The Application of Prolog to Structured Design[1]

T.H. Tse[2], T.Y. Chen[3], F.T. Chan[4],
H.Y. Chen[5] and H.L. Xie[6]

SUMMARY

In this paper, we investigate into the feasibility of applying logic programming to structured design. We propose to use Prolog as a common machinery for the representation of various structured tools. We illustrate through examples how to produce structure charts from data flow diagrams, and evaluate them according to commonly recommended design guidelines. If the structure charts produced are not satisfactory, the inherent backtracking mechanism in Prolog will help to produce other versions for further evaluation.

Key Words and Phrases: Data flow diagrams, logic programming, Prolog, structure charts, structured design.

INTRODUCTION

The application of logic programming to structured analysis and design is an area which has begun to attract much attention. Kowalski [1] suggested that data flow diagrams are different in syntax but equivalent in semantics to a logic-based language of conclusion-condition rules. Docker [2] used Prolog to develop a CASE tool called SAME to specify and exercise data flow diagrams. Steer [3] used a parallel logic programming language Parlog to model data flow diagrams. Tsai and Ridge [4] proposed the use of expert system in assisting structured design, and reported several problems encountered.

Although both structured analysis and structured design depend to a large extent on decision-making processes based on heuristics and human experience, most of the projects above are confined to the analysis phase and do not address the issues in the design phase. Logic programming, which is most suitable for logical inference, should play a prominent role in structured design as well.

One objective of our Software Engineering research group is to investigate into the feasibility of applying Prolog to structured design and develop automated tools to assist human designers in this important phase of the software life cycle. Our recent investigation [5] has contributed an initial effort in this respect and demonstrated the usefulness of Prolog in evaluation of structure charts. We would like to use a common machinery for all aspects of structured design, including the representation and processing of various structured tools. We find that Prolog is more suitable than conventional imperative languages for the purpose. Not only have we solved some problems formerly regarded as difficult, but also obtained better solutions for others which had only been solved partially [4].

---

1. This project is partially supported by a University and Polytechnic Grants Committee Research Grant, and a Research and Conference Grant of the University of Hong Kong.
2. Department of Computer Science, the University of Hong Kong, Pokfulam Road, Hong Kong. Part of this research was done at the Programming Research Group, University of Oxford.
3. Department of Computer Science, University of Melbourne, Melbourne, Australia.
4. School of Professional and Continuing Education, the University of Hong Kong, Pokfulam Road, Hong Kong.
5. Department of Computer Science, Jinan University, Guangzhou, China.
6. Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania.

We assume that readers of this paper are familiar with the fundamentals of Prolog. Please refer to [6, 7, 8] for details.

STRUCTURED SYSTEMS DEVELOPMENT

A number of software tools and techniques have been developed to enhance software productivity for various stages of the software life cycle. Structured systems development methodology stands out to be the most popular and successful [9, 10]. It provides a set of notations which can be used to specify the overall structure and functional requirements of a system in a graphical and hierarchical manner. Such a systems specification may be evaluated and manipulated under a set of well-accepted guidelines.

A *data flow diagram* [11, 12] is a graphical network representing the processes and data interfaces of a system. It is structured in the sense that a lower level represents a decomposition of the higher level. The details of the lowest level are described by *mini-specifications* [11] written in *structured English*. Specifications of the data interfaces and storage information are recorded in *data dictionaries*. A *structure chart* [13, 14] is a graphical tool for describing the hierarchy of modules to be implemented, as well as the algorithmic relationships and communication links among them. Like data flow diagrams, the detailed specifications of individual modules are written in structured English and the communication links are specified by means of data dictionaries.

A data flow diagram is a natural tool for the analysis phase of the software life cycle, while a structure chart is useful in the design and implementation phases. There is a vast difference in graphical outlook between data flow diagrams and structure charts, and the bridging of this gap is vital in ensuring a smooth transition between software development phases. To this end, structured methodology provides us with two important strategies called *transform analysis* and *transaction analysis*. Then the structured chart can be evaluated according to guidelines such as *coupling* and *cohesion*.

Numerous attempts and evaluations may be required before the final structure chart can be produced [11, 13, 14]. Tedious programming would be involved if an imperative language were used to develop an automated system. Implementation in Prolog is much easier since its intrinsic backtracking mechanism is particularly useful in such processes involving trial and error.

REPRESENTATIONS OF STRUCTURED TOOLS IN PROLOG

Tse [15] proposed the use of term algebra as a unified representation of various structured tools, and showed that there are mathematical mappings between them. Since data flow diagrams and structure charts are usually large in size, however, it may be rather cumbersome to handle long and deeply-nested terms. Instead, we propose to use a collection of Prolog predicates or relations to represent the structured tools. Such a representation turns out to be easier to understand and manipulate. In this section, we describe briefly how to represent data flow diagrams, structure charts, data dictionaries and structured English using Prolog.

There are five kinds of components in data flow diagrams, as illustrated in the sample in Figure 1. They are specified using the following predicates:

```
dfd_data_flow(Data).
dfd_file(Node).
dfd_source(Node).
dfd_sink(Node).
dfd_process(Node, Node_reference).
```

where `Node_reference` is a list specifying the reference number usually attached to each node in a data flow diagram. For instance, the node 4.3.2 is given a `Node_reference` of `[4,3,2]`. Similarly, the root node of the entire data flow diagram is given a `Node_reference` of `[1]`.

