ICICLE: Intelligent Code Inspection in a C Language Environment

V. Sembugamoorthy & L. Brothers
Bellcore
Room: RRC-1H225
444 Hoes Lane
Piscataway, NJ 08854

Abstract
ICICLE (Intelligent Code Inspection in a C Language Environment) is an intelligent assistant that augments manual code inspection by providing a rule-based static-debugging tool, hypertext-based browsers for referring to various kinds of knowledge associated with code inspection (e.g., the application domain knowledge and the knowledge of the programming environment), a powerful human interface for preparing comments on the code under inspection and a CSCW (computer supported cooperative work) platform for conducting code inspection meetings. ICICLE has been prototyped on UNIX workstations using a commercial multi-paradigm expert system shell and a window package for implementing computer supported cooperative work. The initial prototype has been significantly enhanced based on the feedback from extensive videotaped usability testing sessions which used a real world application. It is currently being used for live code inspection by one of our initial user organizations. This paper describes the goals, the approach, the architecture, the main components, the usability testing and the future directions of ICICLE.

Introduction
In 1976 Fagan[6] formulated a rigorous manual technique for inspecting software design and code. Since then this technique, with some modifications, has been applied for code inspection in many organizations including Bellcore, IBM[6] and AT&T Bell Laboratories[1]. This rigorous method of using the knowledge of a team of expert developers, designers and domain experts to inspect programs has been found to result in more error-free and understandable software. As a result, code inspection has either replaced or supplemented traditional reviews and walkthroughs [2]. Seeing its benefits through pilot experiments, Bellcore's Software Technology and Systems sector has recently been directed to perform code inspections on 100% of newly developed software.

Though very effective, code inspection is painstaking and requires significant investment of time and human resource. The main reason is that it is a highly knowledge intensive activity. In other words, a code inspector applies a large body of different kinds of knowledge such as design knowledge, programming knowledge and domain knowledge. In general, it is very difficult to find a code inspector who is an expert in all these kinds of voluminous knowledge. Often even experienced code inspectors suffer from cognitive overload. If we can help to solve these problems by computerizing the problem-solving tasks which underlie code inspection as well as the associated knowledge, then the manual code inspection process will become less painful, and more efficient and effective, resulting in the improvement of the quality of software products and productivity of programmers.

With the aim of augmenting and improving manual code inspection, we have interviewed code inspectors, observed and analyzed real-world code inspections, and developed a system called ICICLE (Intelligent Code Inspection in a C Language Environment). It runs on Sun Workstations using the ART multi-paradigm expert system shell, Common Lisp and C languages, and the XView window package. ICICLE was put through 12 hours of videotaped usability testing over the course of two weeks. The testers were developers from one of our initial user organizations. Based on the feedback from this testing we have enhanced ICICLE considerably. This organization is currently using ICICLE for several real code inspections. In

**Sun Workstation is a registered trademark of Sun Microsystems, Inc.
***ART is a trademark of Inference Corporation.
****XView is a registered trademark of Sun Microsystems, Inc.
*****It is the policy of Bellcore to avoid any statements of comparative analysis or evaluation of vendors' products. Any mention of products or vendors in this paper is done where necessary for the sake of scientific accuracy and precision, or for background information to a point of technology analysis, or to provide an example of a technology for illustrative purposes, and should not be construed as either positive or negative commentary on that product or that vendor. Neither the inclusion of a product or a vendor in this paper, nor the omission of a product or a vendor, should be interpreted as indicating a position or opinion of that product or vendor on the part of the author(s) or of Bellcore.
order to determine the efficacy of ICICLE, some of these inspections will also be performed using the conventional manual approach.

As mentioned earlier, a code inspector is expected to possess many different kinds of knowledge such as requirements specification, system design, domain knowledge, knowledge of the programming environment, debugging knowledge, etc. Since the task of acquiring, representing and using all these kinds of knowledge is a huge problem, we are currently focusing on debugging knowledge, domain knowledge and knowledge of the programming environment. Due to its widespread use in Bellcore as well as elsewhere, we chose to construct a code inspection system for programs written in the C language and running under UNIX system. However, ICICLE's modularity should make it easy to extend our work to other languages and operating systems.

