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)

Evaluation of Structure Charts: a Logic Programming Approach*

T.Y. CHEN
Department of Computer Science and Software Engineering
University of Melbourne

C.S. KWOK, W.H. TANG
Centre of Computing Services and Telecommunications
The Hong Kong University of Science and Technology

and

T.H. TSE
Department of Computer Science
The University of Hong Kong

ABSTRACT

We apply the techniques of logic programming to evaluate structure charts. We find that structure charts can be represented naturally in Prolog, and useful information can be derived in a straightforward manner. Standard techniques in the evaluation of structure charts can be formalized, and a few previous problems can be solved easily.

1. Introduction

Structured systems development methodologies have been recognized as some of the most popular methods in software engineering [1]. They involve problem solving techniques and decision making processes which can only be carried out by experienced software engineers. Although a number of CASE tools on structured methodologies have already been proposed, they are not totally satisfactory since it is extremely difficult to have human expertise formalized in conventional terms and implemented by algorithmic programs.

A project has been set up to apply the techniques of logic programming to structured methodologies. This is a relatively unexplored direction. An exception is the work by Kowalski [10], who suggests that we can regard data flow diagrams merely as syntactic sugar for logic programs. Another is the work of Docker [3], who uses Prolog to implement a tool for ''specifying and exercising'' data flow diagrams. A recent paper by Tsai and Ridge [14] reported

* This project is supported in part by a University and Polytechnic Grants Committee Research Grant, a Research and Conference Grant of the University of Hong Kong and a Travelling Grant of the Hong Kong University of Science and Technology.

that quite a few problems have been encountered in an attempt to use expert systems for the evaluation of structure charts. In this paper, we summarize our experience in applying logic programming techniques to evaluate these charts. We find that logic programming is a useful tool for this purpose.

In Sections 2 and 3 of the paper, we shall provide some background information on structured methodologies and logic programming. In Section 4.1, we shall show how to represent structure charts naturally in Prolog, and how to derive useful information from such representation. In Section 4.2, we shall illustrate through examples how we can formalize standard techniques in the evaluation of structure charts, including one or two problem areas suggested by other authors. We hope the research will provide further insight for software engineers into structured methodologies, and guidelines for implementors of CASE tools.

2. Structured Systems Development

Structured methodologies are widely accepted by practising systems developers because of the top-down approach to systems problems and the graphical nature of the representations. A complex systems specification can be decomposed into a modular and hierarchical structure which is easily comprehensible. They enable practitioners to work out blueprints of target systems and to communicate with users easily.

Different structured representations are found to be suitable for different development situations depending on the environment, emphasis and stage of development [1, 11, 13]. In other words, we need more than one of these models during the development process of a typical system. They are converted from one form to another as the needs arise. For example, data flow diagrams are used for systems analysis and structure charts for systems design [2, 12, 18]. Specification details are expressed in a textual form such as structured English or decision tables. The multi-model approach allows the most appropriate representation to be used in a given situation. One distinct feature of all these models is that they support multiple levels of abstraction and refinement. Hence the analysis and modification of a specification can be handled relatively easily.

If we provide practitioners with CASE tools to convert one structured representation to another, the efficiency of systems development can be greatly improved. Unfortunately, the transformation process is not a straightforward matter based on well-defined algorithms. The heuristics are often unquantifiable and, unlike simple rules of thumb, involve many decision factors which can easily be overlooked by less experienced systems designers. Hence the qualities of different designs vary according to the expertise of the practitioners involved. We believe that design constraints and heuristics can be handled more easily within the logic programming paradigm. An expert system should be built to simulate the knowledge and experience of human practitioners.

3. Logic Programming

Logic programming is the study of predicate logic as a programming language [7]. In addition to the simplicity and elegance of its semantics [17], logic programming supports a new programming paradigm, namely declarative programming. An algorithm consists of two parts: a logic component and a control component [8]. The logic component specifies the problem to be solved, and the control component specifies the mechanism to solve the problem. In pure logic programming, the control component is left entirely to the system, while the programmer is only responsible for the logic component. In other words, the programmer just needs to specify *what*

is required and leaves *how* it is done to the system. Thus the programmer is liberated from such problems as overcoming structural complexity and the implicitness of knowledge [4].