Furthermore, we use the following predicate to specify the connections between data flows and processes or other nodes:

```
dfd_couple(Node1, Data, Node2).
```

For example, the sample data flow diagram in Figure 1 is specified in Figure 2.

The components of structure charts (modules, connections and couples) are represented by the predicates

```
sc_module(Module).
sc_structure(Module, Structure_type, Children).
sc_couple(Module1, Data, Module2).
sc_data(Data, Data_type).
```

Here the predicate `sc_module` is used to specify the name of each module. The predicate `sc_structure` is used to specify the parent-child relationship between a `Module`, and a list of submodules, denoted by `Children`. The type of relationship is indicated by `Structure_type` and can either be `sequence`, `selection` or `iteration`. The predicate `sc_couple` is used to specify the data passed between two modules, and `sc_data` further specifies the type of each data couple, namely `atomic`, `composite` or `control`. For example, the sample structure chart in Figure 3 is specified in Figure 4.

A data dictionary is represented by a set of entries, each of which gives the structure of a data item as follows:

```
data_structure(Data, Data_structure).
```

A `Data_structure` can be defined using one or more composition operators, namely:

Operator	Meaning	Corresponding Symbol in Structured Design
`and`	sequence	`+`
`or`	selection	`[]`
`iter`	iteration	`{}`
`opt`	optional	`()`
`atom`	atomic data item	

For example,

```
order = [customer# | customer_name] +(discount)
   + {product# + quantity}
```

is represented by

```
data_structure(order,
   and([or([customer#, customer_name]), opt(discount),
        iter(and([product#, quantity]))])).
```

Structured English is a specification language consisting of a subset of natural English with only a limited vocabulary and limited language constructs. Each sentence may be a simple statement or a compound statement made of the sequence, selection and/or iteration of other statements. For the purpose of the evaluation exercise in our system, we represent the lowest level statements using a Prolog predicate

```
ms_statement(Node, Statement_reference, Verb, Inputs,
   Outputs)
```

where `Node` is the name of the node in the data flow digram which contains the statement, `Statement_reference` is a reference number assigned to each statement for identification, `Verb` is the command of the statement, `Inputs` is the list of data items referred to by the statement, and `Outputs` is the list of data items returned. For example, the statement ‘‘add amount to total’’ in a node `compute_total` may be represented by

```
ms_statement(compute_total, 1, add, [amount, total],
   [total]).
```

SYSTEM OVERVIEW

A structured systems specification (in the form of a hierarchy of data flow diagrams plus data dictionary and mini-specifications) is maintained in a `Dfd_file` in Prolog representation as described in the previous section. The system will (*a*) load the `Dfd_file` into the Prolog database, (*b*) construct a structure chart automatically by inserting its Prolog representation into the database, and (*c*) evaluate the construction using a set of standard criteria. If the result of an evaluation fails to pass the pre-defined standards, the system will backtrack and produce another structure chart for further evaluation. This will be repeated automatically until the result is acceptable. The accepted structure chart is then output to an `Sc_file`. In case none of the results is acceptable, we may relax the pre-defined standards and make further attempts.

Because of the inherent backtracking mechanism of Prolog, the above procedure can be defined using a simple predicate as follows:

```
dfd_to_sc(Dfd_file, Sc_file) :-
   reconsult(Dfd_file),
   dfd_process(Global_root, [1]),
      % Find the root of the entire data flow diagram
   construct_structure_chart(Global_root),
   evaluate,
   output_result_to(Sc_file).
```

The respective sub-programs will be discussed in the following sections.

CONSTRUCTION OF STRUCTURE CHARTS

Transform and transaction analyses are two supplementary strategies for producing structure charts. In transform analysis, we follow the input and output data streams of a data flow diagram to determine the central portion of the system responsible for the main transform of data. In this way, a balanced structured chart can be derived accordingly. In transaction analysis, we try to isolate a transaction centre which captures an input transaction, determines its type, and then processes it in the appropriate branch of the centre.

Yourdon [14] suggests to hide the transaction centres as if they are single processes, and apply transform analysis first. He then suggests the use of transaction analysis to deal with the hidden parts. On the other hand, Page-Jones [13] suggests that a system should first be broken

into suitably tractable sub-diagrams using transaction analysis. Each sub-diagram should then be converted into a structure chart using transform analysis. We propose to follow Yourdon in the main data flow diagram, but the advice of Page-Jones is also taken at the lower levels. Hence we implement the combined strategy as a recursive procedure, thus:

(*a*) Locate a proper level to start the conversion. A commonly suggested heuristics is to choose a level as high as possible but containing at least ten processes.

(*b*) Hide all the transaction centres in the diagram, recursively if necessary.

(*c*) Perform transform analysis on the diagram.

(*d*) Search the structure chart for modules whose corresponding processes contain hidden transaction centres. Expand these processes and perform the procedure on them. Paste the resulting subcharts on to the main structure chart.

(*e*) Recursively perform the procedure on the lower processes which have not been considered in (*a*) above.

