THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TRE3/12

Software Change Contracts

Dawei Qi, Jooyong Yi and Abhik Roychoudhury

March 2012

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

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. General guidelines as well as concrete examples are given to illustrate the usage of our change contracts. We conduct an 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 (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.

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 [25] shows that 14.8%~24.4% of bug fixes in operating systems code are incorrect.

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 for example). Past research has mainly focused on regression testing [6, 11, 23] and regression debugging [22, 26] 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. If programmers can specify the intended change via a formal specification, such issues can be resolved. We present such a mechanism in this paper.

In this paper, we propose the notion of “change contracts” to deal with incorrect program changes. Change contracts specify the *intended* semantic change corresponding to changes in program code. When the *actual* program changes break what is documented in the change contract, an inconsistency 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 [3, 16]. In Design by Contract programming, 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 the 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 same as the output before the change. As our experiments show, checking such default change contracts (which do not involve *any* effort from the 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.

```
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

←

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

(b) Buggy version

←

```
1  void checkIncludePatterns(){
2    ...
3    File f=findFileCaseInsensitive(b,c);
4    if(f.exists()){
5    ...
6  }
```

(c) Original version

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

The contributions of this paper can be summarized as follows. First, we propose the notion of change contracts to prevent incorrect program changes. Secondly, we design a change contract language for Java programs. Our language extends the Java Modeling Language or JML with specific keywords to relate behaviors of program versions. We present the formal semantics of our change contract language. We also show via user studies how various kinds of real life program changes can be specified using our change contract language. Real-life changes from three large Java open source programs (Ant, JMeter, log4j) were investigated in the user studies. In total, users wrote 52 change contracts in our user study. We did not meet any change that cannot be expressed using our change contract language. Last but not the least, we design and implement a system for dynamically checking change contracts by building on top of the Run-time Assertion Checker of JML. We found 10 real-life incorrect changes from the same programs (Ant, JMeter, log4j) and wrote change contracts for them. All 10 incorrect changes are detected by the Run-time Assertion Checker via the change contracts.

2. OVERVIEW

In this section, we show a series of code changes made on a file of a well-known build automation software, Apache Ant [1], and explain how our change contract language and its supporting tool can help with the development and maintenance of programs that change over time.

Figure 1 shows in reverse chronological order how a method `checkIncludePatterns` in file DirectoryScanner.java of Ant was changed over time. The program in Figure 1a is a bug-fixed version of the middle program in Figure 1b. The problem was that `null` could be assigned to variable `f` at line 3 when method `findFile` failed to find file name `c` in the base directory `b` in a case-insensitive way as indicated by the last boolean value. As a result, an NPE (i.e., NullPointerException) was raised at line 4 before. While the fix for NPE is usually as simple as adding a conditional guard as shown in the figure, 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`. In Figure 1c, method `findFile` is used only in a case-sensitive way, and a case-insensitive search is performed by `findFileCaseInsensitive`. A regression-error-causing change was made when these two methods were merged into a new method `findFile` where its last boolean parameter is used to choose a case-sensitivity mode.

