Postprint of article in *Information and Software Technology* **36** (1): 23−33 (1994)

The Use of Prolog in the Modelling and Evaluation of Structure Charts*†

T.H. Tse‡
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
(thtse@cs.hku.hk)

T.Y. Chen
Department of Computer Science and Software Engineering
The University of Melbourne
(tyc@cs.mu.oz.au)

C.S. Kwok
E & M Engineering Department
Mass Transit Railway Corporation
Hong Kong

We summarize our experience in the use of Prolog to model and evaluate structure charts according to standard guidelines in structured design. We discuss how to construct first-cut structure charts automatically from data flow diagrams using transform and transaction analyses, evaluate them using recommended criteria such as coupling, cohesion, morphology and tramp, and improve on the resulting structure charts by means of automatic backtracking.

Logic programming, Prolog, structure charts, structured design

† This project is supported in part by a Research Grant of the University and Polytechnic Grants Committee.

‡ Part of the research was done at the Programming Research Group, University of Oxford under an ACU Visiting Fellowship.

INTRODUCTION

Structured analysis and design methodology has been one of the most popular and successful methods in information systems development[1, 2]. A complex system is specified in the form of a collection of graphical and textual representations, each of which is suited to a different phase of the system life cycle. Each representation has a hierarchical structure, so that users can conceptualize the target system at a high level of abstraction, and then look for details at a lower level. Interfaces between subsystems are defined explicitly and kept to a minimum. One representation of the system is converted to another when we pass from one phase of the life cycle to another. The whole process is based on recommended guidelines laid down by experienced systems developers.

For example, it is recognized that flow graphs are better records of procedures, while hierarchical charts are more suitable for defining program control. Graphical specifications are excellent for presenting overviews of complex systems, but textual languages are better tools for detailed description. Thus data flow diagrams[3, 4, 5] are recommended in the structured methodology for the systems analysis phase, when we wish to capture the way data items move from one user task to another. Structure charts[6, 4, 7] are recommended for the systems design phase, when the overall structure of the target systems becomes more concrete and program control mechanisms need to be determined. Textual mini-specifications[3, 5] are used for defining minute details of individual components of the system.

Conventional CASE tools for structured methodology provide extremely user-friendly graphical interfaces and elaborate cross-referencing facilities. However, although many of them support the transformation of one structured representation to another (such as data flow diagrams into structure charts), this is often done without much consideration on the design philosophy or evaluation criteria such as coupling and cohesion. Most of them do not support the comparison of the relative merits of various alternatives for selecting the most appropriate design choice. This is because design decisions in structured methodology are not well-defined algorithms[6], but are imprecise heuristics collected through the experience of practitioners. Such recommendations cannot be programmed easily in conventional imperative languages, on which the conventional CASE tools are based. We must provide systems designers with a new generation of CASE tools with the intelligence to automate the decision-making processes. Logic programming techniques could be used to simulate the expertise.

In particular, the application of logic programming to structured design is an area which is relatively underexplored. Most of the related work in structured methodology, such as Goble[8], emphasizes only on analysis. Kowalski[9] has suggested that data flow diagrams are equivalent in semantics to logic programs, but little is said about structure charts. Docker[10] has used Prolog to develop a CASE tool known as SAME to simulate the behaviour of data flow diagrams. Tsai and Ridge[11] have reported on the problems encountered in developing expert systems for the evaluation of structure charts, but have not proposed a feasible solution.

In this paper, we present our experience* in the use of Prolog as a modelling and evaluation tool for structure charts. We discuss how to construct first-cut structure charts from data flow diagrams using transform and transaction analyses. We then illustrate how to evaluate the resulting charts using standard recommended criteria such as coupling, cohesion, morphology and tramp, so as to improve on the charts by means of automatic backtracking.

---

* A prototype system has been implemented using Arity Prolog. Interested readers from academia may contact the first author for a copy of the programs at a nominal handling charge.

Basic knowledge of Prolog is assumed in this paper. Readers may like to refer to Bratko[12], Clocksin and Mellish[13], or Sterling and Shapiro[14] for more information.

MODELLING OF STRUCTURE CHARTS

A unique feature of logic programming is that one can use a set of relations to represent a data structure. We shall not discuss in detail the relative merits and dismerits of term-based and relation-based representations (see Kowalski[15], for example). Our contention is that a structure chart is usually large, and hence would be rather cumbersome to be encoded as one huge term[2]. On the other hand, the use of individual relations allows us to add new modules or other components more easily when the needs arise.

Thus, a structure chart is modelled by a set of Prolog predicates. They cover the standard components, namely the modules, the substructures of the modules, the data items, and how the data items communicate between modules.

Consider, for example, a typical structure chart often quoted in standard texts such as Page-Jones[6]. It is an interactive system for updating files, as shown in Figure 1. It is modelled in Prolog as follows:

Modules

We define the modules using an `isModule` predicate. One clause is specified for each module in the system. Examples are:

```
isModule(updateFile).
isModule(getValidTrans).
isModule(putNewMaster).
```

Substructures of Modules

We then specify the control structure of submodules in each module. The predicate `structureOf(Parent, ControlType, ChildModules)` indicates that a `Parent` consists of a list of `ChildModules`, which are linked together by one of the `ControlTypes` "`sequence`", "`selection`" or "`iteration`". Thus:

```
structureOf(updateFile, iteration,
  [getValidTrans, getMaster, updateMaster, putNewMaster]).

structureOf(getValidTrans, sequence,
  [getTrans, validateTrans]).

structureOf(putNewMaster, sequence,
  [formatMaster, writeMaster, askIfUserWantsToContinue]).
```

Data Items

We define the data items in a structure chart using an `isData` predicate. The first argument of the predicate is the name of a data item. The second argument tells us whether it is an `atomic` item, a `record`, or a `control` flag. Examples are:

```
isData(validTrans, record).
isData(newMaster, record).
isData(continueResponse, control).
```

Communications between Modules

Finally, we define the communications between modules using the predicate `couplingBetween(Module1, Module2, Data)`, which indicates that `Data` is passed from `Module1` to `Module2`. Examples are:

```
couplingBetween(getValidTrans, updateFile, validTrans).
couplingBetween(updateFile, putNewMaster, newMaster).
```

Thus a structure chart is specified fully using the above four kinds of predicates.

CONSTRUCTION OF STRUCTURE CHARTS FROM DATA FLOW DIAGRAMS

We have also defined predicates for the specification of data flow diagrams, such as

```
isBubble(Node).
isSource(Node).
isSink(Node).
isFile(Node).
isDataFlow(DataFlow).
childrenOf(Node, ChildNodes).
dataFlowBetween(Node, Node2, DataFlow).
```

These predicates are similar to those for structure charts and hence detailed explanation or examples will not be given here.

Suppose a data flow diagram has been specified as facts in a Prolog database, and suppose that the bubble in the context diagram is `RootNode`. In order to convert it into a structure chart and to save the result as `StructureChartFile`, we should issue a goal:

```
?- structureChartFor(RootNode, StructureChartFile).
```

The predicate is defined as follows:

```
structureChartFor(RootNode, StructureChartFile) :-
  convert(RootNode),
  evaluate,
  saveAs(StructureChartFile).
```

Thus a first-cut structure chart is inserted into the Prolog database, and then evaluated. If the evaluation is not satisfactory, the system will backtrack automatically and produce another version of structure chart for evaluation. When the evaluation is successful, or when the user does not want any further improvement, the final structure chart representation is saved as `StructureChartFile`. Examples of the evaluation of the structure chart using the `evaluate` predicate will be given in the next section.

