# Structure Clashes – An Alternative to Program Inversion

R. G. DROMEY* AND T. A. CHORVAT

*Programming Methodology Research Group, Computing and Information Technology, Griffith University, Nathan, Brisbane, 4111, Australia*

***A method for handling boundary structure clashes is presented. It provides a simple alternative to Jackson's program inversion technique. It is a constructive approach based on the use of prototypes. Development begins by solving the corresponding simpler problem that has no structure clashes. This solution is then used to guide the solution of the original problem. Programs constructed in this way preserve the correspondence between the program control structure and the data structure. They should therefore be easier to maintain than their counterparts developed using Jackson's program inversion method.***

*Received May 1988, revised August 1988*

## 1. INTRODUCTION

Often the processing of information is not synchronized. This results in structure clashes [Jackson-75]. A *boundary structure clash* is defined as an incompatibility between two or more data structures involved in a transfer of information. A situation where data must be read in records of one size and written in records of another size is typical of such problems. In this case, the input of records and the output of records is not synchronized. If such problems are not carefully and systematically dealt with they can lead to complex, and potentially error-prone, programs.

The existing method of solving boundary structure clash problems involves decomposition of the problem with respect to data structure into two or more simpler problems. The solutions to these simpler problems are then composed to provide the total solution. Standard techniques for implementing the composition involve use of co-routines and separate processes communicating through a shared buffer or pipe. When these facilities are not available Jackson's technique of program inversion can be used.

The advantage of Jackson's method lies in the relative simplicity of the functions produced by the decomposition with respect to data structure and the straightforward way inversion composes those functions [Hughes-79]. However, decomposition destroys that correspondence between the program structure and the original data structure. There is then a need to compose the functions and to communicate data and control information between them. This adds to the complexity of the solution.

The question that needs to be asked is whether the added complexity and loss of correspondence associated with decomposition and inversion are really necessary for solving boundary structure clash problems. Hughes [Hughes-79] obtained a result that shows the boundary structure clashes can be solved using Jackson's basic method of matching data structure and program structure. The *forced synchronization* of loops suggested here realises this result.

The forced synchronization method for resolving boundary structure clashes relies on the application of a prototyping strategy. What we are suggesting is *that the basic program structure used in solving the problem where there is no structure clash should be preserved in solving the corresponding problem where there is a structure clash.* The control structure of the prototype solution should be designed to match the data structure in the sense advocated by Jackson.

The guiding structuring principle for prototyping is the *principle of structural simplification*. It may be stated as follows:

*Principle of structural simplification*
*"For any problem for which there is a structure clash there is a corresponding simpler problem for which there is no structure clash."*

This principle can be used to identify an appropriate structure for a program that must resolve a structure clash.

## 2. FORCED SYNCHRONIZATION OF LOOPS

To introduce the concept of forced synchronization of loops, we will consider a very simple copying problem.

### 2.1 Buffer-copy problem

Suppose we want to input data from a file consisting of homogenous data objects grouped in records of one size, $M$, and then output the data objects grouped in records of a different size, $N$. Here, the prototype problem corresponds to the case where the input, transfer of information, and output of data are all synchronized (i.e.: $M = N$). The input of data is accomplished by filling the input buffer *in*[1 .. *M*]. This is achieved by a cell to *getrec*, which accepts three arguments – *in*, where to put the data; $M$, the maximum amount of data to get, and $m$, the actual amount of data placed into *in*. In this case the processing is merely to copy the input buffer *in*[1 .. *M*] to the output buffer *out*[1 .. *N*]. The data is written out by a call to *putrec*, whose arguments identify where the data is, and the size of the record to output. All of this occurs before there is a need for a new call to *getrec*. Processing will need to continue until the input is exhausted. *Getrec* returns a value of zero in parameter $m$ when there is no more data to *actually* place into the buffer. Matching the program structure with the data structure using Jackson's JSP method we arrive at the prototype solution below.

* To whom correspondence should be addressed.

