Innovations Syst Softw Eng (2010) 6:311–318
DOI 10.1007/s11334-010-0136-x

ORIGINAL PAPER

# SCRUB: a tool for code reviews

**Gerard J. Holzmann**

Received: 25 December 2009 / Accepted: 23 July 2010 / Published online: 6 August 2010

**Abstract** This paper describes a tool called Source Code Review User Browser (SCRUB) that was developed to support a more effective and tool-based code review process. The tool was designed to support a large team-based software development effort of mission critical software at JPL, but can also be used for individual software development on small projects. The tool combines classic peer code review with machine-generated analyses from a customizable range of source code analyzers. All reports, whether generated by humans or by background tools, are accessed through a single uniform interface provided by SCRUB.

**Keywords** Peer code review · Static source code analysis · Coding rules · Coding guidelines · Coding standards · Compliance verification

## 1 Introduction

Code review is an important part of any serious software development effort. The objective of a review is to make sure that code is understandable, well-structured, and free from design flaws and coding defects. The review process is frequently also used to make sure that code complies with project-specific coding guidelines and institutional coding standards (e.g., [5]).

Many different methods have been proposed for conducting code reviews, ranging from highly structured processes (e.g., [1,2]) to informal, collaborative peer reviews (e.g., [9]). The more formal processes are based on a line-by-line inspection by a group of experienced developers, in extended meetings. Experience shows, though, that the line-by-line reviews lose most of their benefit if more than a few hundred lines of code is covered per review session [6,7].

In larger projects, conducting detailed formal code reviews can quickly become excessively time consuming. If, for instance, three people review one million lines of code, covering maximally 500 lines per day, the process would take 2,000 days to complete (8–10 years). Looking at it differently, if 3 months are available, the same process would take 30–40 teams of reviewers. With three developers per team, this would consume the fulltime efforts of 90–120 developers for the 3-month period. Clearly, this type of investment will not be possible in all but the rarest cases (e.g., [8]).

If we look more carefully at the code review process, we can see that a large portion of that process is typically spent on things that can also be automated. This applies to all issues that relate to coding style, compliance with coding standards, the use of defensive coding techniques, and the detection of common types of defects. We can significantly reduce the human review effort that is required by leveraging the use of tool-based analyzers. The advantage of tool support becomes more evident as a code base grows. Especially in these cases, tools can perform routine checks more thoroughly and consistently than human peer reviewers. The Source Code Review User Browser (SCRUB) tool that we describe here is meant to leverage the availability of state-of-the-art static source code analysis techniques to support a significantly more efficient code review process.

### 1.1 Leveraging tool-support

Static source code analyzers have become quite effective at identifying common coding defects in large code bases with a low rate of noise. The leading commercial tools

G. J. Holzmann (✉)
Laboratory for Reliable Software,
Jet Propulsion Laboratory/California Institute of Technology,
Pasadena, CA 91109, USA
e-mail: gholzmann@acm.org

include Codesonar, Coverity, and Klocwork, each of which can achieve a low rate of false-positives in the warnings that they generate. Still, the time required to analyze one million lines of code with these tools can vary. The slowest, and most accurate, tools can take several hours to analyze the code; the fastest tools can deliver results in minutes. In each case, though, the tools produce results that would be difficult to realize with purely human-based code inspection. Each tool has different strengths, so there is often surprisingly little overlap between the reports from different tools, even when they are used to search for the same types of defects. In our application, we therefore use multiple analyzers, to increase the chances that the important defects are caught. The use of multiple analyzers in this context can be compared with the use of additional human reviewers can improve the quality and reach of a human peer review process.

Compilers have also gained notable strength in their code analysis capabilities. With all warnings enabled, the output from standard compilers like gcc can be used as a light-weight (and generally fast) form of static analysis.

Analysis tools do of course also have their limitations. Although they can accurately spot coding defects and suspicious code fragments, they are unlikely to uncover higher-level design errors or algorithmic flaws. To catch these higher level flaws we must rely on the use of stronger design verification tools, e.g., logic model checkers [4], whose operation can less easily be scripted, and of course the human insight of peer reviewers. The use of tools can strengthen the code review process in important ways, and reduce the burden on human reviewers to scrutinize ever line of code, but they cannot replace the human reviewer completely.

When we choose to run multiple sophisticated source code analyzers in parallel, it is not realistic to expect developers (and reviewers) to become knowledgeable in the operation of each separate tool, or to keep track of which tools should be considered "best-in-class" at each point in time. Each tool typically comes with its own interface to capture the results of an analysis, which in some cases is accessible only through a web-browser. A single unified and secure interface that can give access to all tool results, as well as peer-generated input, is clearly preferable.