To construct structure charts from data flow diagrams, two supplementary strategies are normally recommended: transform and transaction analyses. In transform analysis, we follow the input and output data streams of a data flow diagram to determine the central portion responsible for the main transform of data. In this way, a balanced structure 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 in the centre. We combine the suggestions of Page-Jones[6] and Yourdon[7] on transform and transaction analyses to produce a first-cut structure chart, and specify a recursive conversion procedure thus:

(a) Use breadth-first strategy to expand each node in the data flow diagram by recursively zooming into its children, grandchildren and so on, until sufficient details* have been shown.
(b) Find the transaction centres in the data flow diagram and reduce each one of them into a single node.
(c) Perform transform analysis on the resulting data flow diagram to produce a structure chart.
(d) Re-expand the hidden transaction centres.
(e) Recursively perform the conversion procedure on the leaf modules in the structured chart produced above, until they cannot be expanded further because they have no children in the original data flow diagram.

An example illustrating the conversion procedure is shown in Figure 2. The corresponding Prolog predicate `convert` is as follows:

```
convert(RootNode) :-
  childrenOf(RootNode, []).

convert(RootNode) :-
  breadthFirstExpand([RootNode], NodeList),
  hideTransactCentres(NodeList, _, NewNodeList),
  transformAnalysis(RootNode, NewNodeList),
  expandTransactCentres,
  findall(Module,
    ( isModule(Module), structureOf(Module, _, []) ),
    LeafModuleList),
  continue(LeafModuleList).

continue([]).

continue([Module|OtherModules]) :-
  convert(Module),
  continue(OtherModules).
```

Here the predicate `transformAnalysis` is involved with transform analysis and `hideTransactCentres` and `expandTransactCentres` are involved with transaction analysis. These predicates will be explained in detail in the next two subsections.

Transform Analysis

A data flow diagram contains a central transform plus afferent and efferent streams[6, 7]. The central transform is the collection of nodes which make up the major function of the system. An afferent stream is a string of nodes which start off by reading data from a physical source, and then convert it

* We follow the recommendation that we must expand each node into a minimum of "more than 9 nodes", unless the node happens to have less than 9 children and grandchildren.

into a more abstract form suitable for the central transform. An efferent stream, on the other hand, is a string of nodes which convert output data from the central transform into a more physical form suitable for output to the real world. Tsai and Ridge[11] suggest that, in order to identify the central transform, user input must be required. We would like, however, to automate the structured design process as much as possible by defining recommended heuristics using Prolog predicates.

In order to identify the central transform, we should first of all identify the nodes which are potentially part of an afferent or efferent stream. The criteria are:

(a) There is a close resemblance between the input and output data for the node. For instance, both the input and output contain identical names qualified by different prefixes, such as `validTrans` and `confirmedTrans`.

(b) The name of the node does not contain a verb implying an abrupt change between its input and output data, such as `update`.

We then identify the central transform as the remaining nodes.

One question often raised is, in the absence of real human expertise, is there a chance of isolating the wrong nodes as the transform centre? We should note that transform analysis is only a recommended "strategy" for transforming data flow diagrams into structure charts, instead of an "algorithm"[6]. A good transform centre thus found would reduce the number of backtracking required for determining the best structure chart. If, however, we have isolated a different transform centre by making the wrong assumptions on afferent and efferent flows, we may still arrive at a similar structure chart, albeit after a number of unnecessary backtracking.

The following is the program for transform analysis:

```
transformAnalysis(Node, NodeList) :-
  newModule(Node),
  transformCentreOf(NodeList, TransformCentre),
  convertTransformCentre(TransformCentre, TransformRoot),
  convertAfferent(Node, NodeList, TransformRoot,
    TransformCentre),
  convertEfferent(Node, NodeList, TransformRoot,
    TransformCentre),
  addStructure(Node).
```

The details of the predicates in the program are as follows:

(i) The predicate `newModule(Node)` creates a new parent module known as `Node`. This is done by inserting a fact `isModule(Node)` into the Prolog database.

(ii) The predicate `transformCentreOf` identifies the transform centre from a given `NodeList`, thus:

```
transformCentreOf(NodeList, TransformCentre) :-
  ( findall(Node,
      ( member(Node, NodeList),
        inputOutputSimilar(Node) ),
      AffAndEffStreams)
    ;
    findall(Node,
      ( member(Node, NodeList), noAbruptChange(Node) ),
      AffAndEffStreams) ),
```

```
    subtract(NodeList, AffAndEffStreams,
      TransformCentre).

inputOutputSimilar(Node) :-
    dataFlowBetween(_, Node, Input),
    dataFlowBetween(Node, _, Output),
    removePrefix(Input, KeywordOfInput),
    removePrefix(Output, KeywordOfOutput),
    KeywordOfInput = KeywordOfOutput.

noAbruptChange(Node) :-
    verbOf(Node, Verb),
    not isAbruptChange(Verb).
```

Here the predicate `removePrefix` removes the lower case letters from the identifier of `Input`, starting from the left to the right, until an upper case is encountered. The remaining string of characters is stored in `KeywordOfInput`. The predicate `verbOf` extracts the `Verb` from the identifier of `Node`. The predicate `isAbruptChange(Verb)` will hold if the Verb causes abrupt changes between its input and output data. Information on abrupt changes is stored in advance in a knowledge base of the system. Users may extend the knowledge base should the needs arise.

(iii) The predicate `convertTransformCentre` creates a sub-structure-chart for `TransformCentre`, with `TransformRoot` as the root. The identifier for `TransformRoot` is constructed by putting "do_" in front of the original identifier for the first node in the transform centre.

```
convertTransformCentre(TransformCentre,
    TransformRoot) :-
  member(FirstTransformNode, TransformCentre),
  not( member(Node2, TransformCentre),
       dataFlowBetween(Node2, FirstTransformNode, _) ),
  concat('do_', FirstTransformNode, TransformRoot),
  assertz(isModule(TransformRoot)),
  buildSubchart(TransformRoot, TransformCentre).

buildSubchart(_, []).

buildSubchart(Root, [Node|NodeList]) :-
  assertz(isModule(Node)),
  ( dataFlowBetween(_, Node, Data),
    [! assertz(couplingBetween(Root, Node, Data)) !],
    fail
    ; true ),
  ( dataFlowBetween(Node, _, Data2),
    [! assertz(couplingBetween(Node, Root, Data2)) !],
    fail
    ; true ),
  buildSubchart(Root, NodeList),
  assertz(structureOf(Root, sequence,
    [Node|NodeList])).
```

(iv) The predicate `convertAfferent` converts each afferent branch into a sub-structure-chart, and adds the data couples from its `Subroot` to `Node` and those from `Node` to `TransformRoot`.

```
convertAfferent(Node, NodeList, TransformRoot,
    TransformCentre) :-
  ( member(Node2, NodeList) ; not isBubble(Node2) ),
  [! not member(Node2, TransformCentre),
      dataFlowBetween(Node2, Node3, Data),
      member(Node3, TransformCentre),
      concat('get_', Data, Subroot),
      assertz(isModule(Subroot)),
      assertz(couplingBetween(Subroot, Node, Data)),
      assertz(couplingBetween(Node, TransformRoot,
        Data)),
      convertOneAfferentBranch(Node2, Data,
        NodeList) !],
   fail
   ; true.

 convertOneAfferentBranch(Node, Data, NodeList) :-
   concat('get_', Data, Subroot),
   ( isBubble(Node, _), !,
     assertz(isModule(Node)),
     dataFlowBetween(Node4, Node, Data4),
     ( member(Node4, NodeList)
       ; not isBubble(Node4, _) ),
     concat('get_', Data4, NewSubroot),
     assertz(isModule(NewSubroot)),
     assertz(couplingBetween(NewSubroot, Subroot,
       Data4)),
     assertz(couplingBetween(Subroot, Node, Data4)),
     assertz(couplingBetween(Node, Subroot, Data)),
     assertz(structureOf(Subroot, sequence,
       [NewSubroot, Node])),
     convertOneAfferentBranch(Node4, Data4, NodeList)
     ;
     % Node is source or file:
     assertz(couplingBetween(Node, Subroot, Data)) ).
```

(v) The predicate `convertEfferent` is similar to `convertAfferent`.

