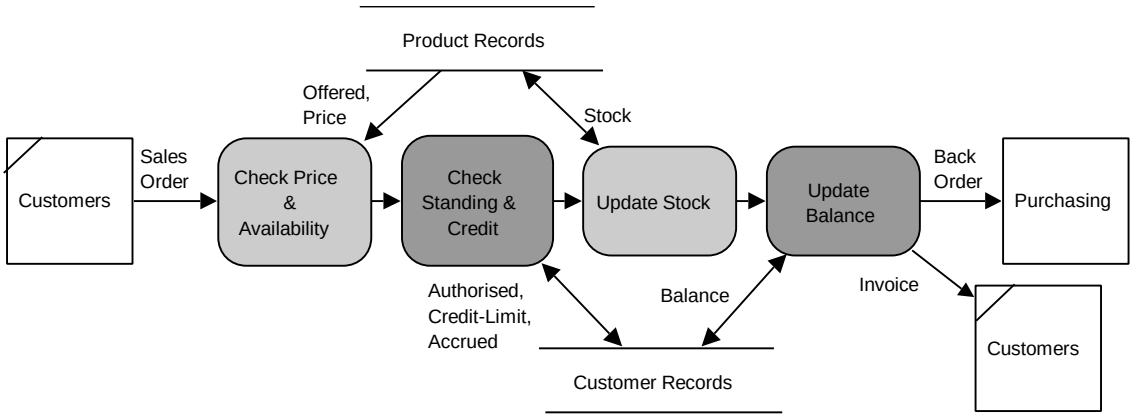

Figure 4: A possible DFD for Serv-U-Rite's sales order system.

Of the many possible DFD's that preserve the *determines* relation, what decides which is the best? A good heuristic is to choose processes to access variables that can share the same record. For example, *Authorised*, *Credit-Limit* and *Accrued* should be grouped into a single record. A second heuristic is to choose processes to avoid accessing variables that *cannot* share the same record. For example, a product variable such as *Stock* cannot share the same record as a customer variable such as *Balance*. This is why we shaded the vertices in Figure 3. To derive Figure 4, we considered combining vertices with the same colour, but not those with different colours. We can see that Figure 4 is the DFD with the least number of processes that satisfy these principles.

The reason for the first heuristic is that it is quicker to access one record containing several variables than several records containing one variable.

The reason for the second heuristic is less obvious: the DFD of Figure 4 processes each product independently of every other, and each customer independently of every other. This allows many records to be worked on in parallel. It also allows sequential file processing. Although a process that updated both *Stock* and *Balance* could still allow several clerks to work in parallel, they would contend for use of both files. If the work were sub-divided by customer account number, clerks would still contend for product records. In a stock shortage, it would also be hard to maintain a first-come, first-served policy correctly. Corresponding objections would apply if the work were sub-divided by product code. A process that accesses variables from two different records is usually more complex and less efficient than two processes that access them separately. In short, it pays to exploit *independence*.

Why not combine access to *Stock* with access to *Offered* and *Price*? All three variables could share the same product record. The answer is that the combined process would have to come both before and after the customer credit check. Depending how you look at it, either this is a logical impossibility, or it is a confused way of describing two different processes. Even if the *Pricing* and *Stock Control* functions were done within the same office they would remain distinct tasks, done at different times. On the other hand, if we *really* combined them by doing them at the same time, we would need to do the credit check at the same time too. Unfortunately, this would violate the second heuristic: accessing variables that cannot share the same record. In fact, although Figure 4 shows only two data stores, it would be better to have four, to keep the *Stock* and *Balance* records separate from the rest.

SYSTEM FRAGILITY

Sensitivity to Requirements

Suppose, as a change to company policy, Serv-U-Rite decides *not* to place back orders when orders cannot be filled. Since the requirements are now *simpler*, surely the existing system will be able to cope with this. Wrong! A customer's commitment will now depend on whether goods are in stock. In fact, *Accrued* becomes redundant, because a customer's commitment becomes simply the balance they owe.