The procedure can be specified and hence implemented in Prolog as follows:

```
construct_structure_chart(Root) :-
   expand_nodes([Root], Node_list, 10),
   ( Node_list = [], !
     ;
     transact_analysis(Node_list, _, New_node_list),
     transform_analysis(New_node_list, Root),
     expand_transact_centres,
     findall(Node,
        ( member(Node, Node_list),
          % Nodes without children:
          not sc_structure(Node, _, _) ),
        Leaf_node_list),
     zoom_into(Leaf_node_list) ).

zoom_into([]).

zoom_into([Node|Other_nodes]) :-
   construct_structure_chart(Node),
   zoom_into(Other_nodes).
```

In the rest of this section, we shall explain transform and transaction analyses in more detail, since the two together constitute the most important step in the preparation of a structure chart, resulting in a reasonably acceptable form for further evaluation.

Transform Analysis

A data flow diagram contains a transform centre, the afferent streams and the efferent streams. The transform centre is the collection of processes which make up the major function of the system. An afferent stream is a string of processes which start off by reading data from a physical source, and then convert it into a more abstract form suitable for the transform centre. An efferent stream, on the other hand, is a string of processes which convert output data from the transform centre into a more physical form suitable for output to the real world.

We can know for sure that some nodes must belong to the transform centre. They satisfy either of the following conditions:

(*a*) The node name contains a verb such as `subtract` which would alter the values of input data items;

(*b*) Statements in the mini-specification of the node contain verbs which would alter the values of input data items.

We define the `Core_transform_centre` as the minimal sub-data-flow-diagram containing these nodes.

We also know for sure that some nodes must belong to the afferent streams. As recommended by standard guidelines, we follow each input stream and identify those nodes whose input and output data names are similar except a qualifier. Examples are `order` and `valid_order`, or `formatted_debit` and `confirmed_debit`. We define `Core_afferent` as the minimal afferent streams containing these nodes. The nodes between `Core_afferent` and `Core_transform_centre`, not identified by the above criteria, can possibly be regarded either as part of the afferent streams or part of the transform centre. Our system would initially treat all of them as part of the afferent streams. The treatment on `Core_efferent` and that on the nodes between `Core_efferent` and `Core_transform_centre` are similar.

Structure charts can then be constructed and evaluated, such as by promoting a boss or hiring a boss, as described in detail in [13]. If, however, the structure charts thus constructed are unacceptable, the system will backtrack automatically and reduce the afferent and efferent streams. This will be repeated until the result is acceptable, or until the afferent and efferent streams cannot be reduced further.

The following, then, is a program for transform analysis:

```
transform_analysis(Node_list, Root) :-
   split(Node_list, Transform_centre, Afferent, Efferent),
   convert(Root, Transform_centre, Afferent, Efferent).
```

The predicate `split` divides a given set of processes in a data flow diagram into the transform centre and the sets of afferent and efferent processes as described. The predicate `convert` transforms them into sub-structure-charts and hangs them under the `Root` module. These predicates are defined as follows:

```
split(Node_list, Transform_centre, Afferent, Efferent) :-
   core_transform_of(Node_list, Core_transform_centre),
   core_afferent_of(Node_list, Core_afferent),
   semi_afferent_of(Node_list, Core_transform_centre,
      Core_afferent, Semi_afferent),
   semi_afferent_subpaths_of(Semi_afferent,
      Core_transform_centre, Semi_afferent_subpaths),
   append(Semi_afferent_subpaths, Core_transform_centre,
      Temp_transform_centre),
   subtract(Semi_afferent, Semi_afferent_subpaths,
      Remaining_semi_afferent),
   append(Core_afferent, Remaining_semi_afferent, Afferent),
   core_efferent_of(Node_list, Core_efferent),
   semi_efferent_of(Node_list, Core_transform_centre,
      Core_efferent, Semi_efferent),
   semi_efferent_subpaths_of(Semi_efferent,
      Core_transform_centre, Semi_efferent_subpaths),
   append(Temp_transform_centre, Semi_efferent_subpaths,
      Transform_centre),
   subtract(Semi_efferent, Semi_efferent_subpaths,
      Remaining_semi_efferent),
   append(Remaining_semi_efferent, Core_efferent, Efferent).
```

```
convert(Root, Transform_centre, Afferent, Efferent) :-
   retractable_assertz(sc_module(Root)),
   convert_afferent(Afferent, Root),
   convert_transform_centre(Transform_centre, Root),
   convert_efferent(Efferent, Root).

convert_transform_centre(Transform_centre, Root) :-
   hire_a_boss(Transform_centre, Root)
   ;
   promote_a_boss(Transform_centre, Root).
```

where the predicate `retractable_assertz(clause)` inserts the `clause` into the Prolog database in forward execution, but removes it in the case of backtracking.

Transaction Analysis

Transaction analysis consists of three steps:

(1) Find each transaction centre by locating the respective first node and all its subsequent branches.

(2) Reduce each of them into a single node for the ease of transform analysis.

(3) Re-expand the nodes afterwards.

If there is more than one transaction centre in a given data flow diagram, transaction analysis will be executed recursively.

