The Michael Jackson Design Technique: A study of the theory with applications

C.A.R.Hoare
edited and introduced by Daniel Jackson

Abstract

This paper reproduces a report written by Hoare in 1977, in which he explains the fundamental ideas of Jackson's programming method, JSP. The notions that Hoare uses for this purpose – program traces, selective traces, and combination of selective programs – appeared in the following years in the conceptual foundation of his process algebra, CSP. Hoare shows that the key step in the JSP method, in which multiple stream structures are merged to form a single program structure, can be viewed in two phases. First, each stream – represented as a regular expression – is transformed by applying simple algebraic equalities into an equivalent structure, matching the structures obtained from the other streams. Second, these structures are merged. The paper also includes a wider discussion about the scope and nature of JSP. A new introduction provides some context, and presents in full the examples that Hoare alludes to in his report.

Introduction

In March 1977, Tony Hoare wrote a short paper explaining the essence of JSP, the program design method presented two years earlier in Michael Jackson's book, *Principles of Program Design* [4].

Hoare and Jackson had studied classics together at Merton College, Oxford, twenty years earlier, and both had subsequently worked on large software projects and had become dissatisfied with existing programming methods. Hoare was appointed professor at Queen's University in Belfast in 1968, and over the next few years developed the seminal ideas in semantics and verification for which he was to receive the Turing Award in 1980.

Jackson remained in industry (at the consultancy John Hoskyns & Company), and began work on a new method. In 1970, he left Hoskyns and founded his own firm, Michael Jackson Systems Limited, to develop the method fully. By 1977, JSP – standing for 'Jackson Structured Programming', a name coined by the company's Swedish licensee in 1974 – was widely known, used in the United States, Europe and Asia, taught in several universities, and had been adopted (under the name 'SDM') by the UK government as its standard program design method.

JSP is unusually rigorous, not only amongst the approaches known as 'methods' at the time of its invention, but even by comparison with programming methods today. Indeed, for most problems, JSP is systematic enough that two programmers can be expected to produce identical solutions to a given problem. This rigour – demanded also

by the tools built to support JSP – relied on the notation having a simple and precise semantics. Jackson had provided such a semantics informally; JSP diagrams, it turned out, were essentially regular grammars. There was, however, no mathematical theory of JSP that might explain why the steps were sound, and what kinds of programs could be handled. It is these questions that Hoare began to answer in his short paper.

The paper is interesting for another reason too. Hoare's explanation of JSP in terms of traces and projections connects it to his own work on CSP (Communicating Sequential Processes). The same year that he wrote this paper, Hoare returned to Oxford, becoming head of the Programming Research Group – following the death of Christopher Strachey (who, incidentally, had introduced Jackson to programming as his teacher at Harrow School in the early 1950s). Over the next few years, Hoare developed CSP, culminating in the book of 1985 [3].

The Contributions of JSP

Because JSP advocated basing the structure of a program on the structure of the *data* that it processes, it has often been misunderstood. In JSP, 'data' does not mean the internal data structures representing the state of the program. Nor does it refer to the processing of inputs that are in-memory data structures (although JSP can be used effectively in this way, as for example in a current approach to functional programming [1] that it inspired). Rather, 'data' in JSP refers to a stream being processed, and its structure is that imposed on it when parsed as input or generated as output.

The premise of JSP is that any sequential stream of data can be treated in the same way, whether it consists of records read from a file on disk or a magnetic tape (dominant when JSP was invented), or of events from a user interface or network. The developers of Unix, in which the sequential stream plays a fundamental role, had the same idea. In fact, one of the features of JSP-COBOL, the preprocessing tool sold by Jackson's company, was its decoupling of programs from the physical properties of the devices, in just the same way that Unix's 'special files' allowed tape drives, network cards and other peripherals to be accessed as if they were disks, through a single API [10].

In this sense, JSP was a precursor not only to CSP but more generally to the view of programs as stream processors. The key insight of JSP is that a program to process a stream is best structured in the same way as the stream itself. When only the input stream has significant structure, JSP reduces to recursive descent parsing: a grammar is written for the stream and the code traverses the grammar in the (now) obvious way. Even today, this simple idea is not always well understood, and it is common to find programs that attempt to parse iterations of iterations without nested loops, or that become contorted with jumps, flags and exceptions because read operations are not placed early enough in the stream to resolve ambiguities.

JSP was more than this though. It identified the source of many of the difficulties that plagued data-processing programs as incompatibilities between stream structures. A program that processes (as inputs or outputs) streams with distinct structures cannot be structured on the basis of one stream alone (although a related method [11] attempted to do exactly that). Instead, a program structure must be found that generalizes over all the streams, reducing to the structure of each stream when projected onto its events.

This idea is the focus of Hoare's paper. He gives both a semantic view – in terms of 'selective traces' – of how a single program structure can be compatible with multiple streams, and an operational view – in terms of algebraic identities on regular expressions – of how the stream structures are manipulated to obtain the merged program structure.

Many aspects of JSP are not addressed by the paper. In some cases, the stream structures are fundamentally incompatible, and cannot be reconciled by the kinds of transformations that Hoare formalizes. JSP calls these situations *structure clashes*, and classifies them into three classes: *ordering clashes*, in which streams share elements but the elements occur in different orders; *boundary clashes*, in which the elements occur in the same order but are grouped differently; and *interleaving* or *multithreading clashes*, in which one stream can be split into (sequential) components, with the elements of each component corresponding to a subsequence of the elements of another stream.

JSP prescribes a solution for each kind of clash. For an ordering clash, the program is broken into two phases, connected by a sort or by a data store that can be accessed in either order. For a boundary clash, an intermediate stream is created, and the program is either run in two phases, or two programs are run in parallel with a stream between them. To eliminate the stream, one program can be run as a coroutine of the other. COBOL, of course, did not provide coroutines, so JSP offered a program transformation (automated by the JSP-COBOL tool) that converted stream reads and writes into calls, and let the programmer choose whether the reader or writer had initiative without changing to the code.

For an interleaving clash, a separate thread was executed for each interleaving. Jackson discovered (as he explained in the penultimate chapter of the JSP book [4]) that most information systems could be viewed as stateful transformers that generated reports from interleaved event streams. These streams could be grouped according to type. The input stream presented to a banking system, for example, might include events related to accounts, and events related to customers, and could thus be viewed as an interleaving of a set of account streams (each containing updates to a particular account) and a set of customer streams (each containing updates regarding a particular customer). Thus Jackson's system development method, JSD, was born [7]. Each type of stream became a process type; instances of a process could be executed concurrently, or could be scheduled on demand, with their states frozen in a database between uses.

This is, of course, much like object-oriented programming (OOP), but with some important distinctions. JSD was more flexible than OOP in terms of scheduling and concurrency. The treatment of scheduling issues as secondary (and their implementation by program transformation) provided an important decoupling. It also made JSD suitable for the design of real-time systems. Moreover, JSD insisted on a clean separation between the objects that represented the state of the world, and the processes that computed functions for output. In OOP, the coupling between these has been an endless source of difficulties that patterns such as *Visitor* [2] and strategies such as aspect-oriented programming [9] try to mitigate. (The 'join points' of aspect-oriented programming are perhaps just indications of how to form JSP correspondences between a base program and its aspects.)

Another major focus of JSP was its treatment of recognition problems – when a record's interpretation could not be resolved until after reading (sometimes an unbounded number of) subsequent records. To handle this difficulty, JSP prescribed multiple readahead, and when this was not feasible, positing a certain interpretation and backtracking if it turned out to be incorrect (an idea due to Barry Dwyer, who worked with Jackson early on in the development of JSP).

Perhaps, however, none of these particular technical ideas was the essential contribution of JSP – nor even the part of the book most widely quoted, namely Jackson's optimization maxim ('Rule 1: Don't do it. Rule 2 (for experts only). Don't do it yet – that is, not until you have a perfectly clear and unoptimized solution.') [4, p. vii]. Rather, the key contribution was the idea that a programming method should identify *difficulties* inherent in the class of problems addressed. The difficulties provide not only a rationale for the method's prescribed solutions, but a measure of progress. Partly in reaction to those who believed that a successful method was one that made all difficulties magically vanish, Jackson formulated this idea explicitly (many years after JSP) as the *Principle of Beneficent Difficulty* [8]:

> *Difficulties, explicitly characterized and diagnosed, are the medium in which a method captures its central insights into a problem. A method without limitations, whose development steps are never impossible to complete, whose stages diagnose and recognise no difficulty, must be very bad indeed. Just think about it. A method without limitations and difficulties is saying one of two things to you. Either it's saying that all problems are easily solved. You know that's not true. Or else it's saying that some problems are difficult, but the method is not going to help you to recognise or overcome the difficult bits when you meet them. Take your choice. Either way it's bad news.*

So the identification of difficulties marks progress not only in a single development, but in the field as a whole. A method's difficulties characterize the class of problems it addresses. In Jackson's view, the limitations of JSP are not an embarrassment to be hidden from potential users. On the contrary, the fact that its limitations are so clearly spelled out – in terms of the difficulties the method addresses, and by omission the difficulties it fails to address – makes it a worthy tool in the software developer's repertoire.

