Knowledge-Based Code Inspection with ICICLE

Laurence R. Brothers, Velusamy Sembugamoorthy, and Adam E. Irgon , Bellcore

ICICLE[1] (intelligent code inspection in a C language environment) is a multifaceted software system developed with components from several technologies, including AI, computer-supported cooperative work (CSCW), and software technology. It is intended to support the process of formal code inspection within the software development cycle. This chapter reports on ICICLE in the context of its successful deployment at Bell Communications Research, Inc. (Bellcore) for code inspection and discusses its design and operation with particular emphasis on its AI components.

The Problem: Code Inspection

In this section, we review software development and the telecommunications industry. We also analyze the techniques of code inspection, manual and intelligent.

Software Development and the Telecommunications Industry

Modern telecommunications companies are heavily dependent on software in virtually all aspects of running their businesses. When a customer calls the telephone company and asks for telephone service, customer service representatives use a host of software systems to determine the customer's current status, verify information about the customer, determine what services the customer might want (often supported by sales advice expert systems), and initiate the customer's order. Initiated orders are sent to other software systems that provision the order, selecting and configuring the facilities and equipment that can implement the customer's requested services. Other systems monitor and operate the network on an ongoing basis: electronic switching systems connect calls; software routes calls in the network dynamically; and software systems monitor the health of the network, providing alarms as needed and in some cases fixing problems automatically.

The systems used to run the day-to-day business of the telephone companies are referred to collectively as operations support systems (OSSs). Development of OSSs is a major part of the services of Bellcore, which was created at AT&T's divestiture in 1984 to provide centralized services to the divested Bell Operating Companies.

This ubiquity of software in telecommunications results in a strategic edge for companies deploying advanced software systems. At the same time, bugs in telecommunications software systems can be catastrophic, as the recent network breakdown in the New York metropolitan area illustrated (Russell 1991). The loss of customers' goodwill because of network outages such as this one can have enormous financial impact on a company. At best, the act of fixing bugs in telecommunications software systems is expensive; we estimate that in 1990, Bellcore spent approximately $65 million fixing bugs. Accordingly, the pressure to make software development quality and productivity gains is powerful.

Manual Code Inspection

In an influential paper, Fred Brooks (1987) argued that no techniques will fundamentally transform software development quality and productivity. Rather, gains will be incremental, as in the case of code inspection. *Code inspection* is a phase of software development intermediate between implementation and testing. It is a rigorous, formalized process that is rapidly replacing informal code reviews in companies such as IBM (Dobbins 1987; Fagan 1976), AT&T Bell Labs (Ackerman 1984), BellNorthern Research (Murray 1985), and Bellcore because it offers significant software quality and long-term maintenance benefits.

Inspection involves using the knowledge of a team of expert developers, designers, and application-domain experts to achieve reduction of errors and more understandable software. Ideally, code inspection should be performed on all new or significantly changed code, a goal Bellcore has set. Such inspection pays off handsomely in software quality gains but also carries a heavy cost. Code inspections are painstaking, time consuming, and knowledge intensive. Code inspectors must apply a wide variety of knowledge, including design, programming, and application-domain knowledge. In general, it is difficult to find a code inspector who is an expert in all these areas. Often, even experienced code inspectors suffer from cognitive overload.

The nature of code inspections makes them expensive and often unpopular with software developers. The tendency of software development projects to be behind schedule has been well documented (Brooks 1975). In addition, customer demands for new, increasingly sophisticated services supported by rapidly changing technology put software development organizations under pressure to develop systems faster, cheaper, and better, often at the expense of code inspection.

Code inspection is broken into several phases, the two most important of which are comment preparation and the code-inspection meeting.

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 and time spent are obtained. These data are used for monitoring the effectiveness of code inspection in the organization.

Several difficulties are associated with these two phases. In comment preparation, the problems are mainly cognitive, focusing on the inability of a single developer to understand source code written by another in a limited amount of time and once understanding the code to be able comment on it. In the code-inspection meeting, the problems are mainly secretarial and administrative: The rigorous and formal requirements of the code-inspection procedures described by Fagan (1976) and Ackerman and his colleagues (Ackerman 1984; Ackerman, Buschwald, and Lewski 1989) are desirable for the reasons they discuss but also make actual participation in such meetings unpleasant and tedious. These problems are not insuperable, but together, they have combined to make code inspection unpopular in many development organizations and have reduced the overall value of the activity where it does take place.

Intelligent Code Inspection

Given the previously described difficulties with code inspection, our task was simple: to provide systems that seek to eliminate the negative aspects of the process yet accentuate the positives.

ICICLE (Brothers, Sembugamoorthy, and Muller 1990; Rich 1986) is a software system developed at Bellcore that augments and improves on manual code inspection by computerizing many of the problem-solving tasks that underlie code inspection. For some years, C has been the programming language of choice in Bellcore, and accordingly, ICICLE is for C program inspection. ICICLE has been designed to accommodate other languages, such as C++, as they increase in popularity.