```
1  //@ changed_behavior
2  //@  when_signaled (NullPointerException)
3  //@          findFile(b,c,false)==null;
4  //@  signals (NullPointerException) false;
5  void checkIncludePatterns() {
6   ...
7   File f=findFile(b,c,false);
8   if(f!=null && f.exists()){
9   ...
10 }
```

Figure 2: An annotated change contract for the latest change

Now notice that the conditional guard at line 4 in Figure 1c does not yet check whether `f` is null. Nevertheless, an NPE did not occur in the original version in Figure 1c. The reason of this difference is that when there is no file name `c` in base directory `b`, the merged method `findFile` in Figure 1b returns `null` whereas `findFileCaseInsensitive` in the original version as shown in Figure 1c creates a fresh dummy object of type `File`.[1] Apparently, it seems that the developer mistakenly assumed that the merged method `findFile`, when its boolean parameter is set `false` to indicate case-insensitivity, always behaves in the same way as the removed `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 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*. For example, Curt who found the aforementioned unexpected NPE could have reported the NPE problem with the change contract shown in Figure 2. The given annotation describes the following two things. (i) First, line 2~3 describes that there exists a certain input that caused the previous version to signal an NPE; when that NPE was signaled, the boolean expression at the end of the `when_ensured` clause indicates that the NPE is caused by the `null` return value from the `findFile` method. (ii) Second, line 4 describes that the current version cannot signal an NPE as indicated by `signals (NullPointerException) false` when given the same input as signaled the NPE before. Once such a change contract is written, Stefan, who is in charge of maintaining this part of code, should be able to understand the NPE problem.

Our change contract is not only unambiguously understandable but also *automatically checkable*. With the help of a supporting tool

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

provided by us, developers can check whether their code changes indeed match intended changes expressed as change contracts. Thus, change contracts function as test oracles for intended changes.

Our supporting tool automatically checks not only whether intended changes are made to the updated version, but also whether unintended changes are mistakenly made. Note that unintended changes cause regression errors. When an input does not match any given change contracts, previous method and current method are assumed to behave exactly the same for that input by default. Exploiting our by-default-equal assumption, the regression error of our running example could have been detected earlier. When merging two methods that find a file, the changes made to method `checkIncludePatterns` are merely auxiliary, and its behavior was supposed to remain the same. This is the case where our by-default-equal assumption can be exploited to the extreme; no change contract needs to be provided and any behavioral changes are reported as unexpected changes. Our tool can detect such unexpected changes caused by programmer's mistakes.

3. CHANGE CONTRACT LANGUAGE

To express intended program changes, we extend a subset of JML (Java Modeling Language) [5], a de facto lingua franca when giving checkable formal specifications to Java programs. In fact, one of our goals in designing a change contract language is to be as close to an existing popular specification language (in this case, JML) as possible to lower the learning barrier, and our syntactic extension to JML is very limited as will be shown shortly. However, JML or any other specification languages, to the best of our knowledge, is not expressive enough to express program changes over two different versions, and this requires us to propose non-trivial semantic extensions.

Using the suggested specification language, we want to be able to specify behavioral changes that occur between two consecutive versions of a method, i.e., the previous version and the current version. In this paper, we are concerned only about behavioral changes and do not consider most syntactic changes such as modifying the name of a parameter.[2] We, however, allow to add or remove formal parameters of a method, fields, and methods. Lastly, as in JML, we are concerned only with sequential Java programs, and do not consider multi-threading.