An Outline of Hoare's Paper

Hoare's paper starts by connecting the structure of a program to the structure of its executions. A program's behaviour is represented by its possible executions, called *traces* (later to become the semantic foundation of CSP); and a program with a regular structure defines a regular language of traces. Hoare defines a program as *annotated* if mock elementary statements have been inserted that have no effect on the state when executed, but which indicate, when they occur in a trace, the beginning or end of a structural block (according to whether the program is 'left' or 'right' annotated). The trace of an annotated program thus reveals the structure of the program that generated it, and by this means, Hoare is able to express the idea of structural correspondence between programs entirely semantically.

Hoare then introduces the *selective trace*, in which only some subset of statements appear – such as the reads of a particular input stream, or the writes of an output stream, or the updates of a particular variable – and the *selective program* in which all but the chosen statements have been deleted. By postponing consideration of conditions and treating them as non-deterministic, Hoare ensures that the selective traces obtained by filtering the traces of the complete program are exactly those obtained from the full executions of the selective program. This notion of selection becomes a principal means of abstraction in CSP (where it is referred to as 'restriction' when applied to traces and 'hiding' when applied to processes).

JSP, Hoare explains, can be viewed as a reversal of this selection process. Instead of obtaining selective programs from a complete program, the complete program is synthesized by merging its selective programs. Since the selective program for a stream is exactly that stream's grammar, the selective programs are readily available. To obtain the complete program, the selective programs have to be brought into *correspondence*. Hoare defines correspondence as a simple property on the traces. Two elementary statements `a` and `b` correspond if they strictly alternate `a; b; a; b;` … etc; similarly, two larger structures correspond if the start of the first alternates with the end of the second. (By left-annotating the first selective program and right-annotating the second, this correspondence of structures becomes a simple alternation of the annotation events.)

Two selective programs can be brought into perfect correspondence by manipulating their structures until they match. These manipulations can be expressed as simple transformations on regular expressions, each based on a well-known algebraic identity. Because these transformations preserve the meaning of the regular expression, the changes made to the selective programs do not affect their trace sets.

This formalization does not tell the whole story. As Hoare notes, correspondences have a deeper semantic significance. Two elementary statements `a` and `b` can only be paired if, in addition to strictly alternating, the data required to generate each `b` can be computed from the `a`'s and `b`'s that preceded it. In JSP, the programmer determines this informally.

The contribution of Hoare's paper is not only an elegant and compelling formalization of the basic concept of JSP. From his formalization emerge some concrete proposals for the method itself:

(a) In Jackson's approach, the merging of selective programs is accomplished often in a single step, which can be justified by a check that applying each selection to the merged program recovers the original selective program. Hoare prefers the merging to be performed in a series of small steps, each formally justified, so that the resulting merged program is correct by construction.

(b) JSP postpones the listing and allocation of operations until the merged program has been obtained. Hoare notes that a selective program could be written not only for each stream, but also for each *variable*. Allocation of operations then becomes just another merging step. As Hoare suggests, this could be a useful teaching tool. Whether it would work well in practice is less clear. The usage pattern of a variable – unlike the structure of an input or output stream – is not given but rather invented, is often guided by the structure of the merged pro-

gram, and is dependent on the usages of other variables. So it may be hard to describe the variables independently before considering their inter-relationships.

(c) Whereas Jackson sees the notion of structure clash as *absolute* – two selective programs clash if the only possible merge places one in its entirety before the other – Hoare prefers to see the notion as *relative*. Whenever correspondences cannot be established all the way down to the leaves of the structure tree, some state must be retained in memory to record the effect of multiple events. Hoare notes that the programmer may have a choice about how far down to go, and might choose to read an entire structure from one stream before writing the entire corresponding structure to another, even if a finer-grained interleaving were possible. Jackson would view this as undesirable, since it sees a structure clash where one does not really exist, and would prefer a tighter program, applying the standard JSP strategy of establishing correspondences as far down as possible. But Hoare's proposal is nevertheless compelling, because it adds some flexiblity to JSP, allowing greater use of intermediate structures. And by JSP standards, most programs written today are quite loose, and take liberal advantage of Hoare's proposal.

Hoare's paper was written not for an academic audience, but as a consulting report. It references example problems (taken from contemporary course materials) and assumes a knowledge of the basic JSP method. To make it more accessible, I have added brief presentations of these problems, based on the versions that appeared in Jackson's book. The paper was written in Hoare's elegant long hand. Beyond typesetting it, I have made only a few changes: notably updating the code fragments to a more modern idiom, inlining some footnotes, and adding section headings.

Multiplication Table Problem

The *Multiplication Table* problem is the first appearing in Jackson's book [4, pp. 3–7]. It is used not so much as an illustration of how to apply the JSP method, but rather to show the difference between a well-structured program, in which the program structure matches the stream structure, and a poorly structured program obtained by coding from a flowchart (a popular technique back in the 1970s). With the correct structure, a variety of simple modifications are straightforward to apply. With the wrong structure, even the most simple modification is awkward: a change to how a line is displayed, for example, cannot be made locally, because no program component corresponds exactly to the displaying of a single line.

The problem is to generate and print a multiplication table in this form:

```
1
2  4
3  6  9
4  8  12 16
5  10 15 20 25
...
10 20 30 40 50 60 70 80 90 100
```

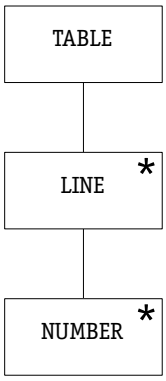

Fig. 1: Structure of Multiplication Table

JSP proceeds by representing the streams as regular grammars, forming a merged program structure from the stream structures, and then allocating operations to it. In this case – somewhat unusually for JSP – the only stream to consider is an output stream, and no merging will be needed. Its structure is simply an iteration of lines, each being an iteration of numbers (Figure 1).

Using a variable r to count rows, and c to count columns, the operations required are the initializations:

1. r = 1
2. c = 1

the updates:

3. r = r + 1
4. c = c + 1

and operations to clear a line, append a number to it, and write the line to output:

5. clear ()
6. append (r × c)
7. write ()

Each operation is allocated (Figure 2) by finding its appropriate component (by asking how often the operation should be applied), and placed in a position relative to the other operations to ensure the required dataflow. So, for example, to place operation (3) that updates the row, we ask: 'how many times is the row variable updated?'. The answer is 'once per line', so we allocate it to LINE, placing it at the end since the new value will be used for the next iteration. Similarly, we place the row initialization at the start of the table; the line-clearing, column initialization and line-writing once per line; and the adding of a new entry and the column update once per number. Although, as Hoare will note, placing the operations requires programmer intuition, it is usually easy; if not, the program structure is almost certainly wrong.

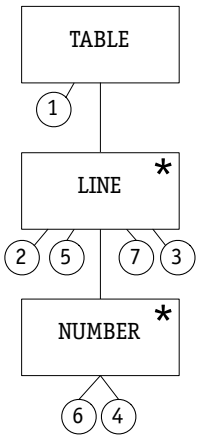

Fig. 2: Multiplication Table structure with allocated operations

Finally, the conditions for the iterations are identified – `c <= r` for number iteration and `r <= 10` for the line iteration – and the code is transcribed from the diagram:

```
r = 1;
while (r <= 10) {
    c = 1;
    clear ();
    while (c <= r) {
        append (r × c);
        c = c + 1;
        }
    write ();
    r = r + 1;
    }
```

(I have chosen the operations to match the original COBOL program, which is not so different from the program one might write today in Javascript, for example, in which `clear ()` might be written as `document.write ("<tr>")`, `append (r × c)` as `document.write ("<td>", r*c, "</td>")`, and `write ()` as `document.write ("</tr>")`. In a language such as C# or Java, the need to write a tab before each number would require a small elaboration of the output stream structure, to distinguish the first number on each line, since otherwise an extra tab would appear the end of each line.)