In the following paragraphs, we summarize some special features of logic programming which are relevant to our study. The language to be used is Prolog (*Pro*gramming in *log*ic), which is regarded as the most successful and practical logic programming language. As an example of a simple Prolog program, consider an `append` procedure for concatenating lists:

```
append([], List1, List1).
append([Head|List1], List2, [Head|List3]) :-
   append(List1, List2, List3).
```

Here "`:-`" is the standard Prolog symbol for "if", "`[]`" denotes an empty list, and "`[Head|List1]`" denotes a non-empty list whose first element is `Head` and whose other elements form another list called `List1`. Thus the first clause of the procedure means that concatenating `[]` with `List1` will end up with `List1` itself. The second clause means that, if concatenating `List1` with `List2` gives `List3`, then concatenating the list `[Head|List1]` with `List2` will give the list `[Head|List3]`.

A procedure in Prolog, such as `append` above, is commonly known as a *predicate*. Its identifier begins with a lower case letter. It is more general than a procedure in a conventional programming language, as we shall see in Section (*a*) below. On the other hand, a variable identifier such as `List1` begins with an upper case letter. A variable in Prolog denotes an element in a domain. Once a variable has been assigned a value, it cannot take a new value. In order to avoid confusion with a variable in a conventional programming language, we call it a *logical variable*.

(*a*) *Variation of Input/Output Parameters*

A parameter in a Prolog procedure can be used either as an input parameter or as an output parameter depending on the context. As a result, a given procedure may have multiple usage. For example, we can use the `append` procedure to concatenate two given lists by specifying a *goal* such as

```
?-  append([i, n], [f, o], Result).
```

to Prolog. We may also use it to split a given list into two by issuing a goal such as

```
?-  append(List1, List2, [i, n, f, o]).
```

In the latter case, there will be more than one solution:

```
List1 = [],        List2 = [i, n, f, o];
List1 = [i],       List2 = [n, f, o];
List1 = [i, n],    List2 = [f, o];
List1 = [i, n, f], List2 = [o];
List1 = [i, n, f, o], List2 = [].
```

We refer to such a case as *nondeterminism*.

(*b*) *Backtracking*

Prolog is a sequential logic programming language. When it encounters nondeterminism, it explores the solutions one by one through a mechanism known as *backtracking*. When a goal succeeds, it carries on to try the next potential solution. When a goal fails, it backtracks to an earlier point and tries an alternative path.

In structured methodology, such as when we convert a data flow diagram into a structure chart, the set of guidelines given by most authors will not guarantee a unique result. The whole process is nondeterministic by nature. We are supposed to make a first-cut design, and evaluate it based on a set of criteria. In case the first-cut design is unsatisfactory, backtracking must be employed to find an improved solution. If we use a conventional programming language to implement the methodology, we shall have to specify the backtracking strategy explicitly. If we use Prolog, however, this can be done automatically and is transparent to the implementor.

(*c*) *Incremental Addition of Knowledge*

The application of logic programming languages to implement rule-based expert systems has been a popular topic. The following example is adapted from Kowalski [10]. Suppose in a medical expert system, we have a rule stating that a patient should take some treatment if (*i*) she has a complaint which the treatment will suppress and (*ii*) the treatment is not unsuitable for her. This heuristic rule used by a human expert can be captured very naturally in Prolog, thus:

```
should_take(Patient, Treatment) :-
   has_complaint(Patient, Complaint),
   suppresses(Treatment, Complaint),
   not unsuitable(Treatment, Patient).
```

The `unsuitable` predicate can be defined in a similar way:

```
unsuitable(Treatment, Patient) :-
   aggravates(Treatment, Condition),
   has_condition(Patient, Condition).
```

The properties of various treatments can be stored in a database of facts:

```
suppresses(aspirin, inflammation).
suppresses(aspirin, pain).
aggravates(aspirin, pepticUlcer).
aggravates(lomotil, impairedLiverFunction).
. . .
```

Incorporation of additional knowledge and expertise can be done incrementally. For example, suppose we learn later on that a treatment is unsuitable to a patient if she is allergic to it. We do not need to alter any of the predicates in the original program, but simply define an additional rule

```
unsuitable(Treatment, Patient) :-
   allergic(Patient, Treatment).
```

In other words, we can start with a prototype with limited heuristics, and then incrementally

increase the ‘‘heuristic power’’ as we know more about the problem domain.