These observations have been the motivation for the development the SCRUB tool. SCRUB provides developers and reviewers with a uniform interface to the source code and all related review and analysis artifacts. The tool runs as a stand-alone application, avoiding the security problems related to web-access by inheriting only existing access rights from the user. The tool, for instance, cannot open any files or folders that the user does not already have access rights to. This means that the tool does not need to enforce or administer any additional security policies.

All analysis results presented through the SCRUB tool's user interface are computed off-line, given that especially for larger projects this computation can take longer than would be appropriate for interactive tool use. Peer comments, though, can be entered interactively at any time by anyone with access to the code and the tool.

In Sect. 2, we describe the code review process that is supported by the SCRUB tool. Section 3 details the design of the SCRUB tool itself. Section 4 presents our experience in the use of this tool on a large JPL project, and Sect. 5 concludes the paper.

## 2 Code review process

The code review process supported by the SCRUB tool consists of three phases.

- *Review*: peer reviewers study the code and enter their comments and observations into the SCRUB tool.
- *Response*: the developer responds to all reports, indicating agreement, disagreement, or the need for more information.
- *Resolution*: a resolution is reached on all pending issues, specifically on disagreements between the developer and the peer reviewers' comments and tool reports.

Tool-based analysis reports are generated for each version of the code checked into the version management system, so no separate step to generate these reports is required: they are always available, and archived for every version of the code. The main steps in the review process are described in more detail below.

### 2.1 The review phase

Although comments and tool reports can be collected on the source code at any time, the formal code review process starts only when a review for a software module is scheduled, and specific peer reviewers are assigned. At this time a date for a *closeout* review is also scheduled.

The peer reviewers are asked to study the code and identify design defects and coding issues that might be beyond the scope of the analyzers (the analysis reports are available to them). Reviewers typically get one or two weeks to do enter their comments into the SCRUB tool, by annotating the code.

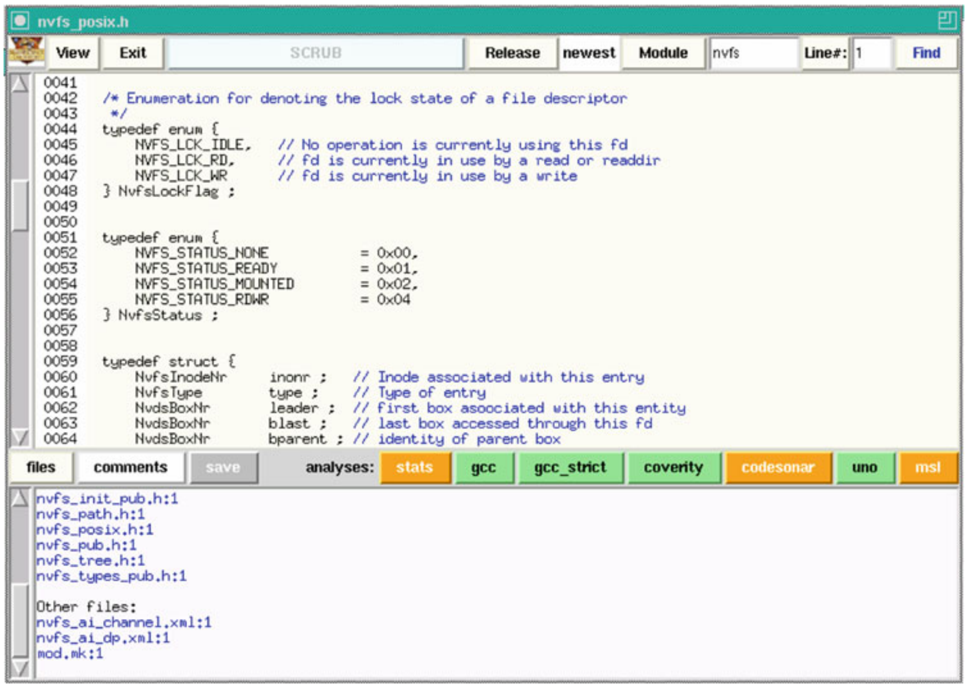
At JPL, the peer reviewers are generally members of the same development team, not directly involved in the development of the module being reviewed. Because they are part of the same development team, the reviewers can be assumed to be familiar with the general context in which a module operates, but they generally will have to acquaint themselves with the specific functional requirements of the module being reviewed. A summary of the coding standard to be used and a review checklist is available to the reviewers in the *View* menu, shown at the top-left of the SCRUB display in Fig. 1.