Figure 5 shows the *determines* graph for this new situation. Its most important feature is that *Balance* determines *Stock* and *Stock* determines *Balance*. The first because the credit check decides whether goods should be sold; the second because customers pay only for what they get. This means that the DFD of Figure 4 is no longer a valid solution. A *correct* implementation of the system would need to combine the credit check, stock update, and balance update in a single step. This is a major setback, because *Stock* is a product variable and *Balance* is a customer variable. If several clerks work in parallel, they will contend for access to the records, resulting in greater complexity and lower efficiency. This cannot be avoided.

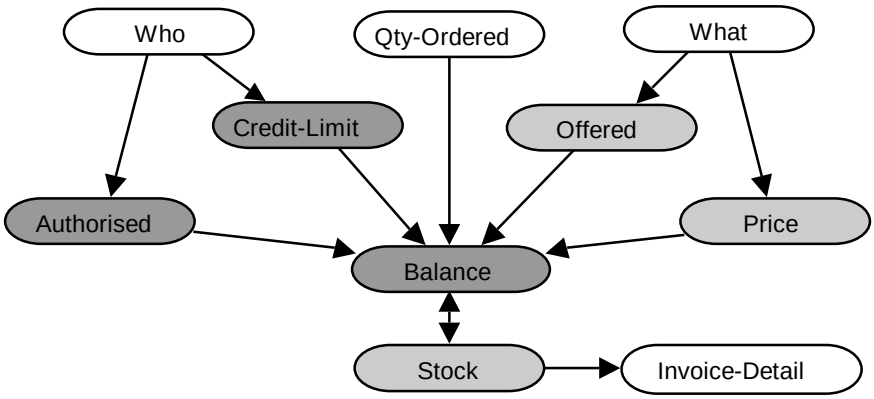

Figure 5: The Determines Relation when Back Orders are Eliminated.

In practice, it is likely that Serv-U-Rite would cling to the DFD of Figure 4 even when it is not strictly correct. This could be done by fudging the requirements. For example, the *Credit Control* department could assume no shortages, then correct customer balances later. This could lead to embarrassment: A customer might order two items: the first expensive, the second cheap. Assume the first sale exhausts the customer's credit, so the second sale has to be rejected. Then it turns out that the first item is out of stock, so it cannot be supplied. Try explaining to the customer why the second item wasn't delivered! The only way this could be done with the DFD of Figure 4 is to allow feedback from the stock update to the credit check. The resulting system would not really be *fair*; our unfortunate customer might still lose out because by the time *Stock Control* have told *Credit Control* about the shortage, and *Credit Control* have allowed the second item, a *later* order from another customer might have exhausted its stock. Even so, this 'solution' might be accepted—after all, the customer will never know!

Sensitivity to Data Representation

A customer's *Accrued* was defined to be a sum of two terms: the amount the customer owes for goods sold, and the value of goods on back order. We may therefore compute the back order value as the difference between *Accrued* and *Balance*. Suppose we store this difference instead of *Accrued*, calling it *Commitment*.

Figure 6 shows the resulting *determines* relation. It has the same problems as Figure 5. The cycle involving *Commitment*, *Stock* and *Balance* means that they must all belong to the same process. *The DFD of Figure 4 has been made invalid without even changing the original requirements.*

How can we find the best set of variables to store? The question does not seem to have a simple answer. One heuristic is to consider how the *determines* graph might be simplified. The cycle in Figure 6 is clearly an embarrassment, and *Commitment* stands out as a nexus in the graph: six other variables determine *Commitment*. By studying the requirements, we see that it is *Credit-Limit* minus the sum of *Balance* and *Commitment* that determines whether an order should be processed. Perhaps we should consider storing this result instead of the variables from which it is computed. Naming this result 'Available-Credit' would lead to a *determines* relation exactly like that of Figure 3, replacing the vertex labelled *Accrued* by one labelled *Available-Credit*.