Downloaded from https://academic.oup.com/comjnl/article/33/2/126/385835 by guest on 09 April 2024

*Prototype structure**

```
  i := 0;
  j := 0;
  repeat
    getrec(in, m, M);                               {Input}
    do i ≠ m →
      out_{j+1} := in_{i+1}; i := i + 1; j := j + 1;
                  {Process (in this case just a copy)}
    od;
    i := 0;
    putrec(out, m); j := 0;                         {Output}
  until m = 0;
```

With this particular type of problem, as soon as we admit the possibility that the input buffer $in[1..M]$ and the output buffer $out[1..N]$ may be different in length (i.e. $M \neq N$) there is no longer any synchronization between the calls to *getrec* and *putrec*. This creates a loop structuring problem where it is necessary to make a call to either *getrec* or *putrec* or both with each iteration.

In our introduction we suggested that a good strategy for handling problems of this type was to exploit the structure of the prototype solution that had no structure clash in constructing a solution that handles the structure clash.

Examining the structure of our prototype solution to the problem we see that it has the basic form:

```
  repeat
    1. input data
    2. process while not requiring input or output
    3. output the processed data
  until termination condition
```

All that is required to handle the situation where the input and output buffers are different in size is to make the input of data and the output of data *conditional.* That is, we only input data *when it is required*, and we only output data *when it is required.* This preserves the prototype control structure in the solution of the more complex problem. We then have the augmented or synchronized structure:

```
  repeat
    1. if require input then input data
    2. process while not requiring input or output
    3. if require output then output the processed data
  until termination condition
```

With this new structure, because of the conditional input and conditional output, there is either input or output with each iteration of the outermost loop. When the input and output buffers are the same size (i.e. $M = N$) there will be input *and* output with each iteration because the two processes are synchronized. This structure with conditional input and output implements the *forced synchronization* of the input and the output. The component of the structure that implements processing while there is no demand for either input or output is referred to as a *pipe*. The pipe controls the flow of information from the input to the output. It intermittently suspends the flow when either the sink (output) is full or the source (input) is empty. Termination of the information flow through the pipe is achieved using the forced termination technique which has been discussed in detail elsewhere [Dromey-85].

Surprisingly, these very simple guidelines can lead to the straightforward solution of a wide range of problems. Applying forced synchronization to the present problem we get:

*Mismatched copy*:

```
  i := 0;                             {Input initialization}
  j := 0;                             {Output initialization}
  repeat
    if i = 0 then getrec(in, m, M) fi;
                          {Conditional Input initialization}
    p := m;
    do i ≠ p →             {Process while not requiring
                                        input or output}
      if j ≠ N → out_{j+1} := in_{i+1}; i := i + 1; j := j + 1;
      [] j = N → p := i
      fi
    od
    if i = M then i := 0 fi;        {Conditional Input
                                         finalization}
    if j = N then putrec(out, j);       {Conditional
      j := 0                       Output finalization}
    fi;
  until m = 0;
  if j ≠ N then putrec(out, j) fi
                               {Final output finalization}
```

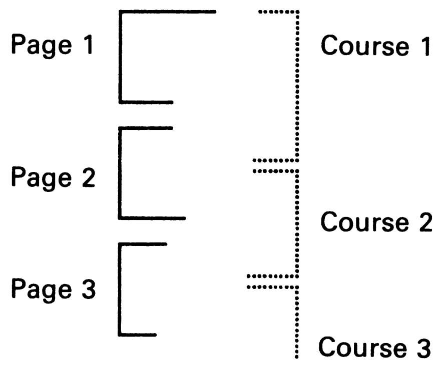
The advantage of the forced synchronization solution to the problem is that it retains the structure of the prototype problem. This keeps input, processing, and output well separated. The separation will make any subsequent change to one of these phases easier.

## 2.2 A Simple Structure Clash

The next example is slightly more complicated. Suppose we have a file of student records (each record consisting of a course descriptor, student name, and mark) which are sorted by course. For example:

| | | |
|---|---|---|
| CS1 | Name1 | 40 |
| CS1 | Name2 | 66 |
| . | . | . |
| CS2 | Name1′ | 75 |
| CS2 | Name2′ | 53 |
| . | . | . |

To process this data it is required to produce a report that is split into pages, with a heading on each page. The file is to be read and student names and marks are to be listed together with the total class count at the end of each course. In this problem there is an arbitrary relationship between course and page. For example:



* For our implementations we have used a variant of Dijkstra's Guarded Commands.

