ソフトウェア工学　121－８
（1998．11．　5）

ソフトウェア自動設計におけるコンポーネント再利用補助のためのデータ辞書

マヘモド．エルホーリー　B.H. ファー　善彌河野

† 〒338-8570 埼玉県浦和市下大久保255
埼玉大学情報システム工学科
Tel. 048-858-3489,　Fax. 048-858-3716
E-mail. elkhouly@cit.ics.saitama-u.ac.jp
http://www.cit.ics.saitama-u.ac.jp/~elkhouly/

あらまし:　　ソフトウェア再利用のアプローチはプログラム開発の生産性向上の技術として提案されている。本論文においては、コンポーネントレポジトリ、各ソフトウェアコンポーネントの定義と同義語を考慮することによりレポジトリからコンポーネントの検索を助けるデータ辞書を提案する。まずコンポーネントをその機能により、クラスに分別する。ユーザに要求されたコンポーネントを発見するために新手法を適用する。この手法では、部分的にマッチしたコンポーネントをそのスーパークラスに統合し、インスタント変数及び似た名前に基づいてそのクラスにをいて最も適したコンポーネントをチェックする。プログラマの要求と検索されたコンポーネントの比較のためには意味チェックを用いる。この手法の利点は正確にマッチしなくても似たコンポーネントを発見できることである。

キーワード　データ辞書, ソフトワェア再利用, 設計

Data Dictionary Support for Reusing Components in Automatic Software Design

Mahmoud EL-KHOULY　Behrouz H. FAR　Zenya KOONO

†Department of Information and Computer Science,
Faculty of Engineering, Saitama University
255 Shimo-okubo, Urawa 338-8570, Saitama, Japan
Tel. +81-48-858-3489,　Fax. +81-48-858-3716
E-mail. elkhouly@cit.ics.saitama-u.ac.jp
http://www.cit.ics.saitama-u.ac.jp/~elkhouly/

Abstract:　Software reuse approach is proposed as a technique to increase productivity of program development. We propose the data dictionary support to help in retrieving the software components that a programmer wants to use from the components repository, definition and synonyms of each component had been taken into consideration. We classify the components to classes according to its functionality. We apply a new technique to find the required component. This technique can generalize the semi-matched component to its super class, and then check the most suitable component at that class according to its instant variables and the similar names, and it can use definition check to compare between the retrieved component with the programmer requirement. The advantage of this technique is that the exact match is not necessary to find a similar software component.

Key words　*Data Dictionary,Software Reuse,Information Retrieval , Design*

1 Introduction

There are many definitions of what software reuse means, but most involve some variation of *the use of engineering knowledge or artifacts from existing systems to build new ones* [1]. This definition encompasses many diverse areas, from application generators, to domain analysis, to design patterns, to classification and search, to management [2]. Our interest is in the use of existing software components, in particular *functions*. Now, research dealing with searching software libraries has principally focused on improving indexing [3,4,5]. Others have much attention towards automated methods for gathering information in response to a query from a user [6,7,8,9]. In our research, first we classify the software components into classes according to its functionality, then we store these components with what we call *data dictionary* in a repository. Second, we establish a new model contains two levels for retrieving components from a repository. In the first level, we retrieve it according to the component specification (name, similar function, ...) and we use a frame based representation to inherit the supper-class characteristics. In the second one, we use a definition check of the retrieved component with the definition required. Third, we use Jackson System Process (JSP) to represent the input and the output data structures which will be the input to our system. By using JSP, different programmers will produce similar programs, which makes it easier for one programmer to understand and maintain anothers's. Fourth, we built a set of converter rules bases which transform the JSP' program data structure to different computer source code's languages (in this paper we use converter rules for pseudo code). We give an example of how to apply the above four steps in automatic software design area, however, the design itself in not our target, but the outcome of the design is. That is why we use JSP in our design phase because its outcome is straight forward to generate pseudo code program automatically.
Figure 1, shows the overall system mechanism applying in an agent environemt. The user specifies his input data, output required and data specification for the terms he uses. The programmer generates only two data structures (one for input and the other for output). By supplying programmer's output to the agent, the agent produces program data structure after retrieving the similar software component from the repository. Then, the agent generates the source code referring to the converter rules base.
If the user ask from beginning for definition check, then the programmer supplies also the data specification table to be considered during retrieving process. We applied the above mechanism also in a multi-agent environment [10].

