

Eliminating Repetitive Errors Found in Consecutive Source Releases

Jiří Slabý
Faculty of Informatics
Botanická 68a
Brno, Czech Republic
xslaby@fi.muni.cz

ABSTRACT

With a use of contemporary static analysis tools, many errors are found in source codes. We introduce a mechanism to prevent recurring of perpetual errors found by the tools in different source releases. The mechanism aims especially at those errors which were not fixed yet or are false-positives which will likely be present with its source code forever. We discuss what algorithms can be used to determine such errors and demonstrate them on output from STANSE run on the Linux kernel.

Categories and Subject Descriptors

D.2.4 [Software]: Software Engineering—*Software/Program Verification*

General Terms

Algorithms

Keywords

Static analysis, error pruning, consecutive version errors

1. INTRODUCTION

Recently, many new static analysis tools emerged (for example [1, 4, 5]) with a potential of finding many long-standing errors in the source code. Mostly the problem with these tools is that they produce a very long list of errors. The reason is much sources present in the project (more potential places for errors) and/or false-positives which are not about to be fixed. From the recurring error perspective, there is a little what can be done about that in one specific release of sources. But it becomes a burden when the developers would have to check the very same subset of reported errors every single release of the product.

In this case, when such an analyzer is run on a project with release cycle short enough, it would be convenient to recognize (a) the errors which were reported in an earlier version

and were not fixed yet¹ and (b) false-positives because they are persistent in time and nobody wishes to see them again and again in every upcoming released version.

Although it may seem that there are almost no requirements on the project when we want to use this technique, it is not that easy. Not only the fast development and quick releases of new versions is a precondition, but also rigid structure of files so that they don't fluctuate over the source tree. Further, a thorough development methodology which does not allow developers to perform inappropriate moves of code parts between files every release and so on. Every change should have its reason, otherwise the change won't be accepted by the project management.

One of the projects with the requirements above satisfied is the Linux kernel for sure. Its release cycle between major versions (e.g. 2.6.33→2.6.34) is relatively short, oscillating between 11-13 weeks. The changes coming into the tree are not accepted unless appropriately reasoned, reviewed and merged by subtree maintainers and finally accepted by L. Torvalds. Also, developers are mostly not allowed to shuffle with sources arbitrarily over the tree, every class of drivers has its own, predetermined, directory.

We already mentioned that there are many static analysis tools out there. To have a fully operational tool working on the Linux kernel, STANSE [6], a tool developed at the Faculty of Informatics, is used specifically in this paper. It contains a base with C parser, error reporting facility, user interface, points-to analysis module and application programming interface for checkers. A checker is a module which gets the parser output and reports errors back to the base which are in turn presented to a user. Currently there are four checkers implemented, in this paper we are interested exclusively in the AUTOMATONCHECKER which is able to find errors described by finite automata.

The paper is organized as follows. First, in Section 2 we will see why the technique is important and what benefits it brings, then Section 3 introduces the technique itself with possible extensions and finally we evaluate it with a use of STANSE output collected from the Linux kernel in Section 4.

¹There are many reasons for this including slow responses to error reports from maintainers. An example of this is at <http://lkm1.org/lkm1/2009/9/23/221> which had to be sent three times and was merged even after 3 months.

Older version	Newer version
<pre>static DEFINE_SPINLOCK(my_lock1); static DEFINE_SPINLOCK(my_lock2); static void fun(int in) { spin_lock(&my_lock1); if (in) return; spin_unlock(&my_lock1); } . .</pre>	<pre>static DEFINE_SPINLOCK(my_lock); static DEFINE_SPINLOCK(my_lock2); static void fun(int a) { spin_lock(&my_lock1); if (a) { spin_unlock(&my_lock2); return; } spin_unlock(&my_lock1); }</pre>

Figure 1: Illustration of false-negative introduced by the *Simple Solution*

2. MOTIVATING EXAMPLE

Before we dive into the details of the algorithms in the Section 3, consider the following piece of code extracted (and also slightly manually macro-expanded for clarity) from `lib/locking-selftest.c` obtained from the Linux kernel:

```
static DEFINE_SPINLOCK(lock_A);
...
static void double_unlock_spin(void) {
    spin_lock(&lock_A);
    spin_unlock(&lock_A);
    spin_unlock(&lock_A);
}
```

The `double_unlock_spin` function intentionally breaks the proper locking policy. The `lock_A` is locked and then unlocked twice, thus leaving the lock in an inconsistent state. However, in this case, it is the sole purpose of the function. It is intended for exercising a mechanism for automatic dynamic locking violations tracking present in the kernel². Thus it is impossible to change or "fix" the code to be correct from the point of view of an automaton which tracks lock states and watches the policy is not violated. Hence it is an error (a kind of false positive) which will be with this part of sources forever.

We did a complete review of errors reported in the Linux 2.6.27 to sort out which of those are false-positives. All false-positives were marked accordingly and true errors reported to corresponding code maintainers (some with appropriate patches). Then we tried to perform the analysis more often, even on release candidate kernels in a weekly period only, and when inspecting these candidate versions, we found out that many of the errors are of the similar kind like this example, i.e. cannot be fixed or were already reported but not fixed yet for some reason.

