This is a revised version of a paper appearing in Proceedings of the Third European Conference on Computer Supported Cooperative Work, Milan, Italy, September, 1993.

Improving Software Quality through Computer Supported Collaborative Review

Philip M. Johnson
Danu Tjahjono
Department of Information and Computer Sciences
2565 The Mall
University of Hawaii
Honolulu, HI. 96822 U.S.A.

Formal technical review (FTR) is a cornerstone of software quality assurance. However, the labor-intensive and manual nature of review, along with basic unresolved questions about its process and products, means that review is typically under-utilized or inefficiently applied within the software development process. This paper introduces CSRS, a computer-supported cooperative work environment for software review that improves the efficiency of review activities and supports empirical investigation of the appropriate parameters for review. The paper presents a typical scenario of CSRS in review, its data and process model, application to process maturation, relationship to other research, current status, and future directions.

1. Introduction

Formal technical review (FTR) is a cornerstone of software quality assurance. While other techniques, such as measures of program size and complexity, or software testing can help improve quality, they cannot supplant the benefits achievable from well-executed FTR. One reason why review is essentially irreplacable is because it can be carried out early in the development process, well before formal artifacts such as source code are available for complexity analysis or testing. Another reason is because no automated process can yet provide the two-way quality improvement in both product and producers possible through review.

However, the full potential of review is rarely realized in any of its current forms. Three significant roadblocks to fully effective review are the following:

Review is extremely labor-intensive. Typical procedures for FTR involve individual study of hard-copy designs or source listings and hand-generated annotations, followed by a group meeting where the documents are paraphrased line by line, issues are individually raised, discussed, and recorded by hand, leading eventually to rework assignments and resulting changes. For one approach to FTR called *code inspection* (Fagan, 1976), published data indicates that an entire man-year of effort is needed to review a 20KLOC program by a team of four reviewers (Russel, 1991). Unfortunately, little automated support for the process and products of review is available. What support is available typically supports only a single facet of review (such as the review meeting), or is not integrated with the overall development environment.

Review is not compatible with incremental development methods. Because of their labor-intensive nature, most organizations cannot afford to review most or even many of the artifacts produced during development. Instead, review is deployed as a "hurdle" to be jumped a small number of times at strategic points during development. While this may be a reasonable tactic for development in accordance with a strict waterfall lifecycle model, more modern incremental and maintenance-intensive development methods prove problematic: there is no effective way to optimally position a small number of review hurdles in the development process.

No methods or tools exist to support the design of prescriptive review methods adapted to an organization's own culture, application area, and quality requirements. Research on review tends to fall into two categories, which we will term "descriptive" and "prescriptive". The descriptive literature describes the process and products of review abstractly, advocates that organizations must create their own individualized form of review, but provides little prescriptive support for this process (Schulmeyer, 1987; Dunn, 1990; Freedman, 1990). Such work leaves ill-defined many central questions concerning review, such as: How much should be reviewed at one time? What issues should be raised during review, and are standard issue lists effective? What is the relationship between time spent in various review activities and its productivity? How many people should be involved in a review? What artifacts should be produced and consumed during a review? The prescriptive literature, on the other hand, takes a relatively hard line stance on both the process and products of review (Fagan, 1976; Fagan, 1986; Russel, 1991). Such literature makes clear statements about the process (Meetings must last a maximum of 2 hours; each line of code must be paraphrased; lines of code must be read at rate of 150 lines per hour; etc). The data presented in this literature certainly supports the claim that this method, if followed precisely, can discover errors. However, the strict prescriptions appear to suggest that organizations must adapt to the review method, rather than that the review method adapt to the needs and characteristics of the organization.

This paper introduces CSRS[1], a computer-supported cooperative work system that is designed to address aspects of each of these three roadblocks to effective formal technical review.

First, CSRS is implemented on top of EGRET, a multi-user, distributed, hypertext-based collaborative environment (Johnson, 1992) that provides computational support for the process and products of review and inspection. This platform allows an essentially "paperless" approach to review, supports important computational services, and facilitates integration with existing development environments. In combination with an adapted review process model, CSRS provides computational support that significantly decreases the labor intensive nature of review.

Second, CSRS is designed around an incremental model of software development. While simply lowering the cost helps integrate review into incremental models, CSRS also provides an intrinsically cyclical process model that parallels the iterative nature of incremental development.

Third, CSRS exploits the use of an on-line, collaborative environment for review to collect a wide range of metrics on the process and products of review. Such metrics generate historical data about review process and products for a given organization, application, and review group that can provide quantitative answers to many of the questions concerning the basic parameters for review raised above.

The remainder of this paper illustrates our approach to FTR and the CSRS environment in more detail. Although our approach can be applied to FTR of a wide range of artifacts produced during software development, we currently concentrate on support for code review, and this paper reflects this orientation. Section 2 introduces CSRS through selected snapshots from a recent review experience. Sections 3 and 4 provide a broader perspective by detailing the CSRS data and process models. Section 5 outlines both formal and informal applications of CSRS to software process maturation. The paper concludes by comparing CSRS to other systems for review in Section 6, and discussing its current status and future directions in Section 7.