In the next section, we introduce the background which includes data dictionary and Jackson system process. In Section (3) we present the retrieval system mechanism from constructing the repository until implementation phase, and we give an example in automatic software design at Section (4). Finally, Section (5) presents the conclusion.

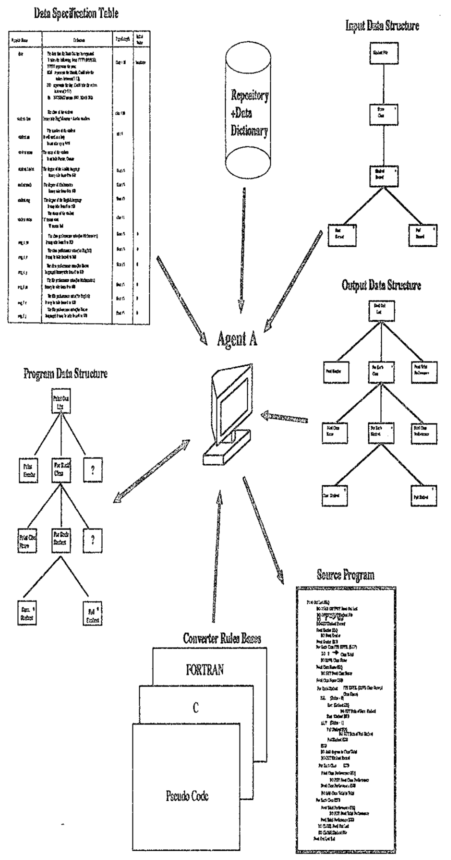


Figure 1: Overall system mechanism

2 Background

2.1 Data Dictionary

The failure of component retrieving is mainly caused by the disaccord of component designers and other agents who want to reuse it. Each component has the corresponding specification (e.g., name, class, definition, ...). Usually, the agents retrieve the appropriate components according to the name or functionality. However, if the keyword of the components at repository and that of the agent disaccord, the retrieval can be failure even if the eligible components exist in the component repository. Moreover, most information retrieval systems use Boolean operations for searching large document, collections. While Boolean operations for information retrieval systems [11] have been criticized, improving their retrieval effectiveness has been difficult [12]. Intelligent matching strategies for information retrieval often use concept analysis requiring semantic calculations at different levels [13]. Ambrosio [14] used two auxiliary dictionaries (*Domain* dictionary defining the relationships between different application and *Term* dictionary defining the semantic and syntactic relationships between concepts) to fulfill the previous shortage. However, using auxiliary dictionaries requires a new query process at that dictionaries. In our repository we included the data dictionary in the components repository, such that no new query process is required to search about the semantic of components. An item in our repository is a "class", in the object-oriented sense. A class consists of a set of "methods" which define its functionality. Each method has a set of "instance variables", "formula", "similar names", "similar function" and "definition", as shown in Figure 2. By this structure, we found that the retrieval process becomes faster than using auxiliary dictionary, the contents of the repository become complete, understandable, clear and readable.

2.2 Jackson System Process

Jackson Structured Programming (JSP) [15] is a method of program design which has arisen within the field of commercial Data Processing (DP) as part of a whole new approach to systems development [16]. JSP is a prescriptive de-

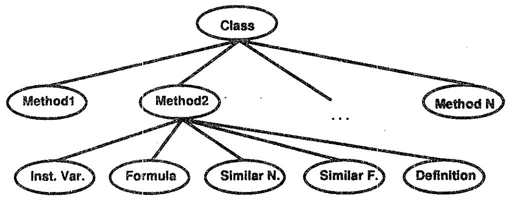


Figure 2: Class hierarchy

sign method which can be taught and can be learned. It enables a trainee programmer to learn how to design programs without resort to trial and error or the 'sit by Fred' approach, in which he/she is asked to sit next to an experienced programmer until by some magical process he/she assimilates the latter's knowledge. Using JSP, different programmers will produce similar programs, which makes it easier for one programmer to understand and maintain another's. That is the reason why we selected JSP method rather than object oriented approaches. Moreover, the theory of JSP cannot tell us how to code a program, but in practice the implications of JSP for coding are immense, such that, we put rules to transform the JSP' program data structure to any programming language's source code.