Stores Movement Problem

The *Stores Movement* problem [4, pp. 59–63] is a more typical JSP problem, in which the program structure is derived from the structure of a single input stream.

In the 1960s, magnetic tape was the only realistic choice for large files. A 2400-foot tape cost about $10 and held 20-60MB; a hard disk that held only 2MB cost about $400. Consequently, company data was typically stored on one of more 'master files' held on tape, with records sorted, for example, by customer number. Transactions were held on

a separate tape as a stream of records sorted on the same key, and were applied in batch by reading both tapes and writing a new tape for the updated master file.

Although such programs were standard, errors – such as mishandling an empty set of records or mistakenly reading beyond a record – were common, and many of the techniques used at the time encouraged the use of flags and complex control flow, resulting in a bug-ridden program with little semblance of structure. JSP identified the essential difficulties and prescribed simple and systematic remedies for them: readahead for recognition difficulties; interposed streams for structure clashes; inversion for overcoming lack of multithreading, etc.

In the *Stores Movement* problem, a file of transactions is to be summarized. The file contains records for orders of parts received and issued by a factory warehouse, each record consisting of a code indicating receipt or issue, a part number, and the quantity. The file has already been sorted into part-number order. The problem is to generate a report with one line for each kind of part, listing the part number and net movement (total received minus total issued). The reader is encouraged to sketch a solution to this problem before proceeding.

The input file is an iteration of groups (one per part number), each group being an iteration of movement records, and each record being an issue or a receipt (Figure 3a).

The operations are:

1. smf = open (...)
2. close (smf)
3. rec = read (smf)
4. display (pno, net)
5. pno = rec.part_number
6. net = 0
7. net = net + rec.quantity
8. net = net - rec.quantity

The program structure is just the input stream structure. Each operation (as explained before) is allocated (Figure 3b) by finding its appropriate component and placing it in a position relative to the other operations that respects dataflow dependences. So, for example, to place operation (1) that opens the file, we ask: 'how many times is the file opened?'. The answer is 'once per stores movement file', so we assign the operation to the top level node, and knowing that the file must be opened before reading, we place it at the start. Operation (7), which increments the net movement, must be executed once for each receipt record, so we assign it to RECEIPT. To place the read operation (3), we ask how much readahead will be required. Since we only need to determine whether the next record is a receipt or an issue record (and whether it belongs to the group currently being processed) single readahead will do, so a read is allocated once immediately after the open, and once after each record is consumed.

The resulting program is:

```
smf = open (...);
rec = read (smf);
while (!eof (rec)) {
```

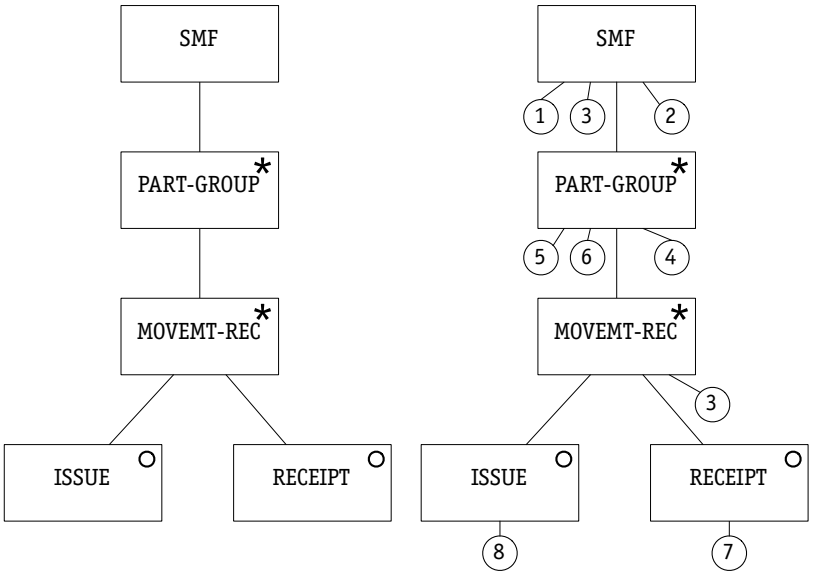

Fig. 3: Stores Movement Problem.
3a (left): Structure of input stream; 3b (right): With operations allocated

```
    pno = rec.part_number;
    net = 0;
    while (!eof (rec) &&  pno == rec.part_number) {
        if (rec.code == ISSUE)
            net = net - rec.quantity;
        else
            net = net + rec.quantity;
        rec = read (smf);
        }
    display (pno, net);
    }
close (smf);
```

This example, although simple, illustrated an important point. The double iteration is essential to match the structure of the input; without it, getting a clear and maintainable program is impossible. And yet the traditional approach to writing such programs at the time ignored this simple observation, and, in place of the nested iteration, used a 'control break' to separate the record groups. Although the term 'control break' is no longer used, the same mistake is still made today. Few programmers would solve the *Multiplication Table* problem with only one loop, but many would fail to see the same nested structure in this case.

Hoare's reference to this problem was actually to a version appearing in course notes (Figure 4b), which differed slightly from the version appearing in the book (Figure 4a), by having the report include a heading. This hardly complicates the problem, but it illustrates an important point from a theoretical perspective: that formalizing the merging process would require, in this case, an additonal elaboration that Jackson skips.

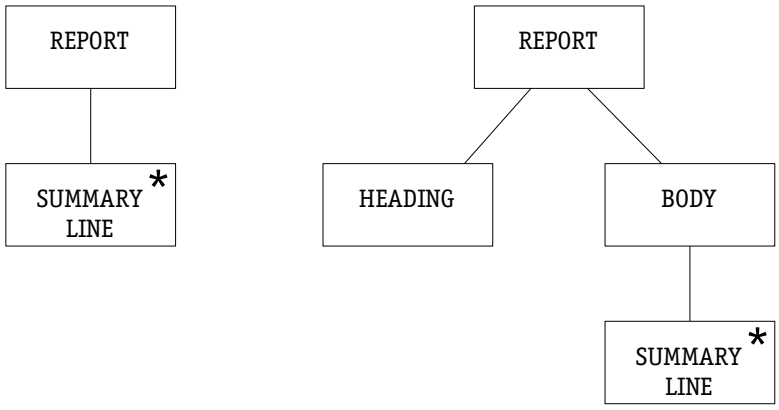

Fig. 4: Structure of output stream for Stores Movement Problem
4a (left): Version appearing in book [3]; 4b (right): Version appearing in course notes

Magic Mailing Problem

The *Magic Mailing* problem appears in the chapter of the book [4, pp. 70–74] that shows how to construct programs that handle multiple streams, and is an example of a 'collating' or 'matching' problem.

Here is the description of the problem from Jackson's book:

> *The Magic Mailing Company has just sent out nearly 10,000 letters containing an unrepeatable and irresistible offer, each letter accompanied by a returnable acceptance card. The letters and cards are individually numbered from 1 to 9999. Not all of the letters were actually sent, because some were spoilt in the addressing process. Not all of the people who received letters returned the reply card. The marketing manager, who is a suspicious fellow, thinks that his staff may have stolen some of the reply cards for the letters which were not sent, and returned the cards so they could benefit from the offer.*
>
> *The letters sent have been recorded on a file; there is one record per letter sent, containing the letter-number and other information which need not concern us. The reply cards are machine readable, and have been copied to tape; each reply card returned gives rise to one record containing the letter-number and some other information which again does not concern us. Both the letter file and the reply file have been sorted into letter-number order.*
>
> *A program is needed to report on the current state of the offer. Each line of the report corresponds to a letter-number; there are four types of line, each containing the letter-number, a code and a message. The four types of line are:*
>
> NNNNNN 1 LETTER SENT AND REPLY RECEIVED
> NNNNNN 2 LETTER SENT, NO REPLY RECEIVED
> NNNNNN 3 NO LETTER SENT, REPLY RECEIVED
> NNNNNN 4 NO LETTER SENT, NO REPLY RECEIVED

When there is more than one stream, JSP proceeds by identifying *correspondences* between the elements of the structures of the various streams. Corresponding elements

then become a single element in the program structure. The programmer can check that the program structure is correct by taking each stream in turn, and projecting the program structure onto the elements of that stream. If the program structure has been obtained correctly, this will recover for each stream a structure equivalent to the original one.

Collating problems such as *Magic Mailing* present a complication. The input streams – here, one for letters and one for reply cards – cannot be understood in isolation from one another; we need to impose a structure on each stream based on its relationship to the others. In the *Magic Mailing* solution in Jackson's book, each stream is represented as an iteration of matched or unmatched records (Figure 5). The report is just an iteration of lines, each having one of the four forms (Figure 6).

