Some Transformations of Relevance to Report Generation

A. G. MIDDLETON

Department of Computer Science, Memorial University of Newfoundland, St John's, Newfoundland, Canada A1C 5S7

This paper presents an example of 'event-based' program transformation, in which events embedded in a program are used as markers to aid the process of program transformation. In this methodology a 'program' is presented as 'base program plus extensions'. The (procedural) base program supplies the raw inputs for a set of calculations, and the set of (non-procedural) extensions express outputs to be derived from these raw inputs. A sequence of transformations are applied to the base program to derive a 'target program' which consumes the raw inputs and produces the outputs specified by the set of extensions. An example of the methodology is presented which suggests that this approach could be of use in the production of report generation programs.

Received March 1981, revised March 1986

1. INTRODUCTION

This work can be related to the following recent developments in software research.

(i) For some time now, there has been an interest in non-procedural programming languages,[10, 11, 16] in which the programmer specifies 'what' is to be done and relies on the language translator to deduce 'how' this is to be done. A similar desire to suppress procedural detail has motivated research into functional programming languages.[1]

(ii) There has been considerable interest in program transformation as a means of converting clear, concise, but (most probably) inefficient programs into programs which are more efficient but (most probably) somewhat opaque.[2, 6, 12] The conversion of such programs is achieved through the use of a sequence of equivalence-preserving transformations.

(iii) Recently, Weiser has shown that sequential programs can be better understood by decomposing them into 'slices',[17] a slice being a group of interrelated program statements which act together to produce a single result. These slices can be regarded as the natural, intellectual 'chunks' of the program which correspond to 'units of thought' used by the programmer to comprehend the program structure.[3]

This research can be related to the above trends as follows.

(i) This paper presents a 'semi-procedural' approach to programming in which part of the program (the base program) is procedural, and the other part of the program (the extensions) is non-procedural. An interesting feature of the system is that the degree of procedurality of a source program can vary between two extremes: (*a*) the source program is completely procedural (no extension is supplied); (*b*) the source program is completely non-procedural (no base program is supplied).

(ii) The transformations which are used differ from those normally used in the following two respects: (*a*) the transformations used are 'event-based'–events embedded in the program are used as markers to indicate those points at which new activities are to be added to a program; (*b*) strictly, the transformations used are not equivalence-preserving, since the transformations cause extra results to be computed by the program. (However, the transformations do preserve the equivalence of all prior results.)

(iii) The addition of a new extension corresponds to the addition of a new 'slice' to the program.[17] For each extension, an interrelated group of statements are added to the program, using events embedded in the program to ensure that each statement in the slice is inserted at the correct point in the transformed program.

2. AN EXAMPLE PROBLEM

In order to make ideas specific, this paper will focus on a single example problem, chosen to suggest the relevance of event-based program transformation techniques to the development of report generation programs.

Suppose that we are processing a file containing one year's listing of sales figures, sorted by the name of the salesperson involved in each sale. A typical entry in this file might be a record of the following form

Name	Amount
Jones	1500

This record would indicate that Jones has sold an item worth $1,500.

Now let us suppose that we wish to extract the following information:

(i) gross sales for the year;
(ii) the total number of salespeople involved in the sales;
(iii) the sales total for each salesperson;
(iv) the salespersons (or salesperson) with the highest yearly sales total;
(v) the number of salespersons who achieved this maximum sales level.

The following base program, which ignores essentially irrelevant details about opening and closing files, would be suitable for supplying the raw data:

```
⟨BEGIN_YEAR⟩
  READ(RFILE), R
  WHILE R ≠ DUMMY DO
    ⟨ITEM: R⟩
    READ(RFILE), R
  ENDWHILE
⟨END_YEAR⟩
```

At this point, the following remarks are in order.

(i) The programming notation used in this paper is informal, and assumes a dynamic treatment of data types. This enables, essentially irrelevant, details concerning type declarations to be suppressed.

(ii) This base program contains three events: ⟨BEGIN_YEAR⟩, ⟨ITEM⟩ and ⟨END_YEAR⟩, and the second of these events has an 'associated value', R. These events will be used to aid the transformation process.

(iii) R will contain a representative record from the file RFILE, which contains the yearly sales summary.