JSP has four types of component. They are: elementary components, which are not further dissected and have no parts, and three composite types: (a) sequence, which has two or more parts occurring once each, in order, (b) iteration, which has one part, occurring zero or more times, and (c) selection, which has two or more parts of which one, and only one, occurs once. The three composite types form an effective structuring system. Each is easy to understand, in the sense that the relationship between its parts corresponds to an intuitively acceptable idea which is simple and easy to remember. Also, it has been shown that any program which can be expressed in a flowchart can be expressed as a structure of sequences, iterations and selections, so there is no restriction on the programs we may write [15].

3 Retrieval System

3.1 Data item

Suppose that we have a first order definition of data item L with signature

L=⟨ C,N,I,R,S,J,M ⟩, (1)

where C is a super class; N is a name of the function; I is a set of instance variables; R is a formula of the function; S is a set of synonym functions' names; J is a set of similar functions that may have different names and M is a definition of the function. We are interested in some formal criteria for obtaining a software component, e.g., a function F, with signature

$F = \langle N, I, R, M \rangle$ (2)

i.e., a function F should retrieve with its name, instant variables, formula and definition.
We use equation (1) in building our repository, while we use equation (2) as what we expect from the retrieving process.

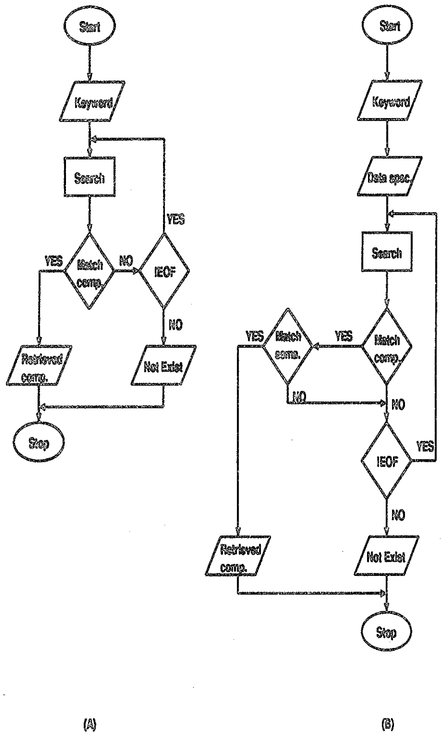


Figure 3: a- Retrieve without semantic check, b- Retrieve with semantic check

3.2 Find and Similar

Def.1 A function F is *exact match* with L iff
$((L(N) = F(N)) \vee (F(N) \in L(S)))\wedge$
$((F(M) \subseteq L(M)) \vee (L(M) \subseteq F(M)))$ (3)

It means that a function F is *exact match* with the data item in the repository if and only if the following two conditions are true: (a) they have the same name; or the name of the required function exists in the list of similar names of that data item. (b) the definition of the required function is subset or equal the definition of that data item; or vice versa.
Def.2 A function F is *match* with L iff

$(L(N) = F(N)) \vee ((F(N) \in (L(S))$ (4)

It means that the function F is *match* with the data item in the repository if and only if they have the same name; or the name of the required function exists in the list of similar names of that data item.
Def.3 A function F is *similar* to L iff

$(L(C) = F(C)) \wedge ((F(N) \in (L(J))$ (5)

i.e., the function F is *similar* with the data item in the repository if and only if their super class has the same name and the name of F exists in the list of similar functions of that data item.
We use the above definitions to construct two important commands in our retrieval system: **Find** which uses definitions 1 and 2 to retrieve *exact match* or *match* components, and **Similar** which applies definition 3 to find *similar* components.

3.3 Query process

The interaction between the agent and the repository of reusable software components must follow a well defined process. This process will control both the visual layer and the access to the repository.
Def.4 Query process system can be formulated as a quadruple