Following the convention of JML, one can write a change contract above a method as an annotation. A change contract should be either between `/*@` and `@*/` or after a line comment marker `//@`. The method following such a change-contract annotation should be a currently-working version. Such a currently-working version at hand is compared to the previous version in a change contract.

3.1 Syntax

We show in Figure 3 a simplified version of our change contract language. The keywords in bold are extensions to standard JML. The semantics of these keywords are explained in details later in this section. We also explain the semantics of some basic JML keywords that are essential to understanding change contract language.

We add eight additional keywords to JML. With those new keywords, our language can declare one kind of type specification (by `induces`) and method specification (by `changed_behavior`), and use three additional kinds of clauses (by `changes`, `when_ensured`, and `when_signaled`) and three more kinds of specification-only expressions (by `\prev`, `\latest`, and `\deprecated`).

type-spec ::= **induces** *pred-or-everything*;   *method-spec* ::= *spec-case-seq*

spec-case-seq ::= *spec-case* [`also` *spec-case*]∗

spec-case ::= **changed_behavior** *clause-seq*
clause-seq ::= [*clause*]∗

clause ::= `requires` *pred*; | `ensures` *pred*; | `signals` *pred*;
| **when_ensured** *pred*; | **when_signaled** (*reference-type* [*ident*]) *pred*;
| **changes** *method-name*(*param-type*);

exp ::= ... | *quantified-exp* | `\result` | `\old`(*exp*) | **\prev**(*exp*)
| `\fresh`(*exp*) | **\latest**(*exp*) | **\deprecated**(*exp*) | **\latest**(*method-name*)

Figure 3: Change contract language as an extension to a JML subset; standard regular expression notation ∗ is used.

Keyword `changed_behavior` declares a method specification case for a change contract.[3] In fact, it is the only kind of specification cases we allow. Although JML allows more kinds of specification cases such as normal-behavior and exceptional-behavior, we do not consider them in this paper. While we believe it should be possible to use our change contracts and ordinary JML contracts at the same time for the same program, we assume in our examples that only change contracts are available in a given program.

Two keywords `when_ensured` and `when_signaled` construct clauses that express the post-conditions of the previous version. The former expresses the post-condition at normal method termination (i.e., termination without throwing an exception), and the latter the post-condition at abnormal method termination (i.e., termination with an exception thrown). Meanwhile, post-conditions of the current version are expressed with JML's `ensures` and `signals` clauses. In the case of `when_signaled` and `signals` clauses, the type of an expected exception is additionally specified.

The grammar in Figure 3 also shows other kinds of clauses we use for change contract. Those clauses except for `changes` are borrowed from JML, and we use them with their original meaning.

The one remaining `changes` clause is to clarify which method of the previous version is supposed to be changed following a change contract. Such a method subject to specified changes is usually the same as the method annotated with a change contract in the current version. In such cases, a `changes` clause can be omitted. Due to Java's method overloading, however, it can be unclear which one among several methods sharing the same name is subject to specified changes. Recall that we assume the existence of missing parameters across the previous and current versions. A `changes` clause comes to the rescue when such ambiguities arise. This clause is also useful when the method name changes across versions.

Keyword `\prev` constructs a \prev expression that accesses the previous-version value from a current-version state; `\prev(`E`)` used at a post-condition of the current version (i.e., used in a `ensures` or `signals` clause) returns the value of E evaluated at the post-state of the previous version reached when the same input is used. Meanwhile, we disallow the use of `\prev(`E`)` in a pre-condition (i.e., inside a `requires` clause) because we assume that input is the same between the two consecutive versions, and thus `\prev(`E`)` is always equal to E at a pre-state.[4] Readers familiar with JML could find the similarity between `\prev` and `\old` of JML. While `\old`

[2] Renaming can be dealt with easily with additional renaming information.

[3] Although JML distinguishes heavyweight and lightweight styles of a method specification, we deal with only the heavyweight style in this paper for the sake of brevity of presentation.