Downloaded from https://academic.oup.com/comjnl/article/33/2/126/385835 by guest on 09 April 2024

The prototype problem in this case is one where each course fits exactly on a page. The solution obtained using Jackson's method may take the form:

*Prototype solution**

```
M = true;
get(data); i := eof(data);           {input initialization}
linecount := 0;
do i ≠ M →
   WritePageHeading(data^.course);           {page
                                   output initialization}
   currentcourse := data^.course; {course output
   classcount := 0;                         initialization}
   m := M;
   do i ≠ m →          {process while there is more
                        data, and it is the same course}
      if data^.course = currentcourse →
         write(data^.name); write(data^.mark);
         linecount := linecount + 1;
         get(data); i := eof(data);
         classcount := classcount + 1
      [] data^.course ≠ currentcourse → m := i
      fi
   od;
   write(classcount); {course output finalization}
   write(PAGEBREAK); {page output finalization}
   linecount := 0;
od;
```

Once again loop control is achieved using the forced termination technique [Dromey-85]. The outer loop will terminate when the input is exhausted. The inner loop will terminate when either there is no more data, or there is a change of course. *m* provides an upper limit on the amount of processing in the inner loop and is used to satisfy proof of termination requirements. *i* always signifies whether the end of the data file has been reached or not.

To handle the case where course and page boundaries do not correspond we need to conditionalize the initialization and finalization mechanisms for input and output and force termination of the inner course-processing loop when a new page is needed, (i.e. when *linecount* = *PAGELENGTH*).

*Forced synchronization solution*

```
M := true
linecount := 0;                  {page initialization}
classcount := 0;               {course initialization}
get(data); i := eof(data);      {input initialization}

do i ≠ M →
   if linecount = 0 then    {conditional page init.}
      WritePageHeading(data^.course)
   fi;
   if classcount = 0 then {conditional course init.}
      currentcourse := data^.course
   fi;
   m := M;
   do i ≠ m →
      if data^.course = currentcourse ∧
         linecount ≠ PAGELENGTH →
         write(data^.name);
         write(dark^.mark);
         linecount := linecount + 1
         get(data); i := eof(data);
         classcount := classcount + 1
      [] data^.course ≠ currentcourse ∨
         linecount = PAGELENGTH → m := i
      fi
   od;
   if data^.course ≠ currentcourse then
      write(classcount);        {conditional course
      classcount := 0                    finalization}
   fi;
   if linecount = PAGELENGTH then
      write(PAGEBREAK);         {conditional page
      linecount := 0                     finalization}
   fi
od;

if classcount ≠ 0 then write(classcount) fi;
                          {course output finalization}
if linecount ≠ 0 then write(PAGEBREAK) fi
                            {course page finalization}
```

* Pascal's I/O has been borrowed and used here.

### 2.3 Multiple Input Structure Clash

The same basic prototyping strategy can easily be extended to handle more than one input and output. As an example suppose we extend our original buffer-copying problem of section 2.1 to a stream addition problem where there are *two* input streams and one output stream. Each of the streams has a different record size.

| *Stream* | *Buffer* |
|---|---|
| Input_A | *A*[1 .. *M*] |
| Input_B | *B*[1 .. *N*] |
| Output | *Out*[1 .. *P*] |

In the prototype problem we assume that the sizes of the records are the same (i.e.: $M = N = P$). Once again we produce the corresponding prototype solution using Jackson's technique of matching data structure to program structure.

*Prototype solution*

```
i := 0;                         {Input_A initialization}
j := 0;                         {Input_B initialization}
k := 0;                          {Output initialization}
repeat
   getrec(A, M, m);             {Input_A initialization}
   getrec(B, N, n);             {Input_B initialization}
   do k ≠ P →
      $out_{k+1} := A_{i+1} + B_{j+1}$;          {Process loop}
         i := i + 1; j := j + 1; k := k + 1
   od;
   i := 0; j := 0;                {Input finalizations}
   putrec(out, k); k := 0        {Output finalization}
until m = 0 ∨ n = 0
```

To resolve the structure clash both the inputs and the output need to be conditionalized and the processing loop must be forced to terminate when either of its inputs are exhausted as well as when the output buffer is filled. Making these refinements we get:

Downloaded from https://academic.oup.com/comjnl/article/33/2/126/385835 by guest on 09 April 2024

*Forced synchronization solution*

```
    i := 0; j := 0;                        {Input initializations}
    k: = 0;                                {Output initializations}
    repeat
¶     if i = 0 then getrec(A, M, m) fi;          {Input_A
                                     Conditional initialization}
¶     if j = 0 then getrec(B, N, n) fi;          {Input_B
                                     Conditional initialization}

      p := P;
      do k ≠ p →  {Information transfer conditional
        if i ≠ m ∧ j ≠ n →   on availability of inputs}
          out_{k+1} := A_{i+1} + B_{j+1};        {Processing}
          i := i + 1; j := j + 1; k := k + 1
        []i = m ∨ j = n → p := k
        fi
      od;
§     if i = M then i := 0 fi;                   {Input_A
                                     Conditional finalization}
§     if j = N then j: = 0 fi;
                        {Input_B Conditional finalization}
      if k = P then putrec(out, k);          {Conditional
        k := 0 fi;                      Output finalization}
    until m = 0 ∨ n = 0;
    if k ≠ 0 then putrec(out, k); fi           {Terminal
                                              finalization}
```

### 2.4 Text Formatting Problem

The next problem that we wish to consider has been discussed by Floyd [Floyd-78]. It may be stated informally as follows:

*'Read lines of text until the input is exhausted. Eliminate redundant blanks between words, and print the text with a maximum of L characters to a line without breaking words between lines. It is also required that leading and trailing blanks should be removed and there should be the maximum number of words possible on each line.'*

In designing a solution to this problem we will first construct a solution for the prototype problem that involves no structure clashes. We will then modify it using forced synchronization to handle the structure clash.

In the prototype solution a line of words is read, the leading and trailing spaces are removed, and multiple spaces between words are discarded. Words are written as soon as they are read.

The prototype implementation obtained by following Jackson's guidelines to handle lines of text containing *one or more* words may take the following form:

¶ Note $m$ and $n$ are zero when there is no more input data.

§ Also note that the conditional finalizations and the conditional initializations may be combined if the same guard is employed. Here, the same condition would be used to signal that all the input has been processed as well as the need to (re)fill the input buffer. The corresponding initialization would indicate that all input has been processed. There are two views of the conditions. Firstly, there is the *transitional* view – where only one guard is required: the condition signals movement from one task to the next. Then there is a more *discrete* view – where there are two distinct guards: one condition signalling the start of a task and another signalling the end of a task.

*Prototype text formatter*:

```
M := true;
i := eof(input);
l := 0;
do i ≠ M →
  getspaces(j); getword(word, w, j);
  getspaces(j);                         {input initialization}
  writeword(word, w);                  {output initialization}
  l := l + w;
  do j ≠ ENDLINE →
    getword(word, w, j); getspaces(j);
                                              {input process}
    write(SPACE); writeword(word, w);
    l := l + w + 1;                          {output process}
  od;
  readln; j := STARTLINE;              {input finalization}
  writeln; l := 0;                    {output finalization}
  i := eof(input);
od
```

To proceed from the prototype solution to handle the structure clash we need to make the initialization for input and output conditional. We also need to make the finalization mechanisms for input and output conditional. The flow of information through the pipe must be suspended when the *next* word to be printed exceeds the output line length limit $L$. In our implementation we use the variable $l$ to hold the current length of the output line and $w$ to hold the length of the next word to be processed. The implementation of the formatter for handling text with non-empty lines takes the following form:

*Text formatter*:

```
M := true;
l := 0; w := 0;                       {output initialization}
j := STARTLINE;                        {input initialization}
i := eof(input);
do i ≠ M →
  if j = STARTLINE then            {start of input line}
    getspaces(j) fi;
  if l = 0 ∧ w = 0 then
    getword(word, w, j); getspaces(j)
  fi;
  if l = 0 ∧ w ≠ 0 then           {start of output line}
    writeword(word, w); l := l + w;
    p := p + w; w := 0
  fi;
  n := ENDLINE;
  do j ≠ n →
    getword(word, w, j); getspaces(j);
    if l + w + 1 ≤ L → write(SPACE);
      writeword(word, w); l := l + w + 1; w := 0
    []l + w + 1 > L → n := j
    fi
  od;
  if j = ENDLINE then
    readln; j := STARTLINE; i := eof(input)
  fi;
  if l + w + 1 > L then              {end of output line}
    writeln l := 0
  fi
od;
```