4. Evaluation of Structure Charts

In the evaluation of structure charts, we are concerned with two issues: how to represent a structure chart and how to formalize the criteria for evaluating the chart. The first issue is a study of the knowledge representation. The second issue is slightly more complex because most of the criteria suggested in the literature are rather vague and imprecise.

4.1 Representation 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 the pros and cons of term-based representation and relation-based representation here (see, for example, Kowalski's text [9]). Our contention is that a structure chart is usually large, and hence it would be rather cumbersome to encode the chart as one huge term [15].

We need to know (*a*) all the modules in the chart, (*b*) all the data items in the chart, (*c*) the organization of the modules, and (*d*) all the communications among modules. As an example for illustration, consider Figure 1 which has been adapted from Page-Jones [12]. It shows part of an interactive system for updating a file. First of all, we identify all the modules in the system by defining the `is_module` predicate:

```
is_module(updateFile).
is_module(getValidTrans).
is_module(getTrans).
is_module(validateTrans).
is_module(getMaster).
. . .
```

Next, we identify all the data items through the `is_data` predicate. The first argument of `is_data` is the name of a data item. The second argument tells us whether it is a simple item (`atomic`), a composite item (`record`) or a flag (`control`).

```
is_data(trans, record).
is_data(validTrans, record).
is_data(continueResponse, control).
. . .
```

Then we specify the organization of the modules. The predicate `structure(ParentModule, Type, ChildModules)` describes the connection between a `ParentModule` and a list of `ChildModules`. The `Type` of connection can be sequence, selection or iteration, which are denoted by the constants `seq`, `sel` or `itr` respectively.

```
structure(updateFile, itr, [getValidTrans, getMaster,
   updateMaster, putNewMaster]).
structure(getValidTrans, seq, [getTrans,
   validateTrans]).
structure(putNewMaster, seq, [formatMaster, writeMaster,
   askIfUserWantsToContinue]).
```

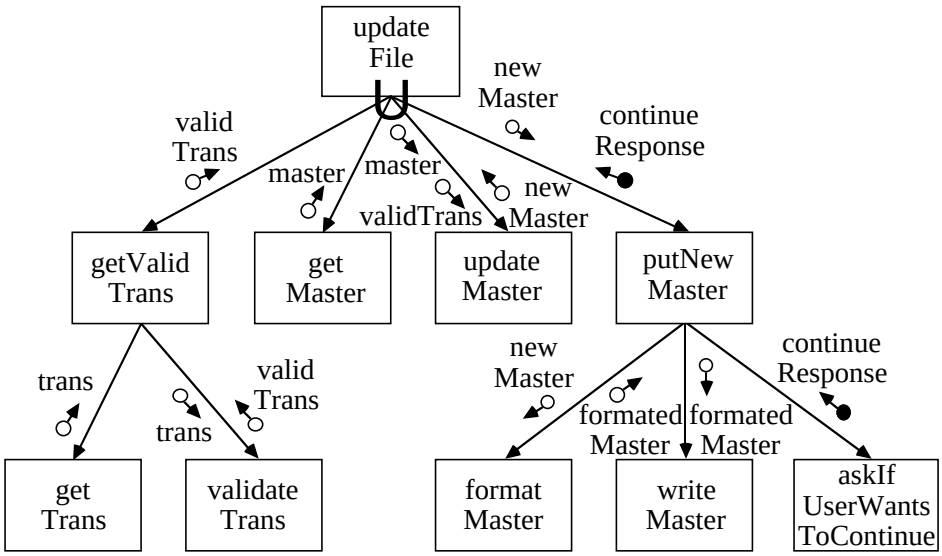

Figure 1 Sample Structure Chart

Finally, we specify the communication between modules. The predicate `coupling(Data, SourceModule, TargetModule)` would hold when `Data` flows from `SourceModule` to `TargetModule`. Thus:

```
coupling(validTrans, getValidTrans, updateFile).
coupling(trans, getTrans, getValidTrans).
coupling(validTrans, validateTrans, getValidTrans).
. . .
```

This completes the specification of a structure chart.