The first node in a transaction centre is a process which inspects the type of each transaction entering the system and routes it to its corresponding branch for processing. We locate this first node (which we shall call `Milestone`) from the set of processes in a data flow diagram using the following predicate:

```
is_milestone(Node_list, Milestone) :-
   member(Milestone, Node_list),
   dfd_couple(_, Input, Milestone),
   setof(Data,
      N^dfd_couple(Milestone, Data, N),
      Data_list),
   is_transact_data(Input, Data_list),
   mutually_exclusive_types(Data_list).

is_transact_data(_, []).

is_transact_data(Input, [Data|Other_data]) :-
   is_part_of(Data, Input),
   is_transact_data(Input, Other_data).
```

where the predicate `is_part_of(Data, Input)` holds if every component of `Data` is a component of `Input`, and `mutually_exclusive_types(Data_list)` holds if there is a common component `X` in each data in `Data_list` such that all the `X`'s are different.

This example illustrates the advantage of using a declarative programming language to help automating structured design. We need only to declare the characteristics of `Milestone`. It is not necessary to specify the procedure for searching all modules. The `is_milestone` predicate will trigger the intrinsic backtracking mechanism and find every `Milestone`. The declarative characteristic of Prolog is useful also in many other parts of the system.

The predicate for transaction analysis may then be written as follows:

```
transact_analysis(Node_list, Temp_node_list,
      New_node_list) :-
   is_milestone(Node_list, Milestone), !,
   findall(Transact_branch,
      ( dfd_couple(Milestone, _, Second_node),
        transact_branch_of(Node_list, Milestone,
           Second_node, Transact_branch) ),
      Transact_branches),
   append([Milestone], Transact_branches, Transact_centre),
   assertz(is_transact_centre(Transact_centre)),
   reduce(Node_list, Milestone, Transact_centre,
      Temp_node_list),
   transact_analysis(Temp_node_list, Temp_node_list2,
      New_node_list)
   ;
   New_node_list = Node_list.
```

Here the predicate `reduce` replaces the entire `Transact_centre` by a single node. If the name of the original `Milestone` is `x`, then the single node will be given a name of `do_x`. The clause `is_transact_centre(Transact_centre)` is a fact inserted into the Prolog database to save the details of the hidden transaction centre for use in the predicate `expand_transact_centres` below. In case there is more than one transaction centre in a given data flow diagram, there will be more than one fact `is_transact_centre(Transact_centre)` inserted. The predicate `fail` in `expand_transact_centres` will cause backtracking to take place and re-expand further `Transact_centre`s thus saved.

```
expand_transact_centres :-
   is_transact_centre([Milestone|Transact_branches]),
   expand_branches(Milestone, Transact_branches),
   concat('do_', Milestone, New_milestone),
   setof(Node,
      D^sc_couple(New_milestone, D, Node),
      Node_list),
   assertz(sc_structure(New_milestone, selection,
      Node_list)),
   fail
   ;
   abolish(is_transact_centre / 1).

expand_branches(_, []).

expand_branches(Milestone,
      [Transact_branch|Other_branches]) :-
   concat('do_', Milestone, New_milestone),
   Transact_branch = [Second_node|Other_nodes],
   transform_analysis(Other_nodes, Second_node),
   dfd_couple(Milestone, Input, Second_node),
   retractable_assertz(sc_couple(New_milestone, Input,
      Second_node)),
   findall(Output,
      ( last_node_of(Transact_branch, Second_node,
           Last_node),
        dfd_couple(Last_node, Output, _) ),
      Output_list),
   retractable_assertz(sc_couple(Second_node, Output_list,
```

```
      New_milestone)),
   expand_branches(Milestone, Other_branches).
```

EVALUATION OF STRUCTURE CHARTS

Some of the structure charts produced by the predicates described in the last section may be reasonably acceptable by users, but others may not be worth considering. We must evaluate the structure charts according to some established guidelines [11, 13, 14], which may include coupling, cohesion, fan-out, fan-in consistency, tramp data, factoring, decision-splitting, morphology, and initializing and terminating modules. The first four are the most important and have been used in our system. Since Prolog programs are declarative in nature, any further expertise in evaluation, as long as it does not contradict existing heuristics, can be incorporated incrementally. The predicate for evaluation is specified simply as follows.

```
evaluate :-
   globally_acceptable_coupling,
   globally_acceptable_cohesion,
   globally_acceptable_fan_outs,
   globally_consistent_fan_ins.
```

Details of each predicate called will be explained in the subsequent subsections.

Coupling

Coupling is a measure of the inter-dependence among modules in a structure chart. A good system should consist of as many independent modules as possible, so that low coupling between modules signifies a well-designed system. An important means of measuring coupling is by inspecting the types of data passing between modules. We can divide coupling into five major types, varying from the best to the worst. Data coupling is the best. It means that two modules communicate through atomic data items. The next one on the list is stamp coupling, which means that two modules communicate through composite data items. Control coupling between two modules means that they communicate through control flags. Common coupling means that modules use global data. Finally, content coupling means that a module refers to (or changes) some data within another module, or refers to (or alters) a statement in another module.* The last two types of coupling are generally regarded as unsatisfactory.