Having supplied a suitable base program, the required outputs of the target program can be specified by the following extensions:

```
(EX_1) GROSS ← SUM(AMOUNT(ITEM),YEAR)
(EX_2) NAME_GROUP ←
            CONTROL_GROUP(NAME(ITEM),YEAR)
(EX_3) NAME_TOTAL ←
              SUM(AMOUNT(ITEM),NAME_GROUP)
(EX_4) VECTOR(TOTALS,NUMPERSONS) ←
                         (NAME_TOTAL,YEAR)
(EX_5) MAXSALES ← MAX(NAME_TOTAL,YEAR)
(EX_6) WINNER ←
               LAST(NAME(ITEM),NAME_GROUP)
               AT (NAME_TOTAL = MAXSALES)
(EX_7) VECTOR(WINNERS,NUMWIN) ←
                             (WINNER,YEAR)
```

The intent of these extensions is as follows.

(i) Gross sales for the year are to be stored in the variable GROSS.

(ii) A 'name group' will contain a subsequence of records with the same NAME-value (i.e. all sales for a particular salesperson).

(iii) NAME_TOTAL provides the total for all sales within a particular name group.

(iv) The one-dimensional array TOTALS is to receive the values of NAME_TOTAL, and NUMPERSONS will record the number of totals transferred to this array.

(v) MAXSALES will record the highest per-salesperson total.

(vi) A 'winner' is any person who achieves this highest sales total (there could be several).

(vii) The names of the winners will be transferred to the one-dimensional array WINNERS, and NUMWIN will record the number of names so transferred.

The above does not represent a substantial problem to current report-generation systems, and the fact that this methodology can solve the problem is, in itself, not considered to be a significant achievement. However, it is considered to be significant that this paper presents a systematic methodology for developing such a program. In the author's opinion, as suggested in Horowitz, Kember and Narasimhan (1985),[4] the higher-level language systems used in data-processing environments are long overdue for systematic treatment by academics. The aim of this paper is to make a contribution towards the more systematic development of such languages.

Before proceeding with the development of the above example, some elementary technical concepts will be introduced in the next few sections of the paper.

3. EVENTS, INTERVALS AND SEQUENCES

3.1 Events

The transformation process depends on the use of 'events' to act as markers to aid program transformation. These markers allow the system to insert new activities at the correct points in the target program. The notation

⟨E⟩

denotes an event name 'E', and the notation

$$\langle E\colon e_1, e_2, \ldots, e_N\rangle$$

associates the tuple of values $(e_1, e_2, \ldots, e_N)$ with the occurrence of the event ⟨E⟩. In the above base program there are three events, ⟨BEGIN_YEAR⟩, ⟨ITEM⟩ and ⟨END_YEAR⟩, and the second of these events has a single value associated with its occurrence.

3.2 Intervals

Events are often used in pairs to form an 'interval'. In the above base program, the pair of events ⟨BEGIN_YEAR⟩ and ⟨END_YEAR⟩ act together to form the interval ⟨YEAR⟩. Typically, an interval will indicate the span of time over which a sequence of data values will be produced.

3.3 Sequences

If ⟨E⟩ is a value-producing event, and ⟨I⟩ is an interval containing ⟨E⟩, then the pair (⟨E⟩,⟨I⟩) denotes the sequence of all those E-values which occur over any instance of the interval ⟨I⟩. Typically, ⟨I⟩ might correspond to the scope of some loop, and the pair (⟨E⟩, ⟨I⟩) denotes the sequence of E-values which correspond to some single execution of the entire loop.

Where no ambiguity will occur, the sharp brackets around event-names and interval-names can be dropped. Thus the pair (E,I) denotes exactly the same sequence as the pair (⟨E⟩, ⟨I⟩).

4. EVENT-BASED PROGRAM TRANSFORMATION

As mentioned above, the methodology presented depends on the use of 'event-based' program transformations. Events embedded in the program are used as markers to direct the insertion of new activities in the program.