2. Review using CSRS

To get the flavor of review using CSRS, this section presents excerpts from a recent review, including illustrative screen shots and metric data. A more complete presentation of both data and process model is provided in Sections 3 and 4.

This review cycle focussed on a object-oriented class implementation called "nbuff" (short for node-buffer) in the generic-interface subsystem of EGRET. Nbuff defines an abstraction that bridges and combines the hypertext "node" abstraction provided by lower-

[1]An acronym for Collaborative Software Review System.

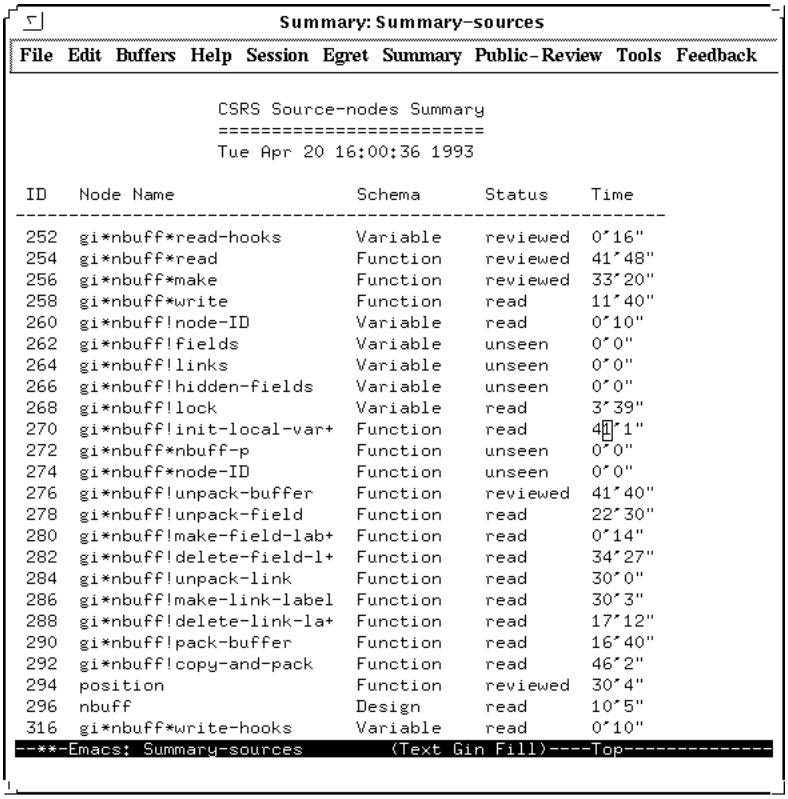

Figure 1. A summary window illustrating the state of review for one reviewer.

level subsystems in EGRET and the textual "buffer" abstraction provided by higher, application-specific subsystems such as those comprising CSRS. The portion of nbuff under review consisted of approximately 500 of the 1100 lines of Lisp macros, functions, and variables in the entire nbuff class.

In CSRS, each program object, such as a function, procedure, macro, variable or data type declaration is retrieved from a source code control system and placed into its own node in a hypertext-style database. After an orientation session to familiarize each review participant with the system under study, a *private review* phase begins. During private review, each member individually reviews the source code without access to the review commentary of others, although non-evaluative questions and answers about requirements and so forth are publically accessable. CSRS provides facilities to summarize the state of review for the reviewer, such as the window displayed in Figure 1. At this point, the reviewer has partially completed private review, as indicated by the fact that some of the source-nodes are reviewed, some have been read but have not been completely reviewed, and some have not even been seen. The total cumulative time spent on each node by this reviewer is also displayed.