Despite the simplicity of the representation method, useful information can be extracted through the underlying deduction mechanism which is transparent to the user. For example, it is easy to find all the input and output data items related to a given `Module` by defining the following predicates:

```
input_data(Module, DataItems) :-
   is_module(Module),
   findall(Data, coupling(Data, _, Module), DataItemsB),
   delete_dup(DataItemsB, DataItems).
output_data(Module, DataItems) :-
   is_module(Module),
   findall(Data, coupling(Data, Module, _), DataItemsB),
   delete_dup(DataItemsB, DataItems).
```

The predicate `coupling(Data, _, Module)` would hold when the given `Module` accepts a piece of `Data` from some arbitrary module (denoted by "`_`"). The predicate `findall(Data, Goal, DataItems)` would hold when `DataItems` is a list of instances of `Data` such that `Goal` succeeds. The predicate `delete_dup(DataItems, Result)` would hold when `Result` is a list obtained by removing all the duplicated elements from the list of `DataItems`.

Furthermore, we can decide whether a given `Module` is a `get` module, a `put` module or a `transform` module by simply adding the following rules:

```
module_type(Module, get) :-
   /* nothing flows into it, but something flows out */
   input_data(Module, []),
   output_data(Module, [_|_]).
module_type(Module, put) :-
   /* nothing flows out of it, but something flows in */
   input_data(Module, [_|_]),
   output_data(Module, []).
module_type(Module, transform) :-
   /* something flows in and something flows out */
   input_data(Module, [_|_]),
   output_data(Module, [_|_]).
```

Here "`[_|_]`" denotes a non-empty list in Prolog.

4.2 Formalization of Evaluation Criteria

In this section, we illustrate how we can apply Prolog predicates to review structure charts according to evaluation guidelines as recommended by DeMarco [2], Page-Jones [12] and Yourdon [18].

(*a*) *Coupling and Cohesion*

Coupling is a measure of the interdependence among different modules. Modules should be loosely coupled, or relatively independent. There are five major types of coupling. Data coupling and stamp coupling mean that two modules communicate through atomic and composite data items, respectively. They are the best type of coupling. Control coupling means that two modules communicate through control flags. Common coupling means that two modules share common data, whereas content coupling means that they share common code. The first three types of coupling can easily be detected from our representation of a structure chart. For example, we can define a predicate `data_coupling(Module1, Module2)` which would hold when modules `Module1` and `Module2` exhibit data coupling.

```
data_coupling(Module1, Module2) :-
   (coupling(Data, Module1, Module2)
      ; coupling(Data, Module2, Module1)),
   is_data(Data, atomic).
```

Stamp and control couplings may be defined in a similar fashion. On the other hand, common and content couplings cannot be determined from structure charts alone. They are in fact implementation oriented and can only be detected at a later stage of systems development.

Cohesion is a measure of the strength of association of elements within a module. It is recommended that elements should be highly cohesive, or strongly inter-related. There are seven major levels of cohesion. Functional cohesion means that the module performs a single identifiable function, and is the best type of cohesion. Sequential cohesion means that data produced in an earlier part of the module will be used in a later part of the same module. In communicational cohesion, the elements in the module process data items in the same file, but not necessarily in any specific order. In procedural cohesion, the elements in the module are related by program control algorithms such as selection or iteration. In temporal cohesion, elements are grouped under one module because they are time-related. In logical cohesion, elements are grouped under one module because they are supposed to have similar behaviour, but actually exhibit minor differences. Coincidental cohesion means that the elements in the module are grouped together for no particular reason, and is the worst type of cohesion.

In our project, Prolog predicates are used to represent the internal functions of a module in a style similar to the representation of a structure chart described in Section 4.1 above. While the Prolog representation of internal functions is straightforward, determining the level of cohesion is not a simple matter. Modules with temporal and coincidental cohesions, for instance, are very similar in appearance. Tsai and Ridge [14] suggest that the expert system must query the user for information.

We note, however, that in real practice, the evaluation procedure for a structure chart would not require us to determine the exact cohesion level in every module. Even if we found the precise level of cohesion for each module, the end result would not be useful because there is no accepted guideline for combining all of them to give an overall measure. Instead, most human designers are only interested in identifying modules with relatively poor cohesion levels and improving them accordingly. For example, in the better classes of modules with functional, sequential and communicational cohesions, their functions refer to some common data. In our system, therefore, we would highlight those modules whose functions do not share any common input/output data.