[4] Recall that even if there are differences between formal parameters, those differences are offset by inserting missing parameters.

makes a value of a pre-state available at a post-state, `\prev` makes a value of the previous version available at the current version.

Keywords `\latest` and `\deprecated` construct boolean-type expressions that are evaluated to true when a given field or formal parameter or a method is newly added or removed, respectively, at the current version. We restrict their uses to `ensures` and `signals` clauses only because of a risk of making the verification condition vacuously true if they appear in other clauses.[5]

Notice that `\latest` expressions are polymorphically used. Both `\latest(f)` for field `f` and `\latest(m)` for method `m` are legal expressions. The latter should be annotated only over method whose name is `m` and not over other methods. The method name of `m` is supposed to be fresh in a sense that the addition of `m` does not lead to method overloading. Therefore, false is returned out of `\latest(m)` if there already exists another method with the same name. Meanwhile, we do not use `\deprecated` for method names.

There are various specification-only expressions in JML including quantified expressions, and we inherit them in our language.

Lastly, keyword `induces` constructs the only kind of type (i.e., class) specification we allow. This is simply a syntactic sugar and can be subsumed by method specifications; annotating a class with `induces` ψ' is equivalent to annotating every method in that class with `ensures` ψ'`;` and `signals (Exception)` ψ'`;`. When fields are added to or removed from a class, this type specification relieves a user from the burden of specifying each method with `\latest` or `\deprecated` expressions. In particular, when a new class is introduced, it is handy to write `induces \everything` to implicitly specify the post-conditions of each method with `\latest` expressions for that method and all fields of the class.

3.2 Non-interference semantics

Different from normal program contract, an expression in change contract may involve program states in two different executions. When we compare the behavior difference between two versions of a method, we assume that the two methods are executed from the same program state with the same inputs. At the same time, the execution of the two methods are not supposed to be affected by each other. To address this issue, we provide non-interference semantics. The key difference of our non-interference semantics from the standard one is that our semantics maintains the separate state for each version of a method. Although references can be shared between these two states, changes on the state of one version does not affect the state of the other version. Such a non-interference property is captured in the non-interference semantic rule shown in Figure 4 in a big-step style; $\Downarrow_e$ and $\Downarrow_c$ represent reduction relations of big-step operational semantics for expressions and commands, respectively. In the figure, c_1 and c_2 are supposed to be commands of each version, respectively, and $c_1 \mid\mid c_2$ denotes the parallel execution of c_1 and c_2. We assume the parallel execution model for the sake of mere convenience, and the same idea could have been applied to the sequential model. Also recall that we are concerned in this paper with only sequential Java programs, and the introduced parallel execution is not intended to interfere with Java's multi-threading.

Now that we have the non-interference rule, we can safely use the following starting configuration: $\langle c_1 \mid\mid c_2, (\sigma, h, \sigma, h)\rangle$, where c_1 and c_2 denotes the method bodies of the previous and the current versions, respectively. Notice that we give the same store σ and heap h to both versions to enforce the same-input assumption.

$$c \in Cmd \quad v \in \textbf{Value} \overset{\text{def}}{=} \textbf{Location} \cup \ldots$$

$$\sigma \in \textbf{Store} \overset{\text{def}}{=} \textbf{Variable} \overset{\text{fin}}{\to} \textbf{Value}$$

$$h \in \textbf{Heap} \overset{\text{def}}{=} \textbf{Location} \overset{\text{fin}}{\to} (\textbf{Field} \overset{\text{fin}}{\to} \textbf{Value})$$

$$\frac{\langle c_1, (\sigma_1, h_1)\rangle \Downarrow_c (\sigma_1', h_1') \qquad \langle c_2, (\sigma_2, h_2)\rangle \Downarrow_c (\sigma_2', h_2')}{\langle c_1 \mid\mid c_2, (\sigma_1, h_1, \sigma_2, h_2)\rangle \Downarrow_c (\sigma_1', h_1', \sigma_2', h_2')}$$