$S = (\alpha, \beta, \gamma, \lambda)$ (6)

where
$\alpha = \{\alpha_1, \alpha_2, ..., \alpha_M\}$ is a set of index terms (e.g., keywords);
$\beta = \{\beta_1, \beta_2, ..., \beta_N\}$ is a set of clauses in the repository, β_i has the form of (1), and each $\beta_i \subseteq \alpha$;
$\gamma = \{\gamma_1, \gamma_2, ..., \gamma_L\}$ is a set of queries each $\gamma_P \subseteq \alpha$;
and $\lambda : \gamma \times \beta \to \Re^+$ is a ranking function that evaluates the relevance between a query and a

clause. In a general form a clause β_j can be denoted as a set of index term-weight pairs

$$\beta_j = (\alpha_{j1}, \omega_{j1}; \alpha_{j2}, \omega_{j2}; \ldots; \alpha_{jn}, \omega_{jn}) \quad (7)$$

where
$\alpha_{jr} \in \alpha$ and $\omega_{jr} \in [0,1]$, r = 1, ..., n, reflect the relative importance of terms $\alpha_{jr} \in \beta_j$.
A query $Q \in \gamma$ can also be denoted as a set of index term-weight pairs

$$Q = (\alpha_{q1}, \omega_{q1}; \alpha_{q2}, \omega_{q2}; \ldots; \alpha_{qm}, \omega_{qm}) \quad (8)$$

where $\alpha_{qs} \in \alpha$ and $\omega_{qs} \in [0,1]$, s = 1, ..., m.
The query process task is to yield a set

$$A = \{l_{a1}, l_{a2}, \ldots; l_{an}\} \subseteq \beta$$

to the query Q with a ranking order of $\lambda(Q, l_{ai})$ and $l \in L$. We use these term-weight pairs in definition check-level.
We have implement this model in the retrieval system as shown in Figure 3.
As shown, data dictionary is an important aspect of our retrieval system. The data dictionary include synonyms (including abbreviation), similarity, superordinate and subordinate names. The element of the retrieval mechanism responsible for the automatic modification of queries is called the Query Process. The data dictionary will be used by the query process to guide the query modification through comparing with the contents of repository. The query process permits the agent to choose one of three approaches. In the first, components are retrieved only if an exact match is found. This is the traditional approach to database querying. In the second, query modification is applied if no exact match could be found. In this case, the data dictionary is checked to try to find similar names. In the third approach, query modification is applied again to find a name of similar function which performs the same action (as shown in Figure 4).

3.4 Query Modification

As we have seen, the query process permits the definition of exact and imprecise queries. The first is treated in the traditional way, by interrogating the repository to find components that satisfy the conditions specified. At this stage the query process uses the command **Find** to search in the repository. If no satisfactory component is found, the query process uses the second type by changing the order variables in the **Find** command to search about similar names. If still no

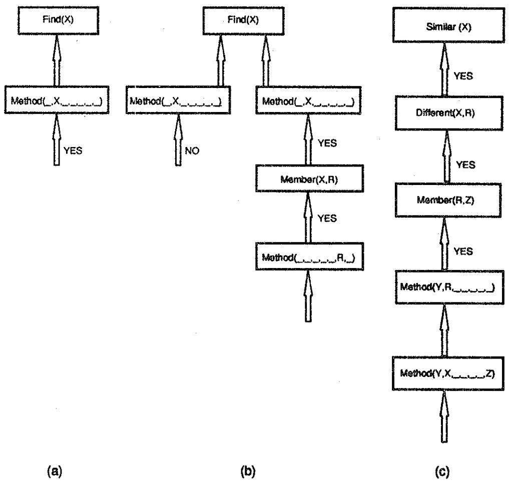


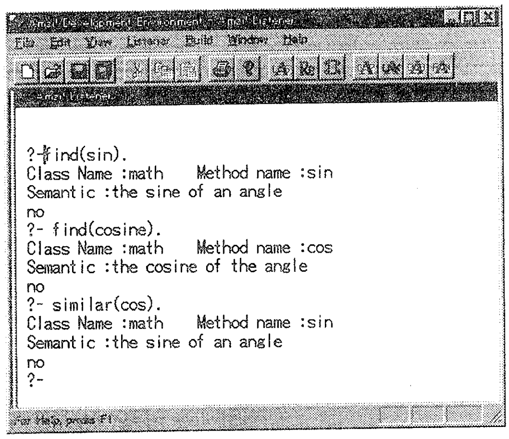
Figure 4: Query process

satisfactory component is found, the answer returned to the agent is null. For the third type, the query process replaces (**Find** command) with (**Similar** command) which uses the data dictionary to find similar functions' names. Therefore, if a query states that it wants all components that have for example, "cosine" as a key concept, the query will consult the data dictionary and recover all the synonyms of "cosine" (e.g., "cos") in the radius stated in the query.

