

Integrating the Structured Analysis and Design Models: an Initial Algebra Approach^{*†}

T.H. Tse
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
Email: thtse@cs.hku.hk

ABSTRACT

Numerous models have been proposed under the name of structured systems analysis and design. Because of the lack of a common theoretical framework, the transition from one model to another is arbitrary and can only be done manually. An initial algebra approach is proposed to integrate the structured models. The algebra defined can be mapped through homomorphisms to Yourdon structure charts and DeMarco data flow diagrams. It can also be linked to Jackson structure text through equations.

Keywords and phrases: algebraic approach, structured analysis, structured design

CR categories: D.2.1, F.3.2, K.6.3

1. INTRODUCTION

Numerous models have been proposed under the name of structured analysis and design. Examples are data flow diagrams (DeMarco, 1979; Gane and Sarson, 1979; Weinberg, 1980), Jackson structure diagrams, Jackson structure text (Jackson, 1975), system specification diagrams, system implementation diagrams (Jackson, 1983), Warnier/Orr diagrams (Orr, 1977) and structure charts (Yourdon and Constantine, 1979). They are widely accepted by practising systems analysts and designers through the simplicity of use and the ease of communication with users. But because of the lack of a common theoretical framework, the transition from one model to another is arbitrary and can only be done manually. Users tend to stick to a particular model not because of its superiority but because of familiarity. Automatic development aids tend to be *ad hoc* and model-dependent.

To solve the problem, there is a need to provide a formal link for the structured models. The initial algebra approach is proposed. It has a rich mathematical linkage with category and algebraic theories.

* © 1986 *Australian Computer Journal*. This material is presented to ensure timely dissemination of scholarly and technical work. Personal use of this material is permitted. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each authors copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder. Permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from *Australian Computer Journal*.

† Part of this research was done at the London School of Economics, University of London under a Commonwealth Academic Staff Scholarship.

But at the same time, the concepts can be simply stated for those who do not want to be involved with elaborate theories.

In this paper we will define the algebra and illustrate how it can be related to Yourdon structure charts, DeMarco data flow diagrams and Jackson structure text by means of homomorphisms and equations. The three models have been chosen for illustration because they represent three distinct classes of structured models.

We shall concentrate on the conceptual framework rather than on formal proofs. Only a knowledge of elementary set theory will be assumed. Readers who are interested in a deeper understanding of algebraic formalism may refer to Cohn (1981) for a general algebraic introduction, to Burstall and Goguen (1982), Goguen et al. (1978), Wagner (1981) and Zilles et al. (1982) for a computer science oriented treatment, and to Goguen et al. (1975) and Wagner et al. (1977) for a category-theoretic treatment.

2. ADVANTAGES OF INTEGRATION

A integration of the structured models is useful for several reasons:

- (a) Specifications can be transformed from one form to another through homomorphisms and equations, as illustrated in the subsequent sections of the paper.
- (b) Different structured models are suitable for different situations depending on the environment (Shigo et al., 1980), emphasis (Colter, 1982) and stage of development (Lauber, 1982). But it has been found that individual models may not be used in some installations because the users are not familiar with them (Beck and Perkins, 1983). Through a transformation system, the most suitable model can be used independently of user familiarity.
- (c) Automatic development aids for one structured methodology can be applied to another through transformations. The development aids described, for example, in Delisle et al. (1982), DeMarco and Soceneantu (1984) and Tse (1985) may therefore be extended to other models.
- (d) In recent years the initial algebra approach has been used extensively in the specification of abstract data types. Examples are Clear (Burstall and Goguen, 1980, 1981; Sannella, 1984) and OBJ (Futatsugi et al., 1985; Goguen 1984; Goguen and Tardo, 1979; Goguen et al., 1983). Interpreters for abstract data types are already available. Although such interpreters are not originally intended for structured analysis and design models, they can nevertheless be adapted to suit our needs, e.g. for validating our specifications.
- (e) In informal specifications such as DeMarco data flow diagrams, a certain degree of omission or “mutual understanding” is permitted. This often leads, however, to ambiguity and misunderstanding. If a formal specification is used, we can enforce predefined standards more easily.

3. ALGEBRAS

Intuitively, an algebra is a family of objects that satisfy a formal structure. To define an algebra A , we must first of all define the formal structure through the concept of a signature. A *signature* consists of a set S of object types, known as *sorts*, together with a family Σ of sets, each set containing *operation symbols* (or simply *symbols*) which connect the sorts. We will use $\Sigma\langle s_1 \dots s_n, s \rangle$ to denote the set of operation symbols which connect the sorts s_1, \dots, s_n to the sort s .