Downloaded from https://academic.oup.com/comjnl/article/33/2/126/385835 by guest on 09 April 2024

```
if $l = 0 \wedge w \neq 0$ then              {output finalization}
   writeword(word, w); $l := l + w$; $w := 0$
fi;
if $l \neq 0$ then writeln fi
```

Some additional comments about this implementation are in order. With each iteration of the outermost loop there may be an initialization for the *input* or *output* of a new line of text. Furthermore, with each iteration of the outermost loop, there may be invocation of a finalization mechanism to terminate either the current output line, or the current input line.

It is interesting to study the effects of complicating the present problem further by introducing another structure clash. For this we may consider a structure clash resulting from the requirement that there should be at most *P* characters on a page. A page is a sequence of the previously described lines of text. In addition, there should be the maximum number of words on a page.

To accommodate this new structure clash in our most recent implementation, we need to add a conditional initialization for new pages and a conditional finalization mechanism for a new page. The only other requirement is to add a condition that suspends information flow through the pipe when it is necessary to move to a new page. Augmenting our previous implementation with these modifications we obtain a paged-line text formatter.

*Paged-line text formatter*:

```
M := true;
$l := 0$; $w := 0$;              {line output initialization}
$p := 0$;                        {page output initialization}
j := STARTLINE;                  {input initialization}
i := eof(input);
do $i \neq M \rightarrow$
   if j = STARTLINE then getspaces(j) fi; {start of
                                            input line}
   if $l = 0 \wedge w = 0$ then            {start of output line}
      getword(word, w, j); getspaces(j)
   fi;
   if $l = 0 \wedge w \neq 0$ then writeword(word, w);
      $l := l + w$; $p := p + w$; $w := 0$
   fi;
      n := ENDLINE;
   do $j \neq n \rightarrow$
      getword(word, w, j); getspaces(j);
      if $l + w + 1 \leq L \wedge p + w + 1 \leq P \rightarrow$
         write(SPACE); writeword(word, w);
         $l := l + w + 1$; $p := p + w + 1$; $w := 0$
      [] $l + w + 1 > L \vee p + w + 1 > P \rightarrow n := j$
      fi
   od;
   if j = ENDLINE then readln;
      j := STARTLINE; i := eof(input)
   fi;
   if $l + w + 1 > L \vee p + w + 1 > P$ then
      writeln; $l := 0$                  {end of output line}
   fi;
   if $p + w + 1 > P$ then write(PAGEBREAK);
      $p := 0$          {conditional page finalization}
   fi
od;
if $l = 0 \wedge w \neq 0$ then writeword(word, w);
   $l := l + w$; $p := p + w$; $w := 0$
fi;                                    {output finalization}
if $l \neq 0$ then writeln fi;
if $p \neq 0$ then
                  {write the final pagebreak if needed}
   write(PAGEBREAK) fi
```

A comparison of this implementation with the previous implementation suggests that the changes needed to accommodate an additional structure clash are relatively minor provided we conform to the guidelines for conditional initialization and finalization. What is somewhat surprising about the last two implementations is that they both retain the same underlying structure that was used for the prototype problem. Furthermore, the extra effort to handle the additional structure clash is small. It is in this domain where forced synchronization offers advantages over program inversion.

## 3. COMPARISON WITH PROGRAM INVERSION

To make a comparison of program inversion and forced synchronization it is useful to use the Buffer-Copy problem discussed in section 2.1. Figure 1 represents the different design steps required to solve this problem using each of the methods.