Having defined the notion of satisfactory/unsatisfactory coupling, an obvious extension is that a structure chart is acceptable if none of its modules have unsatisfactory coupling. Such an extension, however, would be too strict and impractical. A structure chart is usually considered acceptable in practice even if a limited percentage of its modules are not satisfactory. The tolerance limit is specified by the user by means of inserting a fact `maximal_unsatisfactory_coupling(Defined_percent)` into the Prolog database.

Thus, the following predicate `globally_acceptable_coupling` is used to determine the overall acceptability of the structure chart in terms of coupling. As long as the `Percent` of unsatisfactory coupling is not more than `Defined_percent`, the system will backtrack automatically, so that another module is checked. If, however, the `Percent` of unsatisfactory modules exceeds the defined limit, the predicate will fail.

---

* The second aspect of content coupling is the result of poor programming and cannot be detected at the systems design phase.

```
globally_acceptable_coupling :-
   setof(M, sc_module(M), Module_list),
   length(Module_list, No_of_modules),
   maximal_unsatisfactory_coupling(Defined_percent),
      % Reasonable percentage of modules not having
      % satisfactory coupling, as specified by the user
   ctr_set(0, 1),
   member(Module, Module_list),
   unsatisfactory_coupling(Module),
   ctr_inc(0, No_of_unsatisfactory_modules),
   Percent is No_of_unsatisfactory_modules / No_of_modules,
   Percent > Defined_percent, !, fail
   ;
   true. % Percent =< Defined_percent
```

Here `ctr_set` and `ctr_inc` are pre-defined predicates in Arity Prolog, which is used for our system. `ctr_set(0, 1)` initializes the contents of ''counter 0'' to one. `ctr_inc(0, Variable)` moves the latest contents of ''counter 0'' to `Variable` and then adds one to the counter. The predicate `unsatisfactory_coupling(Module)` detects common or content coupling in a module by detecting

(*a*) data which is input to a statement but is neither captured explicitly into the module nor generated by another statement in the same module, and

(*b*) data which is output from a statement but is neither exported explicitly from the module nor generated by another statement in the same module.

The predicate can simply be defined as follows:

```
unsatisfactory_coupling(Module) :-
   ms_statement(Module, _, _, Inputs, _),
   member(Data1, Inputs),
   not sc_couple(_, Data1, Module),
   not ( ms_statement(Module, _, _, _, Outputs),
         member(Data1, Outputs) ), !
   ;
   ms_statement(Module, _, _, _, Outputs),
   member(Data2, Outputs),
   not sc_couple(Module, Data2, _),
   not ( ms_statement(Module, _, _, Inputs, _),
         member(Data2, Inputs) ).
```

Cohesion

Another major design guideline involves cohesion, which is a measure of the strength of functional association of activities within a module. It is commonly recommended that elements in modules should be highly cohesive, that is, strongly inter-related.

There are seven major levels of cohesion. The best level is functional cohesion, where a module performs a single identifiable function. The next level is sequential cohesion, where the data produced in an earlier part of the module will be used in a later part of the same module. In a module with communicational cohesion, the elements process data items in the same file, but not necessarily in any specific order. In a module with procedural cohesion, the elements are related by program control algorithms such as selection or iteration. Temporal cohesion means that elements are grouped under the same module because they are time-related. Logical cohesion means that elements are grouped under the same module because they are supposed to have similar behaviour, but actually exhibit minor differences. The worst level of cohesion is

coincidental cohesion, where the elements of a module are grouped together for no specific reason.

Tsai and Ridge [4] have pointed out the difficulty in determining cohesion levels automatically. Fortunately, we do not need to determine the exact level of cohesion of a module in most cases of evaluation. We say that two statements are related with each other if an output of one statement is also an input to the other. The number of related statements should be relatively large in a module with strong cohesion such as functional, sequential or communicational cohesion. Thus the cohesion of a given module is regarded as satisfactory if there is a reasonably percentage of related statements. Furthermore, the overall cohesion of a structure chart is deemed to be acceptable if there is a reasonable percentage of modules having satisfactory cohesion. This is specified using a predicate `globally_acceptable_cohesion`. Its logic is similar to `globally_acceptable_coupling` and hence not repeated here. It uses the backtracking mechanism of Prolog to call the predicate `satisfactory_cohesion(Module)` and decide whether only a reasonable percentage of `Module`s do not have satisfactory cohesion.

```
satisfactory_cohesion(Module) :-
   findall([Statement1, Statement2],
      ( ms_statement(Module, Statement1, _, _, _),
        ms_statement(Module, Statement2, _, _, _),
        not Statement1 = Statement2 ),
      Statement_pairs),
   length(Statement_pairs, No_of_statement_pairs),
   ( No_of_statement_pairs = 0, !
     ;
     related(Statement_pairs, Related_statement_pairs),
     length(Related_statement_pairs,
        No_of_related_statement_pairs),
     Percent2 is No_of_related_statement_pairs /
        No_of_statement_pairs,
     minimal_related_statements(Defined_percent2),
        % Reasonable percentage of related statements
        % in a module, as specified by the user
     Percent2 > Defined_percent2 ).

related([], []).

related([[Statement1, Statement2]|Statement_pairs],
      [[Statement1, Statement2]|Related_statement_pairs]) :-
   ms_statement(_, Statement1, _, _, Outputs)
   ms_statement(_, Statement2, _, Inputs, _)
   component_of(Outputs, Data_element),
   component_of(Inputs, Data_element), !,
   related(Statement_pairs, Related_statement_pairs).

related([[_, _]|Statement_pairs],
      Related_statement_pairs) :-
   related(Statement_pairs, Related_statement_pairs).
```