Ideally, a code-inspection system should have the following virtues:

First, from the point of view of the user, a code inspector, it should make code inspection much more palatable as a task and easy to perform, reducing cognitive load and secretarial tedium, thus increasing the likelihood that it will actually be done.

Second, from the point of view of the manager of a development project, it should reduce the time and resources needed to inspect code and improve the overall quality of inspections by automating a range of tasks previously done manually and assisting inspectors in better performing some tasks that are not amenable to complete automation.

These goals can be accomplished by targeting the two difficult phases of code inspection with specific functions:

First, 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. A computer-aided code-inspection environment should provide tools to aid static debugging, checking coding standards violations, browsing through various kinds of code-inspection knowledge (such as manual pages, library function specifications), and above all integrate these tools in a user-friendly human interface.

Second, during code-inspection meetings, the team of inspectors meet to reach consensus on problems with the program under inspection. A good computer-aided code-inspection system should support this cooperative effort by computerizing the communications between inspectors, enabling the team members to maintain their focus of attention without being required to resort to paper listings, permit comment integration and recording with a minimum of effort, and also make the secretarial and administrative burden of the meeting as light as possible.

ICICLE Design and Architecture

In this section, we discuss the design and architecture of ICICLE. We examine the integration of multiple technologies for code inspection; ICE, the expert system within ICICLE, the human interface for comment preparation; and the automation of the code-inspection meeting by ICICLE.

Integrating Multiple Technologies for Code Inspection

As a result of the analysis previously described, we realized the need to combine components from several technologies.

To support the efforts of an individual code inspector to understand the code, we needed to provide a sophisticated human interface capable of browsing and navigating source code in different interconnected files. We also needed to be able to present the results of the analyses of software tools such as cross-referencers in an understandable and easy-to-use fashion. The effect of human program understanding in the comment-preparation phase is the compilation of a set of comments about the code. With a knowledge-based debugging tool and conventional debugging tools such as LINT (Johnson 1983), we could provide an automatically generated set of comments. Some of these comments pertain to problems in the code that might otherwise have been overlooked by the human inspector, and other comments might help the inspector to better understand what the original programmer was trying to do.

The activities of the inspectors during the code-inspection meeting are primarily administrative and secretarial in nature but can also occasionally involve some of the same activities as in comment preparation. To support the conduct of the code-inspection meeting, we had to provide groupware to enable inspectors to carry out all the meeting procedures online but still retain the capabilities of the interface used for comment preparation.

The architecture that supports these multifarious activities is shown in figure 1. ICICLE's current components include the following: (1) a human interface for effective presentation of information and interaction with the user (also manages annotation and comments, as provided by either the human user or other automatic subsystems, and contains connections to various software tool interfaces) (2) groupware for supporting the cooperative effort of code-inspection meetings, (3) software tools for static debugging and providing useful program-understanding aids such as cross-referencing information, and (4) an expert system to represent expert developers' knowledge for detecting programming errors and violations of coding standards.

ICICLE's architecture is modular. Because all the analysis tools are interfaced with the human interfaces through text files, it is easy to add new analysis tools (for example, complexity analysis tools, program

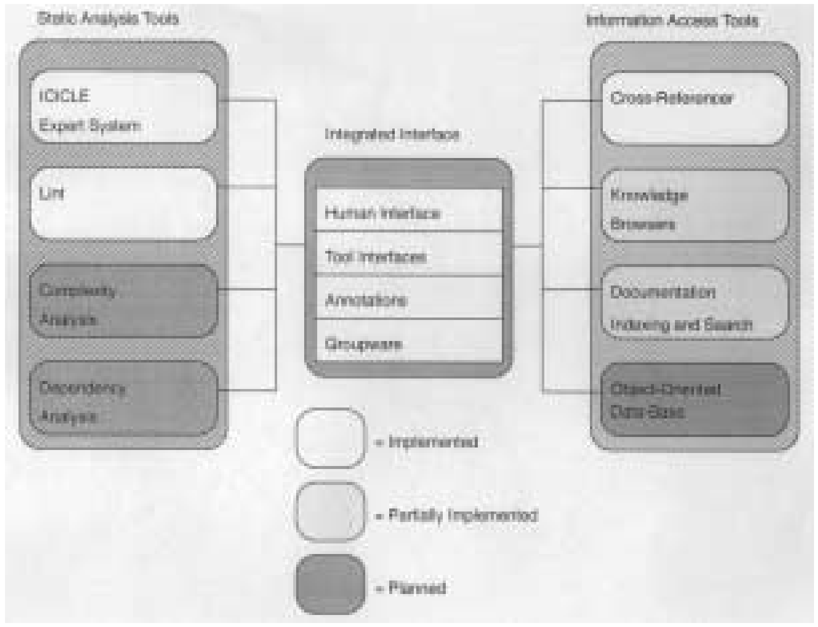


Figure 1. ICICLE *Architecture.*

slice analyzers). Even if some or all of the analysis tools fail for a given source code module (perhaps the program is syntactically incorrect or in an unusual language dialect), ICICLE still provides all the functions necessary to perform a code inspection, albeit with reduced efficacy.