With the Program Inversion method there are two occasions where the correspondence with the original problem may be obscured. Firstly, when the problem is decomposed two sub-problems are created. Secondly,

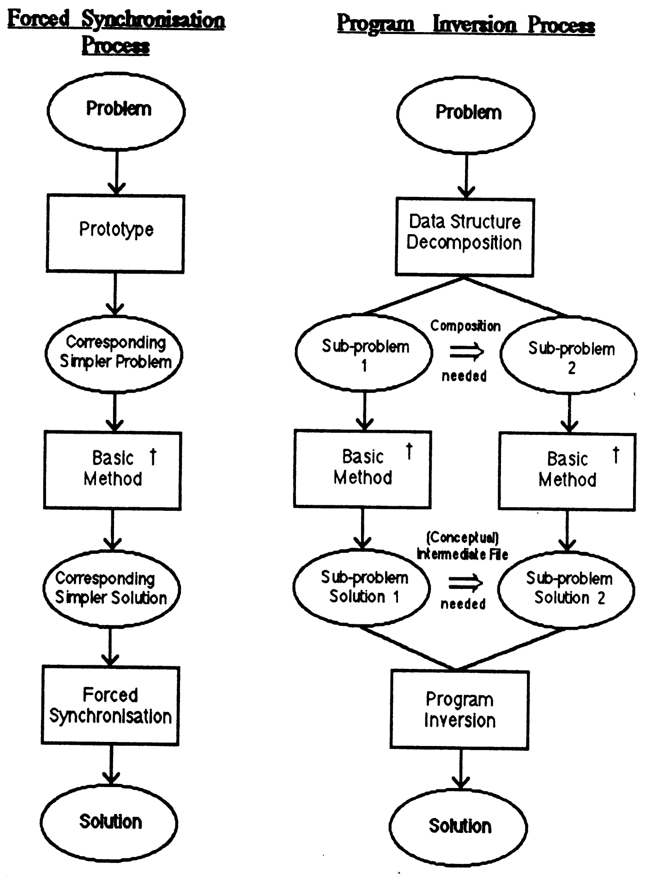


**Figure 1. Alternative methods for solving the Buffer-Copy Problem**

Downloaded from https://academic.oup.com/comjnl/article/33/2/126/385835 by guest on 09 April 2024

when the two sub-problem solutions are composed using program inversion iteration is changed to selection in one of the sub-problems. This sub-problem then becomes subordinate to the other. The solution's main structure corresponds to one of the sub-problems, while a separate procedure is required for the altered solution of the other sub-problem.

With the Forced Synchronization method there are also two occasions where the correspondence with the original problem may be obscured. Firstly, when the problem is prototyped to produce the corresponding simpler problem the number of iterations of each repetition must be adjusted so as to coincide. This is a relatively trivial alteration and the data's structure remains exactly the same. Secondly, when the technique is applied to derive the solution the existing initializations and finalizations are made conditional. This does not alter the program control structure, which still corresponds directly to the original problem's structure.

It is claimed that the solution produced using Forced Synchronization retains a more direct correspondence to the original problem.

It is also claimed that Forced Synchronization provides for a more consistent progression from the synchronized problem to the problem where the structure clash becomes evident. Note that the structure of the data is the same in these two problems. The only change is that the actual number of iterations of the repeated structures are not the same in the structure clash problem.

Consider the situation where we have *already* solved the problem for buffers of the same size (using Jackson's basic method) and we are required to accommodate the more general problem involving buffers of different size. Using program inversion the problem must be reconsidered in order to find the greatest common structure so that the problem may be decomposed. If the problem had warranted the use of decomposition as an abstraction tool then it would have been used for the initial solution. Here, however, decomposition is used solely to handle the boundary structure clash which is now apparent – even though the structure of the data has not changed. So two different program structures are produced for the same data structure where only the number of iterations has changed.

To compare the methods it is useful to describe the input and output using EBNF [Wirth-82]. An arrow ($\rightarrow$) represents the transformation from the input described in the left to the output on the right. Epsilon ($\varepsilon$) is the empty sequence.

Using this notation the original problem may be described by:

$$\{\{\text{byte}\}^M\} \rightarrow \{\{\text{byte}^M\}\},$$

and in the situation where the structure clash occurs we have

$$\{\{\text{byte}\}^M\} \rightarrow \{\{\text{byte}\}^N\},$$

where $M \neq N$.

With Forced Synchronization for the structure clash problem we have

$$\{\{\text{byte}\}^M \mid \varepsilon\} \rightarrow \{\{\text{byte}\}^N \mid \varepsilon\}.$$

† The Basic Method referred to is Jackson's basic method where the program control structure is matched with the data structure.