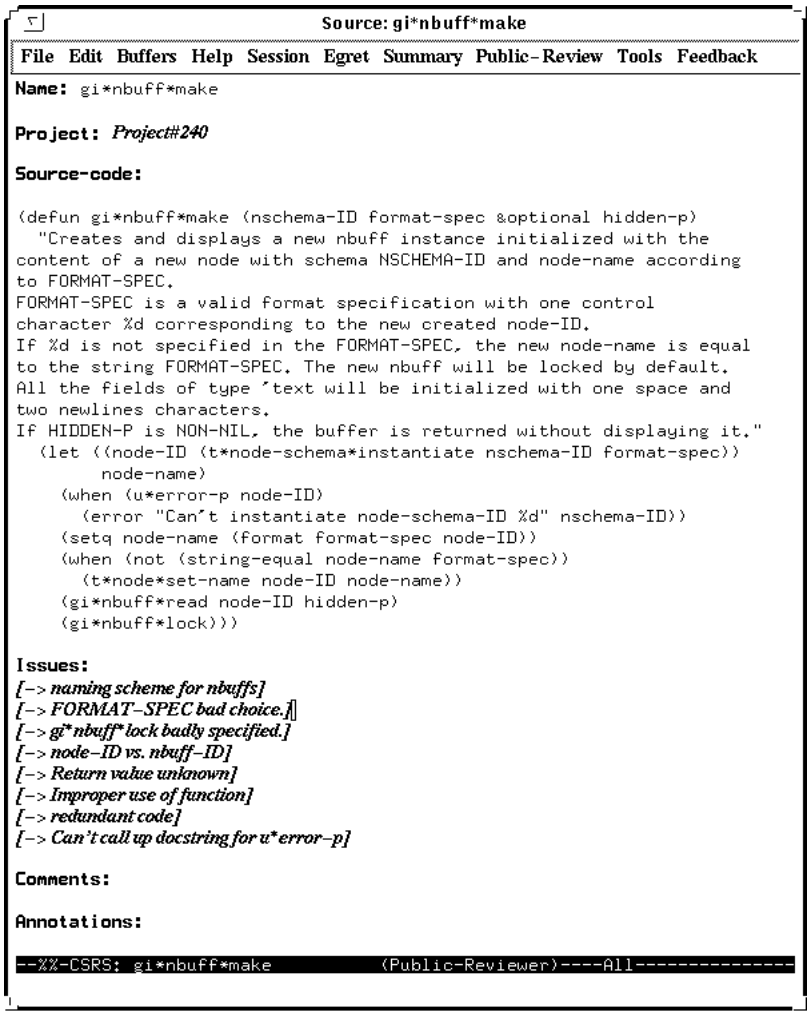

Figure 2. A source node illustrating one of the functions under review.

By double-clicking on a line or through menu operations, the reviewer can traverse the hypertext network from this screen to a node containing a source object under review, as illustrated in Figure 2. In this case, the object under review is the operation gi*nbuff*make. Both pull-down and pop-up menus facilitate execution of the most common operations during this phase, such as creating an issue concerning the current source node under review, or proposing a specific action to address an issue. Once the reviewer is finished with a source object, he explicitly marks the node as "reviewed". Since this is the private review phase, only the issues created by this reviewer are accessable.

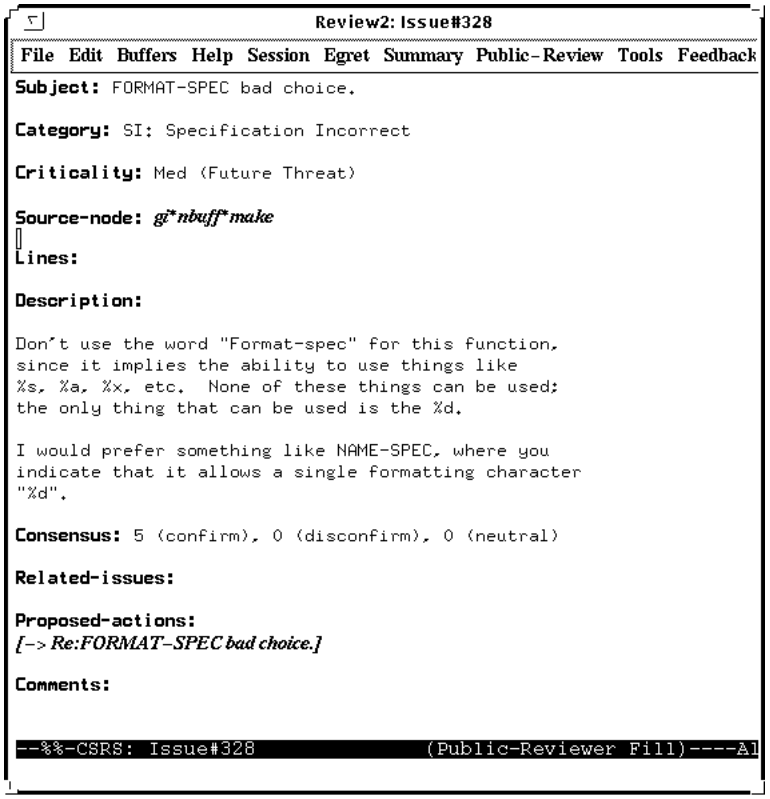

Figure 3. An issue node containing an objection to an aspect of gi*nbuff*make.

CSRS assumes that typical programming environment tools are available to the reviewer, such as static cross-referencing and dynamic behavior information, and thus does not attempt to duplicate that functionality. Part of the benefit of an EMACS-based platform is ease of integration with external programming environment tools (for example, the C/C++ environments XOBJECTCENTER and ENERGIZE, as well as Common Lisp environments by Lucid and Franz provide EMACS front-ends.)

A "standard_issues" menu supports an organization and/or application-specific set of specific issues to raise during private review. CSRS exploits the exploratory type system facilities provided by EGRET to allow reviewers to extend the set of issue types dynamically. In the gi*nbuff*make source node, this reviewer has generated 3 issues. One of them is illustrated in Figure 3.