All transformations have the form

$$\begin{matrix}\text{Old}\\ \text{text}\end{matrix} \Rightarrow \begin{matrix}\text{New}\\ \text{text}\end{matrix}$$

which indicates that 'old text' must be replaced throughout the program by 'new text'. As an example, consider the translation of an extension of the form

S ← SUM(E,I)

which causes the sum of the sequence (E,I) to be stored in the variable S. This extension can be implemented by using the following transformations:

```
⟨BEGIN_I⟩ ⇒ ⟨BEGIN_I⟩
              S ← 0

  ⟨E: e⟩ ⇒ ⟨E: e⟩
              S ← S+e

  ⟨END_I⟩ ⇒ ⟨END_I⟩
              ⟨S: S⟩
```

The effect of these transformations is as follows.

(i) The sum S is initialised to zero every time the event ⟨BEGIN_I⟩ occurs.

(ii) The associated value of ⟨E⟩ is added to S every time that ⟨E⟩ occurs.

(iii) The final value of S is made available whenever ⟨END_I⟩ occurs. Other operators are implemented in a similar manner. For example, an extension of the form

$$N \leftarrow \text{NUMBER(E,I)},$$

which causes N to be used to count the number of occurrences of ⟨E⟩ during the interval ⟨I⟩, can be implemented through the following transformations:

$$\langle \text{BEGIN\_I} \rangle \Rightarrow \langle \text{BEGIN\_I} \rangle \quad N \leftarrow 0$$

$$\langle \text{E: e} \rangle \Rightarrow \langle \text{E: e} \rangle \quad N \leftarrow N+1$$

$$\langle \text{END\_I} \rangle \Rightarrow \langle \text{END\_I} \rangle \quad \langle \text{N: N} \rangle$$

At some later stage, it is intended to provide some means of extensibility, so that new operator definitions can be written in some suitably convenient form, and the required transformations can be derived mechanically from these definitions. However, at present an ad hoc approach is used for each operator required.

5. DESCRIPTORS

This work can be related very closely to the Jackson method of program design.[7, 8] In fact, the method uses 'descriptors' to express the relative sequencing between different events and intervals contained in a program, and these descriptors convey exactly the same information as Jackson-style 'data structures'. In fact, the only difference between descriptors and Jackson-style data structures is the manner in which iteration and selection components are represented. This difference is shown in Fig. 1, which shows a descriptor along with its equivalent Jackson-style data structure. Both structures are intended to represent a group of movement records, each movement record being either an issue record or a receipt record.

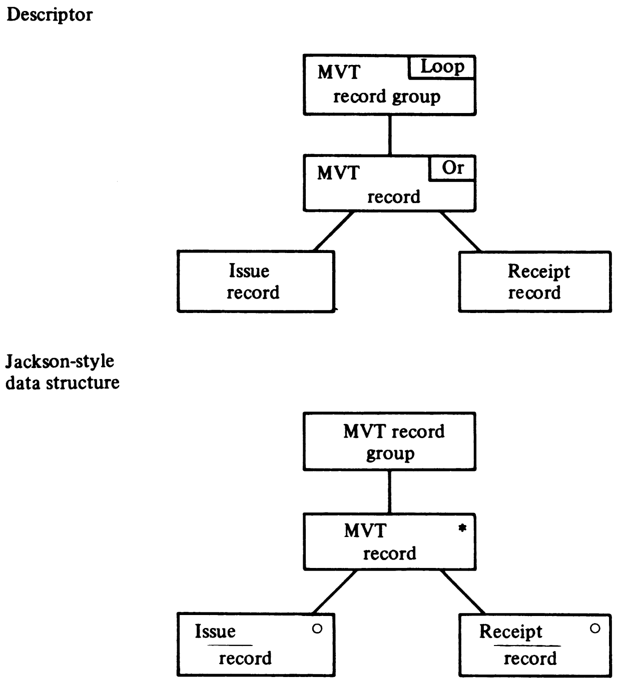

Fig. 1. A descriptor and its equivalent Jackson-style data structure.

