Automatic Fault Localization for Programmable Logic Controllers*

André Sülflow Rolf Drechsler

Institute of Computer Science, University of Bremen
28359 Bremen, Germany
{suelflow, drechsle}@informatik.uni-bremen.de

Abstract. *Programmable Logic Controllers* (PLCs) are widely applied to control safety critical systems. Efficient formal and non-formal methods to detect faulty behavior have been developed, but finding the cause of the buggy behavior is often still a manual process.
Automatic fault localization for PLCs is studied in this paper. Methods for automated debugging are analyzed and compared with respect to accuracy and run time. The experimental results on industrial models show a high accuracy at low run time costs.

1 Introduction

A *Programmable Logic Controller* (PLC) is a re-programmable computer based on sensors and actors that is running a user defined software periodically. That makes a PLC highly configurable and applicable in various industrial sectors, e.g., in nuclear power plants and in railway interlocking systems. The assurance of the correct behavior in safety critical systems is a must. In this work PLCs suitable to control railway electronic interlocking specified to *Safety Integrity Level 3* (IEC61508) are considered.

Model checking of PLC software was proposed in, e.g., [3, 11, 15, 18]. The output of a model checker is either a proof of correctness of the model with respect to a specification or a failure trace, i.e., a counter-example that shows the incorrect behavior. Debugging the observed faulty behavior often relies on manual simulation and is a time consuming task. Automated debugging of faulty behavior in PLC programs has not been considered so far and is in focus of this paper.

Several techniques for automated debugging have been proposed for software (e.g. [22, 2, 13, 9]) as well as for hardware (e.g. [21, 6, 17, 7]). The aim of automated debugging is the highlighting of potential fault locations to an engineer to reduce the complexity for a subsequent manual debugging session. Thus, fixing the faulty behavior remains a manual task to avoid unexpected changes that may be introduced by methods that perform repairs automatically [4].

Explaining debugging techniques compute traces that are similar to a failure trace but fulfill the specification to explain the faulty behavior [10]. The difference between

* This work has been supported in part by the Rail Automation Graduate School (Siemens Transportation, Braunschweig, Germany) and the European Union (project DIAMOND, FP7-2009-IST-4-248613).

the execution of the correct trace and a failure trace reveals potential fault candidates. Program slicing returns statements on the path from a (faulty) observation point to the primary inputs. The analysis is performed statically or dynamically with respect to the actual values of the failure trace [22, 2]. The techniques above do not fully exploit the expected behavior at the observation points and an over-approximation of fault candidates may be returned only.

Model-based diagnosis is more precise by computing fault locations on an abstract model [13]. Using a solver for *Boolean Satisfiability* (SAT) allows to handle large problem instances due to the tremendous improvements in Boolean satisfiability solving [5, 12]. SAT-based debugging [17] partially automates the debugging by finding possible fault locations, i.e., components that can *fix* the faulty behavior. The usability for complex models was shown for debugging hardware [17] as well as software [9].

In this work we analyze methods for automated debugging of PLC programs. Three automated debugging methods are evaluated: (1) static analysis, (2) dynamic analysis, and (3) correction-based debugging.

2 Diagnosis Model

In IEC61131-3 two textual and three graphical programming languages for PLCs are standardized. In this paper we focus on the assembler-like language *Instruction List* (IL). More specifically, *Statement List* (STL) is the input language of the PLC considered [16]. STL is similar to IL and is referenced as IL in the remainder of this paper.

The behavior of an IL program is specified in M lines of code that are sequentially executed in a deterministic order. Each line contains one instruction, composed of an optional label, an operator and an optional operand of $\{variable, constant\}$. The control flow is influenced by jump instructions, that are referencing a label in line r, $1 \leq r \leq M$. IL programs may be additionally divided into sub-programs.

A CDFG is constructed from the IL program to perform automated debugging. Nodes in the CDFG represent data instructions or control predicates, respectively. Data dependency and the control flow are modeled by edges.