Once the source nodes have been privately reviewed, the *public review* phase begins, where reviewers now read and react to the issues and actions raised by others. Each reviewer responds to the issues and actions raised by others through the creation of new issues or actions, creation of confirming or disconfirming evidence nodes to extant issues or actions, and by voting for one or more actions to be taken during the rework phase. In

the nbuff review, over 100 nodes of type issue, action, and comment were created by the moderator and four reviewers during the private and public review phases.

Following public review, the moderator uses CSRS to *consolidate* the review state. Consolidation involves the condensing of information generated during review into a tightly focussed, written consolidation report that delineates the proposed actions, agreements, and unresolved issues resulting from the private and public review phases. CSRS provides automated support to the moderator in traversing the hypertext database and generating a LaTeX document containg the consolidated report.

If all issues arising from the on-line phases are satisfactorily resolved, this consolidation report constitutes the final report and rework activities based upon it can be scheduled immediately. If the consolidation process yields areas of continued debate, a face-to-face meeting is then required to resolve these issues. In the nbuff review, the total of 104 issue and comment nodes were consolidated down to 19 action proposals, of which only 6 were controversial and necessitated a group meeting to resolve. The final resolution of these issues required 35 minutes of group meeting time. (CSRS has not yet been extended to same-place same-time CSCW, and thus was not used in the group meeting.)

The consolidation report constitutes the first-order benefit of CSRS: automated, collaborative support for detection, analysis, and response to defects in a software development artifact. At least as important as this, however, is a second-order benefit of CSRS: support for analysis and improvement of the FTR process itself. CSRS supports this *process maturation* through automated collection of metric data during review. For example, Figure 4 illustrates a spreadsheet (whose data was imported directly from an ASCII file generated by CSRS) that provides just one of the many interesting perspectives on the process and products of this FTR. This important application of CSRS will be explored in more detail in Section 5. The next sections provide more detail on the data and process models of CSRS.

Source code name	Size	Reviewer 1		Reviewer 2		Reviewer 3		Reviewer 4	
		Time	Iss	Time	Iss	Time	Iss	Time	Iss
gi*nbuff*read	25	2:28:47	3	0:49:08	2	0:28:39	0	0:40:43	1
gi*nbuff!pack-buffer	57	0:08:59	0	0:41:33	2		0	0:08:35	1
gi*nbuff!copy-and-pack	51	0:24:18	1	0:29:07	3		0	0:02:37	0
gi*nbuff!init-local-variables	19	0:09:06	1	0:16:57	1	0:02:49	0	0:24:00	0
gi*nbuff!unpack-field	46	0:10:19	1	0:32:28	0		0	0:08:22	0
gi*nbuff*make	20	2:04:44	5	0:36:18	2	0:14:50	0	0:13:38	1
gi*nbuff!make-link-label	34	0:01:39	0	0:40:45	1	0:00:22	0	0:04:15	0
gi*nbuff!unpack-buffer	26	1:46:29	3	0:03:42	0	0:04:22	0	0:17:22	2
gi*nbuff!make-field-label	18	0:00:42	0	0:25:39	0		0	0:03:06	0
gi*nbuff*write	33	0:06:50	1	0:09:54	2	0:07:48	0	0:14:38	0
gi*nbuff!delete-field-label	18	0:00:34	0	0:25:56	1	0:03:44	1	0:02:38	0
gi*nbuff!unpack-link	12	0:20:09	1	0:00:40	0	0:01:48	0	0:06:09	1
gi*nbuff*nbuff-p	12	0:05:11	0	0:01:08	0	0:04:16	0	0:04:15	0
gi*nbuff*node-ID	12	0:00:23	0	0:02:20	0	0:04:40	0	0:08:12	0
gi*nbuff!delete-link-label	31	0:02:08	1	0:02:08	0		0	0:02:38	0
gi*nbuff*read-hooks	5	0:00:11	0	0:07:53	1	0:00:32	0	0:00:35	0
position	12	0:00:50	0	0:00:12	0	0:03:12	1		
gi*nbuff!node-ID	4	0:00:20	0	0:00:08	0	0:00:49	1	0:00:13	0
Total	**435**	**7:51:39**	**17**	**5:25:56**	**15**	**1:17:51**	**3**	**2:41:56**	**6**

Figure 4. An Excel spreadsheet illustrating a portion of the metric data collected during the nbuff review.

3. The CSRS Data Model

The CSRS data model consists of a set of typed nodes and links, where the nodes (in code review applications) correspond to source code under review and related artifacts generated during review, and the links represent relationships among the nodes. Source nodes are further classified into specific program objects such as function, macro, structure, etc. The exact set of program objects is language-specific: for Ada, CSRS would provide a "package" program object, for C++, a "class" object, and so forth. Review nodes are classified into issue, action, comment and evidence. Figure 5 shows the basic relationship among the CSRS nodes and links.