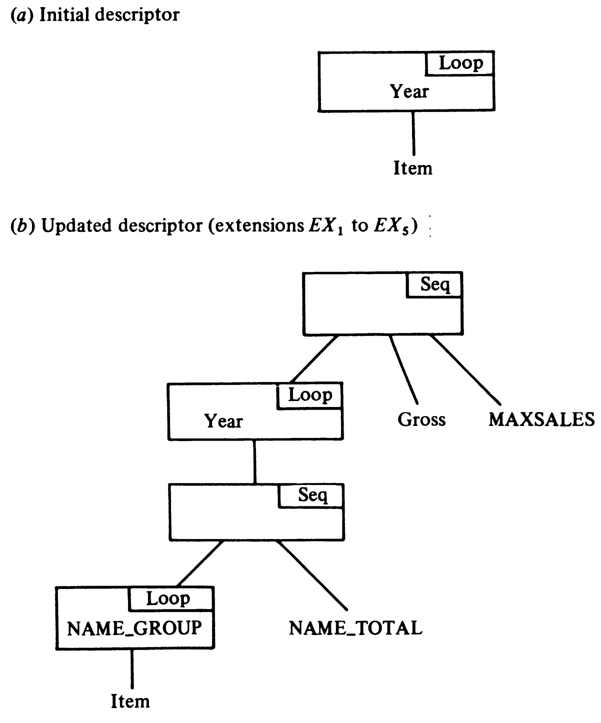

Fig. 2. Updating of descriptors.

6. DATA-BUFFERING

An 'order clash' occurs when data in an input structure does not arrive in the natural order required by a particular computation. Descriptors are used to detect order clashes and to supply the necessary 'data buffering' to resolve them.

An example of this requirement is presented by the translation of the expression NAME–TOTAL = MAXSALES in the extension

(EX_6) WINNER ←
LAST(NAME(ITEM),NAME_GROUP)
AT (NAME_TOTAL = MAXSALES)

In order to deal with such a problem, the descriptor for the target program is updated as each extension is implemented. Initially, before any extension is implemented, the descriptor for the target program corresponds to the descriptor for the base program, and has the form shown in Fig. 2(*a*). After extensions EX_1 to EX_5 have been implemented, the descriptor for the target program has the form shown in Fig. 2(*b*).

Various 'sequence relations' can be deduced from descriptors such as those shown in Fig. 2.[14] In this particular context, the following information can be deduced from the descriptor in Fig. 2(*b*).

(i) There can be several values of NAME_TOTAL for a single occurrence of MAXSALES.

(ii) The values of NAME_TOTAL occur **before** the value of MAXSALES is available.

Unless something is done to save the NAME_TOTAL values, they will be unavailable for comparison with the MAXSALES value by the time that the MAXSALES value becomes available.

To deal with this problem, data-buffering transformations are used to achieve the following effect (see Ref. 14, for details):

(i) The NAME_TOTAL values are saved in a storage buffer as they are produced.

(ii) These values are regurgitated once the value of MAXSALES is available, thus allowing the comparison of each NAME_TOTAL value to MAXSALES.

7. CONTROL GROUPS

Jackson has complained about the inelegance of the well-known 'control break' mechanism.[9] The purpose of the 'control group' mechanism is to provide the more meaningful structure recommended by Jackson.

The control-group structure arises when a sequence is sorted on some key and it is required to partition the sequence into blocks in such a way that items within each block of the partition share the same key value.

We shall now look at the detailed translation of an extension of the form

I′ ← CONTROL_GROUP(E,I),

where (E,I) is assumed to be a sorted sequence of E values, and *I′* will be the control-group interval such that (i) all E values within any occurrence of *I′* are equal; (ii) any two E values which occur in different instances of *I′* are unequal; (iii) all E values fall within exactly one instance of *I′*. The strategy for implementing

I′ ← CONTROL_GROUP(E,I)

is as follows.

(i) Events ⟨*BEGIN_I′*⟩ and ⟨*END_I′*⟩ are required to mark the opening and closing of the interval ⟨*I′*⟩.

(ii) An event ⟨CHANGE_E⟩ is made to occur whenever two successive E values are different.

(iii) The event ⟨*BEGIN_I′*⟩ is made to occur whenever ⟨BEGIN_I⟩ occurs, and the event ⟨*END_I′*⟩ is made to occur whenever ⟨END_I⟩ occurs.

(iv) The event ⟨CHANGE_E⟩ is then made to generate the two events ⟨*END_I′*⟩ and ⟨*BEGIN_I′*⟩ **(in that order)**.

The above mechanism is adequate except that it will not deal with the case where there are exactly zero occurrences of ⟨E⟩ during any single occurrence of ⟨I⟩. (We do not wish to allow the occurrence of a control group containing no E value.) This problem can be dealt with by including the following mechanics.