(*b*) *Tramp Data*

According to Page-Jones [12], a tramp is ''a piece of information that shuffles aimlessly around a system, unwanted by — and meaningless to — most of the modules through which it passes.'' This definition is quite vague, as the terms ''meaningless'' and ''most'' are only intuitive concepts and never formally defined. It is quite difficult to convert such a definition into a conventional programming algorithm. However, experienced designers are able to recognize tramp data in most cases. In Figure 2, for instance, the data item `master` appears meaningless to the modules `getTrans` and `getValidTrans`, and would be regarded as a tramp by most human practitioners. This kind of knowledge can be handled more easily by logic programming than conventional programming.

We may try to specify the concept of meaninglessness using the simple predicate shown below. It checks whether a piece of `Data` passes in and out of a `Module` directly.

```
meaningless(Data, Module) :-
   coupling(Data, Module1, Module),
   coupling(Data, Module, Module2).
```

Before jumping to conclusions, however, let us take a look at the `getValidTrans` module in Figure 1. According to our simple definition, the data item `trans` would be meaningless to the module, and yet most human practitioners do not regard it as a tramp. The reason is that this design actually follows another guideline on the abstraction of data. That is, the `getValidTrans` module hides the physical characteristics of `trans` from modules higher up in the chart. This is especially useful for `get` and `put` modules. A more elaborate definition of meaninglessness should, therefore, involve checking if both the source and destination modules of a data item are children of the same module (or `siblings`):

```
meaningless(Data, Module) :-
   coupling(Data, Module1, Module),
   coupling(Data, Module, Module2),
   not siblings(Module1, Module2).
siblings(Module1, Module2) :-
   parent_child(ParentModule, Module1),
   parent_child(ParentModule, Module2).
parent_child(ParentModule, Module1) :-
   structure(ParentModule, _, ChildModules),
   member(Module1, ChildModules).
```

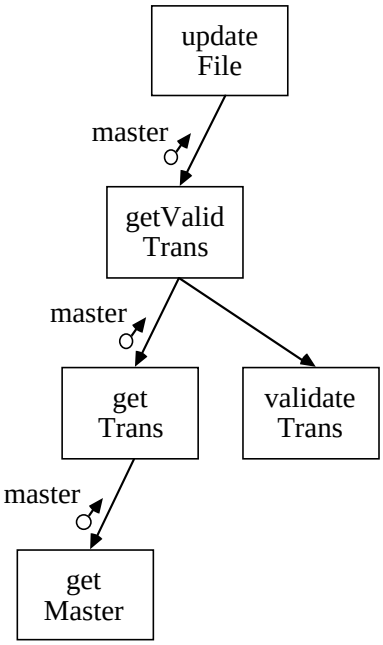

Figure 2 An Example of Tramp Data

Here `parent_child(ParentModule, Module1)` would hold when `ParentModule` is the parent of `Module1`, and `member(Module1, ChildModules)` would hold when `Module1` is an element in the list of `ChildModules`.

Another problematic term in the definition of tramp is ‘‘most’’. It should be clear that it is meant to be a relative measure rather than an absolute one. This can be represented easily in Prolog as follows:

```
tramp(Data, UserDefinedRatio) :-
   input_output(Data, ModuleList),
   length(ModuleList, NoOfInputOutput),
   NoOfInputOutput > 0,
   sub_list(ModuleList, Data, TrampModuleList),
   length(TrampModuleList, NoOfTrampInputOutput),
   Ratio is NoOfTrampInputOutput / NoOfInputOutput,
   Ratio > UserDefinedRatio.
sub_list([], Data, []).
sub_list([Module|ModuleList], Data,
      [Module|TrampModuleList]) :-
   sub_list(ModuleList, Data, TrampModuleList),
   meaningless(Data, Module).
sub_list([Module|ModuleList], Data, TrampModuleList)
      :-
   sub_list(ModuleList, Data, TrampModuleList).
```

where `NoOfInputOutput` is the number of times that `Data` will be input to or output from a module, and `NoOfTrampInputOutput` is the number of times that `Data` will be input to or output from a module which regards it as meaningless. This formalization of tramp leaves the final definition of ‘‘most’’ to the user. For example, the user might consider ‘‘most’’ to mean ‘‘more than half’’, in which case he should pose the query as

```
?-  tramp(Data, 0.5).
```

to the system. In this way, `validTrans` in Figure 1 would not be regarded as tramp data because `NoOfInputOutput` is 4 and `NoOfTrampInputOutput` is 2, and hence `Ratio` is exactly 0.5.

