Towards Automation of Checklist-Based Code-Reviews

Fevzi Belli
University of Paderborn
Dept. of Electrical and Electronics Eng.
D-33095 Paderborn, Germany
belli@adt.uni-paderborn.de

Radu Crișan
University of Paderborn
Dept. of Electrical and Electronics Eng.
D-33095 Paderborn, Germany
radu@adt.uni-paderborn.de

Abstract

Different types of Code-Reviews (Fagan-style Code-Inspections, Parnas-like Active Design Reviews and Walkthroughs) have been found to be very useful in improving the quality of software. In many cases reviewers use checklists to guide their analysis during review sessions. However valuable, checklist-based code-reviews have the principal shortcoming of their high costs due to lack of supporting tools enabling at least partial automation of typical multiple appearing rules.

This paper describes an approach towards semi-automation of some steps of individual review processes based on checklists. The method proposed here is interactive, i.e. reviewers will be enabled to actualize, extend, and check the consistency and redundancy of their checklists. The basic idea underlying the approach is the usage of a rule-based system, adapting concepts of the compiler-theory and knowledge engineering, for acquisition and representation of knowledge about the program. Redundant and conflicting knowledge about the program under study are recognized and solved by means of an embedded Truth Maintenance System. As a result of fault diagnosis, rules for fault classification are used. Software reliability models are applied to validate the results of each review session. The approach has shown promising preliminary results in analyses of conventional C-programs developed in the automotive industry.

1. Introduction

Reviews have been generally accepted in the industry as a valuable tool for the development and management of software. Although the percentage of defects removed through review techniques varies widely (from about 30 to 85) in experimental case studies [16], it is still very high in comparison to other means for program analysis. According to their formality, reviews can be classified as formal reviews [19], walkthroughs [15] and inspections [14]. As stated in some practical investigations [9], most organizations intend to perform formal inspections, but the result of their efforts rarely goes beyond the level of reviews as defined in [18]. Therefore, in this paper we will consider reviews under this most general form. From the four types of review methodologies - individual, peer, public and group - only individual reviews will be considered. Since in many cases program documentations and specifications are either not available or obsolete, software reviews concentrate mostly on already compiled code free from syntactical errors. That is why we emphasize our analysis on source code.

Most industrial review techniques, whether formalized or not, have the usage of checklists which summarize previous experience in fault detection in common. Checklists for code-reviews are usually stated in question forms concentrating on hints, enabling the detection of a program fault or of a class of faults. During a review meeting the program under study is analysed line by line using the checklist as a guide. Detected faults are classified and removed according to their importance after the completion of the meeting, and the initial checklist is improved according to the recently detected faults. However, valuable and accepted in practice, checklist-based methods have been criticized for their high costs and for their nonsystematic approach. The ongoing costs of maintaining checklist-based inspections have been estimated as being approximately 25% of the total software development process costs [16]. The principal explanation of these high costs is given by the lack of supporting tools enabling at least their partial automation. Since such methods generally use project-specific checklists, their reusability for other projects is very poor. Further, checklists for code-reviews are often considered to be firm secrets, an exchange of experience between reviewers being thus impossible.

This paper presents an approach to semi-automate some steps of individual checklist-based code-reviews attempting to both reduce the costs of implementing reviews and assist reviewers in improving their work. The aim is thus
not to replace the reviewer, but to support him in rationalizing his tasks. Therefore, an interaction with the reviewer is required. Since in many cases the reviewer is poorly supplied with documentation helping him to understand the program under study, a practical analysis approach has to provide such documentation by means of reverse engineering. As a starting point for the analysis of a given program, the reviewer is supposed to use an initial checklist consisting of questions hinting to potential failures in resembling programs. By transforming these questions into rules of a deductive knowledge-base, the approach aims to model the way reviewers maintain, augment and prove the consistency of their checklists. Beyond the intuitive way in which reviewers analyze their programs, the system presented here intends to permanently assess the checklists’ success in revealing failures by means of software reliability modeling.