(v) The extension FIRST_E ← FIRST_TIME(E,I) is added. This provides a first-time switch to indicate whether or not there has been a single occurrence in ⟨E⟩ since the beginning of ⟨I⟩.

(vi) The closing of ⟨I′⟩ is made conditional upon the value of this first-time switch.

Having stated the general strategy, let us now look at some of the details. The following transformations allow differences in successive E values to be monitored:

```
⟨BEGIN_I⟩ ⇒ ⟨BEGIN_I⟩
            FIRST_E ← TRUE

⟨E: e⟩ ⇒ IF FIRST_E THEN
            FIRST_E ← FALSE
         ELSE
            IF e ≠ OLD_E THEN
               ⟨CHANGE_E⟩
            ENDIF
         ENDIF
         OLD_E ← e
         ⟨E: e⟩
```

Here, the following remarks are in order.

(i) Monitoring changes in E value (through ⟨CHANGE_E⟩) requires the use of a first-time switch (FIRST_E) for the sequence (E,I). This first-time switch can also be used to monitor the correct closing of the interval ⟨*I′*⟩.

(ii) OLD_E is used to store that E value which precedes the current E value.

The remaining transformations required are as follows:

```
⟨BEGIN_I⟩ ⇒ ⟨BEGIN_I⟩
            ⟨BEGIN_I′⟩

⟨CHANGE_E⟩ ⇒ ⟨CHANGE_E⟩
             ⟨END_I′⟩
             ⟨BEGIN_I′⟩

⟨END_I⟩ ⇒ IF NOT(FIRST_E) THEN
             ⟨END_I′⟩
          ENDIF
          ⟨END_I⟩
```

This completes the transformations required to implement the control-group mechanism. It is worth noting that control groups can be nested, and the interval ⟨*I′*⟩ could itself be partitioned on a secondary key, in the same manner as ⟨*I′*⟩ itself was created.

8. REMAINING TRANSFORMATIONS

Only a few miscellaneous transformations remain to be presented, and the development of the report-generation example is complete.

First, let us dispose of the trivial matter of computing the expressions NAME(ITEM) and AMOUNT(ITEM). Both of these expressions are examples of applying a unary function to a unary event. The general requirement here is to translate an implied extension of the form

FE ← F(E),

where F is a unary function and E is a unary event. Such an extension can be handled quite simply by the following transformation:

```
⟨E: e⟩ ⇒ ⟨E: e⟩
         ⟨FE: F(e)⟩
```

Another outstanding requirement is to translate an extension of the form

VECTOR(V,NV) ← (E,I),

which is intended to cause the sequence (E,I) to be stored in the one-dimensional array V, with a count of the number of values stored to be recorded in NV. This requirement could be treated in the more general context of representing data sequences by abstract operators for iteration (see Ref. 15, for some examples of the use of this approach).

However, to avoid a long digression, we will simply present an ad hoc approach which is adequate to deal with the specific problem at hand. Using an ad hoc approach, the following transformations will suffice to solve this problem:

```
⟨BEGIN_I⟩ ⇒ ⟨BEGIN_⟩
            NV ← 0

⟨E: e⟩ ⇒ ⟨E: e⟩
         NV ← NV + 1
         V[NV] ← e

⟨END_I⟩ ⇒ ⟨END_I⟩
          ⟨NV: NV⟩
```

Now the only remaining problem is to translate the extension

(EX_6) WINNER ←
LAST(NAME(ITEM),NAME_GROUP)
AT (NAME_TOTAL = MAXSALES)

This can be decomposed into the following three sub-extensions:

(EX_{6A}) LN ← LAST(NAME(ITEM),NAME_GROUP)

(EX_{6B}) EQL ← NAME_TOTAL = MAXSALES

(EX_{6C}) WINNER ← LN **AT** EQL

An extension of the form

L ← *LAST*(*E*,*I*)

can be translated easily by using the transformation

```
⟨END_I⟩ ⇒ ⟨END_I⟩
          ⟨L: e⟩
```

where e is extracted from the event ⟨E: e⟩ (A slight problem arises if there are several textual occurrences of ⟨E⟩, but it can be dealt with.)