Given the skeleton structure, we then complete the definition by relating it to real objects. Each sort s is mapped to a set $A\langle s \rangle$, which is called the *carrier* of s . Each symbol \mathbf{q} in $\Sigma\langle s_1 \dots s_n, s \rangle$ is mapped to a function

$$\mathbf{q}_A: A\langle s_1 \rangle \times \dots \times A\langle s_n \rangle \rightarrow A\langle s \rangle,$$

which is called an *operation*.

Let us apply the algebraic fundamentals to structured systems. Conceptually, a structured system is specified by a hierarchy of tasks. Each task consists of a name, a structure, together with the input and output. The structure determines whether the task is elementary, or is made up of subtasks in the form of sequence, selection, iteration or parallelism. The input and output are in the form of data flows related with other tasks, files and the environment.

The signature for structured systems, then, consists of a set S of seven sorts:

$$task, name, struct, input, output, dataflow, flowname$$

and a family Σ of sets of operation symbols:

$$\begin{aligned} \Sigma\langle name\ input\ output\ struct, task \rangle &= \{\mathbf{task}\} \\ \Sigma\langle task^n, struct \rangle &= \{\mathbf{sequ}, \mathbf{seln}, \mathbf{para}\} \\ \Sigma\langle task, struct \rangle &= \{\mathbf{iter}\} \\ \Sigma\langle \Lambda, struct \rangle &= \{\mathbf{elem}\} \\ \Sigma\langle dataflow^n, input \rangle &= \{\mathbf{get}\} \\ \Sigma\langle dataflow^n, output \rangle &= \{\mathbf{put}\} \\ \Sigma\langle flowname, dataflow \rangle &= \{\mathbf{indata}, \mathbf{inflag}, \mathbf{infile}, \mathbf{source}, \mathbf{outdata}, \mathbf{outflag}, \mathbf{outfile}, \mathbf{sink}\} \end{aligned}$$

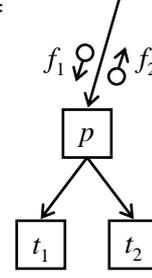
for any positive integer n and where Λ is the empty string.

The sorts of the signature are mapped to the carriers as shown in Table 1. The symbols are mapped to the following operations:

- (a) The operation \mathbf{task}_A specifies the name, structure, input and output of a task.
- (b) The operations \mathbf{sequ}_A , \mathbf{seln}_A , \mathbf{iter}_A and \mathbf{para}_A link up a number of subtasks to form a structure.
- (c) The operation \mathbf{elem}_A indicates that a structure is elementary, i.e. it does not consist of subtasks.
- (d) The operations \mathbf{get}_A and \mathbf{put}_A specify the input and output flows for a task.
- (e) The operations \mathbf{indata}_A , \mathbf{inflag}_A , \mathbf{infile}_A and \mathbf{source}_A specify the name of each input data flow. They also denote, respectively, that the data flow consists of data from some other task, a flag from some other task, data from a file, and data from the environment.
- (f) The operations $\mathbf{outdata}_A$, $\mathbf{outflag}_A$, $\mathbf{outfile}_A$ and \mathbf{sink}_A are similarly used for output data flows.

Different algebras can be defined over the same signature *Homomorphisms*, or functions preserving the signature, can be defined from one algebra to another. Such homomorphisms enable us to forget about minor syntactical differences in various specification methods and concentrate on the major issues. For example, suppose

$$\mathbf{task}_A(p, \mathbf{get}_A(\mathbf{indata}_A(f_1)), \mathbf{put}_A(\mathbf{outdata}_A(f_2)), \mathbf{sequ}_A(t_1, t_2)) = \dots (1)$$



in algebra A , and

$$\begin{aligned} & \mathbf{task}_B(p, \mathbf{get}_B(\mathbf{indata}_B(f_1)), \mathbf{put}_B(\mathbf{outdata}_B(f_2)), \mathbf{sequ}_B(t_1, t_2)) \\ & = \mathbf{procedure } p(f_1, f_2): \mathbf{begin } t_1; t_2 \mathbf{end} \end{aligned} \dots (2)$$

in algebra B . Then a homomorphism mapping the variables p, f_1, f_2, t_1 and t_2 in A to the corresponding variables in B will automatically map (1) to (2).

4. INITIAL ALGEBRA

The algebra that has the richest context is called an *initial algebra*, satisfying the following property:

An algebra A is initial if, for any algebra B over the same signature, there exists a unique homomorphism mapping A to B .

We would like to construct an initial algebra for structured systems using the concept of term algebras. A *term algebra* T_Σ for structured systems is defined as follows:

4.1 Carriers

We regard task names and data names as more fundamental than other variables in our algebra, because these names appear unaltered in a final specification. They are like terminals in the theory of formal languages. We will enlarge the signature by putting in task names and data names as “symbols”. Thus two more sets of symbols are defined:

$\Sigma\langle\Lambda, name\rangle$ = the set of task names,

$\Sigma\langle\Lambda, flowname\rangle$ = the set of names of input/output data flows.