On the other hand, it is easy to find *worse* ways of storing data. Placing *Stock* and *Price* in the same record would be a nuisance; it would create a cycle in any of the *determines* relations we have considered. More subtle, but just as bad, the presence of its *Stock* record (rather than its *Price* record) could indicate whether a product was *Offered*.

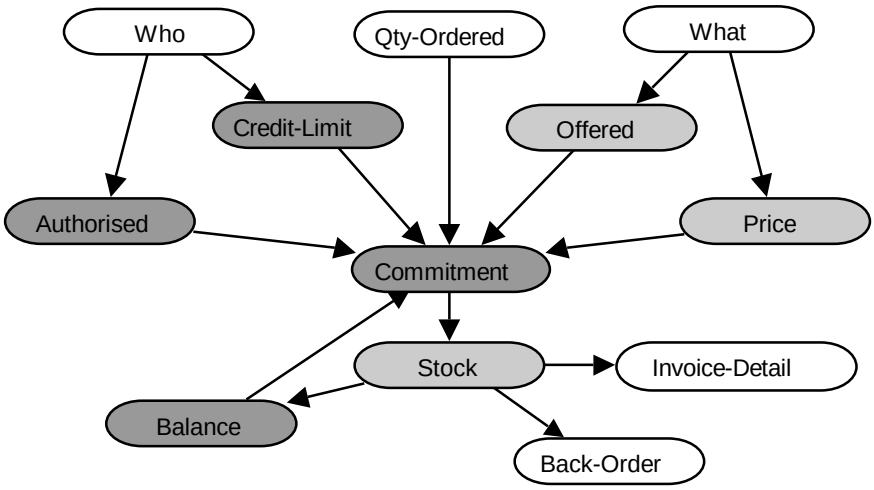

Figure 6: The *Determines* Relation when *Commitment* Replaces *Accrued*.

METHODOLOGY

For completeness, here are further notes on the methodology used (Dwyer 1999).

Multiple Transaction Types

A real system must deal with more than one kind of transaction. In the case study, we may expect that customers will pay their accounts, prices will change, goods will be delivered, new products will be made available for sale, new customer accounts will be opened, etc. The DFD of Figure 4 can handle most of these transactions, and therefore so can Serv-U-Rite's existing departmental structure. We can establish this by checking that the *determines* graph of each transaction type is consistent with the DFD. Equivalently, for each transaction type we can add new edges to the original *determines* graph. For example, a transaction to create a new product would first need to check that the product code was not already *Offered*, then to record its *Price* and set its *Stock* to zero. This would introduce a new edge from *Offered* to *Price* in Figure 3, but it would not invalidate the DFD of Figure 4.

Some transactions are not so easily accommodated. For example, when supplies of a product are replenished, the system first ought to check if there are existing back orders for the product. This would make *Stock* depend on *Back-Order*, introducing another cycle. Since *Stock* is a product variable, but back orders need to be identified by product, customer and date, they must be stored as separate records, and the resulting process would be a complex one. However, clustering back order records by product should solve most of its efficiency problems.

Modes

One type of transaction that is almost certain to cause difficulty is record deletion. This is because for most transactions the presence of a record *determines* other variables, but in a deletion, the other variables *determine* its presence. For example, suppose we decide that a customer record can be deleted (by setting *Authorised* to false) provided the customer's balance is zero. In the *determines* graph of Figure 3, this would introduce a new edge from *Balance* to *Authorised*, completing a cycle, and making the DFD of Figure 4 incorrect. A quick phone call between departments might appear to solve the problem, but the information travels against the flow, and it is possible that the customer might have placed orders already flowing in the forward direction. As a result, the *Accounts* department could finish up billing an apparently non-existent customer.

One way to avoid these problems is for a system to have several *modes* of operation. Modes correspond to different, incompatible DFDs. For example, 'order processing' mode can be shut down, and 'file weeding' mode started up. Many businesses work in different modes at different times of the day, week, month or year. An obvious example of such a mode is stock-taking, for which many businesses close down other operations entirely.