The extension

(EX_{6B}) EQL ← NAME_TOTAL = MAXSALES

can be dealt with by using the data-buffering techniques discussed earlier, and this will produce an event of the form ⟨EQL: b⟩, where b is a boolean value. An extension of the form

EB ← *E* AT B

(where B is a boolean event) can be translated by using the transformation

```
⟨E: e⟩ ⇒ ⟨E: e⟩
         IF b THEN
           ⟨EB: e⟩
         ENDIF
```

9. RELEVANCE TO THE JACKSON METHOD

As was mentioned earlier, this method borrows heavily from the Jackson approach to program construction.[7, 8] Like other researchers,[5, 13] the author feels that further formalisation and automation of the Jackson approach to program design is very desirable.

However, at present the techniques presented here do not generally go much beyond a convenient notation for handling the 'list and allocate operations' step of Jackson's method.[7] In order to make a more substantial contribution towards handling Jackson's methodology it would be necessary to provide more formalisation and mechanisation of (at least) the following types of problem:

(i) structure clashes;
(ii) interleaving problems;
(iii) backtracking problems;
(iv) problems involving data ordering (sorting, merging, collating, etc.).

If problems such as the above can be handled, then it may be possible to chew away at the bottom end of the Jackson design process in such a way that problems which are currently considered to be 'design' problems can be relegated to the level of coding details, with the subsequent delegation of more and more structural detail to a mechanical transformation system.

10. CONCLUSIONS

Data processing professionals have been using higher-level language systems for report-generation problems for some time now. However, so far there has not been much academic interest in the systematic generation of such programs.[4] This paper has presented an attempt to introduce a systematic methodology for the construction of such programs.

Acknowledgement

This work was supported by the Natural Science and Engineering Research Council of Canada, through grant A3406.

REFERENCES

1. J. Backus, Can programming be liberated from the Von Neuman style? *CACM* **21**, 613–641 (1978).
2. R. M. Burstall and J. Darlington, A transformation system for developing recursive programs. *JACM* **24**, 44–67 (1977).
3. J. S. Davis, Chunks: a basis for complexity measurement, *Information Processing and Management*, **20** (1-2), 119–127 (1984).
4. E. Horowitz, A. Kemper and B. Narasimhan, A survey of application generators, *IEEE Software* **2** (1), 40–54 (1985).
5. J. W. Hughes, A formalization and explication of the Michael Jackson method of program design. *Software, Practice and Experience*, **9**, 191–202 (1977).
6. D. F. Kibler, J. M. Neighbors and T. A. Standish, Program manipulation via an efficient production system. *Proceedings, SIGPLAN/SIGART Symposium on Artificial Intelligence and Programming Languages*, pp. 163–173 (1977).
7. M. A. Jackson, *Principles of Program Design*. Academic Press, London (1975).
8. M. A. Jackson, *System Development*. (1983)
9. M. A. Jackson, Getting it wrong – a cautionary tale. In *Tutorial on JSP & JSD: The Jackson approach to Software Development*, IEEE Tutorial (by J. R. Cameron) (1984).
10. B. M. Leavenworth, Non-procedural data processing. *The Computer Journal* **20**, 6–9 (1977).
11. B. M. Leavenworth and J. E. Sammet, An overview of

non-procedural languages, *Proceedings, SIGPLAN Symposium on Very High Level Languages, pp.* 1–12 (1974).

12. D. B. Loveman, Program improvement by source-to-source transformations. *JACM* **24**, 121–145 (1977).
13. R. C. B. Martins and P. A. S. Veloso, Jackson's method for program construction reformalised and extended. *Proceedings, 19th Conference on Information Sciences and Systems, Johns Hopkins University*, pp. 606–609 (1985).
14. A. G. Middleton, Programming by extension: a program construction technique, *Proceedings, 3rd International Conference in Computer Science, Santiago, Chile*, pp. 68–85 (1983).
15. A. G. Middleton and R. B. B. Brake, *Automating Implementation Details for a Limited Data Abstraction*. Technical Report 8511, Department of Computer Science, Memorial University, Newfoundland, Canada (1985).
16. J. T. Schwartz, *On Programming: An Interim Report on SETL Project*. Courant Institute of Mathematical Sciences (1975).
17. M. Weiser, Programmers use slices when debugging. *CACM* **25**, 446–452 (1982).