An abstract model or a concrete model of the executing CPU of the PLC can be used to construct the CDFG. Because a CPU consists of several registers, stacks as well as accumulators the execution of an IL instruction highly depends on those state variables. The usage of a concrete model for the CPU semantic is more complex, but allows to perform CPU specific analyses. For example, registers may be highlighted to observe the faulty behavior, e.g., for a subsequent manual debugging session. A concrete model of the SIMATIC S7 CPU is used in this work. The SIMATIC S7 operates on a 16-bit status word, two 32-bit accumulators, and a nesting stack that stores intermediate results. Operations up to a bit-width of 32 are supported [16].

Technically, the IL program is translated to *SystemC* [20] by augmenting the behavioral information of the underlying CPU as described in [18]. Afterwards, the augmented implementation is analyzed with the parser of [8] to construct a CDFG representation similar to a netlist on RTL [19]. Depending on the chosen instruction, one instruction in IL corresponds to $t, 0 < t$, nodes in the CDFG. For example, the instruction L (Load) operates on the accumulator and requires two assign nodes (one for

each accumulator). Other instructions operate on the 16-bit status word of the CPU, e.g., JC (*J*ump *C*onditional) evaluates and updates four bits of the status word. The reference to the original instruction in IL is kept for each node in the CDFG. Thus, the selection of at least one of the t nodes of an instructions enables to mark the instruction itself.

Faults in software that change the output behavior of the IL program and that are observable at least at one observation point are considered. Thereby, an observation point may be any program state, internal variable, or primary output. Methods for verification are capable to provide one failure trace or in more general a set of m failure traces. The cause of faults in PLC hardware is most likely physical (e.g. due to aging) than logical (e.g. a missing instruction). However, the extension to debug faults in the underlying hardware is a possible extension for future work.

Without loss of generality, let a failure trace consists of:

- an input stimuli I to activate the fault,
- a set of R observation points OP_i and its faulty responses under I: $v_{faulty}(OP_i), 1 \leq i \leq R$,
- a set of R expected responses for each observation point: $v_{expected}(OP_i), 1 \leq i \leq R$

An input stimuli I defines values for primary inputs and values for state variables with respect to a single PLC cycle. The faulty behavior is observable at least at the R observation points. Thus, simulating the PLC program with respect to the input stimuli I yields a pairwise distinct at all observation points: $\forall i : v_{faulty}(OP_i) \neq v_{expected}(OP_i), 1 \leq i \leq R$. The expected responses are automatically obtained by simulating the stimuli I on a reference model.

Components are used to explain the faulty behavior. In general, a component may be of any granularity, e.g., a set of instructions, a single instruction, or a single operand. However, the complexity of diagnosis increases with the granularity. Without loss of generality, single instructions in IL are considered as components in this work only.

A *fault candidate* is a component that *may change* the value at all observation points to the expected values. A *fix* is a component that *changes* the value at all observation points to the expected values: $\forall i : v_{faulty}(OP_i) = v_{expected}(OP_i), 1 \leq i \leq R$. The actual fault site is the position where the fault was injected and it is called *fault site*.

3 Debugging Algorithms

The CDFG and a set of m failure traces ($m \geq 1$) are the input for automatic debugging. Three debugging techniques are analyzed in the following: (1) static analysis, (2) dynamic analysis, and (3) correction-based debugging.

The following sections briefly introduce the methods. For more information about the debugging techniques, we refer the reader to the referenced papers in the respective sections.

3.1 Static Analysis

Independent of any failure trace, a static analysis on the CDFG enables to identify components that have no influence on the output behavior of the IL program. The redundant code fragments manifest themselves as pending nodes in the CDFG, i.e., nodes without successors, and are detectable in linear time (linear in the number of nodes). Only nodes that are in at least one fan-in of any primary output influence the output behavior. Operations that are not in any fan-in are redundant, cannot change the output behavior, and can be removed from consideration. By this, the initial number of fault candidates is reduced. Therefore, the information is also worthwhile for code optimization on the IL program. For example, an assignment to a variable A is removable, if no read operation is applied to A and A is not an output or a state element.

An additional analysis based on static slicing [22] uses the observation points to further reduce the number of fault candidates. A faulty behavior is observable at an observation point OP_i. Thus, all nodes on the path to any OP_i are potential fault candidates. All nodes that are not in the recursive fan-in of any OP_i cannot influence the value at OP_i and do not have to be considered for diagnosis. The nodes on the path from an observation point OP_i are recursively computed in linear time.