Practical reasons brought us to the idea of trying to support the review process as much as possible by means of a dedicated software tool. Our task within a project with a major German automobile company was to assist our partner’s review teams in improving the quality of their software. We have begun with fully manual reviews as performed by our partners, and we have gradually introduced automated checks in the implemented review methodology. Whilst our first review checklists had only five automated general checks, those used at present contain over 80 general checks, and over 20 project specific checks. The length of the review sessions, showing the method’s efficiency, has been constantly decreasing, by about 5% in the early stages, to over 40% in present. In order to take advantage of the facilities offered by other department’s tools we have embedded the analysis environment in the product assurance environment of our department [7].

The paper is organized as follows. The general methodology of the approach is described in section 2. The present stage of implementation of the analysis environment based on this approach is presented in section 3. Section 4 discusses future directions of work. Section 5 gives an overview of related work and concludes the paper.

2. Using reviewers’ knowledge for program analysis

Investigations on human understanding [13] have stated that when analyzing a program, experts combine systematic methods based on layers of understanding as modeled in compiling techniques [1] with heuristics as used in knowledge engineering. This principle has been used in this approach by adapting techniques specific to compiling like parsing, or syntax translation, to knowledge engineering techniques like top-down search, knowledge acquisition or truth maintenance.

2.1. Program post-documentation

Before beginning to analyze programs, e.g. to detect failures or anomalous program constructs, reviewers have to understand the function and the logic of the program under study. The analysis environment implemented using the general methodology presented in this section provides the user with the following documents synthesized by reverse engineering:

- program flow graphs enabling the reviewer to understand the program’s control structure and to generate test cases (testing is the subject of another paper submitted by our group to this conference [12]),
- decision tables enabling the understanding of the program’s logic,
- Halstead and McCabe program metrics.

Beyond their main purpose as means for program understanding, these documents can also facilitate detecting failures. Principles on how flowcharts and decision tables can reveal failures are given in [5].

2.2. Using checklists for knowledge representation

Good software engineers try to reuse parts of solutions with which they are already familiar. They establish a set of rules helping them to detect failures and transform them into checklists used to guide their analysis. In our project we used checklists consisting of two classes of checks:

- general checks hinting on potential failures common compilers and lint-like tools are poor in detecting, or ambiguous program structures easily leading to failures,
- project specific checks derived from knowledge about a particular configuration and specification requirements.

A segment of the much larger checklist used for analyzing C-programs is depicted in Figure 1. Resembling full-length checklists can be found in [15], [5].

2.3. Hierarchical dependencies between checks

Little attention has been awarded to the relationships between checks in industrially implemented checklists. Being informal, the probability of some checks subsuming others is very high. In order to illustrate this, let us consider the example from Figure 2 extracted from a larger C-program analyzed by us within an industrial project.

The programmer’s intentions were obviously those of calculating two numerical values in lines 5 and 6 by using
General checks
G1. Are constants declared with defines?
G2. Can any expansion of macros lead to side effects?
G3. Are all macros correctly paranthesized?
G4. Can unparanthesized macro arguments cause wrong results?
G5. Do all printfs have file arguments?
G6. Are all argument types in printfs matching conversions?
G7. Are equality tests used on floating-point numbers?
G8. Is there any unintended debug code left?
G9. Does the program have a specific exit value?

Project specific checks
P1. Are existing buffers ever too small to hold their contents?
P2. Are there any user-defined functions violating preconditions?
P3. Is the loop nesting depth not excessive?

Figure 1. Segment from a typical checklist

the macro defined in line 1. Although neither customary C-compilers nor lint-like tools give notice of any fault in this program, the results differ from the expected ones. By applying check G3 from Figure 1 to this program, the incorrect macro expansion in line 6 indicates insufficient surrounding of the macro with parentheses. After removing this fault by adding a new pair of parentheses to the macro’s definition, the fault checked by G2 is still persisting. This implies that the check G2 has a higher “priority” than G3. Such observations lead to an inherent fault model of the program in order to capture the internal dependencies between faults. This kind of fault modeling is also motivated by psychological studies [20], stating that expert programmers represent a program in their memory as a partially ordered list of propositions, and that the probability of detecting faults is a function of the fault’s depth in the propositional hierarchy.