We have the following four principal goals in constructing ICICLE, two practical and two theoretical:

- **Our short-term practical goal is to apply the available technologies and construct an immediately useful code inspection environment for Bellcore and its clients.**
- **The long-term practical goal is to use our ICICLE experience in integrating AI (Artificial Intelligence) and software technologies to construct intelligent assistants for other phases of software development cycle, especially reuse-based software design and maintenance.**
- **Our short-term theoretical goal is to evolve ICICLE as a knowledge-rich test-bed to conduct experiments and determine how code inspectors and software maintainers understand code.**
- **Our long-term theoretical goal is to construct a framework for intelligent assistants for different phases of software development.**

This paper is written from the viewpoint of our short term practical goal.

**A Conceptual Analysis of Code Inspection**

In general code inspection includes the following steps:

- **Scheduling:** Once the code is ready for inspection (i.e., sometime after the first clean compilation but before unit-testing), the code inspection coordinator of an organization selects inspectors, sets the date for the code inspection meeting and distributes the source code and other related documents such as requirements and design specifications, if any.
- **Comment Preparation:** Using the distributed materials, code inspectors individually analyze the code and prepare comments relating to bugs, and deviations from coding standards, requirements and design specifications.
- **Code Inspection Meeting:** On the scheduled day, the inspection team meets, discusses the comments prepared earlier, analyzes the code wherever necessary, and finalizes the list of comments. Also certain statistics such as the number and types of errors found, time spent, etc., are obtained. This data is used for monitoring the effectiveness of code inspection in the organization.
- **Post-Inspection Activities:** Comments finalized at the inspection meeting are given to the developer for action. The modified code is reinspected, if necessary.

Of these four steps in code inspection, the most difficult and arduous ones are the comment preparation phase and the code inspection meeting. ICICLE is specifically designed to help code inspectors in both of these activities.

**Comment Preparation**

During comment preparation, code inspectors employ informal static debugging techniques to detect coding errors. The inspectors also work to understand the code and check whether the code meets the requirements and design specifications. Moreover, in order to reduce the possibilities of coding errors as well as to ensure uniformity, portability and understandability of code, software development organizations often require developers to adhere to certain coding standards. Checking violations of these coding standards is another important activity during comment preparation. During these activities, code inspectors employ various kinds of knowledge such as debugging knowledge, application domain knowledge, knowledge of the programming environment. Also they make use of cross-referencing and other kinds of information obtained using various static analysis tools. A computer-aided code inspection environment should provide tools to aid static debugging, checking coding standards violations, browsing through various knowledge chunks mentioned above, and above all integrate these tools in a user friendly human interface.

**Code Inspection Meeting**

Typically code inspection teams consist of three or more inspectors. Three of them are assigned one of the following specific roles: Moderator, Reader, Scribe. The Moderator orchestrates the meeting, resolves any conflicts and also collects inspection statistics. The Reader reads the code aloud and directs the attention of other inspectors to various segments of the code. If any inspector has any comments on the segment pointed to by the Reader, he or she proposes
them. The comments are then discussed and finalized by the team. The Scribe records the finalized comments. The author of the code also attends the meeting and answers the questions of the inspectors. A good computer-aided code inspection environment should support this cooperative effort of inspectors by computerizing the communications between inspectors and the tedious manual recording activities of the Scribe.

Integrating Multiple Technologies for Code Inspection

As mentioned in earlier sections, one of the principal goals of ICICLE is to augment the manual comment preparation and inspection meeting phases of code inspection. To accomplish this goal, we reviewed software and AI technologies and found that certain classes of errors and warnings can be automatically flagged by bug-detection tools such as LINT[11] or DAVE[15]. Also, by acquiring and representing the heuristic rules that expert developers use, one can detect more complex errors. Rule-based systems, due to their ease modifiability of the rule base and the availability of explanation facilities, provide an excellent framework to incorporate the knowledge-rich environment of ICICLE, due to the lack of easily accessible knowledge chunks mentioned above is one of the main problems in manual code inspection, focusing on knowledge representation issues will result in systems that will be immediately useful to code inspectors. Second, capturing the knowledge chunks will be a significant step towards constructing reasoning systems later within the intelligent assistant paradigm, because using the knowledge-rich environment of this approach one can more realistically analyze and automate various strands of reasoning underlying program-understanding. Finally, the computerized knowledge base can also be used for other purposes such as training new members of a project, maintenance, etc.