(*c*) *Flat Subcharts*

Another guideline for evaluating structure charts is that modules higher up in a chart should process ‘‘abstract’’ data, or data with less ‘‘physical’’ characteristics. Thus, a data item called `validTrans` is more abstract than another data item called `trans` because the former has presumably undergone a validation process and hence less physical than the raw `trans`.

But how do we know in general whether a data item is more abstract than another? One possibility is to mimic how a human reader would determine the abstraction level — by looking at the names of the data items. We may try to define a predicate `more_abstract_than(A, B)` which would hold when data `A` is more abstract than `B`, thus:

```
more_abstract_than(A, B) :-
   abstract_prefix(Prefix),
   append_terms(Prefix, B, A).
```

where `abstract_prefix` is defined by a collection of assertions such as

```
abstract_prefix(edited).
abstract_prefix(valid).
abstract_prefix(invalid).
abstract_prefix(matched).
abstract_prefix(unmatched).
```

This is rather unsatisfactory because there is no flexibility in the rule. Any prefix used outside the given vocabulary will not be recognized. Furthermore, it would not work if the system designer used another convention for naming data items. For example, a module which prepares invoices for output may accept `invoice_information` as input and produce `edited_invoice_ information`. Although the second data item carries with it an `edited` prefix, it is in fact more physical than `invoice_information`.

An alternative, as adopted by Tsai and Ridge [14], is to rely on the user to specify the module which will be regarded as the root of a chart, and hence the data processed by the root will be the most abstract. Our aim in this paper is to investigate how to extract as much information as possible from a given structure chart and rely less on additional user input. For example, we may like to check for the abstraction level of data items based on the structural organization of the chart. Consider the excerpts from two similar structure charts, as shown in Figures 3 and 4. The design shown in Figure 3 is better than that shown in Figure 4, because it follows the guideline on the abstraction level of data items, namely that `A` should be transparent to `Parent`. We can detect the anomaly in Figure 4 by means of a `flat_subchart` predicate.

```
flat_subchart(Parent) :-
   /* modules involved */
   parent_child(Grandparent, Parent),
   structure(Parent, seq, [Module1, Module2,
      Module3]),
   /* in case the first child is a get module */
   module_type(Module1, get),
   coupling(A, Module1, Parent),
   coupling(A, Parent, Module2),
   coupling(B, Module2, Parent),
   coupling(B, Parent, Module3),
   coupling(C, Module3, Parent),
   coupling(C, Parent, Grandparent).
```

We can similarly define a rule for detecting an anomaly in a `put` module.

```
flat_subchart(Parent) :-
   /* modules involved */
   parent_child(Grandparent, Parent),
   structure(Parent, seq, [Module1, Module2,
      Module3]),
```