Let P_{ji} be a set of nodes on the path for failure trace j and observation point OP_i. While assuming a single fault, all relevant nodes for debugging (P') are computed by $P' = \bigcap P_{ji}$. For multiple faults the intersection can be empty, e.g., if a faulty behavior is observed on two observation points that have disjunctive input cones. One element from each path has to be selected while assuming multiple faults. However, due to the computational complexity for computing the hitting set, an over-approximation is considered for multiple faults only: $P' = \bigcup P_{ji}$. The union of all nodes from any path are returned as fault candidates in case of multiple faults.

3.2 Dynamic Analysis

Dynamic analysis is applied to debug the IL program with respect to the input stimuli of a failure trace. Instead of debugging all parts of an IL program, the failure trace is analyzed to determine the *sensitive path*.

For this purpose, the failure trace is simulated on the CDFG to obtain the assigned values on each node in the CDFG at first. Afterwards, an affect analysis is performed based on path tracing [1], i.e., a path from an observed faulty behavior at an observation point to the primary inputs is computed [21]. The path is extracted by following the controlling values, i.e., the operations that are responsible for the current value at the observation point $v_{faulty}(OP_i)$. In difference to dynamic slicing [2], path tracing returns statements that are *responsible* for the current value at the observation points instead of statements that *might affect* the value only.

For example, the controlling paths for a logical *AND* with output value 0 are the inputs with value 0 assigned. An input with 1 assigned is not responsible for the output value 0 and thus cannot be responsible for the faulty behavior. If both inputs are controlling paths, the algorithm follows both paths.

The dynamic analysis is performed for all observation points OP_i and all m failure traces in linear time. Thereby, each failure trace is simulated on the CDFG, followed

by path tracing. All nodes on the sensitive path from any observation point are potential fault candidates. Like for the static analysis, either the intersection or the union is computed for all sensitive paths while assuming single faults or multiple faults, respectively [21].

3.3 Correction-Based Debugging

Static analysis and the dynamic debugging technique use information about the observation points OP_i and the input stimuli only. Correction-based debugging is a technique to obtain higher accuracy by considering additionally the expected output responses at the observation points OP_i [17]. The input for the debugging algorithm is a CDFG, a set of m failure traces and the expected responses for each observation point $v_{expected}(OP_i)$.

In contrast to static analysis and debugging based on simulation, the returned fault candidates are proved to fix the failure trace while assuming a non-deterministic behavior of a component. That is, the output behavior of a component is replaced by an unconstrained new input that may have any value assigned. This allows to check whether the application of any correction at the component fixes the faulty behavior at all observation points, i.e., it holds $\forall i : v_{expected}(OP_i) = v_{faulty}(OP_i), 1 \leq i \leq R$.

For each component *correction logic* is inserted in the CDFG as described in [9]. That is, each right-hand side of an expression and all control predicates can be replaced by a non-deterministic value. One *abnormal predicate* for each component controls the activation at all expressions and control predicates from the same component, simultaneously. The CDFG is extended by correction logic, the input stimuli (I) is constrained to the values of the failure trace, and the observation points are restricted to the expected responses ($v_{expected}(OP_i)$).

The initial debugging instance is contradictory, due to the constrained expected responses at the observation points OP_i. Correction-based debugging resolves the conflict by activating exactly k abnormal predicates. Thereby, k gives the cardinality of the fault candidate and may be a single fault ($k = 1$) or a multiple fault ($k > 1$). Starting with $k = 1$, k is incremented as long as the instance is still contradictory. A *cardinality constraint* controls the activation of abnormal predicates. The debugging model is translated into *Conjunctive Normal Form* (CNF) and given to a *Boolean satisfiability solver* (SAT solver) to declare the instance satisfiable or unsatisfiable. After the conflict is resolved, i.e. the instance is satisfiable, fault candidates of minimal cardinality are obtained.

4 Evaluation

The debugging methods are evaluated on concrete benchmarks in the following sections. A detailed case study is presented in Section 4.1 to highlight the pro and cons of the debugging techniques. Experimental results for the application on industrial benchmarks are given in Section 4.2.