DFDs are often drawn showing many modes at once, the flows being labelled with the transaction types that flow along them. Such diagrams are usually too complex to be useful.

Compatibility

In the case study, an order item involves just one customer and just one product. This allows many clerks to work in parallel without contention or deadlock. But what happens if a transaction involves more than one customer or product?

As a simple example of this kind, consider an accounting system that allows money to be moved between accounts, provided that both the payer and payee are *Authorised*. There is then a dependence of *Balance* on *Authorised*, which would be drawn as in Figure 7. The cross decorating the edge from Authorised to Balance is needed because the *Authorised* of the payee determines the *Balance* of the payer, and vice versa. We say that the edge from *Authorised* to *Balance* is 'incompatible'.

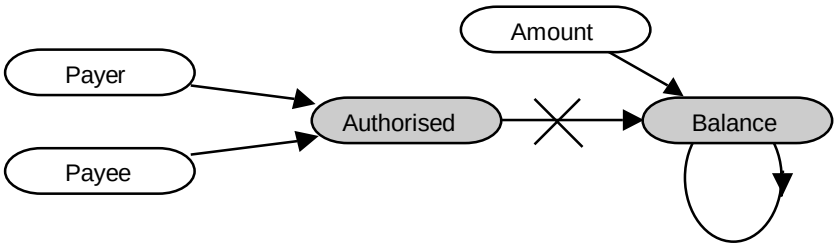

Figure 7: A *Determines* Graph showing Incompatible Variables.

If *Authorised* and *Balance* are dealt with in separate processes, parallel operations are possible. For example, two different clerks could efficiently check the payer and payee in parallel. Then, once both authorisations have been collated, both balances could be updated in parallel.

Combining *Authorised* and *Balance* into a single process would introduce contention. Therefore we avoid creating processes that include an incompatible edge. Actually, we have already seen many examples of incompatible edges. In the case study, any edge linking a product variable and customer variable is incompatible. However, we do not need to mark such edges with a cross, as their status is already clear from the shading of the vertices.

If we added the requirement that the payer's *Balance* must be no less than *Amount*, the payer's *Balance* would *determine* the payee's *Balance*. *Balance* would be incompatible with itself — the loop on it would be marked with a cross — and no implementation could avoid contention.

Deriving the Best DFD

How do we derive the best DFD, in general? We draw the graph of the most frequent transaction's *determines* relation. If this defines a partial ordering, all well and good; but if not, we must find the *strong components* of the graph.

Strong components are maximal sets of vertices such that there is a directed path between every vertex in the set. The components define the smallest processes that can be separated. After clustering vertices that belong to the same component (Aho *et al.* 1972), the resulting graph defines a possible DFD. This DFD contains the maximum number of minimal separable processes. It is *canonical*: only one such graph can be derived from a given system

requirement and set of variables. However, it can usually be improved by merging compatible processes.

Since we want to avoid creating cycles between processes, it is sufficient to consider combining pairs of processes that are either adjacent in the graph, or have *no* directed path between them. To derive Figure 4 from Figure 3, we combined *Authorised* and *Credit-Limit* with *Accrued*, because they were adjacent to it. Then we combined *Offered* and *Price* because they were not connected by a directed path. We did not consider combining *Accrued* with *Balance* because there is a compound path between them — likewise *Price* or *Offered* with *Stock*. In this we were guided by the shading of the vertices. We did not consider combining customer variables with product variables.

Different considerations would apply in optimising Figure 6. It contains a strong component, and therefore a process, involving *Commitment*, *Stock* and *Balance*. This process *must* therefore access records for *Who* and *What*. Since it is a bottleneck that causes contention, the remaining customer and product variables may as well be stored in the same records as *Commitment*, *Stock* and *Balance*. Thus, the best DFD here consists of one rather complex process.