As illustrated in the previous section, any suspected problems or defects in the source are documented in issue nodes. Any questions about the source are recorded in comment nodes. Once issues are identified, action nodes are used to represent their resolution, either by proposing specific solutions, by proposing that the issue be ignored, or by indicating that no solution is known. Similar to source nodes, one may request clarification about issues or actions posted by others through comment nodes, and obtain responses by following respond-to links. The resulting comment may further suggest new issues or actions. Disagreement about specific issues or actions are represented through evidence nodes that disconfirm the corresponding issues or actions. Similarly, support for issues or actions is represented by following a confirming link to an evidence node.

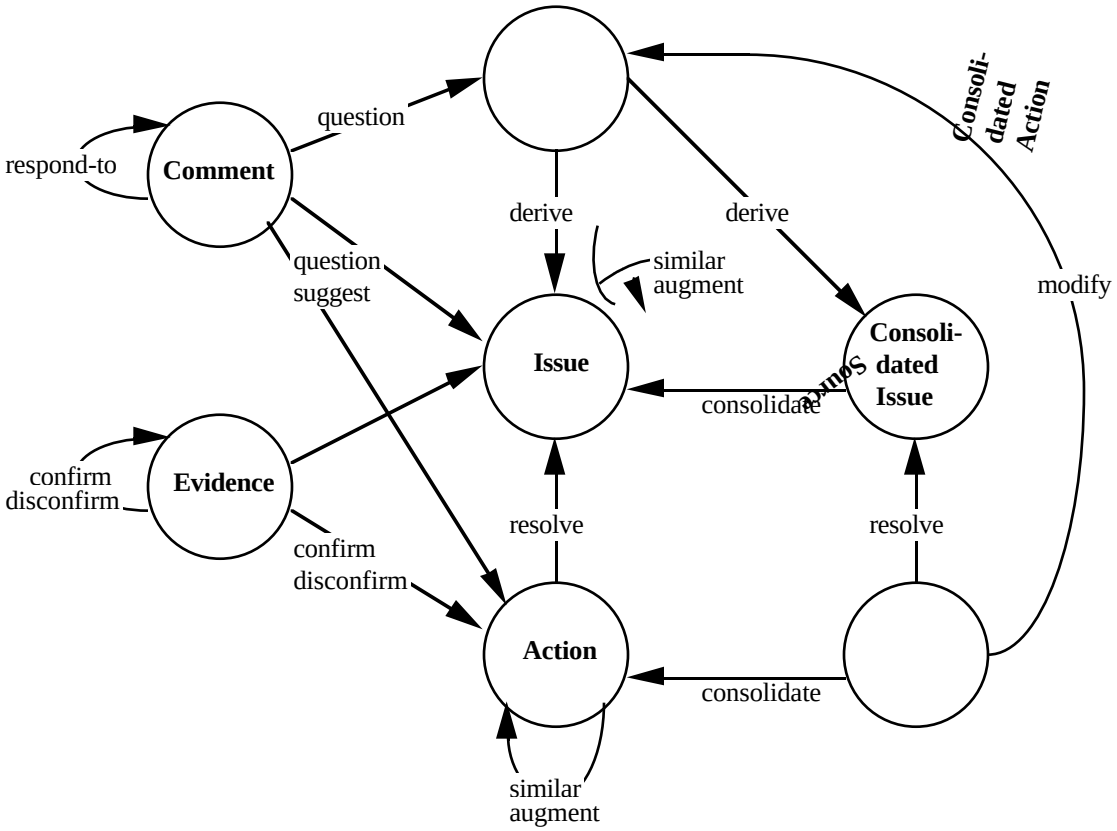

Figure 5. The CSRS data model.

Private review inevitably leads to redundant issue generation, as individuals discover the same problems. During public review, such similarities can be made explicit through the creation of similar-to links. Elaboration of issues or actions can be represented through augment links.

Finally, related issues are summarized into a single consolidated issue, and related actions into a single consolidated action. Actual rework activities are based upon a single consolidated action, rather than by reference to the individual issues and actions raised during public and private review.