To obtain the correspondences (Figure 7), we match each type of output record to its corresponding inputs: type 1 when both input files have matched records, type 2 when a letter record is unmatched, and type 3 when a reply record is unmatched. A type 4 output has no correspondence in the input, since it represents the case in which a record is missing from both files.

The operations are easily identified, and allocated using single readahead, resulting easily in the final program:

```
lrec = read (lfile);
rrec = read (rfile);
n = 1;
while (n < 9999) {
    if (lrec.num == n == rrec.num) {
        write_type_1 (n);
        lrec = read (lfile);
        rrec = read (rfile);
        }
    if (lrec.num == n != rrec.num) {
        write_type_1 (n);
        lrec = read (lfile);
        }
    if (lrec.num != n == rrec.num) {
        write_type_1 (n);
        rrec = read (rfile);
        }
    if (lrec.num != n != rrec.num) {
        write_type_1 (n);
        }
    n = n + 1;
    }
```

Although this final program is correct, the method is not, strictly speaking, applied correctly. As Hoare notes, there is no correspondence between `LREC`, `RREC`, and `LINE`, since there may be more lines that letters or reply cards. The correct solution is to elaborate the input structures further, to distinguish records that are present from records that

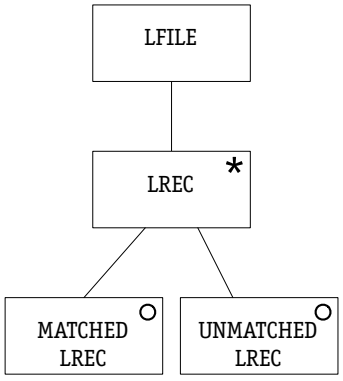

Fig. 5: Structure for letter file of Magic Mailing Problem

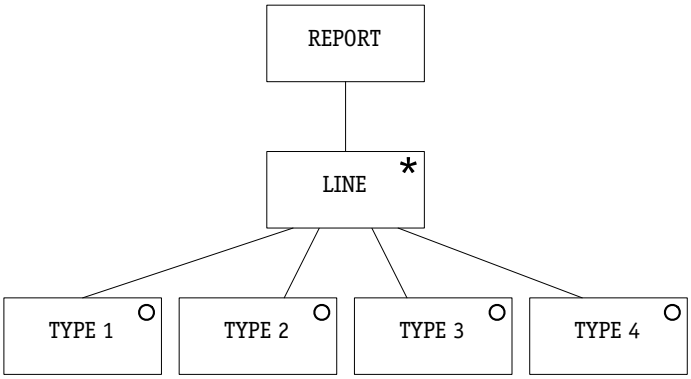

Fig. 6: Structure for report of Magic Mailing Problem

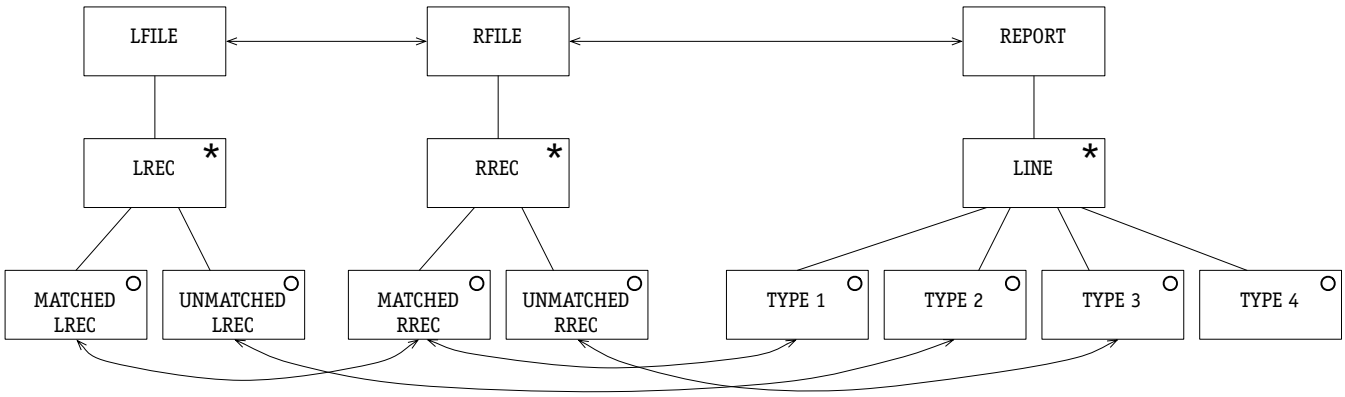

Fig. 7: Correspondences for Magic Mailing Problem

are missing. Jackson does in fact show how to perform this elaboration [4, p.72], but appears not to insist on its use.

The Michael Jackson Design Technique: A study of the theory with applications

C.A.R.Hoare

1 Programs, Traces and Regular Expressions

The execution of a computer program involves the execution of a series of elementary commands, and the evaluation of a series of elementary tests (which, in fact, the Michael Jackson technique tends to ignore). It is in principle possible to get a computer to record each elementary command when it is executed, and also to record each test as it is evaluated (together with an indication whether it was true or false). Such a record is known as a 'trace'; and it can sometimes be helpful in program debugging.

It is obviously very important that a programmer should have an absolutely clear understanding of the relation between his program and any possible trace of its execution. Fortunately, in the case of a structured program, this relation is very simple – indeed, as Dijkstra pointed out, this is the main argument in favour of structured programming. The relation may be defined as follows:

(a) For an elementary condition or command (input, output, or assignment), the only possible trace is just a copy of the command. This is known as a 'terminal symbol', or 'leaf', of the structure tree.

(b) For a sequence of commands (say `P ; Q`), every possible trace consists of a trace of `P` followed by a trace of `Q` (and similarly for sequences longer than two).

(c) For a selection (say `P ∪ Q`), every possible trace is *either* a trace of `P` or a trace of `Q` (and similarly for selections of more than two alternatives).

(d) For an iteration (say `P*`), every possible trace is a sequence of none or more traces, each of which is a (possibly different) trace of `P`. Zero repetitions will give rise to an 'empty' trace `<>`. Thus it can be seen that a program is a kind of regular expression, and that the set of possible traces of the program is the 'language' defined by that expression. In fact, the 'language' will contain many traces that can never be the result of program execution; but for the time being it will pay to ignore that fact.

Of course, a complete trace of *every* action of a program will often be too lengthy for practical use. We therefore need a *selective* trace, which is a record of all actions affecting some particular variable, or some particular input or output file. (Note that a complete trace is a merging of all the selective traces derived from it; but it is not possible to work out which merging it was from the selective traces alone.)

A record of all input instructions on a particular input file will be effectively the same as a copy of the particular data presented to the program on that particular execution of it; and similarly a selective trace of output instructions will be nothing but a copy of the output file. For program debugging such traces would never be required. But the idea of the selective trace is the basis of the whole Michael Jackson Design Technique.

2 Extracting Selective Programs

The theoretical foundation for the technique is that there is a very close relation between a complete program, defining a set of complete traces, and the 'selective' or 'partial' programs defining the sets of all selective traces; indeed, a selective program can be derived automatically from a complete program by the following algorithm:

(a) An elementary command is retained if it refers to the selected variable or file; otherwise, it is replaced by the empty trace <>.

(b) A sequence P ; Q is replaced by P′; Q′, where P′ is the selective program derived from P and Q′ is the selective program derived from Q. If P′ is <>, then it may be omitted (leaving Q′), and similarly if Q′ is <>.

(c) A selection P ∪ Q is replaced by P′ ∪ Q′ where P′ and Q′ are the corresponding selective programs. In this case, an empty program element must not be omitted. However, if P′ and Q′ are *identical*, then one of them can be suppressed.

(d) An iteration P* is replaced by (P′)* where P′ is the selective program derived from P. If P′ is <>, then (P′)* should be taken simply as <> instead of <>*. In P*; P*, the second P* can be suppressed.

Let us taken an example, the *Multiplication Table* problem. The complete program is:

```
r = 1;
(
   c = 1;
   clear ();
   (
      append (r × c);
      c = c + 1
   )* ;
   write ();
   r = r + 1
)*
```

The selective program for r is:

```
r = 1; (r = r + 1)*
```

The selective program for c is

```
(c = 1; (c = c + 1)*)*
```

The selective program for output is

```
(clear (); append ()*; write ())*
```

Jackson recommends that all these programs should be written as tree diagrams, which can easily be done. The main advantage of doing so is that every part of the regular expression must be given its own name. This can also be done by splitting the definition of the regular expression, for example:

```
TABLE  ≙ r = 1; LINE*
LINE   ≙ c = 1; clear (); NUMBER*; write (); r = r + 1
NUMBER ≙ append (r × c); c = c + 1
```

But there remains the advantage of the tree-picture, since it avoids repetition of the 'nonterminal symbols' (LINE, etc); and it prevents recursion.