The primary innovations offered by ICICLE are embodied in (1) the expert system for automatic detection of bugs in C programs, (2) ICE (the ICICLE C expert), (3) a CSCW system for code-inspection meeting support (we refer to this as ICICLE groupware), and (4) the integration of these technologies to enable all the complex and demanding activities of code inspection to run online through the medium of a single system.

ICE: The ICICLE C Expert

Certain classes of relatively simple errors and warnings can be captured by bug-detection tools such as LINT or DAVE (Lukey 1980). To detect more complex errors, one needs to acquire and represent the heuristic rules that expert developers use. Because of the easy modifiability of the rule base and the availability of explanation facilities, rule-based systems provide an excellent framework to represent not only these rules but also those for detecting violations of coding standards. Examples of rule-based debugging systems are FALOSY (Osterweil and Fosdick 1976), MESSAGE TRACE ANALYZER (Gupta and Seviora 1983), and Harandi's (1983) system. The first two systems require the program to be run and are therefore unsuitable for code inspection. Code inspection requires syntactically correct code but is not intended to perform runtime analyses, which are usually performed by a separate testing group. Koenig (1989) compiled more significant debugging knowledge than that represented in Harandi's system. ICICLE has rules to detect many of the characteristic problems reported by Koenig.

ICE contains a YACC[2]-based C grammar parser, which can efficiently accept C code and output a Lisp-readable annotated parse tree in the form of a series of S-expressions. The parser is by no means as complete a parser as, for example, those used by C compilers. Because code presented for inspection must compile without errors, the grammar and actions can be fine tuned to provide exactly the information required by the rules of the expert system component and need not provide robust explanations of syntactical and lexical errors.

ICE's primary inference mechanism is a rule-based system written in ART, a commercial multiparadigm 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 shell's relational pattern-matching inference engine is capable of detecting specific patterns or templates of parse tree nodes corresponding to potential errors, dangerous coding uses, and coding standards violations. If this structural (syntactic) matching is not sufficient to detect some errors or violations, additional Lisp routines can be triggered to perform semantic analysis of the area of the code being focused on by the pattern-matching step. For example, syntactic analysis can find an instance of a pointer being dereferenced, but further semantic analysis is required to determine if dereferencing the pointer is likely to lead to a segmentation fault.

Cliche recognition (Wills 1990; Harandi and Ning 1990; Johnson 1986) is an emerging technology for identifying the function or intention of a piece of code by recognizing a pattern associated with the function. Detection of the intention of a programmer goes a step beyond mere semantic analysis into the more difficult area of pragmatics. This technology can be used to recognize patterns of C traps and pitfalls. It provides a higher-level language-independent framework (for example, PLAN CALCULUS [Rich 1986]) to represent patterns of code and semantic and program-understanding knowledge (for example, the representation in PAT [Harandi and Ning 1990]). Tutoring systems such as PROUST (Johnson 1986) and TALUS (Murray 1985) have also experimented with frameworks to represent debugging knowledge. Compared to the pattern-matching and object-oriented languages available in rule-based system shells such as ART, these higher-level frameworks

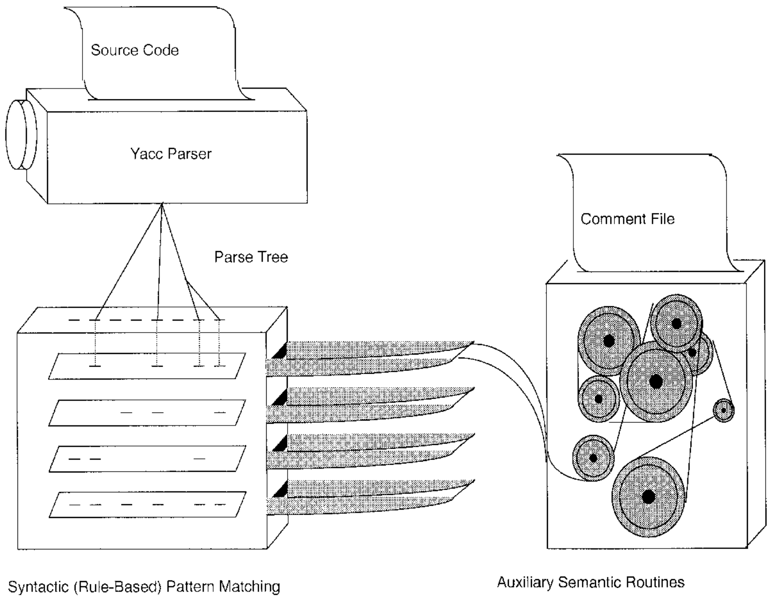

Source Code
Yacc Parser
Parse Tree
Comment File
Syntactic (Rule-Based) Pattern Matching
Auxiliary Semantic Routines

Figure 2. ICE *Architecture.*

make it easier to represent knowledge to catch complex traps and pitfalls that involve sophisticated patterns (delocalized plans [Letovsky and Soloway 1985]) and semantic information dispersed across many procedures and modules. However, none of the reported cliche-recognition systems is scalable to large, real-life application systems implemented in languages such as C (Harandi and Ning 1990; Rich and Wills 1990). Hence, we decided to use an integrated frame- and rule-based expert system shell to implement ICE.