where `component_of(Data_list, Data_element)` will hold if `Data_element` is

Fan-Out

In a structure chart, the fan-out from a module is the number of immediate subordinates to that module. Due to the limits of human beings and according to software development experience, we should limit the fan-out from a module to be no more than seven [16]. This is

implemented using a predicate `globally_acceptable_fan_outs` similar to `globally_acceptable_coupling`. It helps to decide whether or not only a limited percentage of modules have unsatisfactory fan-outs. Such modules can be detected by a simple predicate as follows:

```
unsatisfactory_fan_out(Module) :-
   sc_module(Module),
   sc_structure(Module, _, Children),
   length(Children, N),
   N > 7.
```

Fan-In Consistency

If any given `Module` is called by more than one parent module, say `Parent1` and `Parent2`, the data couples between `Parent1` and `Module` should be consistent with those between `Parent2` and `Module`. This is known as consistent fan-in. We define a predicate `globally_consistent_fan_ins` to ensure fan-in consistency throughout the entire structure chart. This is achieved by calling the predicate `consistent_fan_ins(Module_list)`, which not only checks the fan-in consistency of every `Module` in `Module_list`, but continues the check recursively for every child of the `Module`.

```
globally_consistent_fan_ins :-
   dfd_process(Global_root, [1]),
         % Find the root of the entire data flow diagram
   sc_structure(Global_root, _, Children),
   consistent_fan_ins(Children).

consistent_fan_ins([]).

consistent_fan_ins([Module|Other_modules]) :-
   % Find all Parents:
   findall(P,
      ( sc_structure(P, _, Children),
        member(Module, Children) ),
      Parents),
   % Single parent or consistent couples among parents:
   ( Parents = [_], !
     ;
     consistent_couples(Module, Parents) ),
   ( sc_structure(Module, _, Children), !,
     consistent_fan_ins(Children)
     ;
     true ),
   consistent_fan_ins(Other_modules).
```

where the predicate `consistent_couples` is defined as follows:

```
consistent_couples(Module, Parents) :-
   all_couples_of(Module, Parents, Data_lists),
   same_contents(Data_lists).
```

```
all_couples_of(_, [], []).

all_couples_of(Module, [Parent|Other_parents],
      [Data_list|Other_data_lists]) :-
   setof(Data,
      ( sc_couple(Module, Data, Parent)
        ; sc_couple(Parent, Data, Module) ),
      Data_list),
   all_couples_of(Module, Other_parents, Other_data_lists).
```

CONCLUSION

This paper reports on an attempt to apply logic programming techniques in structured design. We have solved some problems formerly regarded as difficult, and obtained better solutions for other problems which had only been solved partially, such as in the areas of transform analysis, coupling and cohesion. We have developed a Prolog system to implement our proposals. We have found that Prolog is an excellent tool for the automation of structured design because of several reasons. Firstly, it serves as a common notation for representing data flow diagrams, structure charts, data dictionaries and mini-specifications. This facilitates cross-referencing and reduces the probability of errors when one structured tool is converted into another. Secondly, underlying backtracking and pattern-matching mechanisms of Prolog enable multiple design solutions to be found. If a design is not satisfactory according to the evaluation criteria, feasible alternatives can be determined automatically. Thirdly, because of its declarative nature, Prolog supports the incremental incorporation of evaluation criteria and other guidelines on structured design.

ACKNOWLEDGEMENT

We are especially grateful to Mr Raymond Chan of IBM for his invaluable contribution to this project.

REFERENCES

[1] R.A. Kowalski, ‘‘Software engineering and artificial intelligence in new generation computing’’, *Future Generation Computer Systems* **1** (1): 39–49 (1984).

[2] T.W.G. Docker, ‘‘SAME: a structured analysis tool and its implementation in Prolog’’, in *Logic Programming: Proceedings of 5th International Conference and Symposium*, R.A. Kowalski and K.A. Bowen (eds.), MIT Press, Cambridge, Massachusetts, pp. 82–95 (1988).

[3] K. Steer, ‘‘Testing data flow diagrams with Parlog’’, in *Logic Programming: Proceedings of 5th International Conference and Symposium*, R.A. Kowalski and K.A. Bowen (eds.), MIT Press, Cambridge, Massachusetts, pp. 96–110 (1988).

[4] J.J.-P. Tsai and J.C. Ridge, ‘‘Intelligent support for specifications transformation’’, *IEEE Software* **5** (6): 28–35 (1988).

[5] T.Y. Chen, C.S. Kwok, W.H. Tang, and T.H. Tse, ‘‘Evaluation of structure charts: a logic programming approach’’, in *Databases in the 1990s 2: Proceedings of 2nd Australian Conference on Database and Information Systems*, B. Srinivasan and J. Zeleznikow (eds.), Sydney, Australia, pp. 270–284 (1991).

[6] I. Bratko, *Prolog Programming for Artificial Intelligence*, Addison-Wesley, Wokingham, UK (1986).