Figure 2. C-program example

2.4. Rule templates for fault modeling

Rules, and especially rule-chaining mechanisms known from predicate logic provide good support for representing dependency hierarchies. A rule generally consists of premises and actions. Premises describe situations in which actions (e.g. instructions for changing the current state) are to be performed. In the case of rule-based systems which analyzes programs, relevant information about the program under study can be collected by integrating source code instrumentors into rules’ premises. Similar to the way reviewers analyze their programs, the instrumentor scans the program line by line and marks relevant places in the program according to the pattern specified in the premise. This information can then be asserted in a knowledge base with observations about the program (facts). Facts have to match certain patterns specified by rules and can therefore be seen as instantiations of these rules. Keeping track of all instantiations enables the implementation of truth maintenance mechanisms to ensure the consistency of knowledge about the program. This leads to the following template for rules transformed from the initial checklist:

Rule template using hybrid rule chaining and embedding truth maintenance within the verify predicate:

\[ \text{rule-name} \leftarrow \text{not}(\text{fact}), \text{sub-rules}, \text{verify}(\text{fact}), \text{action}. \]

Figure 3. Rule template

In the general form from Fig.3, before firing, rules check if any rule instantiations are already available in the database. Unnecessary repetitions of rule firings (system state change due to action parts of rules) are thus prohibited. After each sub-rule in the premise part of the rule has fired, the newly created fact is verified, and if it doesn’t contradict to any existing fact (e.g. it doesn’t match any pattern of truth maintenance rules) it is appended to the knowledge base. In a truth maintenance rule, pattern specifies a group of facts which constitutes an inconsistency. If the pattern successfully matches against facts, an inconsistency among them has occurred and an action enabling change of corresponding facts’ states is invoked. The hybrid rule-chaining principle embedded in this template is a combination of the fundamental forward and backwards rule-chaining mechanisms [3]. In forward chaining mechanisms, rules with premises matching facts from the database are fired, leading to possible changes in the initial database of facts. In backward chaining mechanisms, those rules fire whose action parts includes a given goal. With the hybrid...
chaining mechanism, rules are tried in order as with backwards chaining, but each rule is used in a forward chaining (modus ponens) way to assert new facts. Using this rule form enables integration of both dependency structures and truth maintenance mechanisms early in the knowledge acquisition stage.

2.5. Assumption based truth maintenance

A particular form of Truth Maintenance System (TMS) called assumption-based TMS (ATMS) [10] was used. The following template for facts is the foundation of the embedded ATMS mechanism:

```
Fact template containing lists enabling ATMS-mechanisms:

 fact_name ([instantiations], [assumptions], [state]).
```

**Figure 4. Fact template**

The first list in the template from Figure 4 contains program instrumentation information gathered by parsing and matching the rule pattern whose firing created the fact. In the second list antecedent facts used to create the fact are tagged by the system building thus a context of assumptions under which the fact is believed. Every fact is in one of two states: in (believed) or out (not believed).

```
State change ment
```

**Figure 5. Example of using ATMS-rules**

The Truth Maintenance System ensures that only facts with well-founded support (believed assumptions) are in. Figure 5 shows how the facts' state change mechanism works. If the system receives new information about the program, leading by pattern-matching with a ATMS-rule, to the conclusion that a contradiction occurred between facts F2 and F3, then all facts having both F2 and F3 in their assumption list will be marked out.

To illustrate this in terms of program analysis, let us consider the example of a rule creating in its action part instances containing information about the number of nested loops. Supposing that the resulting number of nested loops is 4, and if a ATMS-rule is matched indicating a contradiction between this number of nested loops and other requirements regarding a particular loop construct, then all facts basing on both facts are marked not believed. The permanent possibility to change the label value of assumption-sets is a step towards non-monotonic reasoning about programs. The theoretical foundations of non-monotonic extensions of ATMS systems are given in [10].