Here the repeated data structure remains the same. The only addition is the alternation with the empty sequence to force synchronization. The alternation allows for the input of the empty sequence if the next input buffer ($\{\text{byte}\}^M$) is not yet required.

For Program Inversion there is a more significant change in the structure of the problem. Two sub-problems are formed, i.e.

$$\{\{\text{byte}\}^M\} \rightarrow \{\text{byte}\}, \text{ and } \{\text{byte}\} \rightarrow \{\{\text{byte}\}^N\}.$$

The resulting program control structure after composition using program inversion has a less direct correspondence with the data structure of the original problem.

## 4. BOUNDARY STRUCTURE CLASHES

Jackson defines structure clashes as those problems in which a 1-1 correspondence between data components in the data structure and program components in the program structure cannot be found. Since it has been demonstrated that a program component can be formed from the generally provided notions of sequence, selection and iteration to correspond to the data component of unmatched repetition then this case can no longer be considered a 'structure clash'. This is intuitively appealing since boundary structure clashes were the only structure clashes which did not require additional storage of information. Boundary structure clashes can be successfully processed using a single left to right scan of the input – this is in contrast to backtracking, ordering, and multithreading structure clash problems where either the space complexity of the program must be increased or multiple passes of the input used.

This intuitive appeal may be supported by theory. Hughes has argued that Jackson's basic design method is only applicable to generalized sequential machine (g.s.m.) computable functions with input languages definable by deterministic regular expressions. Hughes conjectured that boundary structure clash problems are g.s.m. computable and so theoretically amenable to solution using the basic method without need for decomposition. It was proposed that the other types of structure clash are not g.s.m. computable and so remain as genuine structure clashes.

An alternative definition of correspondence given by Javey [Javey-86] which is more operational in nature is given below. It uses a pair consisting of an alphabet and a regular expression to represent the structure diagrams of JSP. The input alphabet is symbolized by $\Sigma$ and the input regular expression is symbolized by $R$. The pair $(\Delta, S)$ symbolises the output structure.

*Definition.* Two structures $(\Sigma, R)$ and $(\Delta, S)$ are said to correspond if by proceeding from the left to right, in $R$ and $S$, we can form a correspondence between all the components, i.e. symbols or parenthesized regular expressions, of $R$ and $S$. Two components $x$ from $R$ and $y$ from $S$ correspond, written $(x, y)$, if the following conditions are not violated:

1. The generation of $y$ should only require a single left-to-right scan of $x$ (with at most $k$ symbol look-ahead, for a fixed $k$), and
2. $(x^n, y^m) \Rightarrow (n \bmod m = 0)$
   where $x^n$ stands for $n$ repetitions of $x$.

Downloaded from https://academic.oup.com/comjnl/article/33/2/126/385835 by guest on 09 April 2024

Problems for which correspondence cannot be formulated are referred to as *structure clash* problems. Problems which violate the second condition, i.e. $n \bmod m \neq 0$, are referred to as *boundary clash* problems.

The second condition is mainly concerned with the practical aspects of being able to make the input and output loop structures correspond. The condition allows for either a single loop (the case of $n = m$) or a nested loop ($n \bmod m = 0$). The case of $n \bmod m \neq 0$ was excluded since there was not a loop construct which could accommodate it.

Let us consider transformations on program control structure for transforming a repetition of input components to a repetition of output components. The goal of the program transformations is to reduce the space complexity of the program. To transform an input data component to the corresponding output data component we require storage for the whole of the input component and the whole of the output component. If we can gain correspondence with smaller data components then there will be less space required for the transformation. Once again the arrow ($\rightarrow$) represents some mechanism to transform the input on the left to the output on the right. Epsilon ($\varepsilon$) is the empty sequence and operationally may be considered the skip statement. The braces {} indicate repetition. The superscripts represent the number of iterations required.