Although, in general, detection of the intention of a programmer is difficult, several of ICE's rules are designed to detect simple cliches. For example, a common use of the C for loop is to iterate through a fixed-size array. ICE rules can detect possible errors in array bounds and iterator direction based on the expected template provided by this programming cliche. ICE rules do not attempt to detect higher-level cliches, such as sorting functions or access methods; such analysis is beyond the representational and computational capacity of the system. Figure 2 shows the interrelationship of the YACC parser, the parse tree, the pattern-matching rules, and the auxiliary semantic routines.

ICE currently contains about 45 programming heuristics, implemented as a rule-based system. These heuristics can catch many of the traps and pitfalls compiled by Koenig (1989). They fall into several classes:

Standards violations: An example is a failure to initialize automatic variables. These rules are mainly syntactic and, thus, have complete certainty that their firing is correct. Standards violations generally do not reflect actual bugs but are enforced by development organizations to ensure consistent good programming practices across different modules and systems.

Definite programming errors: These errors are serious coding errors, for example, an attempt to dereference the null pointer. These rules are also sure that they are flagging real errors. Most of these rules operate on a fairly small scale, for example, a single expression or operation.

Possible programming errors: Frequently, the system is unable to determine whether a dangerous coding situation is definitely an error; these situations are flagged as such. For example, a for loop that runs from 1 to n (instead of 0 to n - 1) might be correct, but it is atypical enough to flag for further attention. Usability testing has demonstrated that the flagging of false positives is not a problem for users or the system. Because of the high cost of bug correction after code inspection, even a high ratio of false positives to actual bugs detected is acceptable.

All these situations are flagged differently and are displayed as such through the user interface.

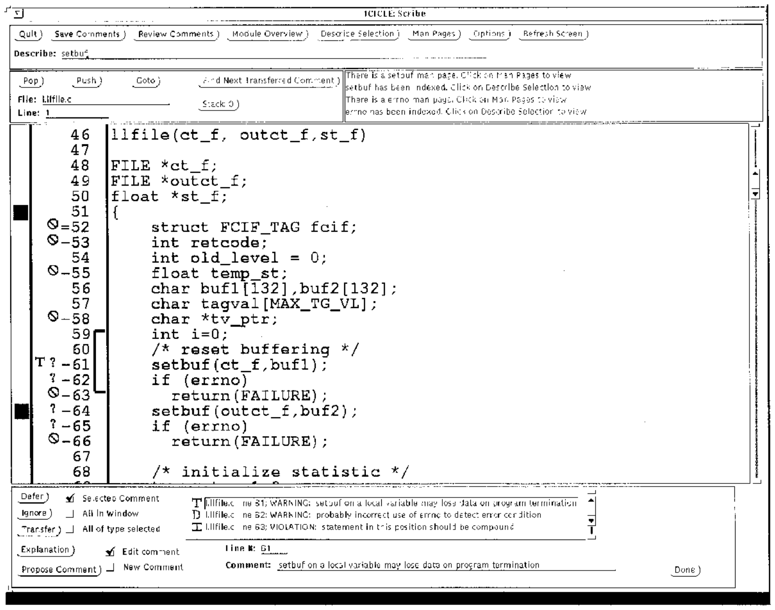
Human Interface for Comment Preparation

In the process of code inspection, various ICICLE features come to the fore in different phases. Both comment preparation and the code-inspection meeting require support for automated analysis, human program understanding, and various secretarial and organizational tasks such as recording and filing annotations, but the style of system use is markedly different during the two phases.

The ICICLE human interface was implemented using the XVIEW tool kit for the X WINDOW system. The functions of X permit easy operation of applications on remote displays, helping to enable the CSCW interface functions. ICICLE can operate on any display device that runs X as long as at least one machine can run the X VIEW client or a client with similar functions.

The human interface has two modalities of operation: comment preparation and code-inspection meeting. The latter mode subsumes the functions of the former, with the addition of groupware to automate the cooperative effort of the meeting. Figure 3 is a screen dump of the ICICLE human interface in its most basic configuration.