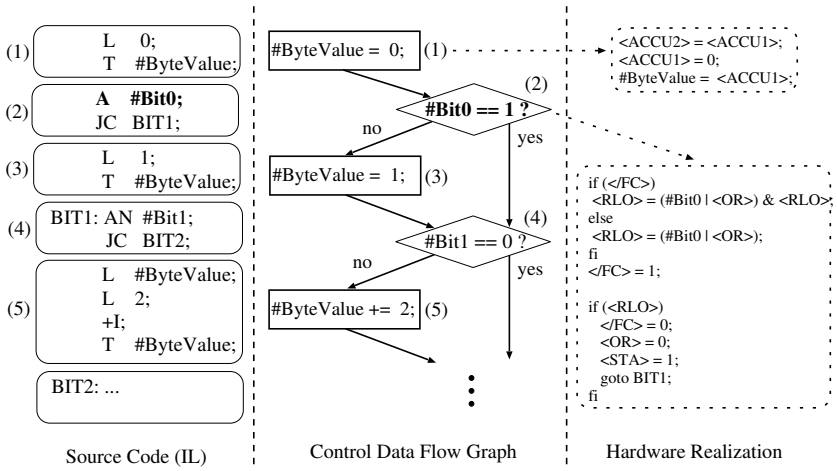

Fig. 1. Faulty program

4.1 Case Study

An example is used to illustrate the approaches. Figure 1 (left) shows a part of a combinational IL program. The program converts the eight inputs ($Bit0$ to $Bit7$) into a byte ($ByteValue$). The inputs ($Bit0$ to $Bit7$) are Booleans and the output ($ByteValue$) is a byte of bit-width 8. The program checks each input bit separately and adds the corresponding value to $ByteValue$.

The program is faulty at checking the value of $Bit0$ (Line 3, marked bold). Here, the logical AND (A) is used instead of the (correct) NAND operator (AN). Therefore, the least significant bit ($Bit0$) is not interpreted correctly and the condition in code block (2) inverts. The computed $ByteValue$ is either too high or too low with a delta of one.

To demonstrate the techniques, the program is partially defined and requires input values for $Bit0$ and $Bit1$, only. The primary output $ByteValue$ is the observation point in the example. Simulating the test case $Bit0 = 1$ and $Bit1 = 0$ leads to the faulty value $v_{faulty}(\text{ByteValue}) = 0$ whereas $v_{expected}(\text{ByteValue}) = 1$ is expected.

Figure 1 (center) shows an abstract CDFG for the IL program. The numbers in brackets are referencing the corresponding code block in IL (Figure 1 (left)). For example, the two instructions in code block (1) initialize $ByteValue$ with 0.

The details of the register changes of the CPU are exemplarily shown for the instructions in block (1) and block (2) on the right-hand side of Figure 1. The internal registers of the CPU are enclosed in '<' and '>'. For example, the two accumulators are denoted by '<ACCU1>' and '<ACCU2>'. As shown in Figure 1, the operations use and modify different registers. Thus, a fault candidate gives additional information to the internal registers where the fault is observable best. Moreover, a fine granular component model may further increase the accuracy of diagnosis by highlighting value changes on register level as potential fault candidates.

Table 1. Diagnosis results

Test case		Static Analysis						Dynamic Analysis						Correction-Based					
Bit0	Bit1	(1)	(2)	(3)	(4)	(5)	LOC	(1)	(2)	(3)	(4)	(5)	LOC	(1)	(2)	(3)	(4)	(5)	LOC
0	0	x	x	x	x	x	12		x	x	x		6		x	x			4
1	0	x	x	x	x	x	12	x	x		x		6	x	x				4
0	1	x	x	x	x	x	12		x	x	x	x	10		x	x		x	8
1	1	x	x	x	x	x	12	x	x		x	x	10	x	x			x	8
Single fault		x	x	x	x	x	12		x		x		4		x				2
Multiple fault		x	x	x	x	x	12	x	x	x	x	x	12		x				2

Table 1 compares the quality of the debugging results with respect to all possible test cases ($2^2 = 4$) (Column *Test case*). The rest of the columns give the diagnosed code blocks ($(1), .., (5)$) for the algorithms and the corresponding number of lines in the IL program (Column *LOC*). An 'x' marks the diagnosis with respect to the test case. The final diagnosis results for the consideration of all four test cases with respect to a single fault assumption and a multiple fault assumption is given in the last two rows. Note, code blocks are highlighted, but the diagnosis is itself performed on instruction level. An (un)marked code block means that none/all instructions within the code block have been marked as fault candidate by the debugging algorithm. None of the debugging algorithms return a subset of the instructions within a code block only.

