SOFTM: a software maintenance expert system in Prolog

Pau, L.; Negret, J. M.

Published in:
Proceedings of the Conference on Software Maintenance

Link to article, DOI:
10.1109/ICSM.1988.10181

Publication date:
1988

Document Version
Publisher's PDF, also known as Version of record

SOFTM: A SOFTWARE MAINTENANCE EXPERT SYSTEM IN PROLOG

L. Pau
Technical University of Denmark
Bldg. 348/EMI, DK 2800 Lyngby, Denmark

J.M. Negret
Battelle Memorial Institute

ABSTRACT: This paper describes a software maintenance (SM) knowledge based system called SOFTM, serving the three following purposes: (1) assisting a software programmer or analyst in his application code maintenance tasks, (2) generating and updating automatically software correction documentation, (3) helping the end user register, and possibly interpret, observed errors on the successive application code versions. The knowledge based system SOFTM is written in PROLOG II, and is largely applicable to application codes written in different programming languages, provided code descriptors can be retrieved. SOFTM does not address any of the syntactic, input-output, or procedural errors normally detected by the syntactic analyzer, compiler, or by the operating system environment. SOFTM is relying on a unique ATN network based code description, on diagnostic inference procedure based on context based pattern classification, on maintenance log report generators, and on interfacing capabilities of PROLOG II to a variety of other languages.

1. INTRODUCTION

1) The current concern about software maintenance is justified by the cost and quality of code repair and updates, while at least maintaining software reliability and performances. The estimated productivity gains expected from software maintenance are, according to Barry Boehm, TRW [45]

Corrective maintenance: 18% of current effort

Adaptative maintenance: 14% of current effort

Perfective maintenance: 7 % of current effort

Update: due to mismatch between user requirements and software specification: *14% of current effort*

The major issues in software maintenance and its role in the software production process, are discussed in [3,8,9,13,14,21,22,24,25,26,27,28,35,43,45,46]. The classical approaches followed are:

- software maintenance personnel selection
- performance goals, and quality control during maintenance
- software maintenance work breakdown
- distribution of responsibilities amongst code users, developers, quality assurance, and maintenance
- audits and user reviews
- problem reporting systems
- maintenance logs
- use of specification formalisms, typically of the SADT model
- use of program design languages
- use of structured techniques to maintain unstructured code
- designing in code maintainability

Amongst the tools in use, or at the research stage, can be mentioned:

- specification and program design languages
- software configuration systems
- conversion of source code in its structural control-flow graphs (e.g.S3, ADA, Z specification languages)
- source code controllers, formatters and comparators
- declarative constructs
- paragraph parsers
- cross referencing and linking facilities (mapping)
- display of data flows
- symbolic debuggers
- test data generation
- sequence analysis tools (for synchronization and recoverability)
- interpreters of abnormal endings
- coverage analysis
- *diagnostic metarules*

Many of the above are still at an early stage, thus resulting in the still overwhelming use of debugging heuristics as the basic software maintenance approach and tool.

2) Some research focusses on knowledge based programming, where code is being written with user driven access thru an intelligent editor to:

Knowledge sources	**Tools**
Data flow models and descriptors	Data flow design
Design rules	Transform aids
Data dictionary	Transaction analysis
Abstract procedures	Procedural templates
Common data structures	Design codes
Structured programming rules	
Common algorithms	

Examples of such related CASE projects are (non exhaustively): Tedium/Tedious Enterprises, Programmers Apprentice/MIT, USE_IT/Higher order Software, R1000/Rational, REFINE/Reasoning systems, Knowledge-based software assistant (KBSA)/Kestrel Institute, Intelligent program editor/Advanced Information and decision systems, POISE Interactive programming environment/Univ. of Massachusetts, INFORM/Univ. Stuttgart [1], KBPA/Esprit [2] and also in AI related CASE research [3,6,13,15,16,21,24, 40,43,44, 45,47].

3) Research has been reported on expert/knowledge based debugging: Message trace analyzer - Source code debugger/Univ. Waterloo [4],SPEAR/Digital Eqipment [5],FALOSY/Univ. Minnesota [7],SOFTM/ Technical Univ. Denmark & Battelle [30], besides [6,8,13,15,19,36,37,41, 47,48].

4) However, little work has dealt so far with knowledge based software maintenance, incorporating some of the relevant approaches and tools mentioned above in 1): see [10,11,17, 20,21,30,42]. It is the purpose of this paper to describe the structure of a software maintenance expert system SOFTM, written in Prolog II [49] , and operating on the source code of a diversity of programming languages.

