Software Change Contracts

Dawei Qi, Jooyong Yi, Abhik Roychoudhury
School of Computing, National University of Singapore
{dawei,jooyong,abhik}@comp.nus.edu.sg

ABSTRACT

Incorrect program changes including regression bugs, incorrect bug-fixes, incorrect feature updates are pervasive in software. These incorrect program changes affect software quality and are difficult to detect/correct. In this paper, we propose the notion of "change contracts" to avoid incorrect program changes. Change contracts formally specify the intended effect of program changes. Incorrect program changes are detected when they are checked with respect to the change contracts. We design a change contract language for Java programs and a dynamic checking system for our change contract language. We conduct a preliminary user study to check the expressiveness of our change contract language and find that the language is expressive enough to capture a wide variety of real-life changes in three large software projects (i.e., Ant, JMeter, log4j). Finally, our contract checking system detects several real-life incorrect changes in these three software projects via runtime checking of the change contracts.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Programming by contract*

Keywords

Software Evolution, Regression Testing, Change Contract, JML

1. INTRODUCTION

"There is nothing permanent except change" - this well-known adage is true for software too. Programmers make changes to introduce new features as required by the evolving software requirements. Programmers also make changes to fix bugs. However, the changes to programs are usually imperfect. The new features might not be completely realized by the changes. At the same time, existing features might get broken by careless changes, which are commonly known as "software regressions". In fact, a recent study [14] shows that 14.8%~24.4% of bug fixes in operating systems code are incorrect.

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.

Regression errors constitute an important class of incorrect program changes. Regression bugs are generated when programmers accidentally break existing program functionality (say in trying to introduce new functionality). Past research has mainly focused on regression testing [3, 4, 13] and regression debugging [11, 15] to eliminate regression errors. Although the goal of regression testing is to detect regression errors, it can hardly distinguish a normal feature update from regression bugs without a proper oracle. Going through the reported "errors" one by one and differentiating unintended differences in program behavior (across versions) from intended program changes is annoying.

The pointed difficulty in regression testing ultimately stems from the fact that programmer's intention behind code changes is seldom expressed. Although intention of changes is sometimes described usually in a form of a change log or a comment, such an informal style of description does not help much in resolving the issue; manual checking is still required. The problem can be resolved if programmers can specify the intended change via a formal specification; formally described intended changes can be checked automatically if they are met by actual program changes.

New idea. We propose in this paper our new idea of "change contracts", a formal specification designed to express intended changes. Change contracts specify the *intended* semantic changes corresponding to changes in program code. When the *actual* program changes break what is documented in the change contract, an inconsistency between intended and actual changes can be detected. If the change contract is properly written, such an inconsistency points out incorrect program changes. Therefore, with the help of change contracts, an incorrect program change can be detected and corrected - prior to checking in such incorrect changes into the code repository.

The concept of change contract is inspired by Design by Contract programming (DBC) [7]. In DBC, programs are checked against contracts to enable early error detection. Contracts typically appear in the form of pre- and post-condition of methods, as well as invariant properties whose correctness is preserved by method execution. However, this early error detection comes at the cost of manually written contracts. This is probably the main reason for the lack of adoption of "design by contract" : programmers are reluctant to write non-trivial specifications.

Compared to program contracts which are recommended in design by contract programming, our change contracts are easier to write. In fact, to detect regression errors, *no change contract* is required at all; we can simply have a default contract which says that the program output after the change should be the same as the output before the change. [1] Checking such default change contracts,

[1] Our change contract language also allows to specify in what sense two outputs are the same if the exact identity of them is not intended.

```
1 void checkIncludePatterns(){
2  ...
3  File f=findFile(b,c,false);
4  if(f!=null && f.exists()){
5  ...
6 }
```

(a) Current version, the bug is fixed

```
void checkIncludePatterns(){
 ...
 File f=findFile(b,c,false);
 if(f.exists()){
 ...
}
```

(b) Buggy version

```
void checkIncludePatterns(){
  ...
  File f=findFileCaseInsensitive(b,c);
  if(f.exists()){
  ...
}
```

(c) Original version

Figure 1: Reverse chronological change history; the leftmost one is the latest one

which do not involve *any (or little)* effort from a programmer, can help reveal many subtle program errors.

The fact that change contracts are easier to write than program contracts comes from the intrinsic nature of change contracts. Program contracts are often specified as pre- and post-conditions of methods. Thus, they specify *what a program method does*, about which the programmer may not always have deep understanding (unfortunately!) in real-life. In contrast, a change contract specifies *how the functionality of a program method is changed* with respect to the old program. The common behavior between two programs, which is usually dominant, does not need to be specified in the change contract. Besides, we allow users to write change contracts at multiple levels of precision. The more precise a change contract is, the more checking is done by our system. The users can choose the level of precision at will. Finally, we note that there exists a large body of code today which completely lacks any formal specification. The concept of change contracts also provides a pragmatic way of adding specifications of intended behavior on top of this huge code base lacking formal specifications.