A number of knowledge representation schemes have been constructed for various aspects of software development such as domain knowledge[4, 8, 7, 3] and program design at many levels[16, 18]. However, when applied to real world applications, these knowledge bases turn out to be very large and acquisition of such large knowledge bases using AI technology is currently very difficult.

Hypertext is a recent technology used to manage complex information in the form of "cards" or pages of text and images. It provides facilities to store information in, and navigate through, a rich structure using a powerful human interface. Hypertext is not considered to be a useful knowledge representation scheme from the viewpoint of AI since a reasoning system cannot currently use unrestricted natural language texts or images to reason. However, as long as the information in a knowledge base will be used by people rather than reasoning programs, hypertext is currently the best technology to acquire and represent large amounts of information such as domain knowledge or programming environment knowledge. Since, for reasons discussed earlier, we shall focus on knowledge representation rather than reasoning aspects of program understanding, we have decided to use hypertext technology to represent domain and other knowledge. However, when we understand the reasoning aspects of program understanding better, we shall use suitable AI knowledge representations to represent domain and programming environment knowledge.

Based on the above analysis, we have designed an architecture for ICICLE. As shown in figure 1, the architecture integrates the following technologies in a flexible and modular framework.

- Human interface technology for effective presentation of information and interaction with the user
- CSCW (Computer Supported Cooperative Work) technology for supporting the cooperative effort of code inspection meetings
- Software technology for static-debugging and providing useful program-understanding aids such as cross-referencing information
Expert system (rule-based) technology to represent expert developer's knowledge for detecting programming errors and violations of coding standards

Hypertext technology to represent knowledge that code inspectors need to understand programs

ICICLE's architecture is very modular. Since all the analysis tools are interfaced with the human interfaces through text files, it is easy to add new analysis tools (e.g., complexity analysis tools). Also, even if an application does not have a well-defined domain and other kinds of knowledge, ICICLE is designed to work, though with reduced functionality.

Human Interfaces
The ICICLE human interface was implemented using the XView toolkit for the X Window System. The use of X permits easy operation of applications on remote displays, enabling the CSCW (computer-supported co-operative work) functionality of the interface. ICICLE can operate on any display device that runs X so long as at least one machine can run the XView client.

The human interface has two modalities of operation: comment-preparation and code-inspection meeting. The latter mode subsumes the functionality of the former, with the addition of groupware to automate the cooperative effort of the meeting.

Comment Preparation Functionality
User Activity in the Comment Preparation Phase The basic activity of the user in this phase is a traversal of the mod-

* X Window System is a trademark of the Massachusetts Institute of Technology.
/* Initialize the logical lines. */
initlltoble();

/* Initialize the grouping levels in the logical level-seq-tbl */
initlmq();

/* Initialize the fcif structure to the cutout of the system */
The Figure 2: Code Window

Interface Features  Figure 2 shows the basic state of the human interface. The large central area is the code window; this window displays the actual source code of a file in the module to be inspected. In addition, the code window displays line numbers, comment symbols, and other decorations and markings meant to annotate the source code. The code window can be split horizontally into two subwindows, each of which can be used to independently traverse the module. This is convenient for viewing a function call and its definition simultaneously, among other purposes.

The symbols to the left of certain line numbers in the code window denote the presence of comments associated with those lines. The leftmost of two symbols designates the "state" of a comment, which may be "Deferred" (i.e. the
user has not yet dealt with this comment, or has put off any decision regarding it), "Ignored" (the user has decided that this comment is incorrect, or is otherwise inappropriate to raise at the code inspection meeting), or "Transferred" (the user has chosen to discuss this comment at the meeting).

The second symbol designates the category of the comment. These categories include: Errors, Warnings, Hazards, Advisories, and Standards Violations.

The user can manipulate the state of existing comments, such as those suggested by LINT, our expert system, or other analysis tools, and can also add his or her own comments. Figure 3 shows a comment window, which was raised in response to a user's mouse click on a line. This window has buttons to change the state of a selected comment, all the comments on the same line, and all comments that have the same text throughout the module. In addition, the user can type in a new comment through this window.

ICICLE also supports user interaction with source-code
objects. As described in figure 4, by mouse clicking on some object such as a variable, parameter, function call, etc., the user can call up whatever information the system has about the given object. In the current version this mainly comprises cross-referencing information. For example, by clicking on an instance of a function call, the user can traverse the module to the site of the function definition, effect pushing the location onto a stack. By pushing and popping function definition locations, the user can effectively traverse the static scoping of a module, greatly reducing the need for paper listings.