The output of comment preparation is a file that contains all com-



Describe: setbuf

File: lllfile.c
Line: 1

There is a setbuf man page. Click on Man Pages to view
setbuf has been indexed. Click on Describe Selection to view
There is a errno man page. Click on Man Pages to view
errno has been indexed. Click on Describe Selection to view

```
   46 llfile(ct_f, outct_f,st_f)
   47
   48 FILE *ct_f;
   49 FILE *outct_f;
   50 float *st_f;
   51 {
 ⊘=52     struct FCIF_TAG fcif;
 ⊘-53     int retcode;
   54     int old_level = 0;
 ⊘-55     float temp_st;
   56     char buf1[132],buf2[132];
   57     char tagval[MAX_TG_VL];
 ⊘-58     char *tv_ptr;
   59     int i=0;
   60     /* reset buffering */
T ?-61     setbuf(ct_f,buf1);
  ?-62     if (errno)
 ⊘-63       return(FAILURE);
  ?-64     setbuf(outct_f,buf2);
  ?-65     if (errno)
 ⊘-66       return(FAILURE);
   67
   68     /* initialize statistic */
```

✓ Selected Comment
☐ All In Window
☐ All of type selected

T lllfile.c line 61; WARNING: setbuf on a local variable may lose data on program termination
D lllfile.c line 62: WARNING: probably incorrect use of errno to detect error condition
I lllfile.c line 63; VIOLATION: statement in this position should be compound

✓ Edit comment
☐ New Comment

Line #: 61
Comment: setbuf on a local variable may lose data on program termination

Figure 3. ICICLE *Human Interface.*

ments and annotations made by the human user and ICICLE components such as ICE and LINT. This file is used as one of several input to the code-inspection meeting. Each inspector is required to review the module individually and prepare separate sets of comments and annotations in separate comment files.

Groupware for Code-Inspection Meeting Support

Figure 4 shows a typical code-inspection meeting situation. During the code-inspection meeting, active analysis of the module's source code is secondary to the discussion of the validity of comments and annotations compiled during the previous comment-preparation phase. Nevertheless, many of the functions required during comment preparation might be needed during the meeting. ICICLE automates the conduct of the code-inspection meeting by supporting the various secretarial and administrative roles assigned to the inspectors during the meeting.

The groupware of the basic ICICLE product requires each inspector to be present, with a personal workstation or terminal in the same room for the duration of the meeting (usually about two hours). To accommodate the requirements of distributed work groups and also afford the possibility of using ICICLE for informal code-review sessions, we ex-

Figure 4. A Bellcore Code-Inspection Meeting.

plored the use of multiple media (voice, image, and video) in the groupware interface to permit inspectors to carry out inspections from their separate offices or across internetworked local area networks between remote sites. ICICLE groupware obviates the need for paper at the meeting (although paper listings and documents can still be used as an auxiliary aid) and greatly enhances the efficiency of code-inspection meeting procedures by streamlining secretarial and administrative procedures through computer support.

For further discussion of ICICLE's groupware and usability testing results and other human-computer interaction issues, see Brothers, Sembugamoorthy, and Muller (1990).

ICICLE: Development, Maintenance, and Use

In this section, we discuss details of ICICLE's development, maintenance, and use.

ICICLE Development

The ICICLE project was started as a research prototype intended to explore issues of code inspection by means of an intelligent assistant approach to program understanding. We understood immediately that many other research prototypes (Harandi and Ning 1990; Johnson

1986; Letovsky and Soloway 1985; Rich and Wills 1990; Harandi 1983) in the field had encountered scaling problems in dealing with such large and difficult phases of software development as specification, design, and implementation (see Sembugamoorthy et al. for more details). We chose, therefore, to focus initially on the relatively isolated phase of software engineering known as code inspection, which we felt existing techniques and hardware could address in real-world situations. Further impetus for our work was the realization that the ordinary code-inspection process was entirely manual and not computer assisted in any way, so that our efforts would not have to conform to a rigid set of expectations in our planned user community (nevertheless, we tried to continue to uphold the principles of the old style of code inspection where this approach was reasonable).

The project was initiated in response to an internal Bellcore policy to include code inspection for all new software. We anticipated that many software developers would be unfamiliar with code inspection and that given the scale of Bellcore's software development efforts, even a slight savings of time or a slight improvement in effectiveness could save many millions of dollars. Later, positive results from usability testing and the joining of our efforts with those of the Bellcore Advanced Software Environment (BASE) development group continued to support our work.

Our initial feasibility prototype took approximately 1-1/2 staff-years for two developers (L. Brothers and V. Sembugamoorthy) to complete. From the beginning, we had the cooperation of a major Bellcore software development group without whose assistance the system could never have been constructed. Developers from this organization permitted us to observe their manual code inspections and also took part in usability testing and field trials of various ICICLE prototypes.

Once our experimental prototype had been tested (approximately 2-1/2 total staff-years, or 1-1/2 calendar-years, of research and development from project initiation), it was time to turn ICICLE into a deployable software system. Approximately three more staff-years, or another year of real time, including the efforts of developers from BASE as well as our own group (Knowledge-Based Systems Development), led finally to the development of the ICICLE software system, which has been deployed internally for several months.