We then consider other transaction types in descending order of frequency. If one of them introduces a cycle into the *determines* graph that would lead to an inefficient DFD, we assign it to a new *mode*, and draw a new *determines* graph for it. If this would cause too frequent switching between modes, we might seek to change the requirements in some way that will eliminate the cycle. For example, although setting *Authorised* to false when *Balance* is zero is incompatible with other transactions, it is easy to introduce a new variable that can be set unconditionally to prevent a customer making further orders. This might satisfy the business requirement equally well. When changing the requirements isn't possible, we can consider whether an incorrect implementation will do. For example, not supplying a customer with goods we have is far safer than supplying a customer with goods we don't have. If errors will happen rarely and won't cause serious problems, we may prefer to live with them.

Consistent Reports

We said in the introduction that systems in which transactions are delayed rarely reach consistent states. How can we report a state of such a system without first having to shut it down? The answer is simple: We treat requests for reports like any other transaction, making them flow along the same pathways and be subject to the same delays as regular transactions. Provided each clerk (process instance) deals with transactions *in their original arrival order*, a request for a report will arrive at each clerk at the correct point in the history of the data. In the case of the library system, if each clerk is asked to report the state of her records when loans before noon have been processed but loans after noon have not, although they may report their results at different times, the results will be consistent.

SUMMARY

System requirements for transactions can be summarised by *determines* relations. Assuming that a system should be considered correct only if it guarantees to produce the same results as a one-transaction-at-a-time reference system, the *determines* relations constrain the DFDs of valid system structures. For some requirements these structures permit contention-free processing, for others they do not. Changes to requirements can destroy the validity of system structures. These changes can be as trivial as changes to how data is represented. Different transaction types may need different DFDs, making it necessary for a system to have several modes.

Because of the fragility of system structure, organisations often process transactions incorrectly. In some cases the errors are corrected through informal communications, but if they are harmless, they may simply be tolerated.

REFERENCES

Aho, A.V., Garey, M.R. and Ullman, J.D. (1972) The Transitive Reduction of a Directed Graph, *Society for Industrial and Applied Mathematics Journal of Computing*, **1**, 131–137.

Conway, M.E. (1963) Design of a Separable Transition-diagram Compiler, *Communications of the ACM*, 6(7).

DeMarco, T. (1978), Structured Analysis and System Specification, Yourdon Press.

Dwyer, B. (1995) "Contention-free Scalable Parallel Batch Processing: Exploiting Separability and Independence", *Technical Report TR95-03*, Dept. of Computer Science, University of Adelaide.

Dwyer, B. (1998a) Separability Analysis Can Expose Parallelism, *Proceedings of PART '98: The 5th Australasian Conference on Parallel and Real-Time Systems*, 365–373, (K.A. Hawick & H.A. James, *eds.*) Springer.

Dwyer, B. (1998b) Separability and Independence in Parallel Systems Design, *Computer Science Technical Report 98-02*, Dept. of Computer Science, University of Adelaide.

Dwyer, B. (1999) *The Automatic Design of Batch Processing Systems*, Doctoral Thesis, Dept. of Computer Science, University of Adelaide.

Flores, F., Graves, F. M., Hartfield, B. and Winograd, T. (1993)"Computer Systems and the Design of Organizational Interaction", in *Readings in Groupware and Computer Supported Cooperative Work*, 504–513, (Baecker, R.M. *ed.*), Morgan Kaufmann.

Gane, C. and Sarson, T. (1979), *Structured Systems Analysis: Tools and Techniques*, Prentice-Hall, Englewood Cliffs, NJ. (1979).

Gray, J. and Reuter, A., (1992) *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann.

Haerder, T., Reuter, A. (1983) Principles of Transaction-Oriented Database Recovery, *Computing Surveys*, Vol. 15, No. 4, Dec. 1983 pp. 287–317.

Schmidt, G. and Ströhlein, T. (1993) *Relations and Graphs*, Springer.

COPYRIGHT