(vi) The predicate `addStructure` completes the structure chart by hanging to `Node` all the modules which have coupling with it. In other words, the afferent modules, the transform centre and the efferent modules will be hung under `Node`:

```
addStructure(Node) :-
  findall(Node2,
    ( ( couplingBetween(Node, Node2, Data)
        ; couplingBetween(Node2, Node, Data) ),
      not parentOf(Node, Node2) ),
    ChildNodes),
  assertz(structureOf(Node, sequence, ChildNodes)).
```

where `parentOf(Node, Node2)` indicates that `Node2` is the parent of `Node`.

Figure 3 shows the result of each step when `transformAnalysis` is applied to the data flow diagram of Figure 2(b). We assume in this illustration that each of the nodes and data items have been given meaningful names in order to facilitate the selection of the transform centre.

Transaction Analysis

Transaction analysis includes two main phases, `hideTransactCentres` and `expandTransactCentres`. Given a `NodeList`, the predicate `hideTransactCentres` will:

(a) Find a `FirstNode`, defined as a node in a transaction centre which inspects the type of each transaction and routes it to the corresponding branch for processing.

(b) Find all the nodes which are linked immediately after the `FirstNode`, and put them in `NextNodeList`.

(c) If all the transaction paths finally merge into a single node, we call it the `MergeNode`.

(d) Find all the transaction nodes in an individual branch between (but excluding) the `FirstNode` and `MergeNode`, and put them in a list `TransactBranch1`. Move the `NextNode` to the head of `TransactBranch1` to form `TransactBranch`.

(e) Collect all the `TransactBranch`es together to form `TransactBranchSubList`. Add the `FirstNode` to the head of `TransactBranchSubList` to form `TransactBranchList`.

(f) Insert the fact `transactBranches(TransactBranchList)` into the Prolog database for use in `expandTransactCentres` later.

(g) Replace the `FirstNode` and all the nodes in each `TransactBranch` by a single `NewFirstNode`. The identifier for the `NewFirstNode` is constructed by putting "`process_`" in front of the original identifier for `FirstNode`. Rename the resulting list of nodes as `TempNodeList`.

(h) Recursively call `hideTransactCentres` for `TempNodeList`, until no more `FirstNode` exists.

(i) Assign the final `TempNodeList` to `NewNodeList`.

The following shows the result of each step when transaction analysis is applied to the data flow diagram of Figure 2(a′).

(a) `FirstNode = n2b.`

(b) `NextNodeList = [n2c, n2d].`

(c) `MergeNode = n2e.`

(d) There are two `TransactBranch`es, one is `[n2c]`, another is `[n2d]`.

(e)
```
TransactBranchSubList = [[n2c], [n2d]],
TransactBranchList = [n2b, [n2c], [n2d]].
```

(f) The Prolog database now contains the fact

```
transactBranches([n2b, [n2c], [n2d]]).
```

(g)
```
NewFirstNode = process_n2b,
TempNodeList = [n1a, n1b, n1c, n2a, process_n2b, n2e, n3a, n3b,
                n3c].
```

(h) The result of the recursive calls of `hideTransactCentres` are as follows:

(aa) `FirstNode = n3a.`

(bb) `NextNodeList = [n3b, n3c].`

(cc) `MergeNode = nil.`

(dd) There are two `TransactBranch`es, one is `[n3b]`, another is `[n3c]`.

(ee)
```
TransactBranchSubList = [[n3b], [n3c]],
TransactBranchList = [n3a, [n3b], [n3c]].
```

(ff) The Prolog database now contains the facts

```
transactBranches([n2b, [n2c], [n2d]]),
transactBranches([n3a, [n3b], [n3c]]).
```

(gg)
```
NewFirstNode = process_n3a,
TempNodeList = [n1a, n1b, n1c, n2a, process_n2b, n2e,
  process_n3a].
```

(hh) Since there is no more `FirstNode`, the recursive call stops here.

(i)
```
NewNodeList = [n1a, n1b, n1c, n2a, process_n2b, n2e,
  process_n3a].
```

The graphical equivalence is as shown in Figure 2(*b*).

The actual predicate for the above procedure is declared as follows:

```
hideTransactCentres(NodeList, TempNodeList, NewNodeList) :-
  firstNodeOf(NodeList, FirstNode),
  findall(Node,
    dataFlowBetween(FirstNode, Node, _),
    NextNodeList),
  findall(TransactBranch,
    transactBranchOf(NodeList, FirstNode, NextNodeList,
      TransactBranch),
    TransactBranchSubList),
  append([FirstNode], TransactBranchSubList,
    TransactBranchList),
  assertz(transactBranches(TransactBranchList)),
  replace(NodeList, FirstNode, NextNodeList, TempNodeList,
    TransactBranchList),
  hideTransactCentres(TempNodeList, TempNodeList2,
    NewNodeList)
  ;
  NewNodeList = TempNodeList.

transactBranchOf(NodeList, FirstNode, NextNodeList,
    TransactBranch) :-
  member(NextNode, NextNodeList),
  ifthenelse(
    mergeNodeOf(NodeList, NextNodeList, MergeNode),
    findall(Node1,
      inTransactBranch(NodeList, NextNode, Node1,
        MergeNode),
      TransactBranch1),
```

```
    findall(Node2,
      samePath(NodeList, NextNode, Node2),
      TransactBranch1)),
  subtract(TransactBranch1, [NextNode], TransactBranch2),
  append([NextNode], TransactBranch2, TransactBranch).
```

The predicate `firstNodeOf` is listed below. It verifies that any `Input` to the `FirstNode` must be routed to one and only one of its various `Output`s.

```
firstNodeOf(NodeList, FirstNode) :-
  member(FirstNode, NodeList),
  dataFlowBetween(_, FirstNode, Input),
  findall(Data,
    dataFlowBetween(FirstNode, _, Data),
    OutputList),
  transactData(Input, OutputList),
  mutuallyExclusive(OutputList).
```

Here the predicate `transactData` will hold if the output data flows in `OutputList` are components of `Input`. The predicate `mutuallyExclusive` will hold if and only if, for any two data flows in `OutputList`, one is not a component of the other, or a permutation of the other, or combination(s) of these. These results are determined through the analysis of the data dictionary, which is beyond the scope of the present paper and will not be discussed here.