[7] W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, 3rd Edition, Springer-Verlag, Berlin (1987).

[8] L. Sterling and E.Y. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, MIT Press, Cambridge, Massachusetts (1986).

[9] M.A. Colter, ‘‘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, pp. 73–96 (1982).

[10] C.A. Richter, ‘‘An assessment of structured analysis and structured design’’, *ACM Software Engineering Notes* **11** (4): 41–45 (1986).

[11] T. DeMarco, *Structured Analysis and Systems Specification*, Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey (1979).

[12] E. Yourdon, *Modern Structured Analysis*, Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey (1989).

[13] M. Page-Jones, *The Practical Guide to Structured Systems Design*, Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey (1988).

[14] E. Yourdon and L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey (1979).

[15] T.H. Tse, *A Unifying Framework for Structured Analysis and Design Models: an Approach using Initial Algebra Semantics and Category Theory*, Cambridge Tracts in Theoretical Computer Science, Vol. 11, Cambridge University Press, Cambridge (1991).

[16] G.A. Miller, ‘‘The magic number seven, plus or minus two: some limits on our capacity for processing information’’, *Psychological Review* **63**: 81–97 (1956).

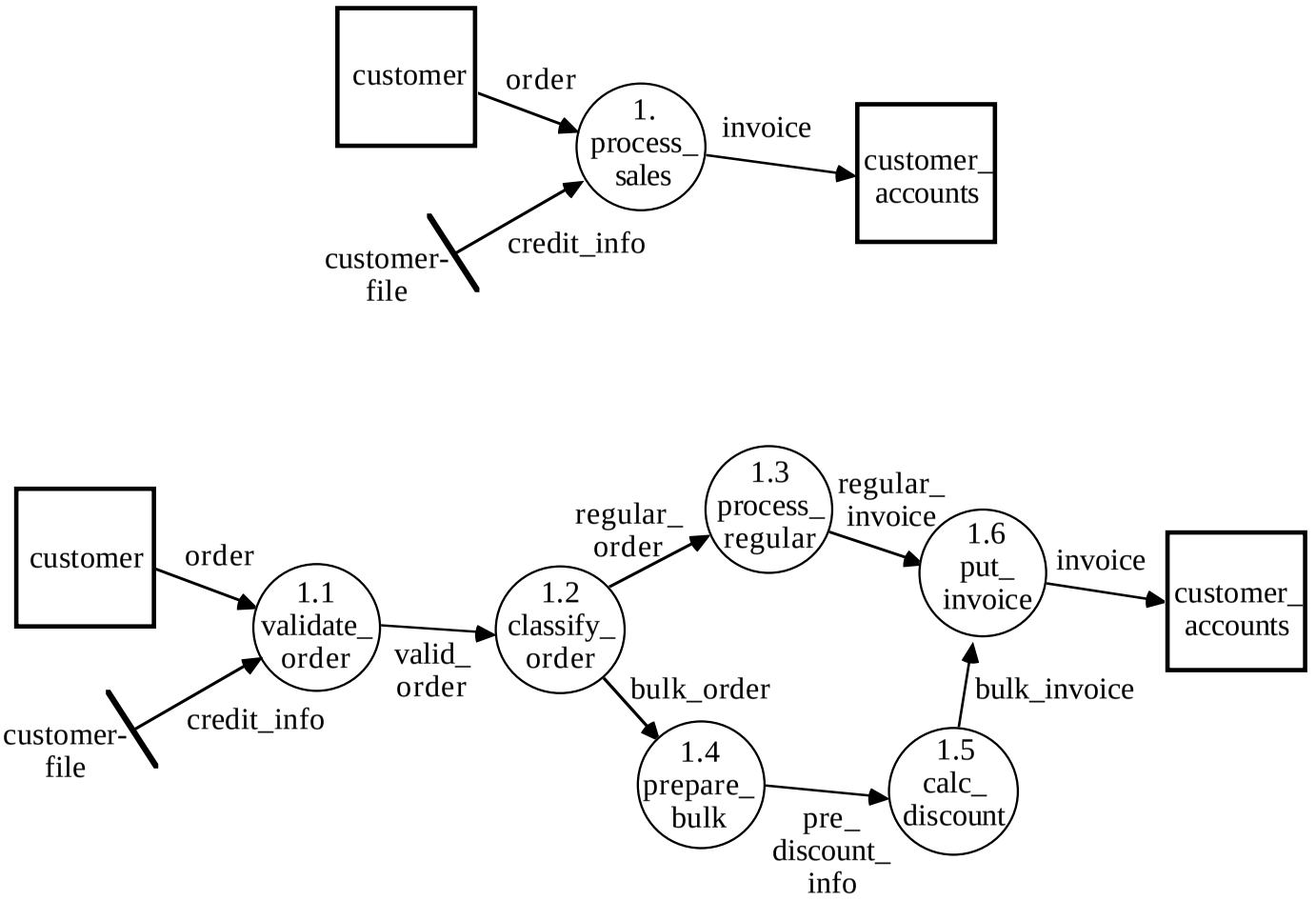

Figure 1 Sample Data Flow Diagram

```
dfd_data_flow(order).
dfd_data_flow(credit_info).
dfd_data_flow(valid_order).
dfd_data_flow(regular_order).
dfd_data_flow(regular_invoice).
dfd_data_flow(bulk_order).
dfd_data_flow(pre_discount_info).
dfd_data_flow(bulk_invoice).
dfd_data_flow(invoice).

dfd_file(customer_file).

dfd_source(customer).

dfd_sink(customer_accounts).

dfd_process(process_sales, [1]).
dfd_process(validate_order, [1, 1]).
dfd_process(classify_order, [1, 2]).
dfd_process(process_regular, [1, 3]).
dfd_process(prepare_bulk, [1, 4]).
dfd_process(calc_discount, [1, 5]).
dfd_process(put_invoice, [1, 6]).

dfd_couple(customer, order, process_sales).
dfd_couple(customer_file, credit_info, process_sales).
dfd_couple(process_sales, invoice, customer_accounts).