The application of static analysis on the example is not very efficient. That is, any path from the primary inputs ends at the observation point $ByteValue$. All nodes are required for the computation of $ByteValue$ and no node is prunable. As shown in Table 1, all diagnoses return all instructions in the IL program as fault candidates.

Dynamic diagnosis is more accurate. The usage of the input stimuli increases the accuracy and some code blocks are pruned (see Table 1). Each test case activates different paths and the final diagnosis depends on the considered test cases. The final diagnosis with respect to a single fault assumption for test case 1 and test case 3 returns code block (2), (3), and (4). The consideration of all four test cases further prunes code blocks and returns code block (2) (the fault site) and code block (4) as potential fault candidates only. However, the cardinality of the fault is typically not known in advance and using all test cases is often not feasible for models with a large number of inputs. Additionally, any fix at (4) cannot fix the faulty behavior. The fault candidates in code block (4) have not been proved to fix the faulty behavior. The final diagnosis without any assumption on the fault cardinality returns all code blocks as potential fault candidates and does not help for debugging.

Correction-based debugging is more accurate in comparison to the first two analyses. Only fault candidates that fix the faulty behavior are returned. Code block (4) is accurately determined to be not a fault candidate. The results for each failure trace are more accurate and the final diagnosis with respect to all test cases returns the block with the original fault site only. The fault candidates are automatically of minimal cardinality and no assumption on the cardinality of the fault has to be made.

Table 2. Efficiency

	Initial	Static Analysis			Dynamic Analysis			Correction-Based		
model	|FCs|	|FCs|	Red.	Time	|FCs|	Red.	Time	|FCs|	Red.	Time
model-1	63	51	19.05	< 1.00	37	41.27	< 1.00	23	63.49	< 1.00
model-2	53	47	11.32	< 1.00	30	43.40	< 1.00	17	67.92	< 1.00
model-3	22	18	18.18	< 1.00	12	45.45	< 1.00	11	50.00	< 1.00
model-4	367	74	79.84	< 1.00	43	88.32	12.81	30	91.83	14.21
model-5	615	130	78.86	1.43	92	85.04	26.52	3	99.51	2.96
model-6	833	728	12.61	9.52	374	55.10	51.99	29	96.52	39.97

4.2 Industrial Software

The debugging algorithms are further evaluated on six industrial programs from the railway interlocking domain. The models have different complexity and use Boolean operations (e.g. logical AND and OR), arithmetic operations (e.g. 16-bit addition), and control flow statements (e.g. conditional jumps). The number of instructions in the models are ranging from 22 to 833 and the models have up to 23 primary inputs, 10 primary outputs, and 31 state variables.

A faulty implementation has been created by injecting a single fault manually in the IL model, e.g. replacing an operator *A* with an *AN*. The failure traces are obtained by applying SAT-based equivalence checking to a high level specification (see [18]). In all cases, the diagnosis is performed with respect to a single fault assumption and a randomly generated single failure trace. Components on instructions level are considered.

All experiments are conducted on an AMD Athlon 6000+ (3GHz) running Linux. The main memory was limited to 4 GB and the run time is measured in CPU seconds. The word level framework WoLFram is the back-end for the analysis [19]. ZChaff is the underlying SAT engine for correction-based debugging [14].

The experimental results are shown in Table 2. Column |*FCs*| gives the number of fault candidates and Column *Red.* denotes the reduction to the initial number of fault candidates in percent. The diagnosis time for the debugging algorithms is shown in Column *Time*.