The second main phase of transaction analysis, `expandTransactCentres`, use automatic backtracking to search for all the hidden transaction centres. For each hidden centre, we recursively invoke `expandEachBranch` to re-expand its `Branch`es, apply `transformAnalysis` to them, and hang the resulting subcharts below the structure chart from the previous phase, as shown in Figure 2(*d*). The predicate `expandTransactCentres` is declared as follows:

```
expandTransactCentres :-
  transactBranches([FirstNode|BranchList]),
  %
  % 'transactBranches' is a Prolog fact asserted
  % by the predicate 'hideTransactCentres'
  %
  expandEachBranch(FirstNode, BranchList),
  concat('process_', FirstNode, NewFirstNode),
  findall(Module,
    couplingBetween(NewFirstNode, Module, _),
    ModuleList),
  assertz(structureOf(NewFirstNode, selection,
    ModuleList)),
  fail
  ;
  abolish(transactBranches/1).

expandEachBranch(_, []).

expandEachBranch(FirstNode, [Branch|OtherBranches]) :-
  concat('process_', FirstNode, NewFirstNode),
  Branch = [NextNode|SubBranch],
  transformAnalysis(NextNode, SubBranch),
  dataFlowBetween(FirstNode, NextNode, Input),
```

```
tryAssertz(couplingBetween(NewFirstNode, NextNode,
  Input)),
findall(Output,
  ( lastNodeOf(Branch, NextNode, LastNode),
    dataFlowBetween(LastNode, _, Output) ),
  OutputList),
tryAssertz(couplingBetween(NextNode, NewFirstNode,
  OutputList)),
expandEachBranch(FirstNode, OtherBranches).
```

where the predicate `tryAssertz(fact)` puts the Prolog `fact` into the database in forward execution, but if backtracking is necessary, the `fact` will be removed automatically by the system.

EVALUATION OF STRUCTURE CHARTS

In this section, we illustrate how we apply Prolog predicates to review structure charts according to evaluation guidelines as recommended in DeMarco[3], Page-Jones[6] and Yourdon[7]. These reviews will help to determine whether the structured charts should be improved. If the result of an evaluation is not satisfactory, the system will backtrack automatically and use alternative conversion procedures to produce other structure charts for consideration. Examples of alternative procedures include promoting a boss instead of hiring a new boss[6], swapping the order of transform and transaction analyses, and expanding the nodes into more children than the first-cut attempt.

We would like to point out we have not yet exhausted the many evaluation criteria as recommended in the literature. We are currently studying the respective Prolog predicates for implementing additional criteria, such as fan-in, factoring, decision-splitting, and initializing and terminating modules. The results obtained so far appear to be promising. Furthermore, we find that we can easily incorporate new guidelines into the Prolog system in an incremental manner.

Coupling

Coupling is a measure of the inter-dependence among different modules. The recommendation in structured methodology is that modules should exhibit a loose coupling. In other words, they should be independent of one another as far as possible. Myers[16] and Stevens *et al*.[17] have identified five levels of coupling between two modules. They are listed in order of preference as follows:

(a) *Data coupling:* Modules communicate through atomic data items.
(b) *Stamp coupling:* Modules communicate through composite data items, or records.
(c) *Control coupling:* Modules communicate through control flags.
(d) *Common coupling:* Modules share common data.
(e) *Content coupling:* A module refers to or changes the inside of another module.

The following predicates help to detect the preferred levels of couplings. The absence of such couplings between two modules will be highlighted as anomalies.

```
dataCoupling(Module1, Module2) :-
  ( couplingBetween(Module1, Module2, Data)
    ; couplingBetween(Module2, Module1, Data) ),
  isData(Data, atomic).
```

```
stampCoupling(Module1, Module2) :-
  ( couplingBetween(Module1, Module2, Data)
    ; couplingBetween(Module2, Module1, Data) ),
  isData(Data, record).

controlCoupling(Module1, Module2) :-
  ( couplingBetween(Module1, Module2, Data)
    ; couplingBetween(Module2, Module1, Data) ),
  isData(Data, control).
```

Cohesion

Cohesion, also known as cohesiveness, is a measure of the strength of association of components within a module. The recommendation in structured methodology is that the components should exhibit a high cohesion among themselves. In other words, they should be inter-related as much as possible. Myers[16] and Stevens *et al*.[17] have identified seven levels of cohesion within a module. They are listed in order of preference as follows:

(a) *Functional cohesion:* The module performs a single inseparable function.
(b) *Sequential cohesion:* A later component of the module makes use of the data produced in an earlier component.
(c) *Communicational cohesion:* Different components of the module refer to data items in the same file, but the order of the components is not important.
(d) *Procedural cohesion:* The components in the module are related by program control structures such as selection or iteration.
(e) *Temporal cohesion:* The components should be done roughly at the same time, such as the end of every month.
(f) *Logical cohesion:* The module tries to accommodate general procedures which may be slightly different depending on the input data or other logical conditions, so that exceptional treatments have to be added here and there.
(g) *Coincidental cohesion:* The components are grouped arbitrarily for no reason.

Tsai and Ridge[11] find it difficult to determine the levels of cohesion of individual modules, and suggest a system which prompts the user for advice. On the other hand, we note that it is not usual as a matter of practice to determine the exact cohesion level for every module in a structure chart. There is scepticism among structured design experts as to whether precise numerical values assigned to cohesion levels have any meaning in real life. Designers would like only to identify those modules which have low cohesiveness and re-arrange them wherever possible. To this aim, our system highlights the modules whose internal components do not share common input/output data, through the analysis of the mini-specifications[18]. This is based on the observation that data must be passed among the components of a module with functional, sequential or communicational cohesion. The analysis of mini-specifications is beyond the scope of the present paper and will not be discussed in detail here.

Morphology

Structure charts should also be evaluated according to their morphology[7], or shapes. One of the criteria is fanout, which is the number of children for a module in the chart. It is recommended that structure charts should have low fanouts. Thus if an afferent module has three or more children, with two or more of them being transform modules, then it is advisable to group some of the children together to form a subtree, headed by an afferent submodule. Similarly for an efferent module. In this way, a module high up in a structure chart does not need to be fully responsible for the behaviour of all its subordinates, but can delegate some of its supervisory function to ‘‘middle management’’. For

example, a chart with an afferent module followed by a transform module, such as `Module3` and `Module4` in Figure 4, would be preferred to that with a flat sequence of transform modules, such as `Module2` and `Module4` in Figure 5. We detect morphological anomaly by means of a `morphAnomaly` predicate.

```
morphAnomaly(Parent) :-
  moduleType(Parent, afferent),
  structureOf(Parent, sequence, [Module1|OtherModules]),
  moduleType(Module1, afferent),
  findall(Module2,
    ( member(Module2, OtherModules),
      moduleType(Module2, transform) ),
    TransformModules),
  length(TransformModules) > 1.

morphAnomaly(Parent) :-
  moduleType(Parent, efferent),
  structureOf(Parent, sequence, ChildModules),
  append(OtherModules, Module1, ChildModules),
  moduleType(Module1, efferent),
  findall(Module2,
    ( member(Module2, OtherModules),
      moduleType(Module2, transform) ),
    TransformModules),
  length(TransformModules) > 1.
```

The predicate `moduleType` above determines whether a given `Module` is `afferent`, `efferent`, or a `transform` module, thus:

```
moduleType(Module, afferent) :-
  isModule(Module),
  parentOf(Module, Parent),
  findall(Data, couplingBetween(Parent, Module, Data),
    []),
  ( childrenOf(Module, [])
    ;
    parentOf(Child, Module),
    findall(Data2,
      couplingBetween(Module, Parent, Data2),
      DataItems),
    findall(Data3,
      couplingBetween(Child, Module, Data3),
      DataItems) ).

moduleType(Module, efferent) :-
  isModule(Module),
  parentOf(Module, Parent),
  findall(Data,
    couplingBetween(Module, Parent, Data),
    []),
  ( childrenOf(Module, [])
    ;
    parentOf(Child, Module),
```

```
    findall(Data2,
      couplingBetween(Parent, Module, Data2),
      DataItems),
    findall(Data3,
      couplingBetween(Module, Child, Data3),
      DataItems) ).

moduleType(Module, transform) :-
  isModule(Module),
  parentOf(Module, Parent),
  findall(Data,
    couplingBetween(Parent, Module, Data),
    [_|_]),
  findall(Data2,
    couplingBetween(Module, Parent, Data2),
    [_|_]).
```

The detection of further morphological anomalies can be defined by adding more rules. We can do this in an incremental manner without being bothered by the detection procedure.