In this approach, if the retrieving request does not contain definition check, then the agent scans the methods names from top to bottom and selects the first one that seems of sufficient interest. The selected method is expanded to allow an assessment of its functionality by a closer inspection of its variables and formula. If it needs to adapt, we generalize that method, by going up to its class, and then turn to other similar method in that class, which is more suitable. But, if the retrieving request contains definition check, then we extract those methods which seem of sufficient interest, and then apply some kind of natural language processing method to check its definition, to find a most suitable method for accurate retrieval.

3.5 Implementation

We implemented the system using Prolog, since Prolog is especially well suited for problems that involve objects- in particular, structured objects-



```
?-find(sin).
Class Name :math     Method name :sin
Semantic :the sine of an angle
no
?- find(cosine).
Class Name :math     Method name :cos
Semantic :the cosine of the angle
no
?- similar(cos).
Class Name :math     Method name :sin
Semantic :the sine of an angle
no
?-
```

Figure 5: AMZI output

and relations between them. The software components were stored as clauses, while the search technique had added at the beginning as predicate rules. By this way, we can add/modify the software components as will as the predicate rules. Also, the feature of learning the structure of new components is existing. Figure 5 shows the implementation of first level using AMZI PROLOG Ver 4.1.

4 Automatic Software Design Example

Consider a student file which contains the name of students, their numbers, their degrees, etc. and we simply want to print out each class with its performance, and the total file performance at the end (as shown in Figure 6). So, the manager of the school (the user) will give to the trainee programmer just the input data file (which contains, student number, student name, class name and student's degrees), and the output list required (which contains some fields not appear at the input data file e.g., class performance). And the user gives also what he means by the variables names which he uses in both input data file and output list required in the form of data specification table. In this table for example *avg-c-m* means *average of students degrees in mathematic at one class.* The programmer will use JSP notations [15] to design both input data structure and output data structure. Then, he/she supplies it to the agent in the form of two dimensional array 'node[row,column] in which node[1,1] means the first node in the chart and node[2,3] means the third node at the second level in the chart. The agent now, has four resources: (1) input

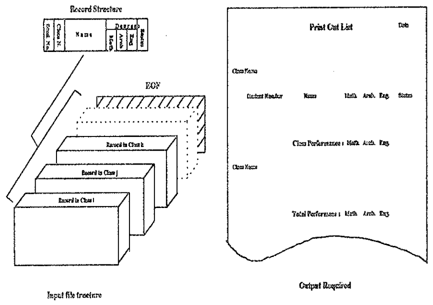


Figure 6: Input example

data structure, (2) output data structure, (3) repository which contains the software component with the data dictionary, (4) converter rules base for pseudo code. Therefore, the agent will use the first three resources to generate the program data structure, however, to produce program data structure, it needs to recognize the functions which suitable to transform the given input to the required output. Therefore, it uses the retrieval system to get a suitable components from the repository. After constructing program data structure, the agent uses the fourth resource *converter rules* (Figure 7) to transform JSP program data structure to pseudo code (as shown in Figure 1). If the programmer is accurate, then he/she can also supply the data specification table as the fifth resource to the agent, such that definition check can take place, as shown in Figure 3. As the result of this approach, we found that the programmer generates only two charts (in the above example, they contain 12 nodes) which is transformed by the agent to (59 line in Pseudo code without comment lines) and (74 line in C without comment lines). Considering that the time required for producing one code line is equal to the time required for producing one of chart's node, then we found that by using our new approach we save 79.6% ~ 83.7% of the original time required, and therefore from the cost. Moreover, existing of different converter rules bases, help in generating the source code with the language which user prefers.

```
Rule 1 : If     its the first node
       then 1) write the name of the node + SEQ
            2) infer all other nodes
            3) write the name of the node + END

Rule 2 : If     the node has no child & no SELECT NOTATION
       then put do before its name.

Rule 3 : If     the node has one or more child and don't contain
                (LOOP or SELECT notations)
       then 1) write the name of the node + SEQ
            2) put do before the name of each child
            3) write the name of the node + END

Rule 4 : If     the node has LOOP notation
       then 1) write the name of the node + ITR UNTIL + node CONDITION
            2) infer all childern
            3) write the name of the node + END

Rule 5 : If     the node has SELECT notation
       then 1) write SEL + the name of the node + YES
            2) infer positive branch
            3) write ALT + the name of the node + NO
            4) infer negative branch
            6) write END
```