2. GENERAL ARCHITECTURE SOFTM

1) On a specific piece of application code C(L), written in a programming language L for which there is an interpretor, compiler, linker, and editor, the actual knowledge based software maintenance system SOFTM carries out essentially error diagnosis and provides for the propagation of corrective changes (see Figure 2):

1. detection of errors: **except** : **a)** syntactic errors detected by the compiler
   b) cross-referencing errors
   c) calculation errors due to a wrong algorithm
   d) search, query or IO errors due to a wrong algorithm
2. location of errors, except 1) a,b,c,d
3. error diagnosis
4. maintenance guidance for C(L)
5. generation of explanation facilities for 1.-4.
6. automatic generation of code maintenance logs, to be incorporated into the C(L) code documentation.

This obeys the diagnostic strategies described in [29] and using context based pattern classification. This is essentially a backward chaining process where root error cause/correction goals are found from their consequences and partially known attributes [50].

2) The knowledge base of SOFTM is divided into three parts:

KB-1: Facts in predicate form, about error types, error localizations, diagnostic classes, the environment, and observables. Observables are passive if measured without modifying code execution, and active if external perturbations are necessary.
KB-2: Code independent kernel rules, applying to the general software maintenance task.
KB-3: Symbolic descriptors of C(L), derived by rewriting in predicate form C(L) features provided by the compiler, the specification language, or the data flow model, in an augmented transition network (ATN) form.

All three part are assigned to different sub-worlds in Prolog, for separation and decomposition; they are edited each by a knowledge base editor specific to each of the three parts. The knowledge base KB-3 is interfaced to other tools as indicated.

3) Regarding inferences and queries, the access is as follows:

- the code developper, can query and update KB-1 alone, and update C(L) thru the editor of that L language; he can also execute C(L)
- the code user, can query KB-1 and run C(L)
- the code maintenance programmer, can query and update KB-1, query KB-3, and update KB- 2.

3. APPLICATION DEPENDENT CODE KNOWLEDGE REPRESENTATION

The code representation is treated as fact predicates in a specific knowledge base world. It consists of:

a) code structure: it describes the information flows between code modules and arguments, stressing the call order during execution. The representation is an augmented transition network (ATN) with attributes, written in predicate logic. The node labels are provided by the linker, and the attributes by a standard run of the code (alternatively by a Petri-net based simulator).

b) module structure: each module of the application code is represented by a frame, attached to the corresponding ATN node. The frame fields are: pointers to I/0 arguments, pointers to internal arguments, ordered textual description of the functional purpose of each successive block in the module (as specified e.g. in structured or functional programming). These frame fields are provided by the linker or compiler, and by the module documentation located in "Comments".

4. KNOWLEDGE REPRESENTATION OF APPLICATION CODE C(L) (KB-3)

1) The relations between modules of C(L) are described by an ATN with the following fact descriptions of each relation in Prolog syntax:

```
mm(m1,m2,ℓ,p)-->;
```

where:	m1	:	is the module name being called from m2
	m2	:	is the module name calling m1, with return to m2
	ℓ	:	list of call sequence numbers in call chronology, terminated by nil
	p	:	name of code or root module

2) The relations between arguments (compiled by the interpreter or debugger) and a module, are described by the facts:

```
am (a,m,line,p)-->;
```

where:	a	:	is an (I/0) argument of module m
	m	:	module name
	line	:	relative address of argument a, or pointer
	p	:	name of code, or root module

3) Each code module is represented by the Prolog frame structure:

```
md (m, ℓ1, ℓ2, b,p) -->
```

where:	m	:	module name
	ℓ1	:	list of I/0 arguments to m represented by relative address pointers in list form, terminated by nil
	ℓ2	:	list of internal arguments in m, which are not in ℓ1
	b	:	pointers to relative address of first line in each functional block of m, in list form, ended with nil
	p	:	name of code or root module

md may be represented also with explicit arguments names, while preserving the same frame structure

4) From the above knowledge descriptions of C(L) it is obvious to estimate the number of steps and processes involved in the code, and to sort them by size.

5) C(L) can also preserve the time/state dependent information necessary to determine what activities are possible in a dynamic environment; in this case, the arguments about time and state are tagged separately for data flow or I/O control.

5. APPLICATION CODE INDEPENDENT KNOWLEDGE BASE

This knowledge base/world, which is application code independent, contains in predicate form, according to the diagnostic strategy of [29]:

Y): observables about the errors, yielding the values of qualitative/continuous measurements on the application code, once instantiated;
L): physical or virtual error locations;
E): generic error type or descriptor, such as data type error, undefined arguments, etc.
C): diagnostic causes (from the domains: design, execution, environment, human errors);
M): generic or specific SM actions affecting:application code programming, application code design, software environment, human factors.