Although also known as “operation symbols”, these symbols are actually not operating on any sort.

Let X denote the enlarged set of symbols, together with three delimiter symbols: “(”, “;” and “)”. The carriers of T_Σ are made up of *terms* in X , i.e. strings of symbols from X . We define the carriers $T_\Sigma\langle s \rangle$ by induction as follows:

- (a) For any symbol \mathbf{q} in $\Sigma\langle\Lambda, s\rangle$, we let the term “ \mathbf{q} ” be in $T_\Sigma\langle s \rangle$.
- (b) For any operation symbol \mathbf{q} in $\Sigma\langle s_1 \dots s_n, s \rangle$, and for any terms “ u_1 ” in $T_\Sigma\langle s_1 \rangle$, ..., “ u_n ” in $T_\Sigma\langle s_n \rangle$, we let the term “ $\mathbf{q}(u_1; \dots; u_n)$ ” be in $T_\Sigma\langle s \rangle$.

4.2 Operations

Operations \mathbf{q}_T in T_Σ are induced from the symbols \mathbf{q} as follows:

- (a) For any symbol \mathbf{q} in $\Sigma\langle\Lambda, s\rangle$, we define \mathbf{q}_T to be the term “ \mathbf{q} ”.

- (b) For any operation symbol \mathbf{q} in $\Sigma\langle s_1 \dots s_n, s \rangle$, and for any terms “ u_1 ” in $T_{\Sigma\langle s_1 \rangle}$, ..., “ u_n ” in $T_{\Sigma\langle s_n \rangle}$, we define $\mathbf{q}_T(u_1, \dots, u_n)$ to be the term “ $\mathbf{q}(u_1; \dots; u_n)$ ”.

It can be shown that the term algebra T_{Σ} thus defined is an initial algebra. It can be mapped by homomorphisms to other algebras over the same signature. For example, we will illustrate how the sample term shown in Figure 1 can be related to a Yourdon structure chart, a DeMarco data flow diagram and Jackson structure text.

5. YOURDON STRUCTURE CHARTS

To illustrate how the terms in our initial algebra can be mapped to structure charts, we must first of all define an algebra Y of Yourdon structure charts (which we will call *Yourdon algebra* for short). The carriers are defined similarly to Table 1. The operations \mathbf{q}_Y are defined as shown in Figure 2. Then the obvious homomorphism will map our initial algebra to the Yourdon algebra. The term in Figure 1, for example, will be mapped to the structure chart of Figure 3.

Can we do the reverse? That is to say, can we define a unique reverse homomorphism from the Yourdon algebra to the initial algebra, and hence get back our term? We find that even though reverse homomorphisms can be created, they are not unique. One reason is that some operations, such as **infile**_Y, **outfile**_Y, **source**_Y and **sink**_Y, are effectively not used in the Yourdon algebra. Another reason is that other operations, although distinct in the initial algebra, may overlap in the Yourdon algebra. For example, both **sequ**_Y and **para**_Y give identical results. The Yourdon algebra must be extended to cater for these operations before a unique reverse homomorphism can be defined. The algebra of *extended* Yourdon structure charts will then be regarded as isomorphic, or equivalent, to our term algebra.

A transformation system is being studied by the present author to provide a computerized link between the initial algebra and the Yourdon algebra. It will be run using Smalltalk under Unix. It will accept any term in our initial algebra as input and generate a Yourdon structure chart automatically. Conversely, the system will also accept a Yourdon structure chart, enquire the user about any extension required, and then convert the chart into a term in the initial algebra.

6. DeMARCO DATA FLOW DIAGRAMS

We can similarly define an algebra D of DeMarco data flow diagrams (or *DeMarco algebra* for short). The carriers are defined similarly to Table 1, and the operations are as shown in Figure 4. The obvious homomorphism will map the terms in our initial algebra to data flow diagrams. The term in Figure 1, for instance, can be mapped to the DeMarco data flow diagram of Figure 5. Furthermore, if we forget about the intermediate task names in the original term, then a flattened data flow diagram can be obtained, as shown in Figure 6.

The transformation system under study (see Section 5) will also accept terms in the initial algebra and generate DeMarco data flow diagrams automatically. Conversely, given a data flow diagram, the system will enquire the user about extensions to the DeMarco algebra and then generate automatically a term of the initial algebra. The system will therefore help the user to convert DeMarco data flow diagrams into Yourdon structure charts.

7. JACKSON STRUCTURE TEXT

We also want to define an algebra J of Jackson structure text (or *Jackson algebra*). This is done through the algebraic concept of equations, and involves the following steps:

(a) We define a preliminary Jackson algebra which has the same carrier and operations as our initial algebra.