The reviewers enter their feedback, working separately, by selecting a source file to be reviewed from the *files* display, shown at the bottom left in Fig. 1. The list generally contains .c and .h source files, but it can also contain .xml and .txt files, and inputs to code generators.

The selection of a file, with a mouse click, brings up a numbered source code listing in the top panel. Another mouse click on a line number in this listing brings up a comment box, shown in Fig. 2, where the reviewer can enter a comment or observation associated with that line, and assign it a priority.

The tool labels each comment with the filename and line-number that was selected, in a field that the user cannot modify. Clicking the *Add Comment* button causes SCRUB to assigns a unique identifier to the comment and to enter it into the review repository. The unique identifier is formed from the user's login name followed by a four-digit sequence number (cf. Fig. 3). The list of all review comments entered in this way are available to all reviewers. A merged list of all review comments will be displayed, for instance, whenever a user selects the *comments* button in the middle menu bar. SCRUB automatically sorts all comments entered by filename and linenumber, to make it easier to correlate the observations on the same fragment of code from multiple reviewers.

Once all comments have been entered the review can enter the next review phase where the module owner is requested to respond to all peer comments and tool reports.

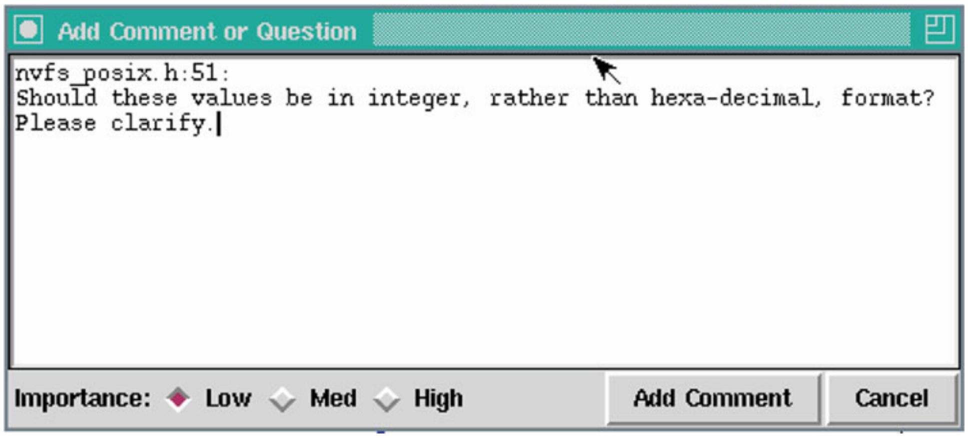


```
0041
0042  /* Enumeration for denoting the lock state of a file descriptor
0043   */
0044  typedef enum {
0045        NVFS_LCK_IDLE,      // No operation is currently using this fd
0046        NVFS_LCK_RD,        // fd is currently in use by a read or readdir
0047        NVFS_LCK_WR         // fd is currently in use by a write
0048  } NvfsLockFlag ;
0049
0050
0051  typedef enum {
0052        NVFS_STATUS_NONE            = 0x00,
0053        NVFS_STATUS_READY           = 0x01,
0054        NVFS_STATUS_MOUNTED         = 0x02,
0055        NVFS_STATUS_RDWR            = 0x04
0056  } NvfsStatus ;
0057
0058
0059  typedef struct {
0060        NvfsInodeNr      inonr ;   // Inode associated with this entry
0061        NvfsType         type ;    // Type of entry
0062        NvdsBoxNr        leader ;  // first box associated with this entity
0063        NvdsBoxNr        blast ;   // last box accessed through this fd
0064        NvdsBoxNr        bparent ; // identity of parent box
```

```
nvfs_init_pub.h:1
nvfs_path.h:1
nvfs_posix.h:1
nvfs_pub.h:1
nvfs_tree.h:1
nvfs_types_pub.h:1

Other files:
nvfs_ai_channel.xml:1
nvfs_ai_dp.xml:1
mod.mk:1
```