Figure 7: Converter rules for pseudo Code

5 Conclusion

The effectiveness of the reuse based approach to software development is strongly dependent on the underlying classification scheme and retrieval mechanism. In this paper we tried to cover both. We began from constructing the repository, and we fulfill its shortage to analyze its contents due to the absence of the semantic meaning of the contents, by supporting it with a data dictionary. In the retrieving point, we described a new model consists of two level of retrieving, retrieve without definition check and retrieve with definition check. For the first level, we presented three approaches to find a software component in the repository which satisfies the rules of (exact match; match; or similar). The advantage of this level is that the exact keywords match is not necessary to find a similar component. We also, presented the application of this new model to automatic software design area. The technique has potential to be applied to other programming areas.

References

[1] W. Frakes and S. Isoda, "Success factors for systematic reuse," *IEEE Software*, pp 14-22, Sept. (1994).

[2] T.J. Biggerstaff and A. J. Prelis, eds., "Software Reusability," *ACM Press*, Vol. 1, New York (1989).

[3] S.D. Fraser, J.M. Duran and R. Aubin, "Software Indexing For Reuse," *Proc. 1989 IEEE International Conference On Systems, Man and Cybernetics*, pp 853-858 (1989).

[4] R. Prieto-Diaz, "Implementing Faceted Classification For Software Reuse," *CACM*, Vol 34, pp 89-97 (1991).

[5] Y.S. Maarek, D.M. Berry and G.E. Kaiser, "An Information Retrieval Approach For Automatically Constructing Software Libraries" *IEEE Transactions On Software Engineering*,Vol. 17, No. 8 Aug., pp 800-813 (1991).

[6] Arens, Y., Chee, C.Y., Hsu, C., and Knoblock, C.A., " Retrieving and integrating data from multiple information sources," *International Journal on Intelligent and Cooperative Information Systems*, 2(2), pp.127-158 (1993).

[7] M.C. Bowman, P.B. Danzig, U. Manber, and M.F. Schwartz, "Scalable Internet Resource Discovery: Research Problems and Approaches," *Communication of the ACM*, 37(8), pp 98-107 (1994).

[8] Tim Oates, M.V. Negendra Prasad and V.R. Lesser, "Cooperative Information Gathering: A Distributed Problem Solving Approach," *Technical Report 94-66, Dept. Of Computer Science University of Massachusetts*, Amherst, (1994).

[9] Shigeru Fujita, Hideki Hara, Kenji Sugawara, Tetsuo Kinoshita and Norio Shiratori, "Agent-Based Support for Reusing Components in Library," *Knowledge-Based Software Eng.*, P.Navrat and H.Ueno (Eds.) IOS Press, (1998).

[10] Mahmoud El-Khouly, Behrouz Far, Aboul-Ella Hassanine and Zenya Koono, "Reuse software components in multi-agent environment, " *Proceeding of 33rd Conf. on Statistics, Comp. Sci. and Operations Researches*, Dec. 12-14 (1998).

[11] Wartik, S., "Boolean operations," *Info. Retrieval: Data Structures Algo.*, Frakes, W.B. and Baeza-Yates, R.(eds)., Prentice Hall PP 264-292 (1992).

[12] Frakes, W.B., "Introduction to information storage and retrieval systems, " *Info. Retrieval: Data Structures Algo.*,Frakes, W.B.

and Baeza-Yates, R.(eds)., Prentice Hall PP 1-27 (1992).

[13] Kaname Funakoshi and Tu Bao Ho "A Rough Set Approach to Information Retrieval," *Studies in Fuzziness and Soft Computing - Rough Sets in Knowledge Discover*, Lech Polkowski Andrzej Skowron (eds)., Physica-Verlag Heidelberg (1998).

[14] "Introducing Semantics in Conceptual Schema Reuse," *CIKM'94, Proceeding of the 3rd Int. Conf. On Info. Know. Management*, PP 50-56 (1994).

[15] Jackson M.A., "Principles of Program Design," Academic Press, (1975).

[16] Ralph Storer, "Practical Program Development using JSP," Blackwell Sci. Pub., (1987).