Now let us suppose that the non-terminal symbols are included in the program, as 'commands' without any effect, except that they are recorded as part of the trace of program execution. We say that a program is *left annotated* if the nonterminal symbol precedes the expression which defines it, and that it is *right annotated* if the nonterminal symbol follows the expression which defines it. These concepts correspond to preorder and postorder traversal of the corresponding tree. For example, the left-annotated version of the *Multiplication Table* problem would be:

```
TABLE;
r = 1;
(
   LINE;
   c = 1;
   clear ();
   (
      NUMBER;
      append (r × c);
      c = c + 1;
   )* ;
   write ();
   r = r + 1
)*
```

And the right–annotated version would be:

```
r = 1;
(
   c = 1;
   clear ();
   (
      append (r × c);
      c = c + 1
      NUMBER
   )* ;
   write ();
   r = r + 1;
   LINE
)* ;
TABLE
```

The advantage of these annotated programs is that we can obtain selective traces and corresponding selection programs that print out only selected *nonterminal* symbols; and thereby we greatly shorten the traces we need to consider. For example, a selective program for LINE and NUMBER would be

```
(LINE; NUMBER*)*
```

which is identical to the tree diagram on which the original structure was based.

Consider now a selective trace containing just two commands (or nonterminals) `a` and `b`. It may happen that the occurrences of `a` and `b` strictly alternate, thus:

`a; b; a; b; ...; a; b.`

A selective program for this trace is simply `(a; b)*`. If *every* such selective trace of a program displays this behavior, we say that the actions `a` and `b` *correspond* in that program. The correspondence of actions is indicated in Michael Jackson's diagrams by drawing a double–ended arrow between boxes containing the actions `a` and `b`.

3 Merging Selective Programs

The basic insight of the Michael Jackson Design Technique is that in the design phase of the program construction, the selection algorithm for regular expressions can profitably be *reversed*; that is that the programmer should first design his *selective* programs, and then he should arrange them to obtain the complete program. The important selective programs are those relating to the input and output files; fortunately, these can usually be designed quite simply, given the specification of the problem to be solved. There remains only the task of merging them, together with the necessary operations on internal variables.

We shall consider the task of merging only *two* selective programs at a time. We will call these the *left* program (usually the input) and the *right* program (usually the output). We shall assume that these are fully annotated, like Michael Jackson's trees; and in considering the traces, we will assume that the left program is left-annotated, and the right program is right-annotated.

Consequently if two nonterminal symbols (internal nodes in the tree) are found to correspond, this means that the *start* of execution of the left subtree below the left node will always strictly alternate with *completion* of the execution of the subtree below the right node of the pair.

In principle, it is always possible to solve the merging problem in the wholly trivial fashion, by putting the entire left program (for input) before the entire right program (for output), thus: `INPUT; OUTPUT`. This would mean that the entire input file is read and stored in internal variables of the program; and only when input is complete does the output start. But this trivial solution will usually be totally impractical, since there will not be sufficient room in internal storage to hold the entire input file before output starts. We therefore wish to obtain a program that economises on space by reusing internal storage occupied by input data as soon as possible after the data has been input.

In principle, the best way to ensure this is to move the *output* instructions as *early* as possible in the merged program, because in general it is possible to overwrite input data immediately after execution of the last output instruction that actually *uses* the data.

So it is a good idea to try to plan for early output even before designing the storage layout of the input data, or how it is going to be reused. (Of course, early output doesn't guarantee success; but success will depend on it).

The basis of the technique recommended by Michael Jackson is the recognition of correspondences between the paired actions, in which an action of the left program

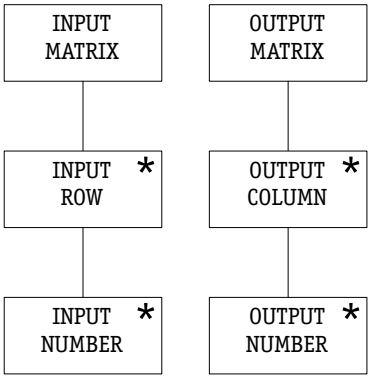

Fig. 8: Structures for Matrix Transposition Problem

is paired with some action from the right program. Recall that corresponding actions alternate in every execution of the program. As explained above, the *entire* input program can *always* be paired with the *entire* output program, since the start of the top nonterminal symbol of the one is always followed (exactly once!) by the end of the top nonterminal of the other. So the essence of the technique must be to find paired actions lower *down* in the tree as well, because this is what is required to permit output operations to be more tightly interleaved with input.

There are two conditions necessary for successful interleaving of a left and right program:

(a) the structures of the two programs must match;
(b) the proposed correspondences between the actions must be feasible.

The second of these conditions is the more difficult. Consider, for example, the *Matrix Transposition* problem (Figure 8). Here we seem to have a good 'structural' match (condition (a)): as always, the top level nodes – here `INPUT-MATRIX` and `OUTPUT-MATRIX` – can be paired. Apparently also, `INPUT-ROW` can be paired with `OUTPUT-COLUMN`. Of course, this would not work for a non-square matrix, which has more rows than columns, or *vice versa*. But this possibility cannot be detected from the diagrams; it must be supplied by the intuition of the programmer. Suppose the programmer *knows* that the input matrix is square, with the same number of rows as columns. Even so, the pairing won't work, because it is *impossible* to print the first column after input of only the first row, simply because there is not yet enough input information available to do so. In fact, output of the *first* column cannot take place until after input of the *last* row. (This example is well explained in Jackson's book [4, pp. 151–153]). The point that I wish to make here is that the identification of the corresponding pairs requires considerable insight from the programmer, and an understanding of what input data is required for the calculation of the value of each item of data output.

When the conditions cannot be satisfied and no interleaving is possible, we have, in Jackson's terminology, a *structure clash*. This is, in fact, a relative term. A structure clash can occur anywhere in the tree, when it proves impossible to perform an output instruction until a whole group of input instructions has been completed. If there is

enough room in main store to hold a sufficiently detailed summary of the input data, then the structure clash causes no problem, and can be ignored. If there is *not* enough room, other measures must be adopted: for example, the use of an intermediate data file, sorting, program inversion, etc. But these are just more sophisticated methods of saving main storage, which must be used when the simpler method of interleaving has failed (at any level). They will not be further mentioned in this study.

Let us move away from the extreme of structural mismatch to the other extreme of perfect structural match, in which every component of the left program is paired with some component of the right program, and *vice versa*, and furthermore every member of a sequence is paired with a member of a sequence, every member of a selection is paired with a member of a selection, and every iteration is paired with an iteration. Actually, it does not matter if there are some completely unmatched subtrees in either of the programs; e.g., the structures of Figure 9, for example, are perfectly matched.

It is now a completely mechanical task to construct a merged program from the left and right programs, together with their pairings. First draw an identical copy of either tree, as far as it matches the other (Figure 10); the two trees are, of course, identical up to that point. Then extend each leaf of the tree as a sequence of two elements, the left one taken from the left program, and the right one from the right program (Figure 11). Any subtrees should be copied too.

4 Transformations

The merging we have considered so far is only the simple case. In general, the left program will have quite a different tree structure from the right program. In this case, it is impossible to apply the simple procedure described above, even when the pairing lines can validly be drawn between the boxes. The solution to this problem is to transform one or both the programs to an equivalent program in such a way that there is an exact match between the paired items. We are therefore interested in transformations that preserve the meaning of a program. In general, the transformations will be the *reverse* of the simplifications that can be made when a program is separated by the process described above (in Section 2). Let us try to catalogue such transformations.

The simplest transformation is just the insertion of an extra box on a line (Figure 12). This is required in the *Stores Movement* problem (Figure 13). Of course, after a bit of practice, one would not always wish to make this elementary transformation explicitly; but to begin with, it may be a good idea to realize the necessity for it.

Another very common transformation is the introduction of a selection, by simply copying complete subtrees (as in Figure 14, where B′ and B″ are identical copies of the subtree B). This transformation is required when the output selective program makes a distinction (like MATCH and NOMATCH) that is not made explicitly in the definition of the input. This technique is the converse of the simplification $Q \cup Q \rightarrow Q$ in regular expressions, and relies on the fact that the same set of languages is defined before and after the transformations.