In addition to information about the source code itself, ICICLE provides a hypertext-based knowledge browser to aid program understanding. The knowledge browser can be invoked on any string in the source code, or on a string typed in by the user. The hypertext "card" shown in figure 5 gives the basic definition of a "service order" (a concept in the domain knowledge base). The user can follow links to subtopics which expand on the definition and answer questions about the concept. By following links to related topics, the user can navigate a network of information pertinent to the domain.

An indexed browser for Unix and other manual pages is also available. When the user selects a string such as a system function call, ICICLE alerts the user to the presence of a "man page" on the topic. "man pages" can be viewed through a scrollable window popped up for that purpose.

**Code Inspection Functionality**

**User Activity in the Code Inspection Meeting Phase**

In this phase, the Reader controls a traversal of the code which is similar to the traversal undertaken by the individual inspectors during their private comment preparation phases. During this traversal, comments are proposed and disposed of, and recorded by the Scribe; at the end of the traversal the Moderator terminates the session and generates various statistical reports.

**Interface Features**

During the code inspection meeting which currently takes place in a single meeting room*, each inspector has an interface, which is linked to that of all the others. Each inspector can see his or her own comments as prepared in the comment preparation phase, and can use all the subsystems available during that phase.

The Reader has a special function in this phase. All the other inspectors' code windows are slaved to that of the Reader, so when the Reader scrolls or visits another file, the other inspectors' code windows synchronously scroll as well. If other inspectors have more than one code window, only one is slaved to the Reader. This function obviates the need for paper listings at the code inspection meeting.

At any time during the reading of the module, any of the inspectors can asynchronously interrupt by proposing one of their comments to the committee at large.** A proposal window containing the text of the comment appears on all the other inspectors' workstations.

The Scribe's special function comes into play when the discussion of a proposed comment concludes. The Scribe's proposal window contains controls to accept or reject a comment, and to set various metrics relating to the comment as required by formal code inspection procedures. In the ordinary code inspection meeting, the Scribe's job is considered the most arduous, as all comments have to be recorded, along with the metrics, with paper and pencil. This functionality greatly relieves the burden on the Scribe.

When the meeting is concluded, the Moderator can terminate all the ICICLE activity and generate the error reports that are required by the code inspection procedure. Previously, this had to be done by hand, as well.

A more detailed description of the human interface for inspection meeting appears in Brothers et al[17].

**Rule-Based Static Debugging**

The first component of our rule-based static debugging system is a yacc-based C grammar parser, which can efficiently accept C code and output a lisp-readable annotated parse tree. This parser is by no means a complete parser as in C compilers. As code presented for inspection must compile without errors, the grammar and actions are fine-tuned to provide exactly the information required by the rules of the expert system component.

The second component is a rule-based system written in ART, a commercial multi-paradigm expert system shell, along with auxiliary routines in lisp. The output of the parser is decomposed into schemata suitable for assertion into the frame knowledge-base used by the expert system. The pattern matching abilities of the Rete Algorithm employed by the expert system shell permit specific patterns of parse-tree nodes to be detected. If this syntactic matching is not sufficient to detect some errors or violations, it may trigger lisp routines to perform semantic analysis of the area of the code being focused on by the pattern matching step.

ICICLE currently has 45 rules. These rules can catch many of the traps and pitfalls compiled by Koenig[13]. They fall into several classes:

*We are exploring the potential for remote code inspection meeting using an audio or audio/visual link in addition to TCP/IP

**Normally this interruption would be preceded by a verbal signal to the other inspectors.
- **Standards violations.** For example, a failure to initialize automatic variables. These rules are mainly syntactic, and thus have complete "certainty" that their firing is correct.

- **Syntactic and semantic errors.** These are serious coding errors -- for example, an attempt to dereference the NULL pointer in a C program. These rules are also "sure" that they are flagging real errors.

- **Warnings.** Frequently, the system is unable to determine whether or not a "dangerous" situation is definitely an error. These situations are flagged as such. For example, the UNIX library call "setbuf" changes the buffering of a stream. If the buffer is an automatic variable, the buffer may be lost when the variable's scope is exited. The programmer may or may not be aware of this, and may have coded around it, but ICICLE is unable to tell.