All debugging algorithms are capable to reduce the initial number of fault candidates. For model-4 and model-5 more than 78% of the instructions are determined by all algorithms to not be a fault candidate. For other benchmarks, static analysis and dynamic analysis provide a rough diagnosis only, e.g. for model-1 and model-2. Here, simulation has advantages over static analysis, but correction-based debugging outperforms both algorithms. Correction-based debugging shows the best diagnostic resolution for all benchmarks. Thereby, the original fault site has been correctly determined by all algorithms. Fault candidates are efficiently detected by all methods and the complexity for a subsequent manual fix of an engineer is reduced. Regarding run time, all debugging algorithms require less than one minute for the diagnosis only.

In summary, all debugging algorithms support the manual debugging task by reducing the number of fault candidates. An engineer can focus on small parts of the whole program only. Automatic debugging gives hints for correction and helps to understand the faulty behavior. The computational overhead is low and the accuracy high.

References

1. M. Abramovici, P.R. Menon, and D.T. Miller. Critical path tracing - an alternative to fault simulation. In *Design Automation Conf.*, pages 214–220, 1983.
2. H. Agrawal and J.R. Horgan. Dynamic program slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256, 1990.
3. G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen. Towards the automatic verificication of PLC programs written in instruction list. In *IEEE conf. on Systems, Man and Cybernetics (SMC)*, pages 2449–2454, 2000.
4. K.-H. Chang, I.L. Markov, and V. Bertacco. Fixing design errors with counterexamples and resynthesis. *IEEE Trans. on CAD*, 27(1):184–188, 2008.
5. M. Davis, G. Logeman, and D. Loveland. A machine program for theorem proving. *Comm. of the ACM*, 5:394–397, 1962.
6. M. Fahim Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith, and M.S.Abadir. Debugging sequential circuits using Boolean satisfiability. In *Int'l Conf. on CAD*, pages 204–209, 2004.
7. G. Fey, S. Staber, R. Bloem, and R. Drechsler. Automatic fault localization for property checking. *IEEE Trans. on CAD*, 27(6):1138–1149, 2008.
8. C. Genz and R. Drechsler. System exploration of SystemC designs. In *IEEE Annual Symposium on VLSI*, pages 335–340, Mar. 2006.
9. A. Griesmayer, S. Staber, and R. Bloem. Automated fault localization for C programs. *Electronic Notes in Theoretical Computer Science*, 174(4):95–111, 2007.
10. Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *Int. J. Softw. Tools Technol. Transf.*, 8(3):229–247, 2006.
11. R. Huuck. *Software Verification for Programmable Logic Controllers*. PhD thesis, Faculty of engineering, University of Kiel, Germany, 2003.
12. J.P. Marques-Silva and K.A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. on Comp.*, 48(5):506–521, 1999.
13. W. Mayer and M. Stumptner. Model-based debugging - state of the art and future challenges. *Electronic Notes in Theoretical Computer Science*, 174(4):61–82, 2007.
14. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conf.*, pages 530–535, 2001.
15. O. Pavlovic, R. Pinger, M. Kollmann, and H.-D. Ehrich. Principles of formal verification of interlocking software. In *Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT)*, pages 370–378, 2007.
16. Siemens. *SIMATIC–Statement List (STL) S7-300 and S7-400 Programming*, 2003.
17. A. Smith, A. Veneris, M. Fahim Ali, and A.Viglas. Fault diagnosis and logic debugging using boolean satisfiability. *IEEE Trans. on CAD*, 24(10):1606–1621, 2005.
18. A. Sülflow and R. Drechsler. Verification of PLC programs using formal proof techniques. In *Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT)*, pages 43–50, 2008.
19. A. Sülflow, U. Kühne, G. Fey, D. Große, and R. Drechsler. WoLFram - a word level framework for formal verification. In *International Symposium on Rapid System Prototyping (RSP)*, pages 11–17, 2009.
20. Synopsys Inc. and CoWare Inc. and Frontier Design Inc. *Open SystemC Inititative*. http://www.systemc.org, 2008.
21. A. Veneris and I. N. Hajj. Design error diagnosis and correction via test vector simulation. *IEEE Trans. on CAD*, 18(12):1803–1816, 1999.
22. M. Weiser. Program slicing. *IEEE Trans. Software Engineering*, 10(4):352–357, 1984.