Tramp

A tramp is "an item of data that, although irrelevant to the function of a given module, has to pass through that module in order to reach another module" [6]. An example of a tramp is shown in Figure 6, where `master` is passed through `getTrans` and `getValidTrans` but is irrelevant to both of these modules.

In order to detect irrelevance, we can start off by defining a simple Prolog predicate, and augment the definition incrementally when more knowledge is available. For instance, we may try to specify irrelevance using a rather simplistic predicate

```
irrelevant(Data, Module) :-
  couplingBetween(_, Module, Data),
  couplingBetween(Module, _, Data).
```

which checks whether a piece of `Data` passes in and out of a `Module` directly.

Let us take a look at Figure 4. According to the simplistic predicate, the data item `B` would be irrelevant to the module `Parent`. We should not consider it as a tramp, however, because it is recommended in structured methodology that two modules on the same level should pass data through their parent rather than directly from one to another, the latter process being known as a pathological connection[7]. We should augment the original definition of irrelevance with this extra knowledge. The improved predicate will also check whether both the source module `M1` and the destination module `M2` of `Data` are children of the `Module` concerned.

```
irrelevant(Data, Module) :-
  couplingBetween(M1, Module, Data),
  couplingBetween(Module, M2, Data),
  not( parentOf(M1, Module),
       parentOf(M2, Module) ).
```

Let us take a further look at Figure 4. According to the predicate, the data item `B` would be irrelevant to `Module3`. But we still do not wish to consider it as a tramp because this design follows

the guideline on morphology, as discussed in the previous section. `Module3` helps to hide the modules working on `A` from modules higher up in the chart. We should augment further the definition of irrelevance. The new predicate will also check whether the passage of data is according to the recommended morphology.

```
irrelevant(Data, Module) :-
  couplingBetween(M1, Module, Data),
  couplingBetween(Module, M2, Data),
  not( parentOf(M1, Module),
       parentOf(M2, Module) ),
  not( parentOf(M1, Module),
       parentOf(Module, M2),
       parentOf(M3, M2),
       couplingBetween(M2, M3, Data) ),
  not( parentOf(M2, Module),
       parentOf(Module, M1),
       parentOf(M4, M1),
       couplingBetween(M4, M1, Data) ).
```

CONCLUSION

In this paper, we have summarized our experience in the use of Prolog to model and evaluate structure charts according to standard guidelines in structured design. We have found that we can construct first-cut structure charts automatically from data flow diagrams using transform and transaction analyses, evaluate them using recommended criteria such as coupling, cohesion, morphology and tramp, and improve on the resulting structure charts by means of automatic backtracking. Prolog has been found to be very useful because:

(a) Although various recommendations have been made on the criteria for good and poor structure charts, there is no prescribed procedure to detect them. But we can simply formulate the criteria as Prolog predicates, and leave the detection procedure to the built-in inference system of Prolog.

(b) Most of the recommendations on the construction and evaluation of structure charts are based on the practical experience of individuals. They are described informally in the literature, and new recommendations are added in the light of further experience on the method. It is very difficult to implement such partial recommendations in a conventional imperative programming language and to add further recommendations without significantly disturbing the original program. But such recommendations can be specified easily as Prolog predicates in an incremental manner.

(c) The guidelines given in structured methodology, such as those for converting a data flow diagram into a structure chart, do not give a unique result. The processes involved are non-deterministic in nature. We are supposed to make a first-cut attempt, and evaluate it based on a set of recommended criteria. In case the design is not satisfactory, backtracking must be employed to find an improved solution. If we use an imperative programming language to implement the methodology, we shall have to specify the backtracking strategy explicitly. This, however, can be done automatically in Prolog and is transparent to the implementor.

ACKNOWLEDGEMENTS

We are especially grateful to Raymond Chan of IBM, H.Y. Chen of Jinan University and H.L. Xie of the University of Pennsylvania for their invaluable contributions in the implementation of the system.

REFERENCES

[1] C.A. Richter, An assessment of structured analysis and structured design, ACM SIGSOFT Software Engineering Notes 11 (4) (1986) 41−45.

[2] 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 11, Cambridge University Press, Cambridge, UK, 1991.

[3] T. DeMarco, Structured Analysis and System Specification, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ, 1979.

[4] L.J. Peters, Advanced Structured Analysis and Design, Prentice Hall, Englewood Cliffs, NJ, 1987.

[5] E. Yourdon, Modern Structured Analysis, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ, 1989.

[6] M. Page-Jones, The Practical Guide to Structured Systems Design, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ, 1988.

[7] 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, NJ, 1979.

[8] T. Goble, Structured Systems Analysis through Prolog, Prentice Hall, Englewood Cliffs, NJ, 1989.

[9] R.A. Kowalski, Software engineering and artificial intelligence in new generation computing, Future Generation Computer Systems 1 (1) (1984) 39−49.

[10] T.W.G. Docker, SAME: a structured analysis tool and its implementation in Prolog, in: Logic Programming: Proceedings of the 5th International Conference and Symposium, R.A. Kowalski and K.A. Bowen (eds.), MIT Press, Cambridge, MA, 1988, pp. 82−95.

[11] J.J.-P. Tsai and J.C. Ridge, Intelligent support for specifications transformation, IEEE Software 5 (6) (1988) 28−35.

[12] I. Bratko, Prolog Programming for Artificial Intelligence, Addison-Wesley, Wokingham, UK, 1990.

[13] W.F. Clocksin and C.S. Mellish, Programming in Prolog, 4th Edition, Springer, Berlin, Germany, 1994.

[14] L. Sterling and E.Y. Shapiro, The Art of Prolog: Advanced Programming Techniques, MIT Press, Cambridge, MA, 1986.

[15] R.A. Kowalski, Logic for Problem Solving, Elsevier, Amsterdam, The Netherlands, 1979.

[16] G.J. Myers, Composite/*Structured Design, Van Nostrand Reinhold, New York, NY, 1978.*

[17] W.P. Stevens, G.J. Myers, and L.L. Constantine, Structured design, in: Advanced System Development / *Feasibility Techniques, J.D. Couger, M.A. Colter, and R.W.* Knapp (eds.), John Wiley, New York, NY, 1982, pp. 164−185.

[18] T.H. Tse, T.Y. Chen, F.T. Chan, H.Y. Chen, and H.L. Xie, The application of Prolog to structured design, Software: Practice and Experience 24 (7) (1994) 659−676.