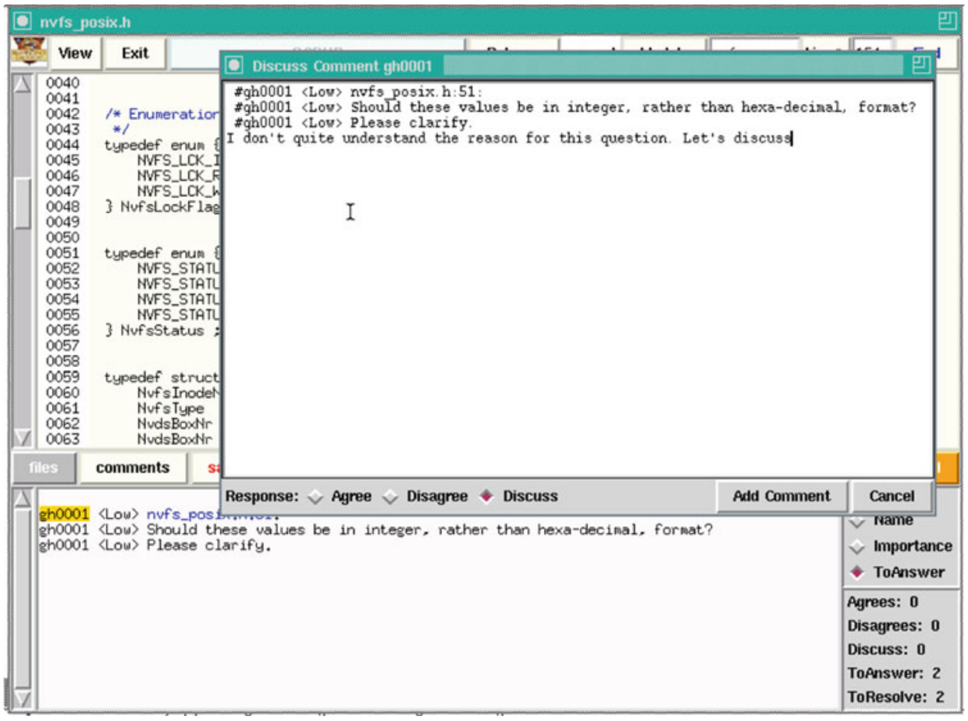
**Fig. 1** Standard SCRUB user interface. The project logo is shown in the top left. The name SCRUB is acronym for "Source Code Review User Browser." Background tools used include the static analyzers Coverity, Codesonar, and Uno [3]



```
nvfs_posix.h:51:
Should these values be in integer, rather than hexa-decimal, format?
Please clarify.
```

Importance: [✓ Low] Med High

**Fig. 2** Comment box displayed after the user clicks on line 51 in the main window shown in the sample display from Fig. 1. The user has selected a low priority comment here



```
/* Enumeratior
 */
typedef enum {
     NVFS_LCK_I
     NVFS_LCK_R
     NVFS_LCK_W
} NvfsLockFlag


typedef enum {
     NVFS_STATU
     NVFS_STATU
     NVFS_STATU
     NVFS_STATU
} NvfsStatus ;


typedef struct
     NvfsInodeN
     NvfsType
     NvdsBoxNr
     NvdsBoxNr
```

```
 #gh0001 <Low> nvfs_posix.h:51:
 #gh0001 <Low> Should these values be in integer, rather than hexa-decimal, format?
 #gh0001 <Low> Please clarify.
I don't quite understand the reason for this question. Let's discuss
```

```
gh0001 <Low> nvfs_posi
gh0001 <Low> Should these values be in integer, rather than hexa-decimal, format?
gh0001 <Low> Please clarify.
```

```
Agrees: 0
Disagrees: 0
Discuss: 0
ToAnswer: 2
ToResolve: 2
```

**Fig. 3** Follow-up and/or developer response prompt, shown when a comment identifier, shown in the lower panel, is selected. The response can be entered in free-form text, and is again tagged with a unique identifier when inserted into the comments list

## 2.2 The response phase

Responses to reviewer comments and tool reports can be entered into the tool at any time, but the last few days before a closeout review are normally reserved for this. A response can be entered by clicking on the comment identifier (which is colored yellow in the *comments* view). This selection brings up the response box shown in Fig. 3. A response category can be selected, with an optional explanation.

Any user can enter a response or a follow-up to any peer comment or tool report in this way, and express agreement or disagreement, but only the module owner's response will determine the actions to be taken in *Resolution* phase of the review (discussed in the next section). The tool itself does not know who the module owner is, but the reviewers of course do.