6. KNOWLEDGE REPRESENTATION FOR THE DOMAIN INDEPENDENT FACT-BASES (KB- 1)

The facts in KB-1 are described by the following Prolog data structures:

1. **Observables:** passive(Y-P) or active (Y-A)
< observable - (type) - p/a, (number of observable), (time-stamp), nil, (sentence defining observable)· (measurement location)· (value of observable) · nil > -->;

2. **Location (L)**
< location, (number), (time-stamp), nil, (sentence defining locations) · (error number) · nil > -->;

3. **Error (E)**
< error, (number), (time-stamp), (likelihood), (error description sentence) · nil > -->;

4. **Diagnosis (C)**
< cause, (number), (time-stamp), (likelihood), (cause name sentence) · (error number) · (cause name sentence) · (error number) ··· nil > -->;

5. **Maintenance (M)**
< correction, (number), (time-stamp), (likelihood of effect), (sentence describing correction) · nil > -->;

6. **Documentation (D)**
< documentation, (number), (time-stamp), nil, (title sentence) · (location number) · nil >-->;

7. INFERENCE PROCEDURES

They consist of (see Figure 2):

a) control structure: it is the Prolog II depth first backtracking with top-to-bottom and left-to-right clause deletion, supplemented by verification and domain dependent predicates; these predicates are supplemented by context sensitive control predicates such as dif(x,y) and freeze(x,p), which implement truth maintenance and conditional propagation [49]

b) explicit inference procedures: they include both a forward and backward chaining, with an observation/goal restriction phase followed by pattern matching on the sets of rules in (c) below, and then by action clauses. The action clauses consist in adding/deleting facts with likelihoods, and by automatically logging them in the SM documentation file;

c) domain dependent inference rules: this rule base contains in predicate form, with a list syntax:

1. detection rules: Y -> E
2. localization rules: E -> L
3. diagnostic rules: ExLxY -> C
4. maintenance rules: ExLxYxC -> M with SM error documentation update
5. incorrectness and insufficiency metarules operating on the ATN
6. sequencing constraint control metarules, to check call sequences and propagate effects of corrections accordingly
7. rule cluster documentation generators for functional code blocks

d) uncertainty representation, by attaching likelihoods to each error (E), cause (C) or correction (M) fact; the likelihoods are propagated and combined along each inference path into an importance qualifier for each hypothesis or goal

The combination of a) b) and c) allows for the automatic propagation of maintenance changes, by asserting into the fact base KB-3 these changes. If new types of errors or locations or corrections are entered into KB-1 through the proper editors, they are automatically accounted for thanks to the Prolog declarative form. Thus the inference procedure in SOFTM uses fully propagation mechanisms and analysis.

8. ENVIRONMENT AND IMPLEMENTATION

The SOFTM knowledge based software maintenance environment in Prolog II [49] has been supplemented, besides the interfaces to the operating system, compiler, and higher level utilities (specification language output, simulator input), by:

- SM documentation explanation facilities
- fact, rule, code structure editors (specialized)
- knowledge base management commands
- query editor for forward chaining (diagnose an error)
- backward chaining editor (reason to possible candidate-corrections/errors)
- likelihood calculations
- time-stamp management on all SM actions
- debugger
- interface between Prolog II and constants, variables, lists in different languages (PASCAL, COBOL, FORTRAN, ADA)
- optional interface to a text database management system containing the full software documentation (e.g. BASIS)

The current implementation is on VAX/VMS for application code written in either COBOL, FORTRAN or ADA, and Prolog itself. Specialized application code languages are also considered, e.g. image processing language, test language, and expert systems [39].

9. KNOWLEDGE EXTENSIONS

The basic information has been collected, although not yet implemented, to enhance the application independent knowledge basis (KB-2) with respect to:

- metarules for identifying modules or module to module relations with similar structures
- measuring debugging/maintenance stress, to estimate likelihoods for detection or corrective actions
- generate and place scope markers to delineate code governed by conditional expressions
- protect from any corrective measure the I/0 arguments
- introduce simple software metric attributes to compare old from revised code
- generate from the call sequences a maintenance plan which obeys module interaction
- inclusion of simple alternate program verification techniques likely to appear in software testing certification standards.