To tackle down this problem, we present an algorithm to match errors from consecutive code releases to avoid repetition of errors presented to developers in error lists. Thus lowering the burden exposed to them when walking through the error list generated from later versions. To the best of our knowledge there is no white paper describing a similar technique.

3. METHODOLOGY

Before we start with the methodology, we first have to know what the output of the tools look like, especially what a

²So-called LOCKDEP, more information are available at <http://lwn.net/Articles/185605/>.

STANSE's output is. Even though the error formatting practices differ tool by tool, in common they contain the source file name where the error lays with exact line too. The report also carries an information about what kind of error is that and possibly what module (checker) generated that. Eventually some tools report another fields like a line column, but this is irrelevant for further considerations.

In the end, the entry (error) in a list of reports RL is a tuple E (e.g. in a database as we will see later) such as:

$$E = (file, line, checker, error_description) \in RL$$

Be a start point two lists of errors RL_1 and RL_2 generated from two different versions of the same project. RL_1 corresponds to an older version, RL_2 to a newer one respectively.

Now, we want to eliminate such tuples E , that correspond to the same error. How the "same error" can be handled is described in the following subsections.

3.1 Simple Solution

The simplest approach is just to compare each tuple from RL_2 to entries in RL_1 and obtaining this result:

$$RL_2^{simple} = \{x | x \in RL_2 \wedge x \notin RL_1\} \quad (1)$$

Obviously there are several problems tightly connected to this approach. First, not all the tuple elements are preserved over source revisions. Namely for example *line* element changes extensively version by version. To make this technique defunct, it is actually enough to add or remove single (even empty) line to/from the file³.

Second, this solution can introduce false-negatives. Consider a code in Figure 1 on the left side where there is a bug in locking. The function `fun` omits to unlock `my_lock1` depending on the input parameter `in`. Given this error is reported to the maintainer of the code, he tries to fix it. Unfortunately, he fixes it wrong (the right side of the Figure) and adds a new error by unlocking a wrong lock, but still being on the same line.

In the end both of the cases the entry will look the same in

³Technically, if the analyzer works on the preprocessed code, it is enough that somebody changes any included file, because the preprocessed output is shifted by the change as a whole.

Version	-rc3	-rc4	-rc4 ^{simple}	%	-rc4 ^{SCM}	%
Atomicity						
Sum	511	600	50			

Table 1: Errors count using none, simple and advanced solution (-rc3 shown only for comparison)

the report list:

$$E = (\text{file.c}, 6, \text{AutomatonChecker}, \text{Locking_imbalance})$$

Inconveniently, the outcome of this is that the second error is hidden from programmer when the *Simple Solution* is used because $x \notin RL_1$ from the equation (1) does not hold.

3.2 SCM Solution

Use git log.

4. EVALUATION

We exercised our algorithm on the errors found in the Linux kernel by STANSE. The two versions we considered are 2.6.35-rc3-next20100618 and 2.6.35-rc4-next20100706, further referred only as -rc3 and -rc4 respectively. That means a development versions where there are over 300 thousand lines changed in 2654 files between them in 18 days. There are no changes in configuration of the kernel (besides newly added drivers, which were ignored) and neither of our tool.

The errors reported by STANSE are represented in an XML form by default. But as we wanted to have a convenient way for combining and querying the data, we decided to use databases and some SQL language for accessing the data. Although there exists infrastructures for combining XML and SQL like SQL/XML [2], we stucked to a simple database system, more concretely to SQLITE [3] which stores and references databases as single files somewhere on a backing store.

To achieve the goal, we wrote a Perl script to transform the XML into a SQLITE database file. Despite each report list resides in a separate database file, it is not a problem because SQLITE allows combining even tables from different databases in its SQL. Thus we can easily reference errors from two versions and store the result into the third⁴.

4.1 Using Simple Solution

First, we will have a look on the *Simple Solution* presented in Section 3.1.

4.2 Using SCM Solution

5. CONCLUSION

We presented an algorithm for pruning errors which are reported repeatedly from consecutive versions of a software project by static analysis tools. The method was implemented and then evaluated on the Linux kernel.

⁴The reason for this is that our web frontend works on a version-per-file basis and needs no changes with this approach.

Acknowledgements

The author is supported by the research centre Institute for Theoretical Computer Science (ITI), project No. 1M0545.

6. REFERENCES

- [1] A. Almassawi, K. Lim, and T. Sinha. Analysis Tool Evaluation: Coverity Prevent. *Pittsburgh, PA: Carnegie Mellon University*, 2006.
- [2] A. Eisenberg and J. Melton. SQL/XML is making good progress. *ACM SIGMOD Record*, 31(2):101–108, 2002.
- [3] M. Owens. *The definitive guide to SQLite*. Apress, 2006.
- [4] N. Rutar, C.B. Almazan, and J.S. Foster. A comparison of bug finding tools for Java. In *15th International Symposium on Software Reliability Engineering, 2004. ISSRE 2004*, pages 245–256, 2004.
- [5] SPARSE. <http://sparse.wiki.kernel.org/>.
- [6] STANSE. <http://stanse.fi.muni.cz/>.