(b) We define four new operations

$$\mathbf{seq}_J: J\langle name \rangle \times J\langle task \rangle^n \rightarrow J\langle struct \rangle$$

$$\mathbf{sel}_J: J\langle name \rangle \times J\langle task \rangle^n \rightarrow J\langle struct \rangle$$

$$\mathbf{itr}_J: J\langle name \rangle \times J\langle task \rangle \rightarrow J\langle struct \rangle$$

$$\mathbf{elm}_J: J\langle name \rangle \rightarrow J\langle struct \rangle.$$

The functions of these operations are shown in Figure 7.

(c) Since input and output are not included in Jackson structure text, we must define two *equations*, or relationships that connect different elements of the algebra together. Thus we have

$$\mathbf{get}_J(\Lambda) = \mathbf{get}_J(d_1, \dots, d_n)$$

$$\mathbf{put}_J(\Lambda) = \mathbf{put}_J(d_1, \dots, d_n)$$

for any data flows d_1, \dots, d_n .

(d) We define four more equations to link up the new operations in (b) with the original operations, thus:

$$\mathbf{seq}_J(p, t_1, \dots, t_n) = \mathbf{task}_J(p, \mathbf{get}_J(\Lambda), \mathbf{put}_J(\Lambda), \mathbf{sequ}_J(t_1, \dots, t_n))$$

$$\mathbf{sel}_J(p, t_1, \dots, t_n) = \mathbf{task}_J(p, \mathbf{get}_J(\Lambda), \mathbf{put}_J(\Lambda), \mathbf{seln}_J(t_1, \dots, t_n))$$

$$\mathbf{itr}_J(p, t) = \mathbf{task}_J(p, \mathbf{get}_J(\Lambda), \mathbf{put}_J(\Lambda), \mathbf{iter}_J(t))$$

$$\mathbf{elm}_J(p) = \mathbf{task}_J(p, \mathbf{get}_J(\Lambda), \mathbf{put}_J(\Lambda), \mathbf{elem}_J)$$

for any task names p and tasks t, t_1, \dots, t_n .

In this way, the obvious homomorphism will enable us to map our terms into Jackson text. For example, the term in Figure 1 will be mapped to the Jackson structure text of Figure 8. Furthermore, the transformation system under study can also be enhanced to accept or generate Jackson structure text.

8. CONCLUSION

Structured analysis and design models can be integrated algebraically. A term algebra has been defined and can be mapped through homomorphisms to Yourdon structure charts and DeMarco data flow diagrams. It can also be linked to Jackson structure text through equations.

As a result, specifications can be transformed from one form to another. The most suitable model can be chosen for a target system independently of user familiarity. Algebraic interpreters may be adapted to validate the specifications. Automatic development aids for one methodology may be applied to another.

ACKNOWLEDGEMENTS

The author is indebted to Ronald Stamper, Haya Freedman and Professor Frank Land of LSE for their invaluable suggestions. He is also grateful to Professors Jim Emery of Philadelphia, Blake Ives of Dartmouth and Joe Davis of Indiana for the most encouraging comments on the project.