In general, multiple structures must be brought into correspondence together. In the *Magic Mailing* problem, for example, there are three structures: the letter file, the reply

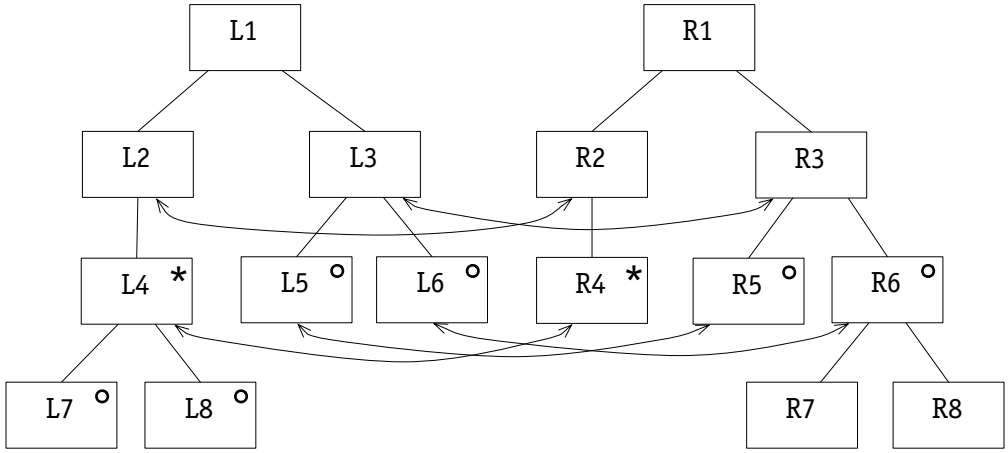

Fig. 9: Structures that are perfectly matched despite unmatched subtrees

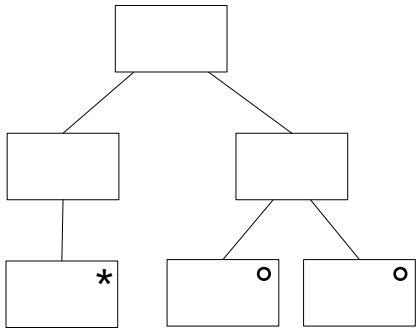

Fig. 10: Constructing a merge: first step

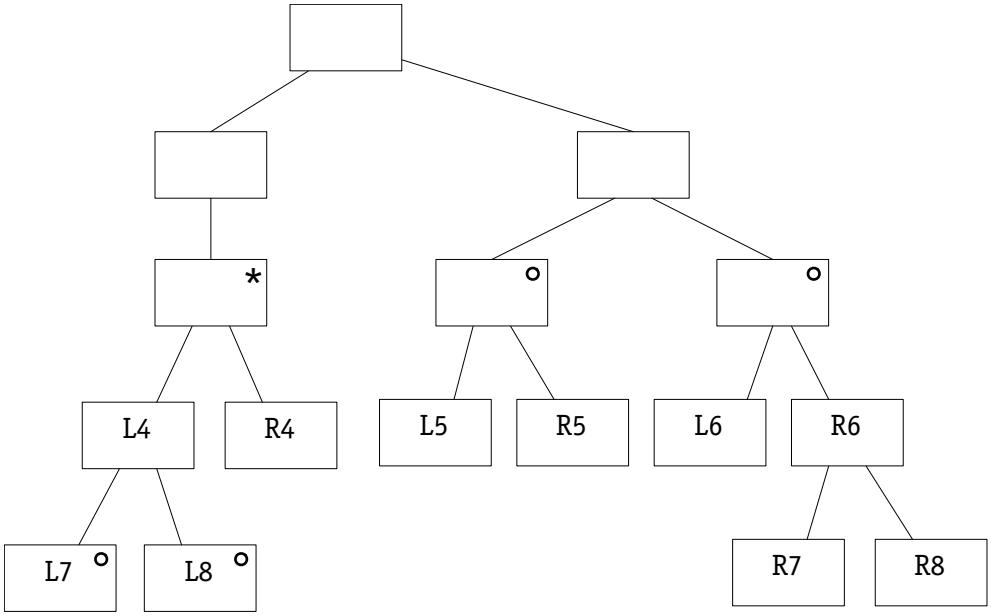

Fig. 11: Constructing a merge: second step

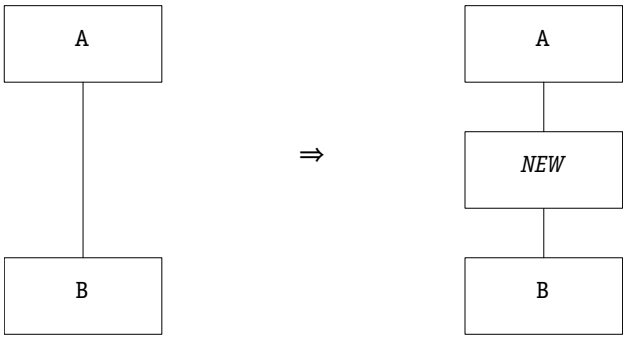

Fig. 12: Simple transformation: adding structure
Before transformation (left); after transformation (right)

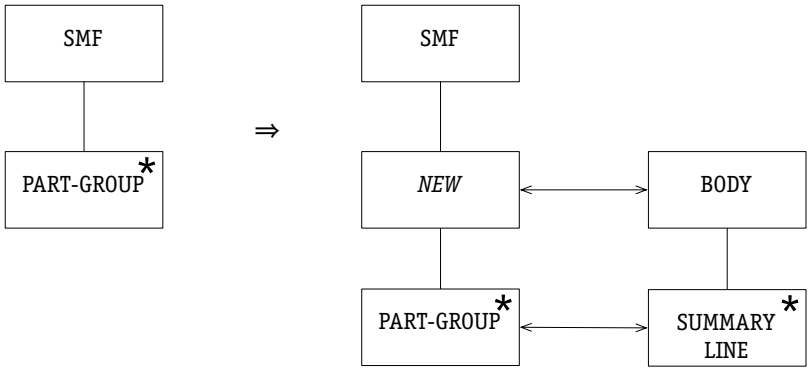

Fig. 13: Applying transformation to Stores Movement Problem
Before transformation (left)
After transformation (right), with correspondences to output stream shown

file, and the report. Here, we shall deal with only two of the tree structures at a time. This is always a valid simplification, and will always find a successful solution if one exists – though it might be a good idea to get a rigorous proof of this fact.

Consider the reply file and the report (Figure 15). It is clear that RREC cannot be paired with LINE, since there are *more* lines than RRECs. The *only* solution to this problem is to introduce a selection (Figure 16), with an empty alternative to correspond to the lines which are written *without* reading an RREC.

This transformation can be made without looking any further down the tree from the LINE* component. The validity of the transformation relies on the fact that the languages defined by

R*

and

<> ∪ R*

are identical. (Perhaps this is a simple case of Jackson's 'interleaving clash' and its resolution). In the *Magic Mailing* problem, it turns out that two further copies of the empty

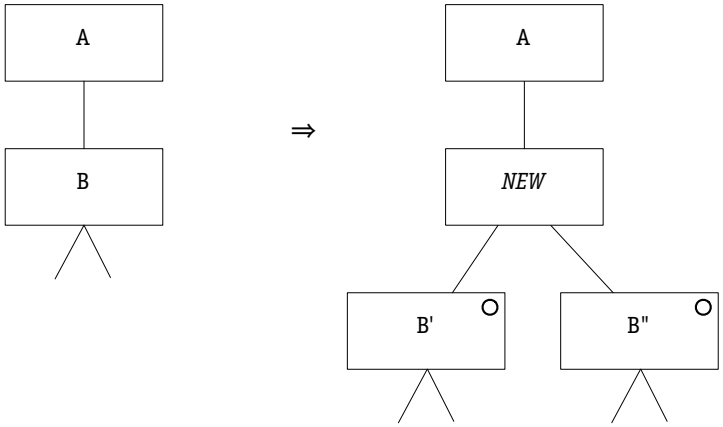

Fig. 14: Introduction of a selection, based on the identity Q = Q ∪ Q
Before transformation (left); after transformation (right)