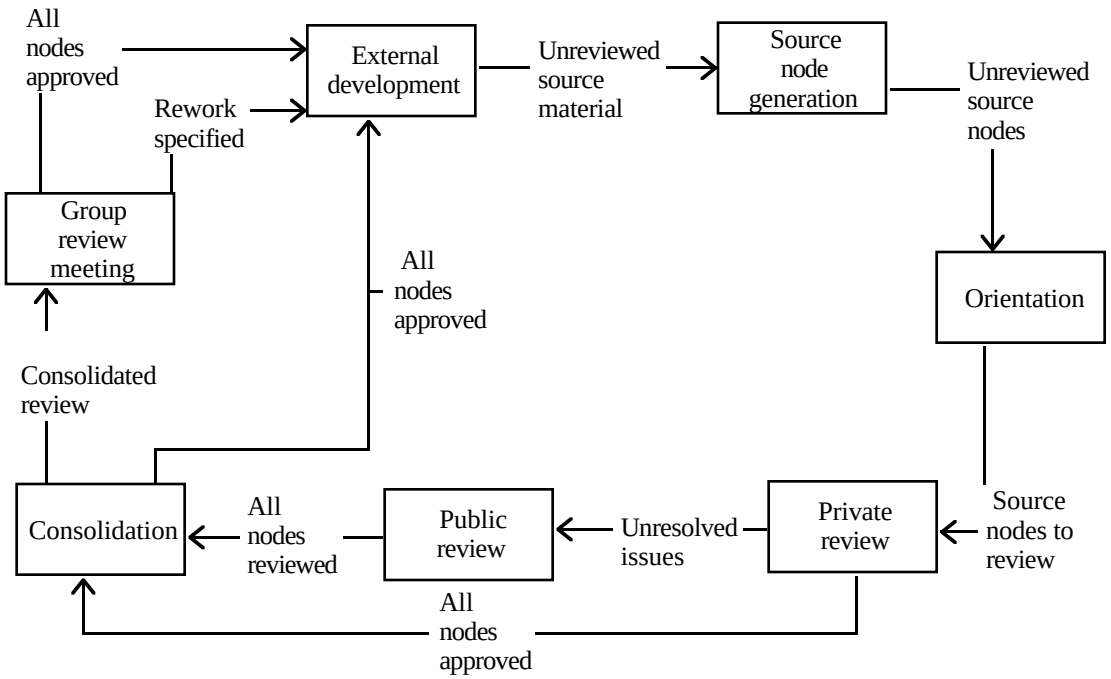

Figure 6. The CSRS process model.

4. The CSRS Process Model

The CSRS process model involves seven phases, as illustrated by the state-transition diagram in Figure 6. The process model constrains the data model by specifying what nodes and links can be legally manipulated during any given phase. The process model also defines specific participant roles similar to the ones defined in Fagan's formal inspection method (Fagan, 1976). The roles include moderator, producer, and reviewer, but eliminates the scribe role.

The next paragraphs describe the phases in CSRS process model, each of which appear as states in Figure 6. Certain administrative procedures such as calling the meeting, selecting participants, and so forth are omitted from this description. For illustrative purposes, the process model is presented in the context of code review, though review of other development artifacts follows the same general procedure.

Source node generation. In this phase, the source code producer with the assistance of the moderator generates source nodes from provided source files. The files are programmatically split into source nodes and annotated as necessary with supplemental information. When the nodes contain source code that has been previously reviewed, backlinks to the consolidated action nodes from the prior review can be created. These links help reviewers check whether prior rework requests have been properly implemented.

Orientation. In this phase, review participants are prepared for private review. Depending upon the nature of review (new review or re-review) and/or group makeup (participants' skills and familiarity with the product), this phase may range from a formal overview meeting with a presentation by the producer about the system structure and behavior, to an informal notification through e-mail noting the presence of new nodes ready for review using CSRS.

Private review. In this phase, reviewers inspect source nodes privately. They create issue, action and/or comment nodes. Issue and action nodes are not publicly available to other reviewers at this time, but comment nodes (clarification about the logic/ algorithm of source nodes) are publicly available. Normally, the producer or moderator responds to comment nodes posted by reviewers. Reviewers must explicitly mark each source node as reviewed, which allows them greater flexibility in their internal review process.

While reviewers do not have access to each other's state during private review, the moderator does. This allows the moderator to monitor the progress of private review. While private review normally terminates when all reviewers have marked all source nodes as reviewed, the moderator may move on to public review at any time.

Public review. In this phase, review participants (including the producer) react to all generated issue and action nodes. Reviewers can create new issue, action or comment nodes based upon existing nodes. They may create similar links between two issue or action nodes upon reading the content of the nodes. They may create new issues or actions that augment the existing ones. They may also post evidence nodes which confirm or disconfirm particular issue, action or other evidence nodes. They can also indicate their agreement about existing actions by voting; this information is used as a consensual indicator among participants about the most appropriate action to take on the issue. This phase normally concludes when all issue, action and evidence nodes have been marked as reviewed by all reviewers. However, control over when public review ends is ultimately in the hands of the moderator.

Consolidation. In this phase, the moderator creates a consolidated representation of the state of review thus far, oriented around the set of actions required based upon review. CSRS supports the moderator in the preparation of a written report containing this information.

Group review meeting. If the consolidated report identifies areas of continued controversy, a group meeting is required. In the meeting, the moderator presents the unresolved issues or actions and summarizes the differences of opinion. After discussion, the group may vote to decide them, or the moderator may unilaterally make the decision. The moderator notes the decision in the written report. The moderator may also declare issues unresolved if no consensus can be reached: this can either imply that informal discussion should proceed after the review or the final resolution will be decided at some later point. Finally, the moderator may also decide to reinspect some source nodes during the meeting as a group (e.g., the nodes that are difficult to understand by the majority of

reviewers). The producer will lead this later activity; the reviewers raise issues and proposed actions verbally, and the recorder records only the consensual issues or actions.

External development. In this phase, actions decided upon by the review process are carried out, as well as any other development activities motivated by non-review concerns (such as changes to functional requirements, platform porting, and so forth.) This phase occurs outside the boundaries of CSRS and results in new or modified source code that has not been explicitly approved through review. External tools can be employed to automatically input non-approved source code into CSRS as required by the overall software development process model.