We proceed with a simple case analysis.

| | | |
|---|---|---|
| **1.** | $\{x\}^m \rightarrow \{y\}^m$ | Two loops in sequence with same number of iterations. Required storage = $m \times \text{size}(x) + m \times \text{size}(y)$ (Since *all* of the input must be obtained before it is transformed to *all* the output.) |
| | $\Downarrow$ | |
| | $\{x \rightarrow y\}^m$ | **Single combined loop** Required storage = $\text{size}(x) + \text{size}(y)$ (Since scope of $x$ and $y$ are limited to one iteration then the space used to store them can be reused.) |
| **2a.** | $\{x\}^{m \times g} \rightarrow \{y\}^g$ | Two loops with common factor in number of iterations. Required storage = $m \times g \times \text{size}(x) + g \times \text{size}(y)$ |
| | $\Downarrow$ | |
| | $\{\{x\}^m \rightarrow y\}^g$ | **Simple Nested Loop** Required storage = $m \times \text{size}(x) + \text{size}(y)$ |
| **2b.** | $\{x\}^g \rightarrow \{y\}^{n \times g}$ | Two loops with common factor in number of iterations. Required storage = $g \times \text{size}(x) + n \times g \times \text{size}(y)$ |
| | $\Downarrow$ | |
| | $\{x \rightarrow \{y\}^n\}g$ | **Simple Nested Loop** Required storage = $\text{size}(x) + n \times \text{size}(y)$ |
| **3.** | $\{x\}^{m \times g} \rightarrow \{y\}^{n \times g}$ | Two loops with factor (where $m \bmod n \neq 0$) Required storage = $m \times g \times \text{size}(x) + n \times g \times \text{size}(y)$ |
| | $\Downarrow$ | |
| | $\{\{x\}^m \rightarrow \{y\}^n\}^g$ | **Nested Loop with internal boundary clash** Required storage = $m \times \text{size}(x) + n \times \text{size}(y)$ If required storage is still too large (or unbounded) then will need to transform $\{x\}^m \rightarrow \{y\}^n$ further (see case 4). |
| **4.** | $\{x\}^m \rightarrow \{y\}^n$ | Two loops in sequence (where $m \bmod n \neq 0$) Required storage = $m \times \text{size}(x) + n \times \text{size}(y)$ |
| | $\Downarrow$ | |
| | $\{x \mid \varepsilon\}^p \rightarrow \{y \mid \varepsilon\}^p$ | where $\max(m, n) \leq p \leq m + n$ This is now amenable to transformation by case 1 to give |
| | $\Downarrow$ | |
| | $\{x \mid \varepsilon \rightarrow y \mid \varepsilon\}^p$ | **Single combined Loop with Forced Synchronization** Required storage = $\text{size}(x) + \text{size}(y)$ (since $\varepsilon$ requires no additional storage). (A number of iterations may be required to fully build up $y$ (or to fully decompose $x$).) |

This analysis shows that all cases (even when $n \bmod m \neq 0$) can be accommodated by appropriate loop and condition control structure. Thus the use of Forced Synchronization obviates the need for condition 2 of Javey's Definition of Correspondence.

## 5. CONCLUSIONS

A practical alternative to program inversion has been presented. Its simplicity makes it an attractive option for solving boundary structure clashes. The method is based on the *principal of structural simplification*. It utilizes *forced synchronization* of loops to implement solutions to boundary structure clash problems. The method preserves the structure of the prototype solution in developing the solution to the structure clash problem. Thus the correspondence between the solution's program control structure and the original problem's data component structure is retained. Furthermore, additional structure clashes are simply and easily accommodated.

## REFERENCES

R. G. Dromey, Forced Termination of Loops. *Software: Practice & Experience* **15** (1), 30–40 (1985).

R. W. Floyd, Paradigms of Programming. *Comm. ACM* **22** (8), 455–460 (1978).

J. W. Hughes, A Formalization of Explication of the Michael Jackson Method of Program Design. *Software: Practice & Experience* **9** (4), 191–202 (1979).

M. Jackson, Principles of Program Design. Academic Press (1975).

S. Javey, The Concept of 'Correspondence' in JSP, Technical Report CS-86-05. Department of Computer Science, York University, Toronto, Ontario, Canada (1986).

N. Wirth, Programming in Modula-2. Springer-Verlag (1982).

Downloaded from https://academic.oup.com/comjnl/article/33/2/126/385835 by guest on 09 April 2024