To support our new idea of change contract, we designed a language for change contract, and developed a contract checking tool. Our change-contract language is an extension of JML (Java Modeling Language) [2]; we extended and modified its syntax and semantics to be able to capture program's behavioral changes over two consecutive versions. To check change contract, we built a runtime-assertion checker on top of OpenJML [8].

In the next section, we explain our new idea of change contract in more detail by an example. After that, in Section 3, we show our early result on change contract from two perspectives. First, we show our preliminary user-study result about usability of our change contract language. We received positive feedback from users that our change contract language is expressive enough to describe changes of real-life software. Second, we share preliminary experience in using our tool for checking change contract. In Section 4, we discuss related work.

2. OUR NEW APPROACH

In this section, we first show code changes made on real-life software. Then, we explain how our change contract can help with the development and maintenance of programs that change over time. Our language for change contract and checking tool will also be explained.

Figure 1 shows in reverse chronological order how a method `checkIncludePatterns` in file DirectoryScanner.java of Apache Ant [1] was changed over time.[2] The program in Figure 1a is a bug-fixed version of the middle program in Figure 1b. The cause of the bug was that `null` could be unexpectedly assigned to variable `f` at line 3. This could happen when method `findFile` failed to find

```
/*@ changed_behavior  /** NPE fix request */
  @  /** findFile can return null */
  @  requires findFile(b,"f",false)==null;
  @  when_signaled (NullPointerException)
  @          findFile(b,"f",false)==null;
  @  signals (NullPointerException) false;
  @*/
void checkIncludePatterns();
```

Figure 2: A change contract for the latest change, i.e., the change from (b) to (a) of Figure 1; in the above, `b` is a field of the enclosing class

file name `c` in the base directory `b` in a case-insensitive way; the last parameter of `findFile` is used to decide case-sensitivity. As a result, an NPE (i.e., NullPointerException) was raised at line 4 in the buggy version. While the fix for NPE is usually as simple as adding a conditional guard as is done for the current version, NPE is pervasive in most Java programs as one of the most common causes of errors. Interestingly, this particular bug was reported by developer Curt while the fix was made by another developer Stefan. Indeed, it is common to see that problems missed by the original developer or a maintainer are found by other developers or even end-users.

In fact, the above NPE is a regression error resulted from a previous change; the same problem did not occur until that previous change was made by yet another developer Matthew. The rightmost version in Figure 1c shows what the same method looked like before an NPE-causing change had been made. Notice that different method `findFileCaseInsensitive` was called then instead of `findFile`. Originally, two different find-file methods were used depending on the case sensitivities required at call sites. A regression-error-causing change was made when these two methods were merged into a new method `findFile` in which its last boolean parameter is used to choose a case-sensitivity mode.

Now notice that the conditional guard at line 4 of the original version shown in Figure 1c does not yet check whether `f` is null. Nevertheless, an NPE did not occur in this original version. The reason for this difference is that when there is no file name `c` in base directory `b`, method `findFileCaseInsensitive` of the original version creates a fresh dummy object of type `File`[3] whereas `findFile` used in Figure 1b returns `null`. Apparently, it seems that the developer mistakenly assumed that the merged method `findFile`, when its last boolean parameter is set `false` to indicate case-insensitivity, always behaves in the same way as `findFileCaseInsensitive` did in the previous version. It is, however, difficult to put the entire blame on the developer because without proper tool support most developers are likely to make similar mistakes.

We now show how change contract can help deal with program

[2] In the figure, we use simplified variable names.

[3] It is created by `new File(b,c)`.

changes described above in various ways. A change contract is essentially a formal specification about intended program changes. Like other formal specifications, change contracts can be used as *unambiguous documentation*. In our example, Curt who found the unexpected NPE could have written a change contract such as the one in Figure 2, and have used it as a medium for a bug report. Basically, such change contract describes when exactly an NPE bug can be observed in the current problematic version, and that the observed NPE bug should disappear in the fixed version.

Change contract language. Before explaining the above change contract example in more detail, let us first give an overview of our change contract language. When describing behavioral changes of a method over two consecutive versions, users would typically want to specify (i) under what common input condition of the two versions, (ii) how different output conditions are observed from those two versions. To handle the former, we use JML's `requires` clause. Unlike in JML, however, the given requires clause is imposed on two consecutive versions at the same time. Meanwhile, to separately describe output conditions for two consecutive versions, we add to JML two additional clauses; namely, `when_ensured` and `when_signaled`. We use these two new clauses to describe the output condition of the earlier version while using JML's existing `ensures` and `signals` clauses to describe the later version's output condition. Following the convention of JML, we distinguish normal and abnormal termination of a method. For the former, `when_ensured` and `ensures` are used; for the latter, one can use `when_signaled` and `signals`. Our change contract language can also deal with method signature changes and field addition/deletion. More detailed description of our change contract language and its formal semantics are available through our technical report [12].