At every stage of ICICLE's development, we followed the principles of user-centered design, with particular emphasis on usability testing. Software developers who have used the ICICLE system have reacted almost uniformly positive to the introduction of the system despite the radical changes it brings to the code-inspection process. As the ICICLE product continues to be used for real code inspections, many changes, both small and large, are being planned to improve the code-inspection process.

ICICLE Maintenance

ICICLE is currently being maintained as an internal software system by BASE. Because every organization has a different set of coding standards and might or might not approve of certain use rules within the ICE expert system, ICICLE must be configured slightly differently for each development group it is deployed for. Fortunately, this configuration is made easy by the alteration of a pair of setup files that define which rules are to be used and also permit development organizations to add certain coding standards of their own. An additional issue is the use by different development organizations of a variety of database languages such as SQL preprocessors. Another configuration item requires such preprocessors to be accessible to the ICE analysis scripts so that all source code can be analyzed to the greatest extent possible.

Current development directions include a transition from ART, a Lisp-based product, to a C-based shell of comparable power. ICICLE will also be ported to use the MOTIF[3] tool kit and will be extended to cover additional languages and environments beyond C and UNIX. BASE treats ICICLE as a component of its general product BASIS, which addresses all phases of the software development process; every one of these components will have to be actively maintained by a staff of developers for the lifetime of the product to support the changing needs and situations of Bellcore software developers.

ICICLE Use

At present, BASE is still deploying ICICLE to its initial client group. Once use patterns are analyzed, and requests for maintenance are received, BASE will update ICICLE's components accordingly. As of this writing, ICICLE has been deployed to several Bellcore development organizations, covering hundreds of potential and actual users. During the course of its development, other development groups were invited to help test the system, and the needs and concerns of different groups were thereby addressed.

Because many Bellcore development organizations use traditional dumb terminals connected to minicomputers or mainframes, ICICLE introduction has been slowed by the need to purchase special equipment (workstations and software licenses). Fortunately, many of these organizations have begun to actively convert their operations to workstation and PC-based configurations, so we anticipate accelerated deployment of ICICLE in the near future.

Looking Back at ICICLE

Here, we examine the impact of ICICLE on Bellcore and the code-inspection process as well as the lessons we learned during its development.

Impact of ICICLE

As anticipated, Bellcore's code-inspection goals were difficult to meet, partially because of the problems attributed to the old regime of manual inspections. Even modest gains in inspection rates and efficiency would be major gains for the productivity of Bellcore's software development organization because of the organization's size.

We characterize the scale of Bellcore's software development effort with the following statistics from 1990: Bellcore produced 18.1 million lines of new or significantly changed code and spent 68,000 hours inspecting 20 percent of this code manually. Inspections resulted in an average fault-detection rate of 7 for every 1000 lines of code inspected. The cost of correcting defects detected during code inspection averaged in the hundreds of dollars. The cost of correcting defects detected during software use averaged $20,000. The correction of defects (not found during inspection or testing) cost $65 million in Bellcore development costs (from a budget of about $400 million), excluding potentially enormous costs from lost productivity in the user community.

Gains from the use of ICICLE fall into the following areas. For each area, we describe our experience to date:

More inspections done: Surveys of ICICLE users indicate an overwhelming preference (over 90 percent) for inspections using ICICLE as opposed to manual inspections. A major barrier to increasing inspection rates is developers' distaste for inspections. Comments from users of ICICLE indicate that it removes many of the most onerous aspects of inspections. ICICLE is currently used on five major software projects. The current version of ICICLE processes Kernighan and Ritchie C. In the near future, versions of ICICLE for ANSI C and C++ will be deployed, making further gains in inspection rates possible.

More errors found: We know that ICE detects errors that many developers are unfamiliar with; we have observed inspectors being surprised by errors detected by ICE. Because analysis tools such as ICE and LINT are capable of automatically finding many classes of errors, ICICLE users are freed to concentrate on more sophisticated and subtle problems, which they otherwise would not have time to look for. This analysis by people is further enhanced by the finely tuned human interface described earlier. Specific data on additional errors found with ICICLE have not been obtained to date because tight development schedules have not allowed for the needed comparative studies. With conservative assumptions of one additional error found with ICICLE for every 1000 lines inspected and just 5 percent of these errors remaining in deployed code, the use of ICICLE could have saved Bellcore approximately $3 million for 1990, when only about 20 percent of the code was inspected.

Less time taken for inspections: We know that ICICLE saves a lot of paper shuffling during comment preparation and inspection meetings and that it eliminates most of the secretarial drudgery and bookkeeping associated with manual inspections. Because our analysis of time spent in manual code-inspection meetings revealed that a large portion of meeting time was wasted in paper shuffling, we believe that ICICLE can be used to save meeting time. Clearly, however, the discovery of more errors for every 1000 might have a countervailing effect on this statistic because more comments will have to be analyzed, discussed, and resolved. Thus, instead of merely reducing meeting time, ICICLE might be said to increase the value of meeting time, however much time is spent.