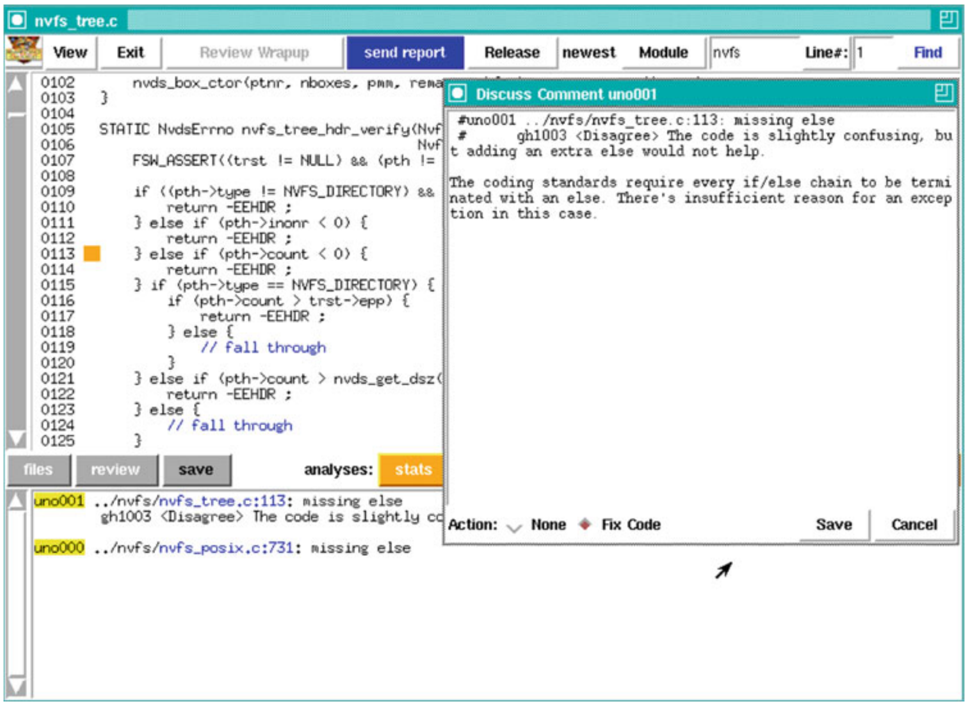
The sequence of comments or observations from one or more users, and concluded with the module owner's response, becomes part of the record of the code review and as such forms an important part of the process. The module owner is *required* minimally to mark all comments and reports with an *Agree*, *Disagree*, or *Discuss* tag, optionally followed by a more detailed explanation. To reinforce this, a failure to enter a response is assumed by SCRUB to default to an *Agree* response, and will automatically lead to an assignment to address the comment in the action list that is generated at the conclusion of the code review.

As we will see in more detail in Sect. 4, the most commonly used response to peer comments and tool reports is an explicit *Agree* response. The least often chosen response is a *Disagree*. The fact that explicit *Disagrees* are less common significantly increases the efficiency of a SCRUB based code review: only *Disagree* and *Discuss* responses are generally discussed in the final resolution phase.

## 2.3 The resolution phase

The final step in the code review process is the only step that requires a short face-to-face meeting of developer, reviewers, and project managers. Each *Disagree* and *Discuss* response from developer is discussed at this meeting. The SCRUB tool is now placed in a separate *Review Closeout* mode (with a selection through the View menu). In this mode, all comments are sorted by level of importance, and with *High* priority *Disagree* and *Discuss* responses listed first, followed by Medium priority and finally *Low* priority *Disagree* and *Discuss* responses, and followed by all remaining *Agree* responses that generally need not be discussed at the meeting. Within each categories comments are sorted by filename and linenumber.

Comment sequences and responses are again selected by clicking the yellow comment identifier, which this time brings up a resolution box that allows for a final designation of *No change required* or *Code fix required* for each comment and tool report, with optional explanation to document the reasons for the choice. This is illustrated for a sample tool report in Fig. 4. Note that a *Disagree* response from a module owner can be overruled in the closeout review with the resolution *Fix Code*.



```
0102        nvds_box_ctor(ptnr, nboxes, pmm, rema
0103    }
0104
0105    STATIC NvdsErrno nvfs_tree_hdr_verify(Nvf
0106                                          Nvf
0107        FSW_ASSERT((trst != NULL) && (pth !=
0108
0109        if ((pth->type != NVFS_DIRECTORY) &&
0110            return -EEHDR ;
0111        } else if (pth->inonr < 0) {
0112            return -EEHDR ;
0113        } else if (pth->count < 0) {
0114            return -EEHDR ;
0115        } if (pth->type == NVFS_DIRECTORY) {
0116            if (pth->count > trst->epp) {
0117                return -EEHDR ;
0118            } else {
0119                // fall through
0120            }
0121        } else if (pth->count > nvds_get_dsz(
0122            return -EEHDR ;
0123        } else {
0124            // fall through
0125        }
```

```
uno001 ../nvfs/nvfs_tree.c:113: missing else
       gh1003 <Disagree> The code is slightly cc

uno000 ../nvfs/nvfs_posix.c:731: missing else
```

```
#uno001 ../nvfs/nvfs_tree.c:113: missing else
#       gh1003 <Disagree> The code is slightly confusing, bu
t adding an extra else would not help.

The coding standards require every if/else chain to be termi
nated with an else. There's insufficient reason for an excep
tion in this case.
```