In terms of program analysis, following contradictions between facts can occur:

- automatically, when creating new facts by rule firing,
- interactively, when the user supplies new information about the program in form of facts.

In the second case, the system establishes, using data abstraction, if the supplied information uncovers any contradictions among existing facts. If such contradictions exist, the system stores the abstracted pattern of the contradictory facts in a new ATMS-rule.

2.6. Internal fault model

Figure 6 shows an example of a fault-model built according to the above mentioned ideas.

The initial fault model is due to the initial checklist (as depicted in Fig.1) and is represented in the left side of Fig.6. In further review stages the user may augment the checklist by adding new checks (e.g. new rules in the analysis program) hereby modifying the internal hierarchy. The system reacts by checking the modified knowledge-base for: unfirable rules, duplicate rules, redundant rules, ambivalent rules and circular inference. Inferring explanations for the existence of a certain fact from existing rules is facilitated by the rules' template and can be deduced from the fault model. Strategies containing decisions on which rule to invoke next when more than one rule may be applicable are embodied in meta-rules. Since ATMS rules consist of knowledge about relationships between other rules, they also belong to this level.

The most difficult task the analysis program is to deal with is handling new facts provided by the user. Giving an explanation for a new fact (as being the case in program analysis when the user supplies the system with information about particular symptoms in the program) is an abductive process. Abductive reasoning is not entirely "logically correct" and involves making hypotheses about the most plausible diagnosis (e.g. rule inference) and discriminating among them basing on plausibility or probability. Following plausibility criteria have been investigated:

- **minimal cardinality** stating that explanatory hypotheses with the fewest number of hypothesized components are to be preferred,
• irredundancy involving that those hypotheses are to be prefered which have no proper sub-hypotheses explaining the new fact

• probability criteria estimating the probability of each inference route.

Meta-rules using the first and the second criteria from above have been included in the system. The greatest problem in handling such reasoning mechanisms occurs regarding the very large number of possible hypotheses heavily increasing the computational complexity of the system. In terms of program analysis this process of discriminating between hypotheses adapts reducing computational lattices in data-flow analysis by compilers using code optimization [1]. Without solving the complexity problems emerging from searching in large spaces of possible combinations, ATMS-based approaches provide valuable support due to their inherent mechanism for environment lattices’ reduction, as depicted by Figure 7.

The right part of the figure shows the environment lattice (set of all possible combinations of facts) corresponding to the hierarchical structure from left. The edges between sets of facts represent subset-superset relations. If the assumption set of R2 (F3,F4) is found to be inconsistent, then all the sets of assumptions that are supersets of R2 (marked with shaded areas in Fig7) are marked out. The initial set of hypotheses involved in diagnosing data flow faults has been reduced using this principle.

2.7. Using fault models for fault classification

Classification of faults due to their severity is a very important organizational request during review processes [5], [22]. Since fault classifications are strongly depending on the program environment and on the reviewer’s strategies in evaluating faults, in this approach we use the interaction with the user to supply the system with rules establishing:

• the number of severity degrees he would like to use,

• subsumption rules between severity grades and checks.

This knowledge is internally represented as meta-rules in the knowledge-base and used in the classification phase. Apart from this interactive part, the system has to assess a fault’s severity automatically when:

• classifications for certain faults are either not existant or irrelevant,

• new rules or new facts are added to the model.

Faults having very serious effects like causing a system’s crash are generally known to the user and are assumed to be already classified. In solving the above situations the system takes advantage of the existence of a fault model from stages prior to the fault classification one. The embedded ATMS-mechanism supports this by determining a list of facts with known severity degrees conditioning the fact’s creation.

Following strategies for fault classification have been studied:

• if the detected fault is presumably having little influence on the program’s behaviour, then the new fault’s severity can be assessed by computing mean values of severity degrees of facts conditioning its appearance,

• if the fault’s effect is of greater importance, simple computations of severity degrees being irrelevant,
thus the system tries to match the pattern of the most resembling rule with known severity degree using heuristic search,

- if this second method doesn’t give acceptable results, the severity of the fault is left unknown and is to be assessed during future review sessions according to the number of fault’s occurrences.