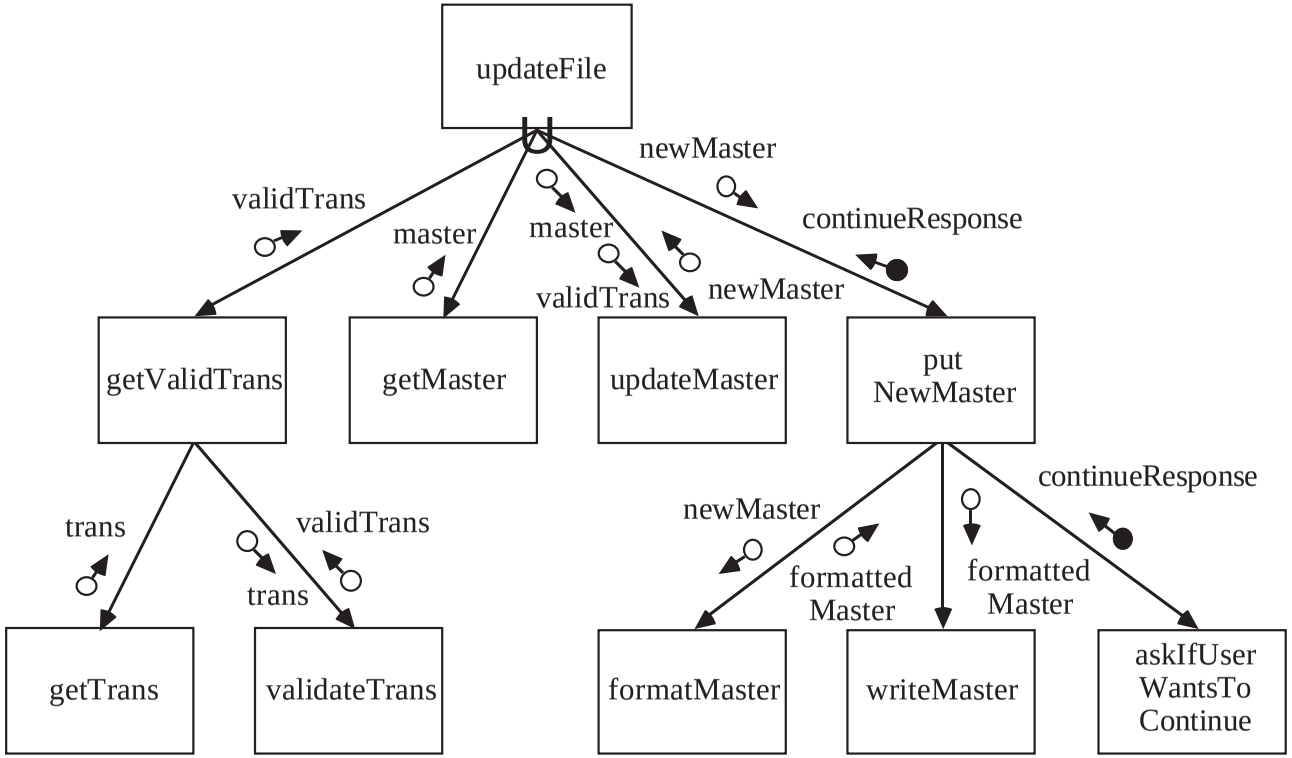


Figure 1 Sample Structure Chart

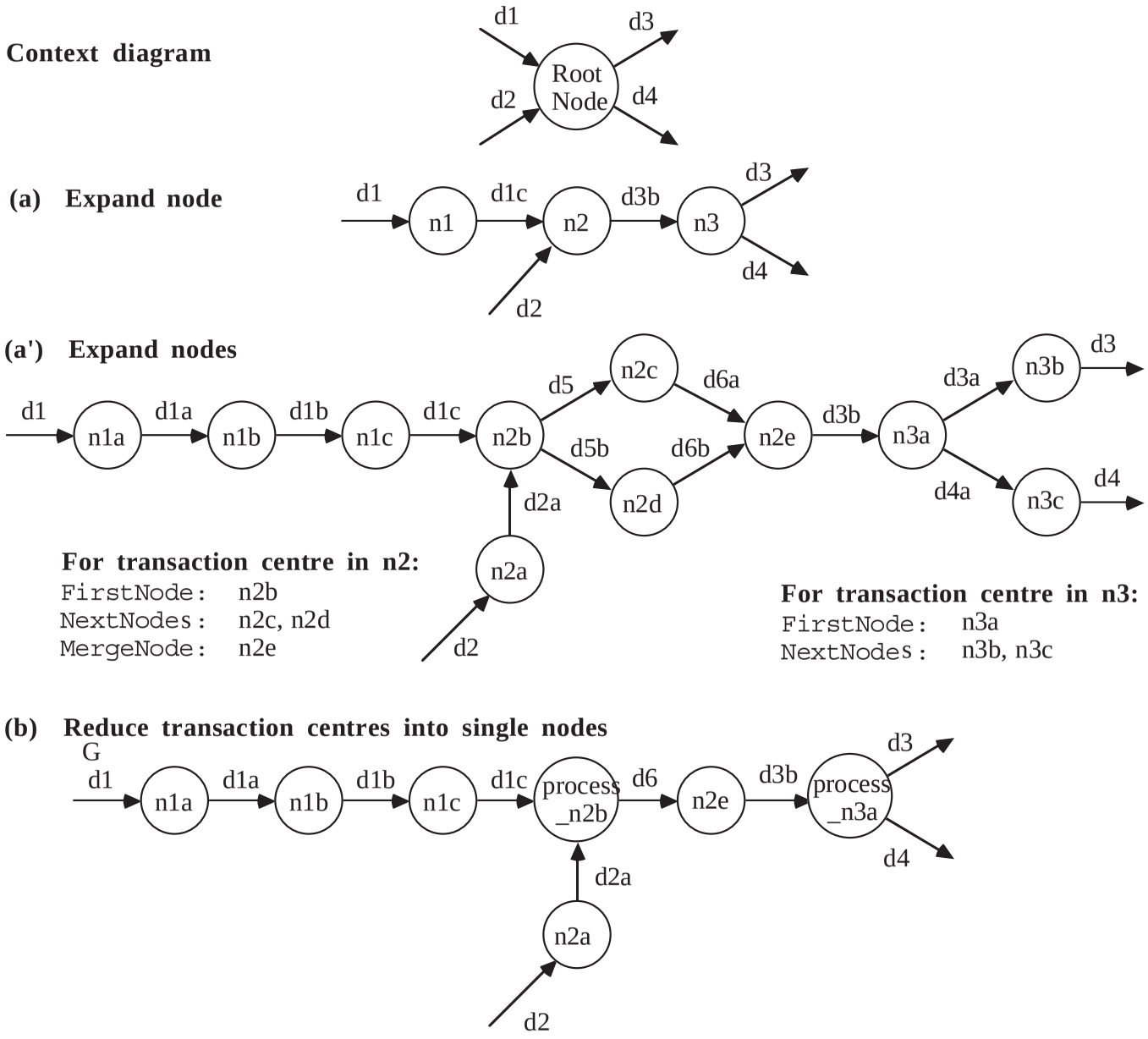
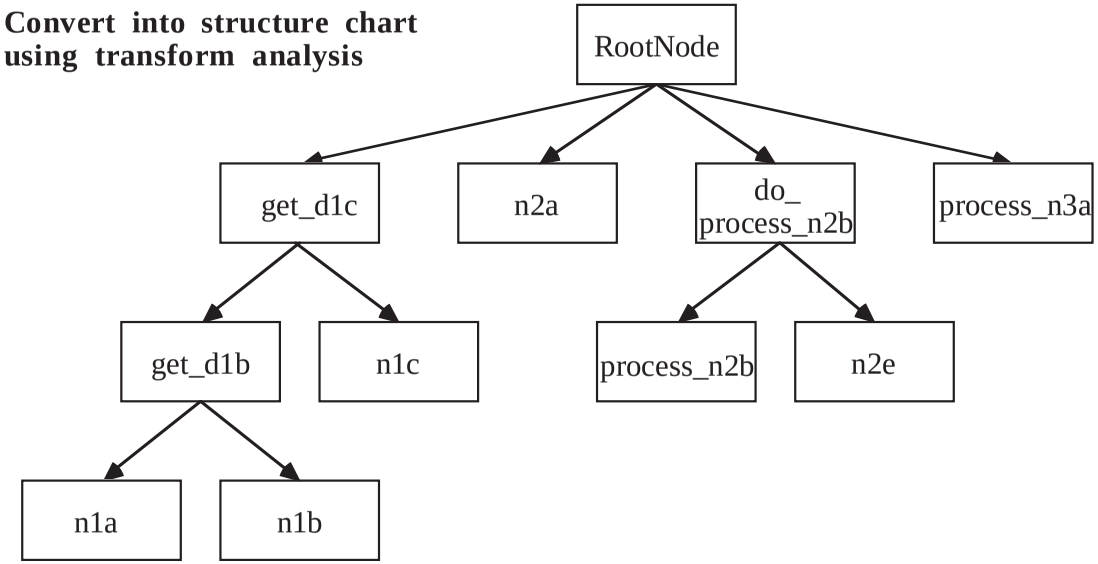