Action: None [✓ Fix Code]

**Fig. 4** SCRUB in Closeout Review mode, showing support for resolving a tool report with a Disagree response from a module owner, which is resolved with a Fix Code designation in a sample closeout review action

The entire review track for each issue that is raised by either human peer reviewers or by analysis tools, including the final disposition of each comment and report, is archived in the SCRUB database for the reviewed version of the code, as a complete record of the review process. A typical record of a discussion can look, for instance, as follows:

tcanham0016 <Low> nvfs_posix.c:1657
tcanham0016 <Low> An unnecessary test. How could this fail?

> rjoshi1168 <Disagree> There are two specific cases where the test could fail, although both are exceptional cases.
>
> scandore2009 <Code Fix Requested> Change the test into an assertion.

The enactment of resolutions reached in the final phase of the code review process is done through an assignment from the project manager to the developer to make all agreed upon changes to the code. When the modified version of the code becomes available, one of the peer reviewers is assigned the task of verifying that all changes were correctly made (and no others). The code is also re-checked, by rerunning all analyzers, to make sure no new issues appeared. The user can select to see only new analyzer reports, not known at the time of the code review.

Although in the examples we have only shown the discussion of peer comments, the working of the SCRUB tool is identical for tool generated reports. The tool reports are available with single button clicks on the middle menu bar, also visible in Fig. 1. All tool reports are imported by SCRUB in a common format that matches the peer entered comments. File and line references are always in a uniform format that is selectable with a button click to bring up the corresponding fragment of code in the source code view panel. This hot-linking of file and line references is supported in all SCRUB panels to facilitate a rapid evaluation of the context for tool and peer reports.

## 3 Tool design

As shown in the figures, the SCRUB display contains two menu bars: one at the top and one in the middle. Below the top menu bar is the main source code view panel. At the bottom of the display, below the middle menu bar, is a separate panel that is used to display the review related information, a files menu, and some general statistics on code quality. The row of buttons on the right-side of the middle menu bar can be used to display analysis reports from the corresponding background tools, where the specific selection of background tools to be used can be customized. Included in the selection of background tools are compiler warnings, both for a basic default compilation of the code and for a strict compilation with all warnings enabled and the pedantic flag set.

### 3.1 Visual feedback

For immediate visual feedback of code quality, the buttons that correspond to tool-reports are colored green when there are no warnings, orange when there are ten or fewer warnings, and red when there are more. The ideal outcome of the code review process is to reach a state where all buttons can turn green. A project manager can quickly assess the quality of the code by browsing relevant modules, checking button colors, and where necessary looking at basic statistics about code size, numbers of warnings, and any unusual features of the module that were detected (such as the use of unusual header files), which are collected in the *stats* view.

The source code view panel is designed to be strictly read-only, since code reviews are always performed for fixed releases of the code. Any changes to the code that are made can only affect future releases of the code, not the current view. This feature also guarantees the validity of the file and line number references in the tool reports and reviewer comments: they will always match the code displayed. All source code references are specified as a filename followed by a colon and a line-number. In large projects, a source files typically have a prefix that identifies the module it is part of, which makes it simple for the SCRUB tool to locate the file. If this fails, the tool will look in the current directory and its sub-directories to find a source file.

The SCRUB database is organized in such a way that there cannot be any conflicts caused by shared access to any of the files used in the SCRUB review process, no matter how many reviewers or on-lookers are browsing the code and entering comments in parallel. SCRUB achieves this without the use of a locking discipline. The benefit is that the user never has to deal with the possibility of delayed writes, stale locks, or corrupted files. Every file used always has a unique single owner who has write permission on that file. All other users can read these files, but not modify their contents. Each user, for instance, creates and stores a separate copy of all merged comments in a file named *comments.username*. When the comments display is updated, all files of this type are sorted and merged, and duplicates are removed before the list is displayed by SCRUB. Because a comment sequence can only grow in length over time, as more responses are added, the merge tool has a fairly simply job in deciding which version of a comment discussion is most up to date: it is always the longest such sequence.

## 3.2 Generating analyzer reports

Generating all the tool reports displayed in SCRUB in a global analysis of all code can be time-consuming and is therefore done offline, once for each new release of the code. For one JPL project we routinely scan about three million lines of flight code, performing detailed analyses for SCRUB that can take up to 10 h per release.