Then, the change contract of Figure 2 means the following. First, the given `requires` clause sets the domain of interest; code change should be considered when method `findFile` returns `null` given the field `b` and file name "`f`". When this input condition holds true, two consecutive versions should behave differently. Let us call those consecutive versions v_1 and v_2 in order of creation. The given `when_signaled` clause describes that an NPE can be signaled in version v_1. This clause also describes the condition that is satisfied when an NPE is signaled. *Only when* that given pair of `requires` and `when_signaled` clauses is satisfied, version v_2 is expected to behave differently from version v_1. Meanwhile, the `signals` clause dictates that an NPE should not be signaled in version v_2; notice that `false` is given as an output condition. Given such a change contract, Stefan, who is in charge of maintaining this part of code, should be able to clearly understand when an NPE is observed and that this problem should be fixed.

Change contract checker. Change contract is not only unambiguously understandable but also *automatically checkable*. Similar to DBC, various levels of change-contract checking is possible from lightweight runtime assertion checking to human-guided full static program verification and to extended static checking in the middle. As a prototype, we built a runtime-assertion-checking tool, and applied it to several sets of real-life software. Our preliminary results will be shown in the next section.

When checking a change contract, it is not only the change that is checked. It is also checked whether unintended changes are mistakenly made. Note that unintended changes cause regression errors. For input that is not of interest of a given change contract, two consecutive versions of methods should lead to the same program state after execution because we assume that the same input is passed to those two versions.

3. PRELIMINARY RESULT

We have conducted two different experiments to evaluate our approach. First, we have conducted an initial user study to receive feedback on the usability of our change contract language. Second, we applied our tool to software changes to see if unintended changes can actually be detected by our tool.

User study. A user study was carried out with two second-year Master's students majoring in computer science. Prior to the user study, they both had no knowledge on program contract and JML. We were mainly interested to see whether our change contract language is expressive enough to describe changes occurring on real-life software. Under the circumstances of limited resources, we took a reverse-engineering approach. The users were first asked to understand subject programs and their changes across different versions. Afterwards, they were asked to write change contracts for those changes.

We selected changes from the Bugzilla database of three open source Java projects, i.e., Ant, JMeter and log4j. All of these are widely used large-scale Java programs; Ant and JMeter have more than 100,000 lines of code each, and log4j has around 13,000 lines of code. Ant is the de facto standard Java build automation tool that helps manage the build process. JMeter is used to test the behavior and performance of various servers such as HTTP and POP3. Log4j is a Java library that eases the logging process in Java. Despite its name, Bugzilla contains not only bug fixes but also numerous changes for new features. To help users understand the programs, we picked only the entries of Bugzilla that provide patch files and discussion of changes.

The user study results are summarized in Table 1. Overall, 52 change contracts were written out of 73 non-refactoring changes. For refactoring, there was no need to write change contracts; a change contract describes behavioral changes, not syntactic changes. Among 52 change contracts, 24 of them listed under "Behavior diff" column describe purely behavioral differences while 28 of them listed under "Add/Delete" column also describe changes that involve adding or deleting fields, methods or parameters.

The users failed to write change contract for some cases. The "Not understood" column of the table accounts for 5 changes that were failed to be understood by the users, for example, due to the lack of source code of third-party libraries. The "Not concerned" column amounts to 7 changes that are not concerned by our change contract at this point, such as changes in synchronization in multi-threaded programs. Lastly, the "Non-code" column accounts for 9 non-Java-source changes such as system environment changes.

Overall, we received positive feedback on the usability of our change contract language. Out of total 80 changes, $(24 + 28 + 5) = 57$ changes were applicable for writing change contracts - the other 22 changes being due to refactoring, multi-threading or environment changes. Out of these 57 changes, the two users could write change contracts for 52 of them using our language.

Tool experience. We built a prototype tool supporting runtime assertion checking of change contract. We were interested to see the capability of the tool in detecting unintended changes, i.e., the changes that do not match a given change contract.