REFERENCES

- L.L. Beck and T.E. Perkins (1983): “A survey of software engineering practice: tools, methods, and results”, *IEEE Transactions on Software Engineering*, Vol. 9, No. 5, pp. 541–561.
- R.M. Burstall and J.A. Goguen (1980): “The semantics of Clear, a specification language”, in *Abstract Software Specifications: Proceedings of the 1979 Copenhagen Winter School*, D. Bjorner (ed.), Lecture Notes in Computer Science, Springer, Berlin, Germany, Vol. 86, pp. 292–332.
- R.M. Burstall and J.A. Goguen (1982): “Algebras, theories, and freeness: An introduction for computer scientists”, in *Theoretical Foundations of Programming Methodology: Proceedings of Marktoberdorf Summer School*, M. Broy and G. Schmidt (eds.), NATO Advanced Study Institute Series, Reidel Publishing, Dordrecht, The Netherlands, Vol. C91, pp. 329–350.
- R.M. Burstall and J.A. Goguen (1986): “An informal introduction to specifications using Clear”, in *Software Specification Techniques*, N.H. Gehani and A.D. McGettrick (eds.), Addison-Wesley, Reading, MA, pp. 363–390.
- P.M. Cohn (1981): *Universal Algebra*, Reidel Publishing, Dordrecht, The Netherlands.
- M.A. Colter (1982): “Evolution of the structured methodologies”, in *Advanced System Development / Feasibility Techniques*, J.D. Couger, M.A. Colter, and R.W. Knapp (eds.), John Wiley, New York, NY, pp. 73–96.
- N.M. Delisle, D.E. Menicosy, and N.L. Kerth (1982): “Tools for supporting structured analysis”, in *Automated Tools for Information Systems Design*, H.-J. Schneider and A.I. Wasserman (eds.), Elsevier, Amsterdam, The Netherlands, pp. 11–20.
- T. DeMarco (1979): *Structured Analysis and System Specification*, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ.
- T. DeMarco and A. Soceneantu (1984): “SYNCRO: A data flow command shell for the Lilit/Modula computer”, in *Proceedings of the 7th International Conference on Software Engineering (ICSE '84)*, IEEE Computer Society, New York, NY, pp. 207–213.
- K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer (1985): “Principles of OBJ2”, in *Conference Record of the 12th ACM Annual Symposium on Principles of Programming Languages*, B. Reid (ed.), ACM, New York, NY, pp. 52–66.
- C.P. Gane and T. Sarson (1979): *Structured Systems Analysis: Tools and Techniques*, Prentice Hall, Englewood Cliffs, NJ.
- J.A. Goguen (1984): “Parameterized programming”, *IEEE Transactions on Software Engineering*, Vol. 10, No. 5, pp. 528–543.
- J.A. Goguen, J. Meseguer, and D. Plaisted (1983): “Programming with parameterized abstract objects in OBJ”, in *Theory and Practice of Software Technology: Proceedings of the International Seminar on Software Engineering*, D. Ferrari, M. Bolognani, and J.A. Goguen (eds.), Elsevier, Amsterdam, The Netherlands, pp. 163–193.
- J.A. Goguen and J.J. Tardo (1986): “An introduction to OBJ: A language for writing and testing formal algebraic program specifications”, in *Software Specification Techniques*, N.H. Gehani and A.D. McGettrick (eds.), Addison-Wesley, Reading, MA, pp. 391–420.

- J.A. Goguen, J.W. Thatcher, and E.G. Wagner (1978): “An initial algebra approach to the specification, correctness and implementation of abstract data types”, in *Current Trends in Programming Methodology, vol. IV: Data Structuring*, R.T. Yeh (ed.), Prentice Hall, Englewood Cliffs, NJ, pp. 80–149.
- J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright (1975): “An introduction to categories, algebraic theories, and algebras”, Research Report RC-5369, IBM T.J. Watson Research Center, Yorktown Heights, NY.
- M.A. Jackson (1975): *Principles of Program Design*, Academic Press, London, UK.
- M.A. Jackson (1983): *System Development*, Prentice Hall International Series in Computer Science, Prentice Hall, London, UK.
- R.J. Lauber (1982): “Development support systems”, *IEEE Computer*, Vol. 15, No. 5, pp. 36–46.
- K.T. Orr (1977): *Structured Systems Development*, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ.
- D.T. Sannella (1984): “A set-theoretic semantics for Clear”, *Acta Informatica*, Vol. 21, pp. 443–472.
- O. Shigo, K. Iwamoto, and S. Fujibayashi (1980): “A software design system based on a unified design methodology”, *Journal of Information Processing*, Vol. 3, No. 3, pp. 186–196.
- T.H. Tse (1985): “An automation of Jackson’s structured programming”, *Australian Computer Journal*, Vol. 17, No. 4, pp. 154–162.
- E.G. Wagner (1981): “Lecture notes on the algebraic specification of data types”, Research Report RC-9203, IBM T.J. Watson Research Center, Yorktown Heights, NY.
- E.G. Wagner, J.W. Thatcher, and J.B. Wright (1977): “Free continuous theories”, Research Report RC-6906, IBM T.J. Watson Research Center, Yorktown Heights, NY.
- V. Weinberg (1980): *Structured Analysis*, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ.
- E. Yourdon and L.L. Constantine (1979): *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ.
- S.N. Zilles, P. Lucas, and J.W. Thatcher (1982): “A look at algebraic specifications”, Research Report RJ-3568, IBM San Jose Research Laboratory, San Jose, CA.

Table 1.

Sort	Carrier
<i>task</i>	set of tasks
<i>name</i>	set of task names
<i>struct</i>	set of structures
<i>input</i>	set of inputs
<i>output</i>	set of outputs
<i>dataflow</i>	set of data flows
<i>flowname</i>	set of data flow names

```

task (process-sales; get (); put ());
sequ (task (get-valid-order; get (); put (outdata valid-order);
  sequ (task (get-order; get (source customer); put (outdata order); elem);
    task (validate-order; get (indata order); put (outdata valid-order); elem)));
  task (process-order; get (indata valid-order); put (outdata invoice-info);
  seln (task (process-local-order; get (indata lc-order); put (outdata invoice-info);
    sequ (task (prepare-local-invoice;
      get (indata lc-order); put (outdata pre-tax-info); elem);
      task (compute-tax;
        get (indata pre-tax-info); put (outdata invoice-info); elem)));
    task (prepare-overseas-invoice;
      get (indata os-order); put (outdata invoice-info); elem)));
  task (put-invoice; get (indata invoice-info); put (outfile invoice); elem)))

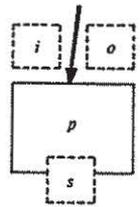
```