Each analysis tool has its own special usage prescripts, so there is no general principle that can be followed to define the analyzer invocations. We use a script, called PreSCRUB for this purpose. The script extracts the information it needs from a build log of the code (e.g., the output of make). All analyzers SCRUB uses work by running a variant of the standard C compiler, using the same command line arguments that were used for the actual compilation. So it normally suffices to extract all compiler calls, and to replace the compiler invocations with the invocation of each appropriate analysis tool, after which the script collects and reformats all analysis results for display in SCRUB.

Many tools can produce their output in xml format, or as plain text which makes post-processing simple. Other tools, e.g., Grammatech's CodeSonar tool, are more tightly integrated with a web-browser. In the latter case the analysis results are retrieved from the web-browser by running additional scripts that retrieve web-pages, extract the essential contents from them, and then reformat the results for use in SCRUB. Most of this is standard shell-scripting, and is easily customized or extended to support additional or different analyzer tools.

For the Coverity tool we run a range of additional checkers, written in Coverity Extend, to check compliance with various coding rules (e.g., [5]). We also run some extra scripts to check for things that are more difficult to specify through the available tools.

Apart from the global analysis of all source code used in a project, SCRUB also supports a mode where the developer working on a single module in a sandbox can invoke the tool and trigger the generation of analyzer reports for a just that module (i.e., without taking into account the code from all other modules). Clearly, such a local analysis can be much faster than the global analyses, taking minutes instead of hours, but the results are not as comprehensive either. To invoke the local analysis the users execute a command called SCRUBME. When the analysis reports are generated, the tool displays them in the local module sandbox. The SCRUBME command works by first executing a standard make command for the module code and intercepting all compiler calls as before. The intercepted calls are used to generate the right scripts for all tool invocations, without requiring assistance from the user. In the same way, much smaller software development efforts by single users can of course also benefit from the use of the SCRUB tool, focusing just on the generation of tool reports for their code, without necessarily using the peer review components of the tool.

As noted, tool reports can be generated by a broad range of tools and scripts, as long as the output conforms to the simple standard format that the tool recognizes. In principle the tool could be used for any type of review. Written prose, for instance, could be annotated by reviewers in the same way, and spelling and grammar checkers could in this case provide additional feedback as tool reports.

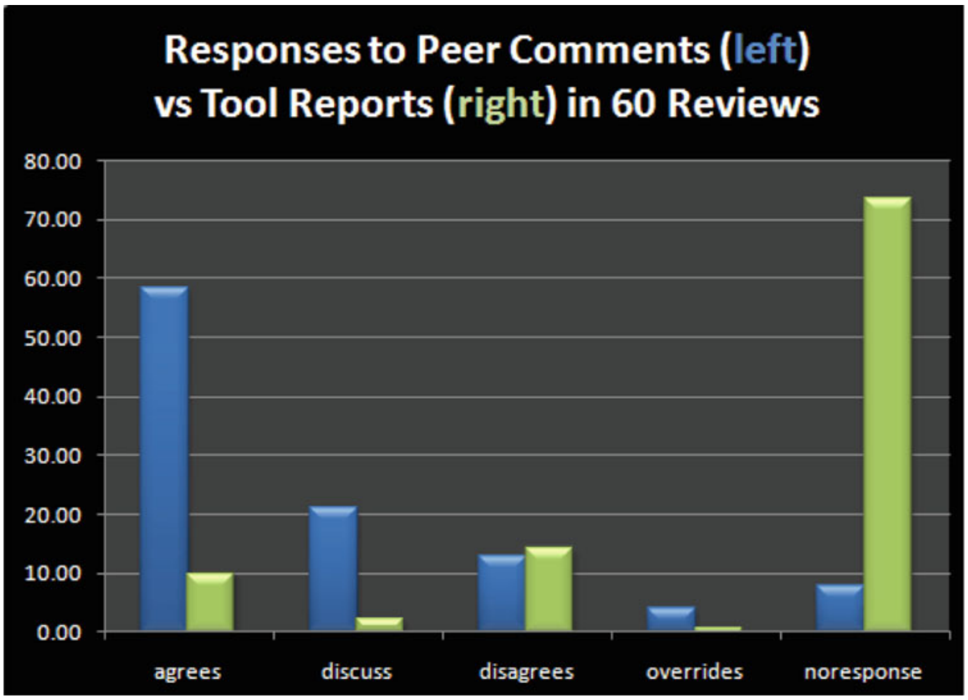


**Fig. 5** Developer responses to peer comments and tool reports. Averaged over 60 code reviews of a total of 227 KSLOC (counted after stripping comments and blank lines)

## 4 Experience and synopsis

The SCRUB tool has been in continuous use on a large flight project at JPL for well over 2 years, with encouraging results. Both developers and project managers tend to like the tool and appreciate the ease with which analysis results are made available in a single framework.