2.8. Validation

Reviews being reiterated according to the obtained results, the reviewer should be supplied after each session with information about the efficiency of his checks. Since program metrics are not widely accepted as means for result validation, software reliability models have been preferred for this purpose. The PRORool environment [4] interfaced with the analysis environment and having several reliability models already implemented provides therefore valuable support. After each review session the reviewer may add into a file characteristics of new found faults like date of fault discovery, fault severity degree, probable fault cause, and date of closing fault removal process. Using the interface, PRORool handles this file as an input, estimates initial models’ parameters like number of inherent faults or fault intensity decay, and finally depicts the fault intensity curve for the selected models. Knowing the estimated number of inherent faults and the shape of the reliability curve, the reviewer can both estimate how many faults he still has to detect and approximately how much time he needs to reach a given reliability level. The used models were all optimistic, meaning that it was assumed that during the fault removal no new faults were generated. This implies that the program reliability was presumed to increase during the review process. The Hyperexponential Model, the Generalized Goel-Okumoto Model and S-shaped Models (Yamada-Ohba) were shown to give the best correlation results between estimated and real-life data in a project using a similar methodology [6]. Estimations resulting from applying pessimistic models and especially from combinations of optimistic and pessimistic ones have yet to be verified.

3. Implementation issues

The general methodology presented in the previous section has been implemented in an environment for C-program analysis.

Following advantages offered by Prolog [3] have motivated us to use this language for the implementation of the analysis program:

- a Prolog program is itself a knowledge-base consisting of rules and facts thus offering the most compact representation for the hierarchical model,
- Prolog predicates like assert or retract enable dynamic modifications of the database as requested by our approach,
- Prolog’s implicit backtracking mechanism provides good support for implementation of hybrid rule chaining,
- Prolog’s pattern matching and unification features support data abstraction,
- the great level of abstraction of this programming language enabled us to keep the analysis program into acceptable dimensions.

3.1. General structure of the analysis environment

The analysis environment consists of five components: instrumentor, translator, ATMS-rule generator, consistency checker and program analyser. Figure 8 depicts the system
level architecture. The translator and ATMS-rule generator transforming informal informations provided by the user into predicates of the analyse program are not yet automated.

Figure 8 also illustrates how the three stages of the review process: initialization, analyse and augmentation were simulated within the system. During the initialization phase, the system assists the user in understanding his program as described in subsection 2.1, and translates the initial checklist from verbal question form into rules of the analysis program. In the analysis phase the instrumentor performs static analysis of the source code (in the existing version only by parsing) as described in section 2.4 and generates the instrumented code. The program analyser reestablishes the intern hierarchy as described in subsection 2.4, verifies matching contradictions as described in subsection 2.5, classifies eventually emerging faults like described in subsection 2.7 and produces an analysis report. Finally, in the augmentation phase, new rules or facts supplied by the user are transformed into new ATMS-rules if detecting new faults. The consistency checker verifies the consistency of the modified knowledge base and establishes diagnoses for new facts as described in subsection 2.6.

3.2. Implementing and using the analysis program

The following Prolog predicate from Fig.9 was used to implement rule G2 from Fig.1.

The rule template applied here has been described in subsection 2.4. Line 2 of the above program tests the existence of any fact having the name macr_increment, and if no such fact is found calls two sub-rules macros.parentheses (G3) and respectively increment_possible. In line 5 the assumptions underlying the facts created by the two sub-rules are appended, and the fact is created in line 6. In line 7 contradictions between the new facts are verified, and if no such contradiction occurs the severity of the new fault is assessed (in line 8) according to its list of assumptions. Finally, the fact is added to the database in line 9.

Figure 10 gives a sample session showing how the system works. The C-program example and the hereby used
checklist have been presented in subsections 2.2 and 2.3.