- **Hazards.** Some situations merely have dangerous potential. For example, an "=" assignment operator in a conditional expression could be a typographical error for the "==" comparison operator. However, the "==" is just as frequently intentional.

- **Advisories.** There are a number of very obscure usages in C and UNIX system which typical programmers may be unaware of. For example, when a stream is opened in "r+" mode, output must be followed by an "fseek" call before input.

All the above kinds of situations are flagged differently, and are displayed as such through the user-interface.

**Usability Testing**

ICICLE is undergoing continuous usability testing. One full phase of testing was completed, and another is currently under way. In the first phase, ICICLE was put through 12 hours of videotaped usability testing over the course of two weeks. The testers were developers for ICICLE's initial target domain. In addition to videotaping, testers filled out an initial questionnaire detailing their experience with code inspection and workstation use, as well as questionnaires following each testing session. ICICLE was tested in both comment-preparation and code inspection meeting modes.

Our focus was specifically on usability issues, based on the principle of user-centered design. For the purposes of this test phase, we were interested in users' reactions and suggestions to the exclusion of performance evaluation (ICICLE's performance evaluation will occur in another sequence of tests).

In general user acceptance was high, despite the discovery of more than twenty significant problems in ICICLE's design -- either in the area of poor implementation of a feature, wrong implementation of a feature, or absence of a desired feature.

A few representative examples of the suggestions we received as a result of usability testing:

- It should be possible to view the entire module, noting analyzed files, and files which have yet to be analyzed.

- In code inspection meeting mode, lines with transferred comments should be marked with 'T' s, just as in comment preparation mode, lines with Deferred comments are marked with "D" s.

- Inspectors should be able to "point" to objects on other inspectors' screens.

- When an inspector proposes a comment, the name of the originator of the comment should be displayed in the proposal window, so the other inspectors can see who the originator of the comment is.

A number of these problems were easily correctable; others required (and still require) more work. Given the inherent inertia of developed systems, it is likely that many of these problems might never have been corrected if we had skipped usability testing.

**Current Status And Future Directions**

ICICLE is currently being used in a Bellcore development organization for live code inspections. In order to measure the usefulness of ICICLE, some of the inspections will be performed manually as well as using ICICLE. The number of detected defects in the inspected code, the time taken for detecting them and, most of all, the comments of the code inspectors regarding the ease and comfort of the ICICLE environment compared to that of manual code inspection should enable us to determine the efficacy of ICICLE.

Our focus in prototyping ICICLE has been on human interface issues, and developing a rule-based static debugging tool as well as a flexible architecture to integrate both AI and non-AI systems for enhancing code inspection. Though, through our knowledge browsers, we have demonstrated the use of domain and other kinds of knowledge for program understanding, we deliberately chose not to develop a powerful hypertext system that will work smoothly with our XVindow-based human interface. Also, we selected an off-the-shelf cross-referencing information generating tool as a placeholder for a more powerful system, since beginning in the second half of 1990 we will be integrating ICICLE with an advanced development environment being developed separately at Bellcore. In this project a powerful hypertext system as well as a sophisticated software information abstraction tool are being developed.

Our experience with ICICLE is encouraging; extensions in four different directions are possible. Firstly, ICICLE can be extended to inspect programs written in other programming environments such as PL/1 in MVS. To accomplish this it is
necessary to formulate rules for standard violations, and traps and pitfalls for the new programming language. Also, a tool for generating cross-referencing information for programs written in that language as well as a browser for the knowledge about the new programming environment should be added to ICICLE. Secondly, parts of ICICLE such as the expert system for detecting standard violations and traps and pitfalls as well as the knowledge browsers can be made available to developers and maintainers, and not be restricted to code inspection. Thirdly, more kinds of knowledge such as design knowledge and requirements specifications can be added to ICICLE. This will enable code inspectors to easily refer to this information while inspecting code. Finally, more analysis tools such as complexity analysis tools, data-flow and control-flow generating tools will help inspectors to understand the code better.

Acknowledgments

We thank Bob Martin, Dave Kessell, Carl Lewis, Chris Riley and Adam Irgon for their managerial support; Adam Irgon, Mary Lu Walling and Hassan Sriniidhi for their comments on this paper; Dave Ensign, our code inspection expert, for discussing and sharing his expertise; Michael Muller and Mary Lu Walling for assisting us with usability testing.

References