Impact on the code-inspection process: Following our analysis of ICICLE use (for more detail see Brothers, Sembugamoorthy, and Muller [1990]), we determined that computerized code inspection can significantly alter the nature of the special roles (moderator, reader, and scribe) assigned for traditional code-inspection meetings. For example, because the scribe has much less work to do in an ICICLE-moderated inspection meeting and because the moderator performs no special functions within ICICLE, we suggested that these roles be merged. Such a merger would permit smaller inspection teams, consequently allowing the performance of more inspections or less cost in staff time. If the size of average inspection teams was reduced from four to three, and the number of inspections stayed constant, Bellcore could save approximately $1.5 million in inspection costs.

As discussed previously, ICICLE is intended not merely to improve metrics such as the number of code inspections and the errors found but, more importantly, to increase the value of code inspections as such. As ICICLE permeates our software development organization more thoroughly, we expect a consensus to emerge among developers about the value of ICICLE in each of these areas and to obtain better statistics to demonstrate this success.

Lessons Learned

As a result of our work, we have gained insights into numerous issues regarding the application of AI techniques to problems in software engineering. Following are some of the most significant. Not all these insights are new or original, but inasmuch as other efforts in the field have sometimes neglected them, we can repeat them here with new emphasis:

Scalability: Unless research prototypes are designed with eventual deployment among real user populations for real problems in mind, they will remain, at best, studies. We were forced to abandon several promising directions that seemed valuable in prototype form because of the improper amount of resources they would consume when scaled to real-world situations. For example, we had hoped to represent a significant amount of project-specific knowledge within ICICLE to expand the reasoning capabilities of ICE and also provide more sophisticated assistance for program understanding. However, the knowledge-acquisition problem forced us to retreat from the representation of knowledge to the presentation of information in its stead.

Ripeness: In the area of code inspection, we found a phase of the software development process that was ripe for exploitation. Because it was entirely manual and at the same time regarded as exceedingly difficult and onerous, we could introduce radical changes into existing procedures without social engineering among the target user population, and we could almost guarantee from project inception that our system would be received favorably. Even systems that can objectively be demonstrated to provide performance that is superior to existing systems might fail if the user population does not perceive the need for the new systems or if the cost of changing over is too great.

Problem integration: Our system was designed from the start to address all major aspects of code inspection, from individual comment preparation to group code-inspection meetings to form and report generation. ICICLE would have been much less valuable had it only supported part of the process, even had this support been even stronger for this phase than the actual system now provides. We were forced to integrate multiple technologies to provide a product capable of dealing with the whole problem of code inspection.

Problem isolation: Despite our need to address the whole problem of code inspection, we were able to avoid having to construct a system to deal with the manifold other problems of software development in general. Had our system been required to address other issues in requirements, specification, design, implementation, or maintenance, we would have been unable to ever deploy a usable system.

Additionally, we learned some less abstract lessons about the specific systems we developed and deployed:

Human interface: Regardless of the success of the ICE software analysis expert system, the human interface is undoubtedly the most critical component. An intelligent assistant program, at least in the style of ICICLE, can be rendered worthless by a human interface that makes interaction with the program difficult or otherwise provides less than optimal performance. We were frequently forced to alter the human interface in response to requests by usability testers and based on our observation of users' interactions with the system.

ICE: Although our rule-based framework enabled us to quickly write and modify both simple and powerful rules for error detection, we eventually found that we had employed this complex pattern-matching system in a few cases to discover errors that could have been more efficiently found by much simpler systems. In fact, we now use a simple SED-based[4] system to detect certain categories of errors that were unnecessarily written in a form much more wasteful of resources. We found that in this case, it was important to trade off simplicity and consistency of approach for efficiency and eventually achieved an order-of-magnitude increase in performance. Of course, most rules continue to operate within our expert system shell language, but we have been forced by concerns for efficiency to constantly reevaluate our choice of rules and rule formats.

Groupware: Through detailed task analysis, we were able to construct a groupware system based on a limited set of primitive operations nonetheless capable of supporting all our communications requirements for the code-inspection meeting. Despite its small size and relative simplicity, the development of the ICICLE groupware took an unexpected amount of development, testing, and refinement. The interaction of multiple users, combined with the synergetic increase in the interaction complexity of other ICICLE subsystems operating in a groupware, made the system unusually hard to develop and test. Nevertheless, we consider this ICICLE component one of the most valuable ICICLE subsystems.

Software tools: We had hoped to directly employ many vendor tools to either detect errors or assist users with program understanding. Unfortunately, we found most such software tools to be either too inflexible to use for our purposes or too isolated and self-contained to connect to our tool framework. We still plan to adopt more such tools but have found our choices more limited than we had expected.