REFERENCES

[1] G. Fischer, H.D. Boecker, The nature of design processes and how computer systems can support them, in P. Degano, E. Sandewall (Ed), "Intgegrated interactive computing systems", North Holland, 1983, 73-86 (INFORM system, University of Stuttgart)

[2] K. Poulter, Representing programming knowledge in the KBPA, in G.J.P. Katz (Ed), "ESPRIT'85: Proc. of the meeting", North Holland, 1986 (KBPA Franz-Lisp/C prototype)

[3] K.A. Frenkel, Toward automating the software development cycle, Comm. ACM, Vol 28, no 6, jun 1985, 578-89

[4] N.K. Gupta, R.E. Seviora, An expert system approach to real time system debugging, Proc. 1st IEEE Conf. on AI applications, 1984, p. 336 (Message Trace analyzer in Prolog, Univ. Waterloo)

[5] SPEAR, Expert systems J., Vol 1, no 2, October 1984, p 98 (SPEAR computer error log analyzer at Digital equipment)

[6] M. Schindler, AI begins to pay off with expert systems for engineering, Electronic Design Magazine, Aug. 9, 1984, 106-146

[7] R.L. Sedlmeyer, W.B. Thomson, P.E. Johnson, Diagnostic reasoning in software fault localization, Proc. IJCAI-83, Karlsruhe 1983, Vol 1, 29-31 (FALOSY master file update software fault finding, Univ. Minnesota)

[8] H.J. Hindin, Intelligent tools automate high-level language programming, Computer Design Magazine, May 15, 1986, 46-56

[9] D. Gustafson, A. Melton, A model for software maintenance, Proc. IEEE/ACM Conf. on software maintenance (CSM.87), Austin, TX, Sept. 1987

[10] B. Terry, R.D. Cameron, Software maintenance using meta-programming, same ref. as [9]

[11] F. Cross, An expert system approach to a program' s information/maintenance system, same ref. as [9]

[12] D. Richard Kuhn, A source code for maintenance, same ref. as [9]

[13] R.L. Baber, The Spine of software: designing provably correct software, Wiley, 1988

[14] M.W. Evans, J. Marciniak,Software quality assurance and management, Wiley, 1987

[15] C. Rich, R.C. Waters (Ed), Readings in AI and software engineering, Morgan Kaufman Publ., 1986

[16] B. Liskov, J.Guttag, Abstraction and specification in program development, MIT Press, Cambridge (MA), 1986

[17] L.B. Alperin, B.I. Kedzierski, AI based software maintenance, Proc. 3rd IEEE Conf. on AI applications, Orlando (FL), Feb. 1987

[18] Proc. conf. on reliability and robutsness of engineering software,Computational Mechanics Institute, Como (Italy), Sept. 1987

[19] M.A. Ould, C. Unwin, Testing in software maintenance development, Cambridge Univ. Press, 1986

[20] S.S. Yau, S.S. Liu, A knowledge based software maintenance environment, Proc. IEEE COMSAC'86, IEEE Computer Society, Chicago, 6-10 Oct. 1986

[21] D.A. Higgins, Data structured software maintenance, Dorset House Publ./Wiley, 1987

[22] J. Mastow, Toward better models of the design process, AI Magazine, Spring 1985

[23] GL.B. Kotik, A.. Rockmore, D.R. Smith, Use of REFINE for knowledge based software development, Proc. 4 th Int. Workshop on software specification and design, IEEE Catalog TH 0181-8, April 1987

[25] M. Lubars, M.T. Harandi, Intelligent support for software specification and design, IEEE Expert, Vol 1, no 4, winter 1986, 33-41

[26] I. Sommerville, Software engineering, Addison Wesley, 1985

[27] NTIS, Annotated bibliography on software maintenance, Superintendent of documents, Washington DC, S/N 003-003-02756-1, 1986

[28] Software maintenance, IEEE Computer society Press, Cat. 0-8186-0002-0. EH-0201-4, 1983

[29] L.F. Pau, Failure diagnosis and performance monitoring, Marcel Deker, N.Y., 1981

[30] L.F. Pau, J.M. Negret, SOFTM: a software maintenance expert system, Battelle Memorial Institute, Geneva, 1985

[31] T.S. Chow, Testing software design method by finite state machine, IEEE Transactions on software engineering, May 1979

[32] R.L. Glass, Software reliability guidebook, Prentice Hall, 1979

[33] J.E. Hopcroft, J.D. Ullman, Introduction to automata theory, languages and computation, Addison Wesley, 1979

[34] P.P. Howley, A comprehensive software testing methodology, Second software engineering standards application Workshop, SanFrancisco, May 1983

[35] Standard for software test documentation, ANSI/IEEE std 829-, 1983

[36] J.M. Morin, J. Perez, S. Xanthakis, Méthodes de modélisation pour la génération de tests de recette, Institute génie logiciel, T/0052/D/T, 1985