We again took a reverse-engineering approach to retrieve correct change contracts and buggy program changes. We exploited bug-fixes found in the code repository for the same three open-source projects as we used for the user study. Let v_3 be a bug-fix version found in one of those repositories. We searched backward in that repository for a previous version v_2 where the bug of interest was introduced. It is obvious that the change made to v_2 from its very previous version v_1 was incorrect and caused the bug. In

Subject prog.	Refactoring changes	Applicable changes			Not applicable changes		Total changes
		Behavior diff	Add/Delete	Not understood	Not concerned	Non-code	
Ant	4	13	15	3	3	5	43
JMeter	1	5	6	1	4	0	17
log4j	2	6	7	1	0	4	20
TOTAL	7	24	28	5	7	9	80

Table 1: User study results on the usability (expressiveness) of our change contract language

other words, the changes from v_1 to v_2 are buggy. Meanwhile, a change contract could hypothetically have been written when v_1 was changed, and a correct one should amount to the changes between v_1 and v_3 while skipping v_2.

We applied our tool to buggy changes for validating the use of change contracts in bug detection. We tested 10 buggy changes over the three subject programs (Ant, Jmeter and log4j), and our tool could report change-contract violation for every case. Admittedly, the efficacy of our runtime-assertion-based contract checker depend on the test input used. To make our checker more efficient, it would be desirable to be able to generate a test suite that is potentially likely to violate a given change contract. We plan to work towards that direction in the future exploiting our previous work on test generation to expose changes in evolving programs [10].

4. RELATED WORK

We earlier mentioned that our work was inspired by DBC. A large body of studies has been conducted around DBC and its enclosing motif, program specification and verification, over many years. What is new in change contracts as compared to those studies is that with change contracts we focus on changes between two programs unlike in traditional methodologies where the main focus is given to a single program.

Controlling program changes has been an interesting topic in software engineering community during recent years. One popular method to control program changes is to report differences at various levels between versions of a program. Such difference reports are generated by various means such as symbolic execution [9], rule learning [6], and test case generation [5] to name but a few. To the best of our knowledge, all of those methods are performed on a post-mortem basis; given program changes, differences are inferred and processed. At the end, a user should review generated difference reports to see if those changes are actually intended. With change contract, we take an opposite approach; a user first describes intended program differences before making a change to a program. Then, it is automatically checked if given program changes meet their change contracts.

5. CONCLUSION AND DISCUSSION

In this paper, we have proposed the notion of "change contracts" as the specification of intended program changes. Incorrect changes can be easily detected when checked with respect to their change contracts. Since change contracts only focus on behavioral differences across consecutive program versions, they can be easier to write than program contracts. We have presented an overview of our change contract based on our formal language and tool that support change contracts. Our early results from change contracts are promising. Our contract language (based on JML) seems to be expressive enough to describe changes of real-life software according to our user study.

There still remains an important question, however. How can we evaluate more thoroughly the usability of our change contract? Will change contract provide more benefits than its cost of writing? Although our small user study served its purpose with positive feedback, we need to extend the user study via collaboration with more users. On one hand, we seek for collaboration opportunities at the conference. On the other hand, we also hope to receive informative feedback on our change contract language.

ACKNOWLEDGEMENTS

We thank Tushar Mehta and Tao Sun for participating in the user study. This work was partially supported by a Ministry of Education research grant MOE2010-T2-2-073 (R-252-000-456-112 and R-252-100-456-112) from Singapore.

REFERENCES

[1] Apache Ant. `http://ant.apache.org/`.

[2] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.

[3] Y. Chen, D. Rosenblum, and K. Vo. Testtube: A system for selective regression testing. In *ICSE*, pages 211–220, 1994.

[4] R. Gupta, M. Harrold, and M. Soffa. An approach to regression testing using slicing. In *ICSM*, pages 299–308, 1992.

[5] W. Jin, A. Orso, and T. Xie. Automated behavioral regression testing. In *ICST*, pages 137–146, 2010.

[6] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *ICSE*, pages 309–319, 2009.

[7] B. Meyer. Eiffel: The language and environment. *Prentice hall press, 300*, 1991.

[8] OpenJML. `http://sourceforge.net/apps/trac/jmlspecs/wiki/OpenJml`.

[9] S. Person, M. Dwyer, S. Elbaum, and C. Pasareanu. Differential symbolic execution. In *FSE*, pages 226–237, 2008.

[10] D. Qi, A. Roychoudhury, and Z. Liang. Test generation to expose changes in evolving programs. In *ASE*, pages 397–406, 2010.

[11] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. DARWIN:an approach for debugging evolving programs. In *FSE*, pages 33–42, 2009.

[12] D. Qi, J. Yi, and A. Roychoudhury. Software change contracts. Technical Report TRE3/12, National University of Singapore, 2012. Available at `https://dl.comp.nus.edu.sg/dspace/handle/1900.100/3588`.

[13] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *TSE*, 27(10):929–948, 2001.

[14] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram. How do fixes become bugs? In *FSE*, pages 26–36, 2011.

[15] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *FSE*, pages 253–267, 1999.