dfd_couple(customer, order, validate_order).
dfd_couple(customer_file, credit_info, validate_order).
dfd_couple(validate_order, valid_order, classify_order).
dfd_couple(classify_order, regular_order, process_regular).
dfd_couple(process_regular, regular_invoice, put_invoice).
dfd_couple(classify_order, bulk_order, prepare_bulk).
dfd_couple(prepare_bulk, pre_discount_info, calc_discount).
dfd_couple(calc_discount, bulk_invoice, put_invoice).
dfd_couple(put_invoice, invoice, customer_accounts).
```

Figure 2 Sample Representation of Data Flow Diagram in Prolog

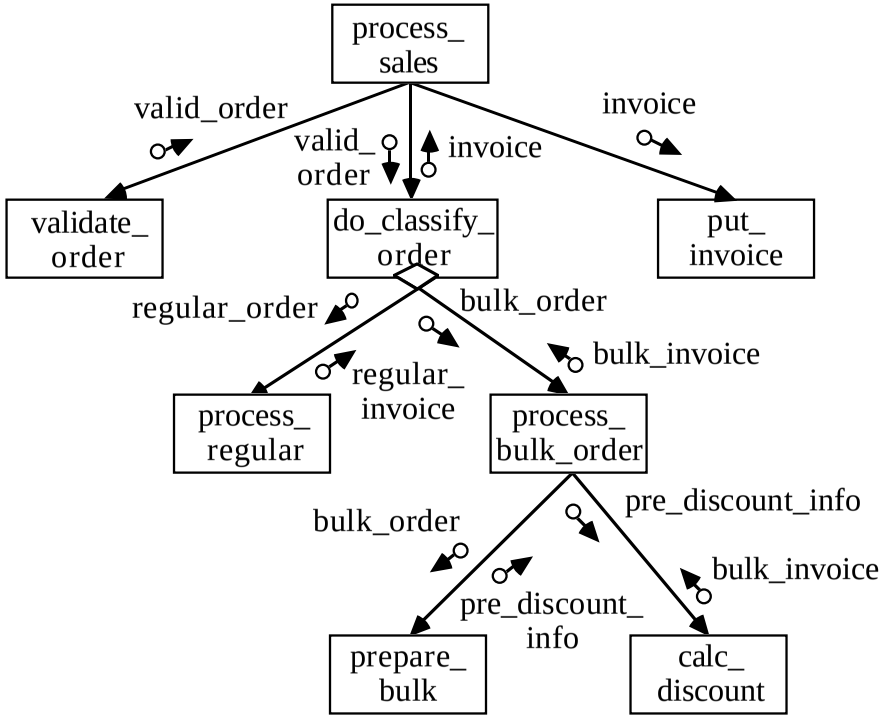

Figure 3 Sample Structure Chart

```
sc_module(process_sales).
sc_module(validate_order).
sc_module(do_classify_order).
sc_module(put_invoice).
sc_module(process_regular).
sc_module(process_bulk_order).
sc_module(prepare_bulk).
sc_module(calc_discount).

sc_structure(process_sales, sequence,
   [validate_order, do_classify_order, put_invoice]).
sc_structure(do_classify_order, selection,
   [process_regular, process_bulk_order]).
sc_structure(process_bulk_order, sequence,
   [prepare_bulk, calc_discount]).

sc_data(valid_order, composite).
sc_data(invoice, composite).
sc_data(regular_order, composite).
sc_data(regular_invoice, composite).
sc_data(bulk_order, composite).
sc_data(bulk_invoice, composite).
sc_data(pre_discount_info, composite).

sc_couple(validate_order, valid_order, process_sales).
sc_couple(process_sales, valid_order, do_classify_order).
sc_couple(do_classify_order, invoice, process_sales).
sc_couple(process_sales, invoice, put_invoice).
sc_couple(do_classify_order, regular_order, process_regular).
sc_couple(process_regular, regular_invoice, do_classify_order).
sc_couple(do_classify_order, bulk_order, process_bulk_order).
sc_couple(process_bulk_order, bulk_invoice, do_classify_order).
sc_couple(process_bulk_order, bulk_order, prepare_bulk).
sc_couple(prepare_bulk, pre_discount_info, process_bulk_order).
sc_couple(process_bulk_order, pre_discount_info, calc_discount).
sc_couple(calc_discount, bulk_invoice, process_bulk_order).
```

Figure 4 Sample Representation of Structure Chart in Prolog