[37] G.J. Myers, The art of software testing, John Wiley and sons, New York, 1979

[38] P. Zave, A comprehensive approach to requirement problem, Proceedings of COMPSAC, 1979

[39] L.F. Pau, Prototyping, validation and maintenance of knowledge based systems software, Proc. IEEE 3rd Conf. on expert systems in government, Washington DC, Oct. 1987

[40] D. Partridge, AI: applications in the future of software engineering, Elis Horwood/Wiley, 1986

[41] Proc. ACM Symp. software engineering for high level debugging, SIGPLAN Notices Vol 18, no 8, Aug. 1983, ACM Order 593830

[42] C.A. Dyer, Expert systems in software maintainability, Proc. 1984 Annual reliability and maintainability symp., IEEE Publ. 0149-144 X/84, 295-299

[43] M. Schindler, Through automation, software shapes itself to the task at hand, Electronic Design, July 25, 1985, 87-

[44] H. Abelson, G. Sussman, The structure and interpretation of computer programs: a LISP perspective, MIT Press, 1985

[45] A.J. Rockmore, Knowledge based software turns specifications into efficient programs, Electronic Design, July 25, 1985, 105 - 112

[46] S. Bendifallah, W. Scacchi, Understanding software maintenance work, IEEE Trans. Software engineering, Vol SE-13, no 1, Jan 1987

[47] H. Wertz, Automatic correction and improvement of programs, Ellis Horwood/Wiley, 1986

[48] J. Loecky, K. Sieber, The foundations of program verification, Wiley-Teubner, 1987

[49] F. Giannesini, H. Kanoui , R. Pasero, M. van Caneghem, PROLOG, Addison Wesley, 1987.

[50] L.F. Pau, A survey of expert systems for failure diagnosis and maintenance, Expert Systems J., April 1986

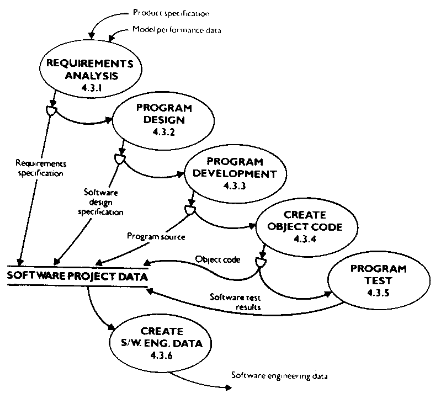

Figure 1: Software engineering CASE cycle

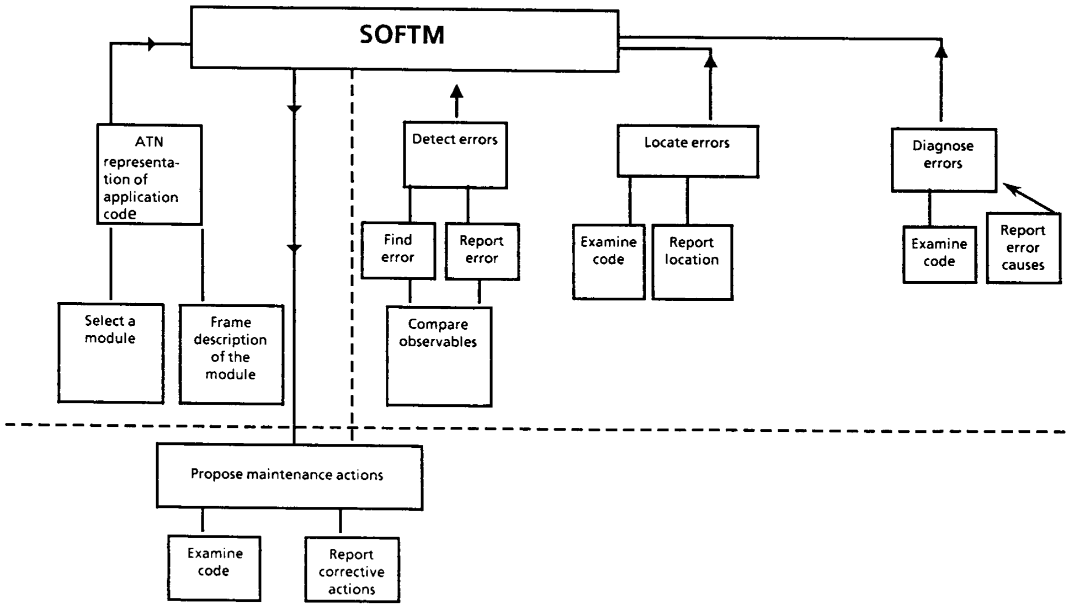

Figure 2: SOFTM inference functions