Our work on ICICLE has not only been of use in the development of a system to help users to accomplish a difficult and time-consuming task, it has also reinforced our beliefs in the basic principles of applied research and exploratory development. Our diligent task analysis, integration of diverse technologies, and, above all, our commitment to the philosophy and procedures of user-centered design helped to ensure that our research hypotheses could be developed into a working software system capable of dealing with serious problems in the software development life cycle.

Acknowledgments

We are grateful to Carl Lewis and other members of the Bellcore Provisioning Order Control System Development Department who provided critical assistance in usability and field testing of various ICICLE versions. ICICLE could not have been deployed without the work of Reva Leung, Bob Kayel, and Eric Jung of the Bellcore Advanced Software Environment Department. The continued encouragement and support of our own management, namely, Chris Riley, Dave Kessell, and Bob Martin, was essential to our success.

Notes

1. ICICLE currently runs under UNIX on Sun Microsystems workstations using the ART expert system shell and the X WINDOW system. The human interface was built using the XVIEW tool kit.
2. YACC, yet another compiler compiler, is a common UNIX tool.
3. An example is ART-IM.
4. SED, the stream editor, is a commonly available filter language available on UNIX systems.

References

Ackerman, A. F. 1984. Software Inspections and the Industrial Production of Software. In *Software Validation*, ed. H. L. Hausen. New York: Elsevier.

Ackerman, A. F.; Buschwald, L. S.; and Lewski, F. H. 1989. Software Inspections: An Effective Verification Process. *IEEE Software*. 6(3) (May): 31–36.

Brooks, F. 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*.

Brooks, F. 1975. *The Mythical Man-Month*. Reading, Mass.: Addison-Wesley.

Brothers, L.; Sembugamoorthy, V.; and Muller, M. 1990. ICICLE: Groupware for Code Inspection. Presented at the Computer Supported Cooperative Work Conference, Los Angeles, California, October.

Dobbins, J. H. 1987. Inspections as an Up-Front Quality Technique. In *Handbook of Software Quality Assurance*, eds. G. G. Schulmeyer and J. J. McManus, 137–177. New York: Van Nostrand Reinhold.

Fagan, M. E. 1976. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal* 15(3): 182–211.

Gupta, N. K., and Seviora, R. E. 1983. An Expert System Approach to Real-Time System Debugging. In Proceedings of the First Conference on Artificial Intelligence Applications, 282–288. Washington, D.C.: IEEE Computer Society.

Harandi, M. T. 1983. Knowledge-Based Program Debugging: A Heuristic Model. In Proceedings of the 1983 Softfair, 282–288.

Harandi, M. T., and Ning, J. Q. 1990. Knowledge-Based Program Analysis. 2(1) (January): *IEEE Software*: 74–81.

ohnson, W. L. 1986. *Intention-Based Diagnosis of Errors in Novice Programs*. San Mateo, Calif.: Morgan Kaufmann.

Johnson, S. C. 1983. LINT, a C Program Checker. In UNIX Programmer's Manual, vol. 2. Murray Hill, N.J.: Bell Labs.

Koenig, A. 1989. *C Traps and Pitfalls*. Reading, Mass.: Addison-Wesley.

Letovsky, S., and Soloway, E. 1985. Strategies for Documenting Delocalized Plans. In Proceedings of the Conference on Software Maintenance, 144–151. Washington, D.C.: IEEE Computer Society.

Lukey, F. J. 1980. Understanding and Debugging Programs. *International Journal of Man-Machine Studies*: 189–202.

Murray, W. R. 1985. Heuristic and Formal Methods in Automatic Program Debugging. In Proceedings of the Ninth International Joint Conference on Artificial Intelligence, 15–19. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Osterweil, L. J., and Fosdick, L. D. 1976. DAVE: A Validation Error Detection and Documentation System for FORTRAN Programs. *Software Practices and Experience* 6: 473–486.

Rich, C. 1986. A Formal Representation for Plans in the PROGRAMMER'S APPRENTICE. In *Readings in Artificial Intelligence and Software Engineering*, eds. C Rich and R. C. Waters. San Mateo, Calif.: Morgan Kaufmann.

Rich, C., and Wills, L. M. 1990. Recognizing a Program Design: A Graph Parsing Approach. *IEEE Software*. 7(1) (January): 82–89.

Russell, G. W. 1991. Experience with Inspection in Ultralarge-Scale Developments. *IEEE Software*: 25–31.

Sembugamoorthy, Y., and Brothers, L. 1990. ICICLE: Intelligent Code Inspection in a C-Language Environment. Presented at the Computer Science and Applications (COMPSAC) Conference.

Wills, L. M. 1990. Automated Program Recognition: A Feasibility Demonstration. *Artificial Intelligence* 45:113–172.

When the Public Network Dies. 1991. *Networking Management*: 31–35.