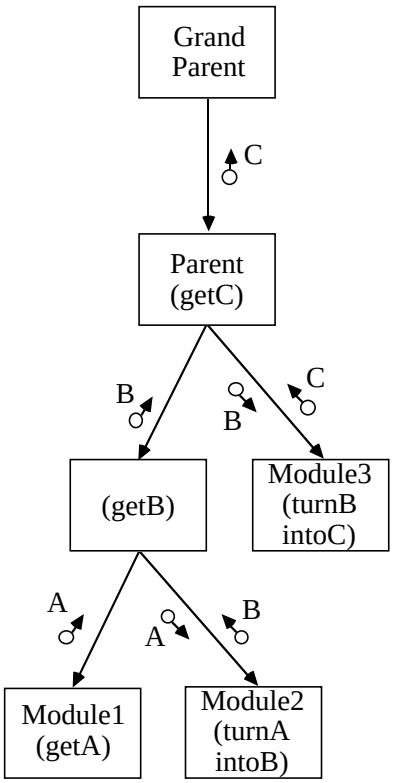

Figure 3 Get Modules according to Recommended Guidelines

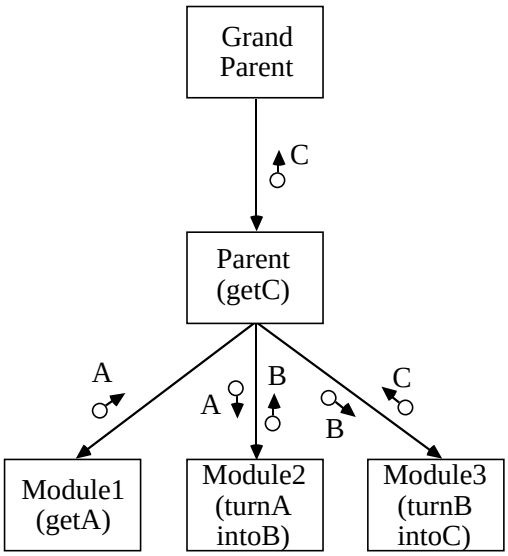

Figure 4 Example of a Flat Subchart

```
/* in case the third child is a put module */
module_type(Module3, put),
coupling(A, Grandparent, Parent),
coupling(A, Parent, Module1),
coupling(B, Module1, Parent),
coupling(B, Parent, Module2),
coupling(C, Module2, Parent),
coupling(C, Parent, Module3).
```

5. Findings and Conclusion

This paper is part of a long term effort to apply techniques of logic programming to structured systems development methodologies. We have described our experience in the evaluation of structure charts. We find that logic programming is a useful tool in this respect. We can represent structure charts naturally, and derive meaningful information in a straightforward manner. Standard techniques in the evaluation of structure charts can be formalized, and a few previous problems can be solved easily.

We find that Prolog is a very natural tool for the detection of anomalies in structure charts. There are several factors contributing to this: Firstly, we may not be fully aware of all forms of anomalies at the beginning, since most of them are based on the experience of individual practitioners and are not formally defined in the literature. Such forms of anomalies can be specified in a Prolog program incrementally. Secondly, we are more interested in what an anomaly looks like and not very much concerned about the algorithm to detect it. If we use Prolog, we can formulate procedures for detecting problematic structure charts by simply translating the anomalies into predicates, and leave the detection mechanism to the system. Thirdly, identifiers in structure charts will have the same name if and only if they refer to the same object [2]. Hence they are more naturally defined by logical variables in Prolog than variables in a conventional assignment-based programming language.

Because of the strong formal relationship [15] between structure charts and other structured tools such as DeMarco data flow diagrams [2], Jackson structure diagrams and Jackson structure text [5, 6], it is hoped that our attempt to use Prolog in analysing structure charts can also be applied in alternative structured notations. Our recent experience [16] confirms that this may indeed be the case. Thus the application of logic programming techniques to structured methodologies is a promising area from both research and practical points of view.

Acknowledgements

The authors are indebted to H.Y. Chen, H.L. Xie and other members of the ALPSE (Application of Logic Programming to Software Engineering) Group for their support of the project.

References

[1] 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.), Wiley, New York, pp. 73−96 (1982).

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

[3] 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, Massachusetts, pp. 82−95 (1988).

[4] C.J. Hogger, ‘‘Prolog and software engineering’’, *Microprocessors and Microsystems* **11** (6): 308−318 (1987).

[5] M.A. Jackson, *Principles of Program Design*, Academic Press, London (1975).

[6] M.A. Jackson, *System Development*, Prentice Hall International Series in Computer Science, Prentice Hall, London (1983).

[7] R.A. Kowalski, ‘‘Predicate logic as a programming language’’, in *Proceedings of the 1974 IFIP Congress* (*Information Processing ’74*), J.L. Rosenfeld (ed.), North-Holland, Amsterdam, pp. 569−574 (1974).

[8] R.A. Kowalski, ‘‘Algorithm = logic + control’’, *Communications of the ACM* **22** (7): 424−431 (1979).

[9] R.A. Kowalski, *Logic for Problem Solving*, North-Holland, Amsterdam (1979).

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

[11] R.J. Lauber, ‘‘Development support systems’’, *IEEE Computer* **15** (5): 36−46 (1982).

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

[13] O. Shigo, K. Iwamoto, and S. Fujibayashi, ‘‘A software design system based on a unified design methodology’’, *Journal of Information Processing* **3** (3): 186−196 (1980).

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

[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] 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): 659−676 (1994).

[17] M.H. van Emden and R.A. Kowalski, ‘‘The semantics of predicate logic as a programming language’’, *Journal of the ACM* **23** (4): 733−742 (1976).

[18] 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).