Figure 2 Conversion of Data Flow Diagram into First-Cut Structure Chart

(c) Convert into structure chart using transform analysis



(d) Re-expand hidden transaction centres, recursively perform the procedure, and hang the subcharts below the original structure chart

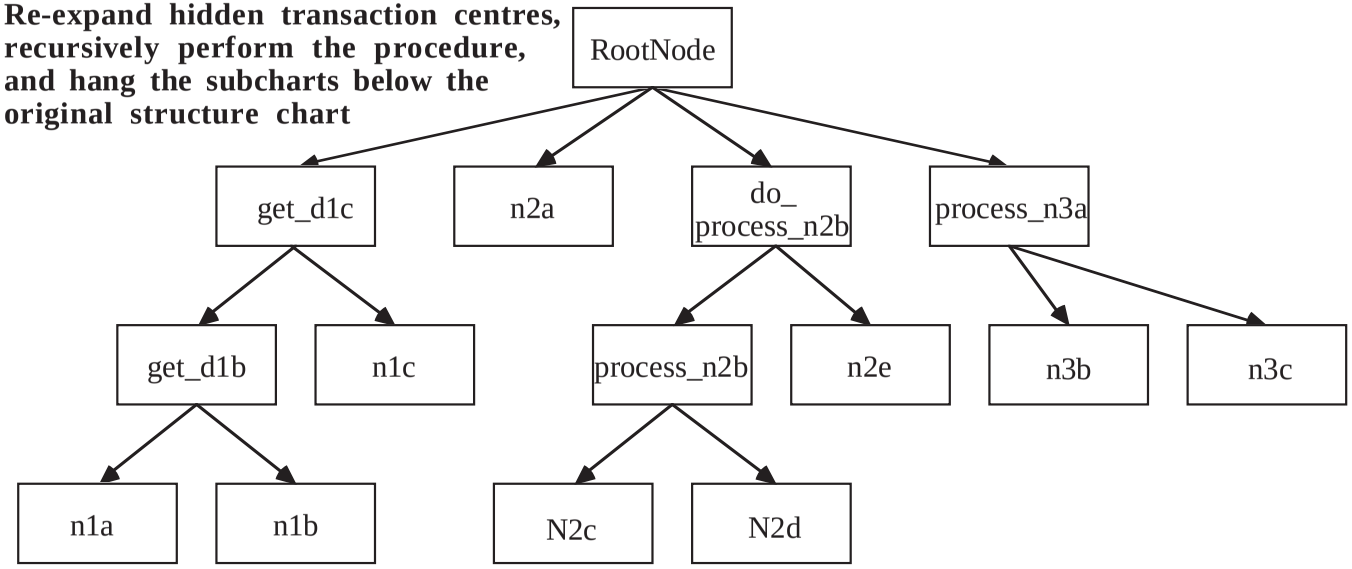


Figure 2 Conversion of Data Flow Diagram into First-Cut Structure Chart (Continued)

PREDICATE: **RESULT:**

`newModule`

RootNode is created in the Prolog database.

`transformCentreOf`

The `TransformCentre` includes `process_n2b` and `n2e` .

`convertTransformCentre`

RootNode

(New additions to the Prolog database are shown in bold type.)

do_ process_n2b

process_n2b **n2e**

`convertAfferent`

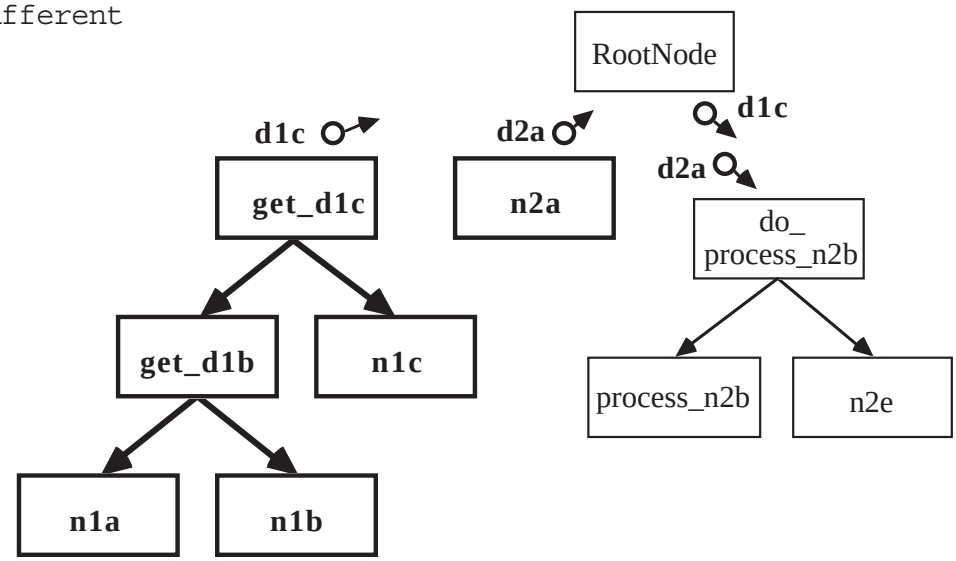
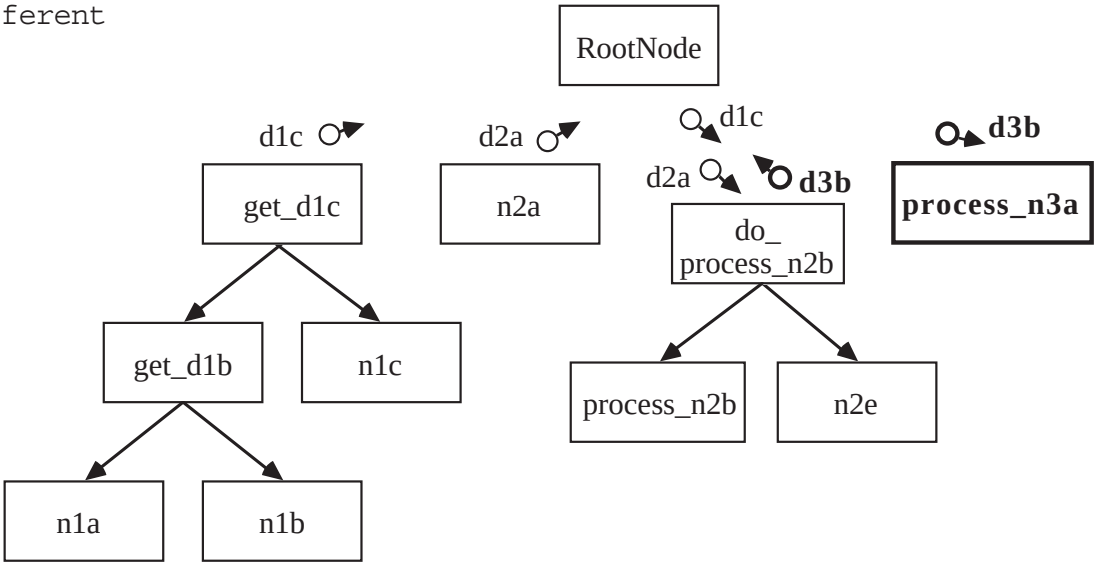