5. Software Process Improvement using CSRS

As illustrated in the scenario of code review in Section 2, the use of a specialized computer-supported cooperative work environment for review provides major benefits. First, it results in a richly linked repository for all artifacts of review, providing a resource that facilitates rework, project scheduling, access to design/maintenance rationale, and so forth. Second, it decreases the dependence in traditional review methods on same-time same-place group work, by providing an additional avenue for collaboration. Third, it provides automated support for the roles of producer, moderator, and reviewer, and essentially eliminates the role of scribe, thus decreasing the labor-intensive manual activities implicit in traditional review methods.

However, the use of CSRS provides an additional, equally significant benefit: the ability to easily instrument review in order to collect metrics that facilitate maturation of its process and products.

CSRS measures the review process by creating a log of each session, where timestamped events are generated by an appropriate subset of the user actions. Currently, actions such as retrieval of a CSRS node, locking and unlocking, node and link creation and deletion, and so forth are all time-stamped and spooled to a log. This logfile of events is then post-processed off-line to generate more condensed, useful information such as that illustrated in the spreadsheet in Figure 5.

Depending upon the organization, application, and group characteristics, some degree of process maturation may be possible via informal analysis of the historical values of these metrics from prior reviews. For example, an organization might find relatively simple correlations between such aggregate metrics as the number of source code lines, the amount of time spent during review, the number of issues and actions raised, and the number of participants. Such analysis can allow rough quantitative boundaries to be placed around review parameters, such as "the number of reviewers should be between 4 and 7", or "the number of lines of code to be reviewed should not exceed 800, nor should the number of source code nodes exceed 12."

On the other hand, CSRS can also be used to provide more formal and robust empirical data concerning the review process. For example, some researchers have claimed that the use of standardized checklists can improve the quality of review, by forcing reviewers to explicitly "check off" each item in a list as being satisfied by the artifact under study (Knight, 1991). This claim can be disputed by arguing that if the standardized checklist is not comprehensive, faults not covered by the checklist will be far less likely to be uncovered. CSRS provides environmental support for empirical investigation of this issue. For example, this particular question could be investigated via a simple repeated measures experimental design where a set of reviewers and a set of source nodes are split into two groups, and each participant reviews and each node is reviewed both with and without a standardized checklist. Such an experiment could also answer such questions as: Is my checklist comprehensive? Do checklists lengthen or shorten the time of review? Do checklists improve the productivity of review in terms of issues identified per unit time?

Having now discussed the CSRS environment, process and data model, and application to process maturity, the next section turns to a contrast of CSRS with other research concerning review.

6. Related Work

While the idea of facilitating review process with computer support is not a new one, we know of no other system that supports review as comprehensively as CSRS. Other computer based review systems, such as ICICLE (Brothers, 1990) support only the face-to-face group meeting by synchronizing and sharing the code window among participants; the use of hypertext technology is basically untyped, non-context sensitive, and limited to source code annotation.

CSRS provides similar benefits to those shown for other forms of computer-mediated meeting support (Nunamaker, 1991). This research indicates that activities such as private review, public review, and moderator consolidation can be more effective and efficient than traditional face-to-face meetings. Such activities decrease process losses such as attention blocking, air time fragmentation, domination, and many others. In fact, educational materials on software inspection published by Software Engineering Institute (Deimel, 1991) discuss these exact problems in the context of review meetings: the moderator dominates inspection, the producer is under attack, the reader is too fast, and so forth.

Martin and Tsai (1990) have shown that having *n* small teams inspect the same source is more effective (in the sense of generating more faults) than a single large team working together. Our past experience with traditional inspection also showed that faults detected by different reviewers overlapped very little. Nunamaker (1991) describes this same phenomena as reduction in free-riding by individual team members. The private review phase in CSRS is designed in light of this research.

The CSRS instrumentation also helps reduce free-riding. Freedman (1990) states that a major responsibility of the review leader is to ensure that participants come prepared to the review, discusses the seriousness and frequency of insufficiently prepared review members, and suggests several ways to address the problem. For example, they suggest deliberately leaving out one page of the review artifact when distributing materials to reviewers---those who adequately prepare will notice its absence and contact the leader. CSRS supports a much less devious approach: the environment records which source nodes have been visited, how long they have been visited, and whether they have been marked as reviewed by each member. Since each member's summary window presents such information, and since this information is known to be available to the moderator, there is no point in trying to "bluff" one's way through the review meeting.

On the other hand, group review has its own benefits, since participants can learn from each other, issues raised by one participant may stimulate other participants to raise new issues, and a more objective evaluation can be obtained (Nunamaker, 1991). CSRS follows the private review phase by a public review phase in order to benefit from group review as well.

The CSRS data model is similar to the one used by gIBIS (Conklin, 1988) for issues exploration and deliberation. However, the CSRS data model is specialized to review, and thus allows more specific computational support and metrics collection. Also in contrast to gIBIS, our data model is strongly tied to the process model and thus context-sensitive: most artifacts can only be manipulated at certain phases defined in the process model.