Figure 5 shows some statistics for the first year of use, covering 60 separate code reviews for a total of 227,041 lines of source code (excluding comments and blank lines). Only a relatively small portion of the peer comments (12.7%) and tool reports (14.2%) trigger an explicit *Disagree* response from the developer, with respectively 4 and 0.1% of these responses ultimately overruled in a final review meeting. Through this process, 84% of all peer comments and 80% of all tool-generated reports are fixed in the code, and the remaining comments and reports are recorded, with written justifications, as not requiring a code modification. Also visible in Fig. 5 is that a non-response (which is interpreted by SCRUB as a default *Agree*) is relatively rare for peer comments (about 7.8%), but more common for tool-reports (73%). The average shown in Fig. 5 is dominated by a small number of modules that had a very large number of tool reports, that were not all individually addressed by the developer.

Because *Agree* responses normally need not be discussed in code review closeout sessions, this saves a significant amount of time in face to face meetings.

The total number of tool reports for these 227 KSLOC of source code reviewed in this first full year of use of the SCRUB tool was 7,412 which averages to roughly one report per 31 lines of code. The total number of peer comments on the same code base was 3,862, or approximately one comment for every 59 lines of code. This means that there were almost twice as many tool reports than peer comments at the final review. Although this data point is already encouraging, and emphasizes the importance of tool based checks, it significantly understates the actual contribution of the tools to the review process. We noted earlier that all tool reports are available for all releases of the code: long before a code review is initiated. The vast majority of the tool reports are fixed well before a code review is started. In our measurements the number of tool reports fixed before the code review is held is approximately *four times* larger than the number of tool reports that remain for the code review to process. The reports that remain, therefore, typically are the ones that the module owner is more likely to *Disagree* with. Still, also from those, the majority ends up as a required code fix when the review is concluded. The use of automated source code analysis, therefore has proven itself to be a major factor in the effectiveness of the SCRUB tool at JPL.

## 5 Conclusion

In this paper, we have described a relatively new tool that is in current use at JPL for conducting code reviews. The tool has the perhaps unique distinction that it has become popular among both developers and managers. It is used for all code reviews, and consulted daily by most developers to guard and track the quality of their code. We believe that a tool such as SCRUB can provide an essential part of a thorough code review process, especially when the size of a project grows to millions of lines of source code. In the code review process at JPL we give priority to the review of all manually written code. An increasing amount of code for our missions is also generated from high-level formats, which are included in the code review process. We subject all auto-generated code to the same analyses as the manually written code, but clearly fixes to such code are made in the code generators, and not in the generated code itself.

The SCRUB tool was developed to support the code review process for one specific, large software development project at JPL (the Mars Science Laboratory, or MSL, mission, which is scheduled to be launched to Mars in October 2011). Our plan is to configure the tool for use on other projects at JPL in the near future, which puts it on track for adoption as the standard code review tool for all software development at JPL.

**Acknowledgments** This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The members and leadership of the MSL software team have provided significant support and detailed feedback that has been essential in the development and improvement of the SCRUB tool, their enthusiastic support is gratefully acknowledged.

## References

1. Fagan ME (1976) Design and Code inspections to reduce errors in program develop-ment. IBM Syst J 15(3):182–211
2. Fagan ME (1986) Advances in software inspections. IEEE Trans Softw Eng 12(7):744–751
3. Holzmann GJ (2002) Static source code checking for user-defined properties. In: Proceedings of conference on integrated design & process technology (IDPT), Pasadena, CA, USA. http://spinroot.com/uno/
4. Holzmann GJ (2004) The spin model checker: primer and reference manual. Addison-Wesley, Reading, MA
5. Holzmann GJ (2006) The power of ten: rules for developing safety critical code. IEEE Computer 39(6):95–97
6. Russell GW (1991) Experience with inspection in ultralarge-scale developments. In: IEEE software, pp 25–31
7. http://smartbear.com/docs/BestPracticesForPeerCodeReview.pdf Fig. 2
8. http://www.fastcompany.com/magazine/06/writestuff.html
9. Williams L (2001) Integrating pair programming into a software development process. In: Proceedings of 14th conference on software engineering education and training, Charlotte, NC, USA, pp 27–36