Figure 3 Example on the Running of `TransformAnalysis`

PREDICATE: **RESULT:**

`convertEfferent`



`addStructure`

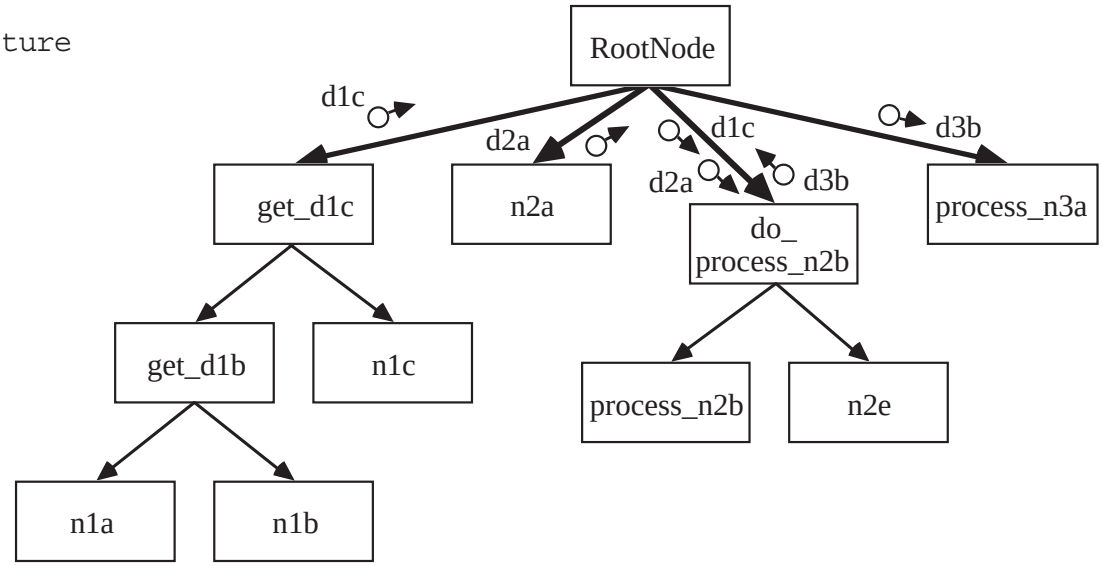


Figure 3 Example on the Running of `TransformAnalysis` (continued)

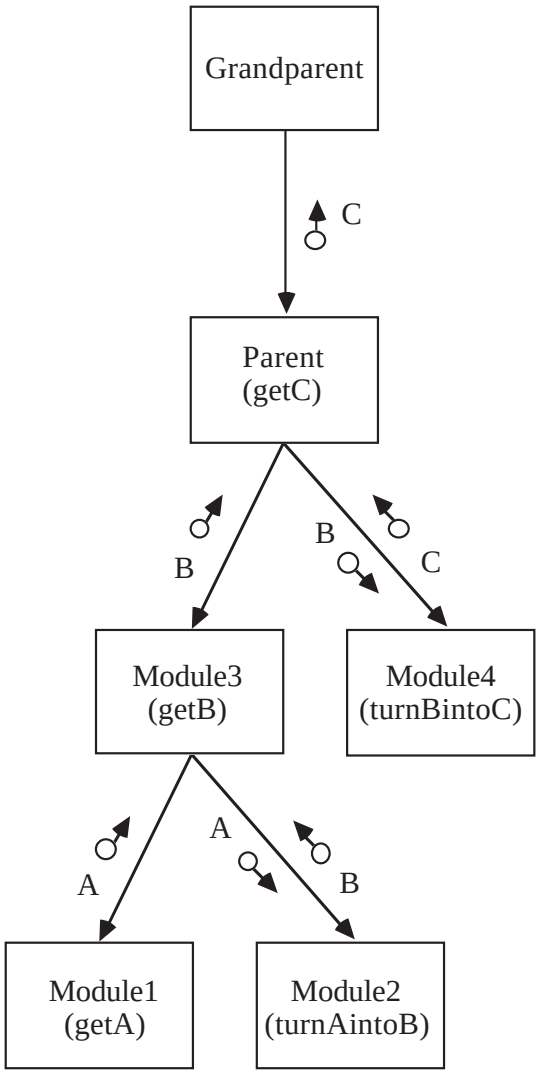


Figure 4 Get Modules According to Recommended Morphology

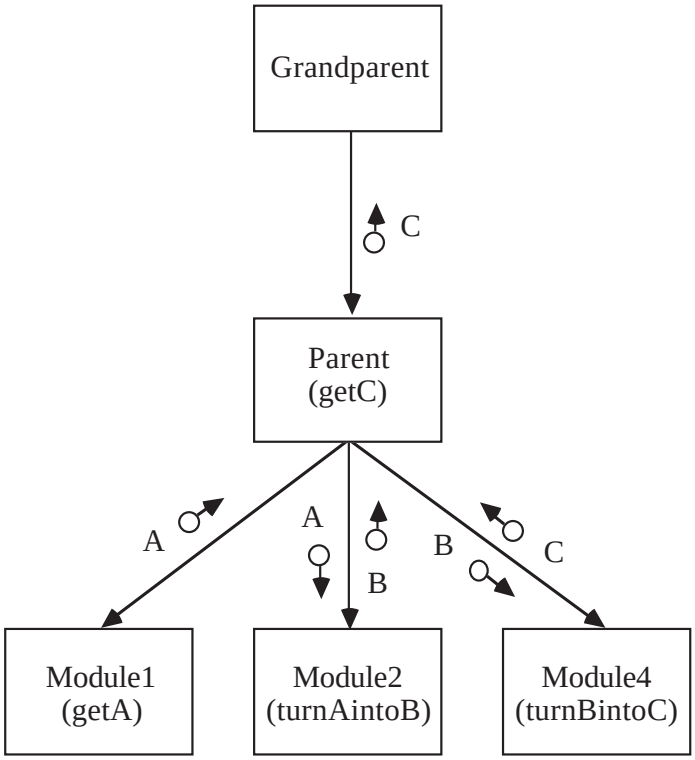


Figure 5 Example of Morphological Anomaly

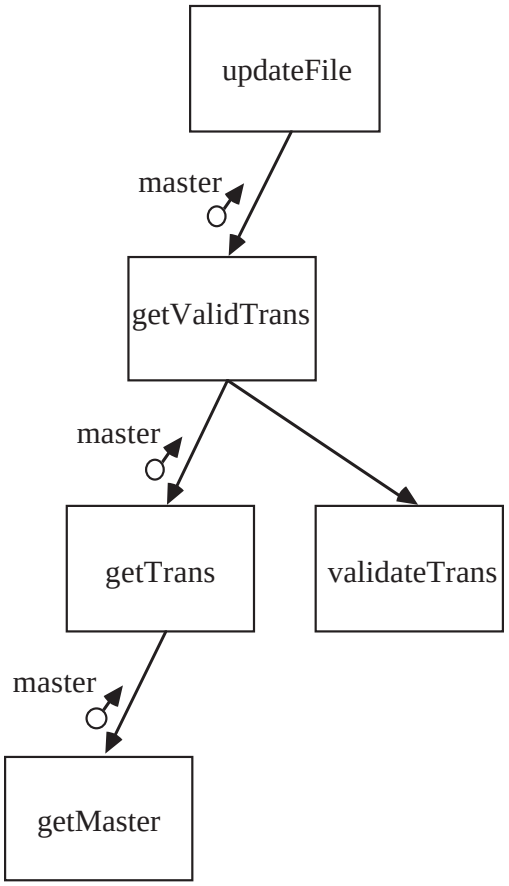


Figure 6 Example of Tramp Data