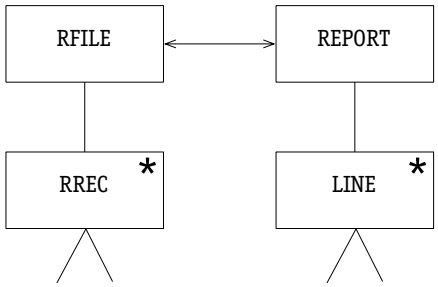

Fig. 15: Unmatched structures in the Magic Mailing Problem

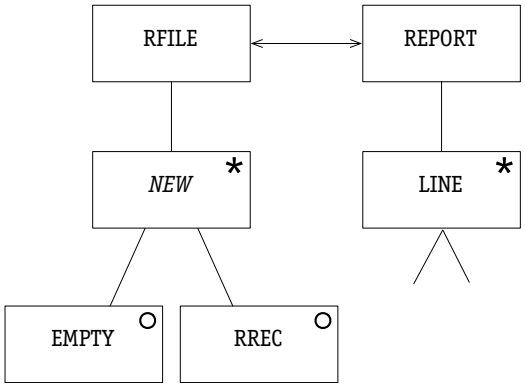

Fig. 16: Applying the transformation based on the identity
R* = <> ∪ R* *to the Magic Mailing Problem*

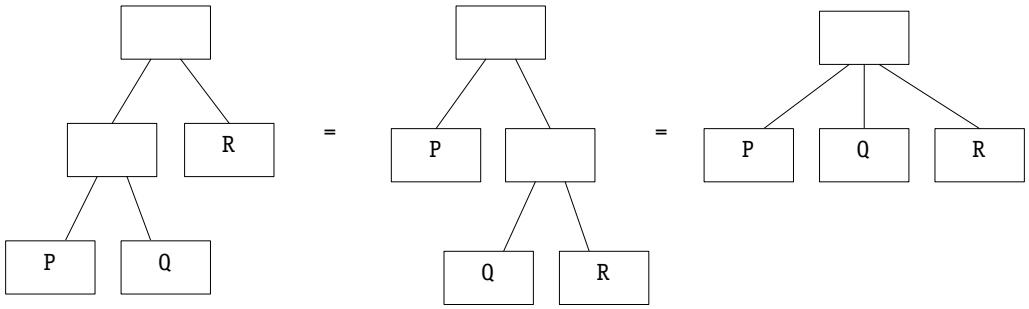

Fig. 17: Transformations based on the identity (P; Q); R = P; (Q; R) = P; Q; R

alternative must be made to match the structure of the line. This splitting of alternatives has already been justified by the fact that (R ∪ R) = R. Then exactly the same transformations must be made in the letter file. In Jackson's solution, all these transformations are made implicitly and simultaneously on both files, thereby obscuring the essential simplicity of the operations.

These two examples of identities over regular expressions can be used to transform the left and right selective programs to make them match prior to merging. It might be interesting to list some of the simpler identities, and construct examples of their useful application; eg.

(P; Q); R = P; (Q; R) = P; Q; R

which is shown diagramatically in Figure 17. Similar diagrams can be drawn for the following:

(P ∪ Q) ∪ R = P ∪ (Q ∪ R) = P ∪ Q ∪ R
P; <> = <> ; P
P ∪ P = P ∪ P ∪ P = P
P ∪ Q = Q ∪ P
R* = (R ∪ <>) *
R * = (R ∪<>); R* = R*; (R ∪ <>)
P; (Q ∪ R) = (P; Q) ∪ (P; R)
(P ∪ Q) ; R = (P; R) ∪ (Q; R)

and so on.

The correct application of these rules could, of course, be checked by computer. But, unfortunately, the computer cannot help in applying the rules in finding correspondences, because this depends intensely on the programmer's intuition about what needs to be done before what. Nevertheless, we can now give the programmer very much more explicit advice about how to apply Michael Jackson's Design Technique.

(a) Initially, one need consider only two partial programs at a time; any others can later be merged with the result of the first two, one after the other.

(b) Start by drawing the correspondence between the root nodes of the tree. Then consider only *one pair of nodes* at a time, one from the left program and one from the right. Check that all nodes higher than the considered nodes have

already been successfully paired (that is, all nodes on the path back to the root have been paired).

(c) If the nodes cannot be paired, try transforming one or both of the nodes by any of the valid transformations listed above.

(d) If you can't succeed, abandon this pair of nodes, and try any other possible pair.

(e) When the tree of paired nodes has been extended as far as possible, check that there is enough room in main storage to accommodate the input data before it is output. This must be done at every point where it was impossible to obtain a correspondence.

(f) If not, try to establish *further* pairs, even though the parent nodes have not been paired. If this can be done, then we have an example of Michael Jackson's 'structure clash' (in this case, a 'boundary clash') which can be resolved by his technique of an intermediate working file, and by program inversion.

Here again, the establishment of the pairs requires an understanding of the nature of the problem, and cannot be done automatically. For example, in the *Telegrams Analysis* problem [4, pp. 155–160] the 'words' of the input file are in the same order as the 'words' required by analysis; but in the case of the numbers of the *Matrix Transposition* problem, this is not so (even though there are the same number of them!). Only the human programmer can distinguish between the two cases.

5 Summary

To summarise this rather rambling discourse, I will give brief answers to a number of questions which were posed before the start of the study.

What is the meaning of a structure diagram?

It is a pictorial representation of a regular expression; the language defined by the regular expression contains all possible traces of execution of the commands which feature as terminal nodes of the tree.

Are there any other programs that cannot be written by the Michael Jackson Design Technique?

In principle there are not; because every program trace is a repetition of the union of all the terminal commands (ie, the regular expression comprising the set of *all* strings over the given alphabet). But this description would be extremely unhelpful. The Michael Jackson Technique is most useful when it is possible to construct rather deep structure trees, and when the structure of the program execution is determined by the structure of its input/output data. It is less useful for the development of clever algorithms, where the content of the program is difficult to determine, and the course of the calculation is dominated by tests on the internal data (ie, while-conditions and if-conditions).

What is the meaning of the 'correspondences' drawn as horizontal double-ended arrows between nodes in two structure diagrams?

A pair of nodes correspond if for *every* execution of the program, the start of execution of the left node alternates strictly with end of execution of the right node.

What is the taxonomy of tricks available to transform a structure to bring it more into correspondence with some other structure?

It is possible to use any valid identity between regular expressions. A number of these identities can be listed and illustrated by example.

Is it possible to write a computer program that will check whether an original and a transformed structure diagram describe the same data or program?

Yes; but the program would be polynomial space-complete (see [12, 13]). This means that there is no efficient algorithm known; the best known algorithms are based on an exhaustive search of all possible methods of matching, which will be quite inefficient in many cases. But perhaps a more serious problem is that we probably do not want an *exact* isomorphism between the languages described; there is no unique way of describing a program as a regular expression; and the programmer should be at complete liberty to give slightly more accurate or less detailed descriptions of his data, if this will help him to find useful correspondences.

Is it possible to detect whether correspondences have been correctly filled in?

Yes. It is trivial, provided that *all* the needed extra nodes have been inserted. All that is necessary to check is that whenever two nodes correspond,

(a) Their parent nodes correspond.
(b) They are the same kind of nodes (o , * , or plain).

In fact, Michael Jackson does not usually put in all the extra nodes and lines that are needed. Presumably it would be possible to define some machine-checkable conventions which would permit these extra nodes to be omitted. But for teaching purposes, it might be better to make the students write them in, to begin with.

What is a structure clash?

A structure clash is whatever prevents the chain of correspondences between branches of two diagrams to be extended.

(a) Some structure clashes can be resolved by making valid transformations on one or both structures.
(b) Some structure clashes can be tolerated because there is room enough in main store to hold all of the data required.
(c) In some cases the clash is not resolvable, since the required output data simply cannot be computed from the preceding input data. There is no automatic way of detecting this case; it must rely on the programmer's judgment.
(d) In case (c) it sometimes happens that a correspondence *can* be established lower down in the tree. In this case we have a definition of a boundary clash, for which Jackson's methods provide an excellent solution.

How do you list all the executable operations which will be required, and how do you put them in the proper place?