InspeQ (Knight, 1991) is a review method whose contribution primarily rests in its detailed description of a set of review phases, each of which contains an explicit checklist for reviewers to follow. CSRS can be easily tailored to the InspeQ process model. In fact, as illustrated in Section 5, CSRS can even be used to empirically determine whether or not such a tailoring is desirable.

Finally, in contrast to traditional formal inspection (Fagan, 1976), CSRS supports proposals for a course of action for each identified issue. Our experiences with traditional inspection revealed to us that if actions are prohibited from the review process, the programmer who is assigned to fix the errors often misunderstands the "real" issue. Thus, action nodes can serve as issue clarification.

7. Current Status and Future Directions

The CSRS data and process model has been under development since 1991, with refinements motivated by a variety of review experiences at both code and design level. The current implementation of CSRS employs a Unix/X windows environment, with a database server back-end written in C++ (Wiil, 1990) and a Lucid EMACS front-end interface. Such a choice of environments makes CSRS easily integrable with current development practices. For example, source code can be read from RCS or SCCS, and

CSRS can automatically generate e-mail messages to inform reviewers of the state of review.

CSRS displays a property typical to most CSCW systems: introducing collaborative support into a work process dramatically changes the work process. Fagan's code inspection method, for example, is intimately tied to a work process lacking CSCW support, and we are finding that many of the concepts (such as the role of scribe) and constraints (strict prohibition of action discussion) to be irrelevent or inappropriate for a CSCW-based FTR tool.

Currently, CSRS is in active use within our research group, and we plan to release a public domain version for external evaluation within the next year. Our current work concentrates on improving the usability of the system and the precision and expressiveness of the metric information. We are beginning to build a database of review data based on our personal experience with CSRS. A primary goal of public release is to obtain data about review across a spectrum of applications, organizations, and review methods.

A long-range goal for CSRS is explicit support for learning during the review process. We believe that the greatest potential review has for improving software quality comes not from its ability to uncover faults in programs, but from its ability to uncover faults in programmers. Review holds incredible potential to teach programmers how to write correct, readable, and maintainable software, yet such a benefit is rarely recognized, much less explicitly supported. We hope to use CSRS to discover effective ways to support this aspect of software review.

Acknowledgments

We gratefully acknowledge the other members of the Collaborative Software Development Laboratory, who have contributed greatly to this research: Dadong Wan, Kiran Kavoori, and Robert Brewer. Support for this research was provided in part by the National Science Foundation Research Initiation Award CCR-9110861 and the University of Hawaii Research Council Seed Money Award R-91-867-F-728-B-270.

References

L. Brothers, V. Sembugamoorthy, and M. Muller (1990): ICICLE: Groupware for code inspection. In *Proceedings of CSCW'90*, pp. 169-181. ACM Press.

Jeff Conklin and Michael L. Begeman (1988): gIBIS: A hypertext tool for exploratory policy discussion. In *Proceedings of CSCW'88*, pp. 140-152. ACM Press.

Lionel E. Deimel (1990): *Scenes of Software Inspections: Video Dramatizations for the Classroom.* Software Engineering Institute, Carnegie Mellon University.

Robert Dunn (1990): *Software Quality: Concepts and Plans.* Prentice Hall.

Michael E. Fagan (1976): Design and code inspections to reduce errors in program development. *IBM System Journal*, 15(3):182--211.

Michael E. Fagan (1986): Advances in software inspections. *IEEE Transactions on Software Engineering*, SE-12(7), pp. 744-751.

D. P. Freedman and G. M. Weinberg (1990): Handbook of Walkthroughs, Inspections and Technical Reviews. Little, Brown.

Philip M. Johnson (1992): Supporting exploratory CSCW with the EGRET framework. In *Proceedings of CSCW'92*, ACM Press.

John C. Knight and E. Ann Myers (1991): Phased inspections and their implementation. *Software Engineering Notes*, 16(3):29-35.

Johnny Martin and W. T Tsai (1990): N-fold inspection: A requirements analysis technique. *Communications of the ACM*, 33(2):225-232.

J. F. Nunamaker, Alan R. Dennis, Joseph S. Valacich, Douglas R. Vogel, and Joey F. George (1991): Electronic meeting systems to support group work. *Communications of the ACM*, 34(7):42--61.

Glen W. Russel (1991): Experience with inspection in ultralarge-scale developments. IEEE Software, (9)1.

G. Gordon Schulmeyer and James I. McManus (1987): Handbook of Software Quality Assurance. Van Nostrand Reinhold.

U. Wiil and K. Oesterbye (1990): Experiences with hyperbase-a multi-user back-end for hypertext applications with emphasis on collaboration support. *Technical Report 90-38*, Department of Mathematics and Computer Science, University of Aalborg, Denmark.

Edward Yourdon (1989): *Structured Walkthrough.* Prentice-Hall, Fourth Edition.