Figure 1. Sample Term in Initial Algebra

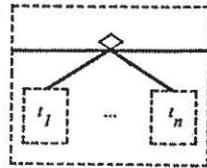
$\text{task}_Y: Y\langle \text{name} \rangle \times Y\langle \text{input} \rangle \times Y\langle \text{output} \rangle \times Y\langle \text{struct} \rangle \rightarrow Y\langle \text{task} \rangle$
 $\text{sequ}_Y: Y\langle \text{task} \rangle^n \rightarrow Y\langle \text{struct} \rangle$
 $\text{seln}_Y: Y\langle \text{task} \rangle^n \rightarrow Y\langle \text{struct} \rangle$
 $\text{para}_Y: Y\langle \text{task} \rangle^n \rightarrow Y\langle \text{struct} \rangle$
 $\text{iter}_Y: Y\langle \text{task} \rangle \rightarrow Y\langle \text{struct} \rangle$
 $\text{elem}_Y: \rightarrow Y\langle \text{struct} \rangle$
 $\text{get}_Y: Y\langle \text{dataflow} \rangle^n \rightarrow Y\langle \text{input} \rangle$
 $\text{put}_Y: Y\langle \text{dataflow} \rangle^n \rightarrow Y\langle \text{output} \rangle$
 $\text{indata}_Y: Y\langle \text{flowname} \rangle \rightarrow Y\langle \text{dataflow} \rangle$
 $\text{inflag}_Y: Y\langle \text{flowname} \rangle \rightarrow Y\langle \text{dataflow} \rangle$
 $\text{infile}_Y: Y\langle \text{flowname} \rangle \rightarrow Y\langle \text{dataflow} \rangle$
 $\text{source}_Y: Y\langle \text{flowname} \rangle \rightarrow Y\langle \text{dataflow} \rangle$
 $\text{outdata}_Y: Y\langle \text{flowname} \rangle \rightarrow Y\langle \text{dataflow} \rangle$
 $\text{outflag}_Y: Y\langle \text{flowname} \rangle \rightarrow Y\langle \text{dataflow} \rangle$
 $\text{outfile}_Y: Y\langle \text{flowname} \rangle \rightarrow Y\langle \text{dataflow} \rangle$
 $\text{sink}_Y: Y\langle \text{flowname} \rangle \rightarrow Y\langle \text{dataflow} \rangle$

such that

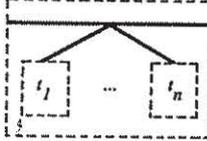
$\text{task}_Y(p, i, o, s) =$



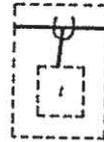
$\text{seln}_Y(t_1, \dots, t_n) =$



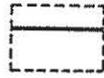
$\text{sequ}_Y(t_1, \dots, t_n) = \text{para}_Y(t_1, \dots, t_n)$



$\text{iter}_Y(t) =$



$\text{elem}_Y =$



$\text{get}_Y(d_1, \dots, d_n) =$



$\text{put}_Y(d_1, \dots, d_n) =$



$\text{indata}_Y(f) = \downarrow f$

$\text{inflag}_Y(f) = \downarrow f$

$\text{outdata}_Y(f) = \uparrow f$

$\text{outflag}_Y(f) = \uparrow f$

$\text{infile}_Y(f) = \text{source}_Y(f) = \text{outfile}_Y(f) = \text{slak}_Y(f) = \wedge$