After having activated the graphical means offered by the environment in the PROGraph menu as shown in subsection 2.1, the reviewer's next analysis step is to check specific points from the checklist relevant to the program. Therefore, by activating the View button from the Checklist submenu, the program to be analysed (in this case the C-Program) as well as the analyse program (Prolog program) are displayed. According to the predicates of the analysis program and to his purposes, the user can select all or some checks from the list. By activating a checkpoint the corresponding rule in the Prolog knowledge base fires, creating a new fact if a fault is detected. By activating one of the buttons of the Report submenu the significance of the detected failure is reported in a lint-like warning style. Figure 10 summarizes the corresponding windows generated during these stages.

The prototype system was embedded in our department's product assurance environment, including the test environments PROTest [7] and PROCom [12] and the reliability assessment environment PRORool [4].

![Figure 10. Windows from the Checklist menu](image)

![Figure 9. Prolog predicate for rule G2](image)
4. Future work

The approach presented in this paper is still evolving in terms of implementation and of its evaluation in practice. Future work is clearly needed in the following directions:

- Improvement of the interaction with the user. In the present stage of implementation, user defined checks cannot be automatically transformed into Prolog rules for the analysis program. We are currently working on the integration of a user-supporting language template enabling the user to take no notice of the internal analysis program.

- Developing further rules for fault classification aiming to reduce as much as possible the interaction with the human analyser.

- Evaluating the "depth" of program code instrumentation requested by the analysis program. In present, source code instrumentation is done by means of top-down parsing algorithms aiming to detect a finite number of semantic faults by searching for predefined potentially faulty language constructs. The treatment of low-level faults from lexical and syntactic levels of understanding is left to the compiler. Clearly delimiting these fault layers is difficult. Valuable information from lexical layers like variable symbol tables are not supplied to the user by the compiler. Grammar-based analysis for certain classes of faults may improve the detecting abilities of the system.

- Studying specific forms of variation of rules. Evaluating how refinements or augmentations of rules affect the general performance of the system.

- Optimizing the internal model representation. The usage of object-orientation and frames is suggested by resembling hardware applications. The effect (whether positive or negative) of implementing such methods in our systems is to be evaluated.

- Selecting, adapting and permanent training of a combination of software reliability models showing the best results in review process modeling.

- Evaluating the size and the granularity of checklists.

5. Related Work and Conclusions

While industrial-strength automated review tools do not exist, a number of attempts have been made to find effective solutions to the problem. The most numerous are experimental studies dealing with management of relevant code review information. Such studies range from direct applications of basic principles underlying review techniques [17], to detailed project case studies [23]. Recent papers from this area of research [24] describe review-support tools designed for software development managers. Such tools are little more than syntax-directed editors providing little automated support for program analysis.

In contrast to the just mentioned class of investigations, other research projects [11] emphasize on the analysis process itself and not on its results. Plans, first introduced formally by Rich, were used in most of these approaches to guide the acquisition and representation of knowledge about the program under study. Some tools using plans as knowledge representation like DESIRE [8], RHET [2] or Cake [21] have embedded justification-based TMS. Cake, the most comprehensive tool under its class, requires seven analysis layers for knowledge representation. As stated by the developers, no methods are given for classifying detected faults.

Being still in the implementation and evaluation stage, the analysis environment based on the approach described in this paper wants to take advantage of both organizational strengths of the first class of investigations and of systematic methodologies specific to the second class mentioned above. We believe to be moving in the right direction towards automation of individual program reviews by:

- reducing the layers of internal program representation to a minimal size,

- using checklists from generally accepted review techniques instead of plans,

- enabling knowledge augmentation based on non-monotonic reasoning,

- permanent classification and validation of results,

- providing a concise framework enabling the human analyser to rationalize his work.

The presented analysis environment can be effectively deployed together with the other product assurance tools developed by our group. By including it before the testing stages (as performed with PROTest and PROCom), and in conjunction with PROTool for result validation, the number of detected faults can be significantly increased.

Acknowledgments

This work has been supported by Mitsubishi Electric Europe, project Modelling and Implementation of a Flexible Robot Cell.

Our thanks are due to the anonymous reviewers for valuable hints and comments.
References