This is in principle the same task as finding correspondences between two structure diagrams. I would recommend that a structure diagram should be drawn for each in-

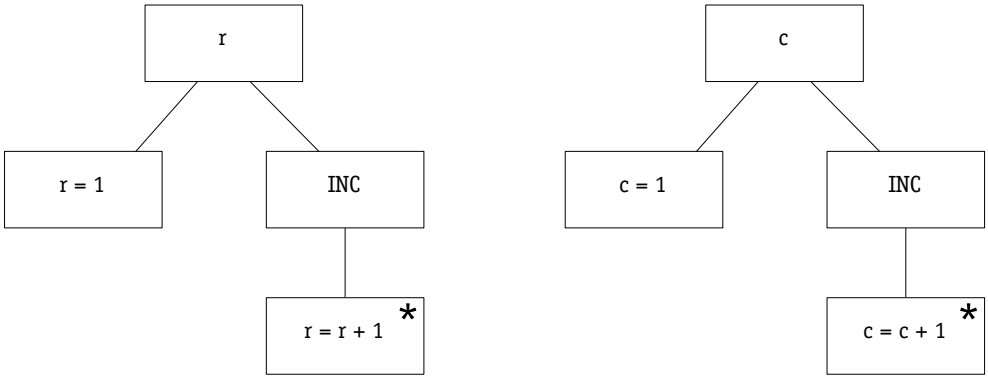

Fig. 18: Structure diagrams for variables

ternal variable of the program, just as it is for the input and output. For example, in the *Multiplication Table* problem, we could construct diagrams for c and r; and then establish the correspondences and merge the diagrams in the same way as any other structure diagrams (see Figure 18).

How much of the Michael Jackson Technique relies on programmer judgment, and how much is just routine formal manipulation?

All the part that involves construction of structure diagrams, and established of correspondences, relies on the judgment of the programmer. The great advantages of the Michael Jackson Technique are that:

(a) It provides a framework within which the programmer can make *one decision at a time*, with high confidence that all decisions will mesh properly with each other.
(b) It provides a framework within which he can describe and justify his decisions to his managers, colleagues, subordinates, and successors; and gives these the ability to understand them.
(c) It provides a framework within which alternate decisions can be easily seen, and cheaply explored, until the best decision is recognized and adopted.
(d) It ensures that program documentation is constructed during *program design*, when it is most useful. The usual practice of documentation after testing is *guaranteed* to be useless. If the documentation is not a help to the program writer himself, it will hardly be helpful to the unfortunate maintainer!

Nevertheless, there are several aspects of the Michael Jackson Technique that are routine:

(a) Given a program, it is possible to work back to the original structure diagrams from which it was created. This is a useful check on the correctness of the program, before it is tested.
(b) The final construction of a COBOL program is fairly routine; though there are still good opportunities for skill in local optimization. This task can therefore

be delegated to coders, who do not have the necessary judgment to design the program, or to postpone their passion for optimization to a stage when it is relatively harmless.

Are there any suggestions for shortening the learning time for the Michael Jackson's Technique?

I think it may help to give exercises which show how the structure diagram 'generates' the input data, the output data, and the traces of program execution. Also, perhaps some exercises in constructing selective traces, and selective programs *from* the complete ones. I believe that it is useful in teaching to show how to *check* the solution to a problem *before* trying to get the student to find his own solution. Similarly for correspondences, a list of permissible transformations to a structure should help considerably.

The consideration of only *two* structure diagrams at a time, and only *one* pair of nodes at a time should be helpful. The use of the *same* technique of structure diagrams and correspondences for internal calculations should help too.

Are there any improvements in the Michael Jackson technique that you could recommend before it is packaged for use in the U.S.A.?

Apart from the possible changes in teaching strategy listed in my answers above, I would be very reluctant to suggest any changes without considerable further thought and experience. The fact is that Michael Jackson is one of our most brilliant original thinkers in the field of programming methodology. Not only is his academic distinction of the highest; he also has long experience of commercial data processing, and has seen his ideas tested in practice over a wide range of environments and applications. Finally, he is a brilliant and experienced teacher; and he has as good understanding of the capabilities and limitations of his likely audience, and he has adapted his material to meet them. That is why I am not qualified to make authoritative suggestions for change in his Technique. If I were so bold to suggest a change, I would very much prefer to discuss the suggestion thoroughly with Michael Jackson himself, not only because he would be the most qualified to comment, but because I have always found in him the true scientific spirit, which is eager to understand and evaluate ideas different from his own; and admit freely if he finds them better.

References (Introduction)

[1] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, 2001.

[2] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.

[3] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[4] Michael Jackson. *Principles of Program Design*. Academic Press, 1975.

[5] Michael Jackson. The Origins of JSP and JSD: a Personal Recollection. *IEEE Annals of Software Engineering*, Volume 22, Number 2, pp. 61–63, 66, April–June 2000.
[6] Michael Jackson. JSP In Perspective. *SD&M Pioneers' Conference*, Bonn, June 2001.
[7] Michael Jackson. *System Development*. Prentice Hall, Englewood Cliffs, New Jersey, 1983.
[8] Michael Jackson. *Software Requirements and Specifications: A Lexicon of Principles, Practices and Prejudices*. Addison-Wesley and ACM Press, 1996.
[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming. *Proceedings of the 11th European Conference on Object-Oriented Programming*, Mehmet Aksit, Satoshi Matsuoka (Eds.), Jyväskylä, Finland. *Lecture Notes in Computer Science*, Vol. 1241. Springer-Verlag, June 1997, pp. 220-242.
[10] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Bell System Technical Journal* 57, 6, Part 2 (1978) pp. 1905–1930.
[11] Jean Dominique Warnier. *Logical Construction of Systems*. John Wiley & Sons, Inc., 1981.

References (Original Paper)

[12] Alfred V. Aho, J.E. Hopcroft, Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. Page 403, Exercise 10.28.
[13] L. Stockmeyer & A.R. Meyer. Word Problems Requiring Exponential Time. *Proc. 5th Annual ACM Symposium on Theory of Computing*, May 1973, pp. 1–10.

Acknowledgments (Introduction)

Daniel Jackson is grateful to Maria Rebelo for entering the text of the original manuscript; to Dick Nelson (the originator of the name 'Jackson Structured Programming') for retrieving old course notes; to Michael Jackson for answering questions about JSP and its history; and to Tony Hoare for his encouragement in the publishing of this paper. The historical comments in the introduction are drawn in part from earlier papers [5, 6].

4.2 Text - Correspondence

The following is a simple problem involving two data structures - one input data structure and one output data structure.

'The stores section in a factory issues and receives parts. Each issue and each receipt is recorded on a punched card: the card contains the part-number, the movement type (I for issue, R for receipt) and the quantity. The cards have already been copied to magnetic tape and sorted into part-number order. The program to be written will produce a simple summary of the net movement of each part. The format of the summary is:

STORES MOVEMENTS SUMMARY

A5/132	NET MOVEMENT	-450
A5/197	NET MOVEMENT	1760
B41/728	NET MOVEMENT	7

. . .

No attention need be paid to such refinements as skipping over the perforations at the end of each sheet of paper.'

The first step of the design procedure, the data step, is to draw data structures of all the files in the problem. The result of the data step is:

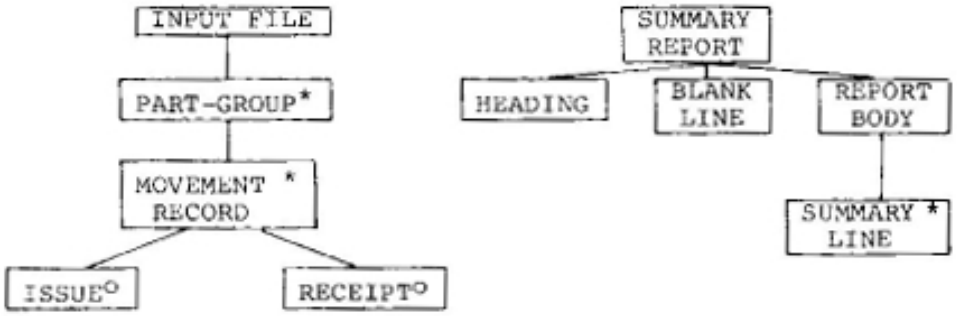

Fig. 19: Sample page from original course notes

The Michael Jackson Design Technique (0164)

A study of the theory with applications.

C. A. R. Hoare

March 1977.

The execution of a computer program involves the execution of a series of elementary commands, and the evaluation of a series of elementary tests.*
It is in principle possible to get a computer to record each elementary command when it is executed, and also to record each test as it is evaluated,* (together with an indication whether it was true or false). Such a record is known as a "trace"; and it can sometimes be helpful in program debugging.

It is obviously very important that a programmer should have an absolutely clear understanding of the relation between his program and any possible trace of its execution. Fortunately, in the case of a structured program, this relation is very simple — indeed, as Dijkstra pointed out, this is the main argument in favour of structured programming. The relation may be defined as follows:

* In fact the Michael Jackson technique tends to ignore these tests.

Fig. 20: Sample page from Hoare's original report