Figure 2. Operations in Yourdon Algebra

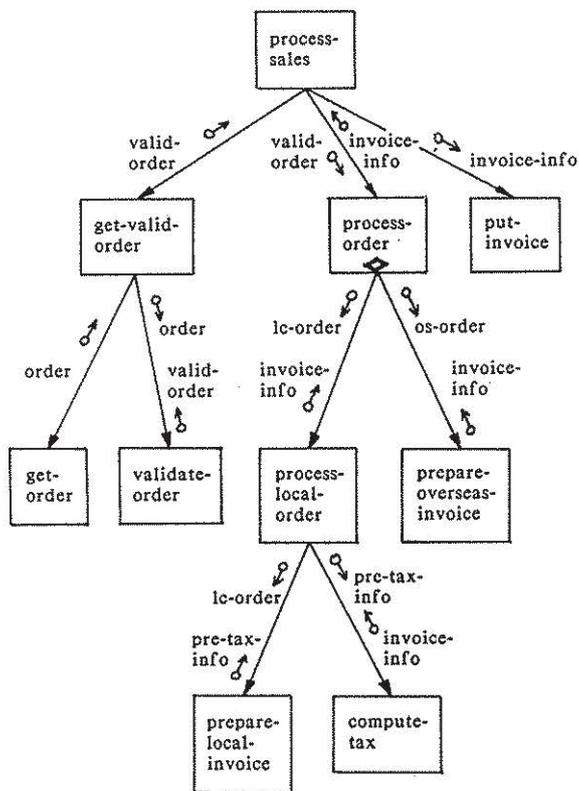
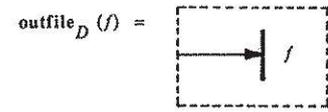
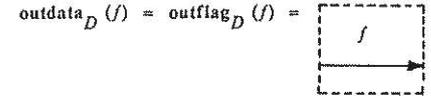
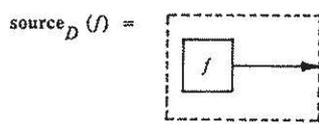
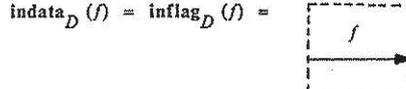
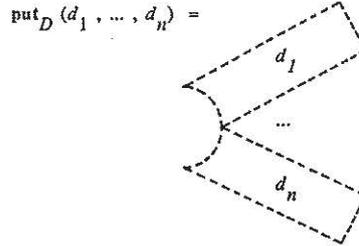
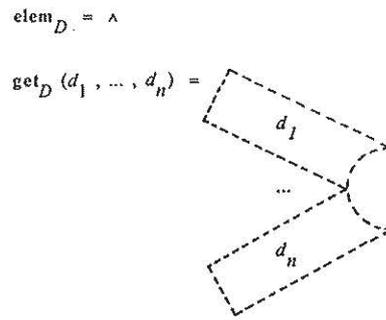
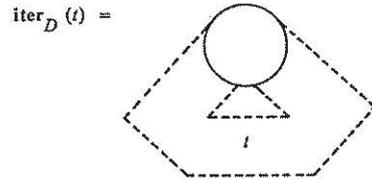
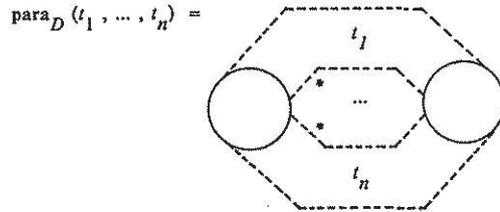
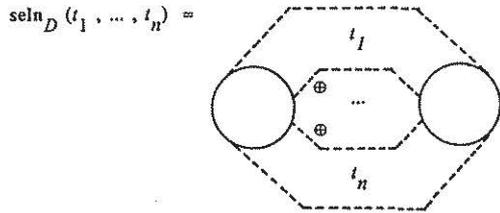
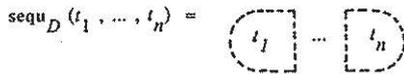
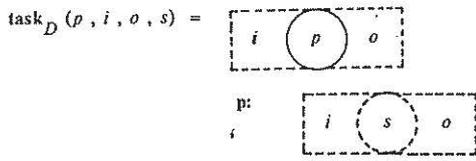


Figure 3. Sample Structure Charts in Yourdon Algebra

$\text{task}_D: D\langle \text{name} \rangle \times D\langle \text{input} \rangle \times$	$\text{elem}_D: \rightarrow D\langle \text{struct} \rangle$	$\text{source}_D: D\langle \text{flowname} \rangle \rightarrow D\langle \text{dataflow} \rangle$
$D\langle \text{output} \rangle \times D\langle \text{struct} \rangle \rightarrow D\langle \text{task} \rangle$	$\text{get}_D: D\langle \text{dataflow} \rangle^n \rightarrow D\langle \text{input} \rangle$	$\text{outdata}_D: D\langle \text{flowname} \rangle \rightarrow D\langle \text{dataflow} \rangle$
$\text{sequ}_D: D\langle \text{task} \rangle^n \rightarrow D\langle \text{struct} \rangle$	$\text{put}_D: D\langle \text{dataflow} \rangle^n \rightarrow D\langle \text{output} \rangle$	$\text{outflag}_D: D\langle \text{flowname} \rangle \rightarrow D\langle \text{dataflow} \rangle$
$\text{sels}_D: D\langle \text{task} \rangle^n \rightarrow D\langle \text{struct} \rangle$	$\text{indata}_D: D\langle \text{flowname} \rangle \rightarrow D\langle \text{dataflow} \rangle$	$\text{outdata}_D: D\langle \text{flowname} \rangle \rightarrow D\langle \text{dataflow} \rangle$
$\text{para}_D: D\langle \text{task} \rangle^n \rightarrow D\langle \text{struct} \rangle$	$\text{inflag}_D: D\langle \text{flowname} \rangle \rightarrow D\langle \text{dataflow} \rangle$	$\text{sink}_D: D\langle \text{flowname} \rangle \rightarrow D\langle \text{dataflow} \rangle$
$\text{iter}_D: D\langle \text{task} \rangle \rightarrow D\langle \text{struct} \rangle$	$\text{infile}_D: D\langle \text{flowname} \rangle \rightarrow D\langle \text{dataflow} \rangle$	

such that



subject to three further conditions:

- (a) If $p = \wedge$, then $\text{task}_D(p, i, o, s) =$
- (b) If $s = \wedge$, then $\text{task}_D(p, i, o, s) =$
- (c)

Figure 4. Operations in DeMarco Algebra

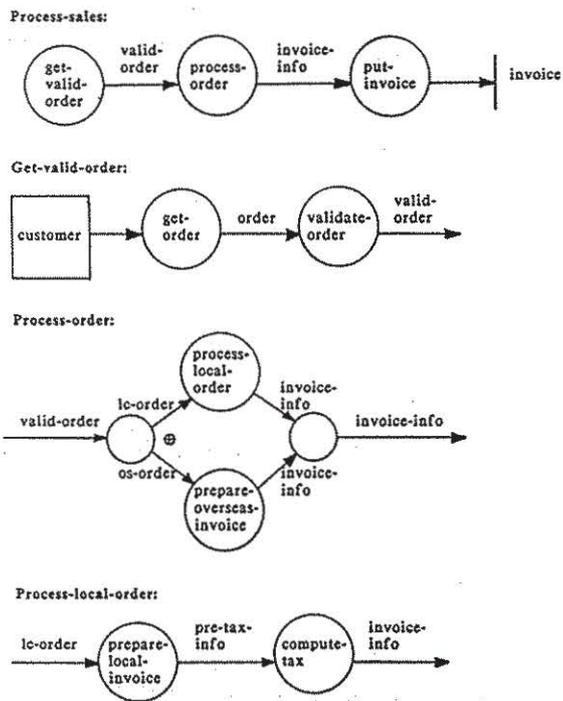


Figure 5. Sample Data Flow Diagram in DeMarco Algebra

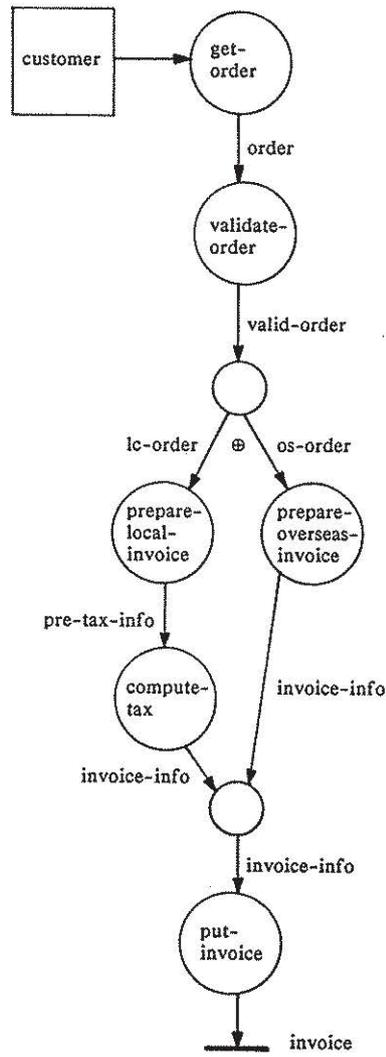


Figure 6. Further Data Flow Diagram in DeMarco Algebra

$$\text{seq}_J(p, t_1, \dots, t_n) = \begin{array}{l} p \text{ seq} \\ t_1 ; \\ \dots ; \\ t_n ; \\ p \text{ end} \end{array}$$

$$\text{sel}_J(p, t_1, \dots, t_n) = \begin{array}{l} p \text{ sel} \\ t_1 ; \\ p \text{ alt} \dots ; \\ p \text{ alt} t_n ; \\ p \text{ end} \end{array}$$

$$\text{itr}_J(p, t) = \begin{array}{l} p \text{ itr} \\ t \\ p \text{ end} \end{array}$$

$$\text{elm}_J = \begin{array}{l} p \end{array}$$

Figure 7. New Operations in Jackson Algebra

```
process-sales seq
  get-valid-order seq
    get-order;
    validate-order
  get-valid-order end;
  process-order sel
    process-local-order seq
      prepare-local-invoice;
      compute-tax
    process-local-order end;
  process-order alt
    prepare-overseas-invoice
  process-order end;
  put-invoice
process-sales end
```

Figure 8. Sample Structure Text in Jackson Algebra