Figure 4: Non-interference rule; $\Downarrow_e$ and $\Downarrow_c$ represent reduction relations of big-step operational semantics for expressions and commands, respectively.

$$\frac{\langle c_1 \mid\mid c_2, (\sigma, h, \sigma, h)\rangle \Downarrow_c (\sigma_1', h_1', \sigma_2', h_2') \qquad \langle E, (\sigma_1', h_1')\rangle \Downarrow_e v}{\texttt{ensures} \vdash \langle \texttt{\textbackslash prev}(E), \sigma_1', h_1', \sigma_2', h_2'\rangle \Downarrow_e v}$$

$$\frac{\langle c_1 \mid\mid c_2, (\sigma, h, \sigma, h)\rangle \Downarrow_c (\sigma_1', h_1', \sigma_2', h_2') \qquad \langle E, (\sigma, h)\rangle \Downarrow_e v}{\texttt{ensures} \vdash \langle \texttt{\textbackslash old}(E), (\sigma_1', h_1', \sigma_2', h_2')\rangle \Downarrow_e v}$$

$$\frac{\begin{array}{c}\langle c_1 \mid\mid c_2, (\sigma, h, \sigma, h)\rangle \Downarrow_c (\sigma_1', h_1', \sigma_2', h_2') \\ \langle E, (\sigma_1', h_1')\rangle \Downarrow_e \bot \qquad \langle E, (\sigma_2', h_2')\rangle \Downarrow_e v\end{array}}{\texttt{ensures} \vdash \langle \texttt{\textbackslash latest}(E), (\sigma_1', h_1', \sigma_2', h_2')\rangle \Downarrow_e \mathit{true}}$$

$$\frac{\begin{array}{c}\langle c_1 \mid\mid c_2, (\sigma, h, \sigma, h)\rangle \Downarrow_c (\sigma_1', h_1', \sigma_2', h_2') \\ \langle E, (\sigma_1', h_1')\rangle \Downarrow_e v \qquad \langle E, (\sigma_2', h_2')\rangle \Downarrow_e \bot\end{array}}{\texttt{ensures} \vdash \langle \texttt{\textbackslash deprecated}(E), (\sigma_1', h_1', \sigma_2', h_2')\rangle \Downarrow_e \mathit{true}}$$

Figure 5: Semantics of `\prev(`E`)` in the context of a `ensures` clause as compared to the one for `\old(`E`)`, and semantics of `\latest(`E`)` and `\deprecated(`E`)`; c_1 and c_2 represent the method bodies of the previous and the current versions, respectively.

With this non-interference rule and other standard semantic rules defined for expressions and commands, one should be able to reduce the starting configuration $\langle c_1 \mid\mid c_2, (\sigma, h, \sigma, h)\rangle$ to the final state $(\sigma_1', h_1', \sigma_2', h_2')$; i.e., $\langle c_1 \mid\mid c_2, (\sigma, h, \sigma, h)\rangle \Downarrow_c (\sigma_1', h_1', \sigma_2', h_2')$. Such a final state provides us with separate post-states for each of two versions; (σ_1', h_1') and (σ_2', h_2') amount to the post-state of the previous and the current version, respectively.

3.3 Semantics of \prev

As mentioned earlier, `\prev` expressions can be used in a change contract to access the post-state value of the previous version. They can appear in `ensures` or `signals` clauses. In Figure 5, we provide the semantics of `\prev` expressions. Semantics of `\old` expressions is also provided in the figure for the sake of comparison.

Suppose that expression `\prev(`E`)` appears in an `ensures` clause. Then, evaluating `\prev(`E`)` should be the same as evaluating its sub-expression E in the post-state (σ_1', h_1') of the previous version. We use the notation "`ensures` $\vdash$" in the above semantics to designate the clause context in which the expression at the right-hand side of $\vdash$ is evaluated. In the case of the `signals`-clause context, the semantics of `\prev` expressions remains the same except for the context change from `ensures` to `signals`. In other contexts of clauses, the use of a `\prev` expression is disallowed.

Notice the semantic difference between `\prev` and `\old` expressions. Evaluating `\old(`E`)` is the same as evaluating E in the common pre-state (σ, h).

As a concrete example, consider the following code changes; the previous version increases field `f` by one whereas the current version by two.

```
void m() { f++; } → void m() { f+=2; }
```

Suppose that before method `m` is called, field `f` has value `0`. Then,

[5] Consider `requires \latest(x);` where `\latest(x)` becomes false.

Clause Type	Default Clause
requires	requires true;
when_ensured	when_ensured true;
when_signaled	when_signaled (Exception) true;
ensures	ensures true;
signals	signals (Exception) true;

Table 1: Default clauses of a change contract

the previous and the current version respectively change the value of f to 1 and 2. According to our semantics, the value of \prev(f) is 1 whereas the value of \old(f) is 0.

3.4 Semantics of \latest and \deprecated

We mentioned that \latest expressions can be used to indicate that fields, formal parameters of a method, or methods are added to the current version. To enforce the same-input assumption, missing parameters and fields of one version are copied from the other version. After such copying is done, we initialize the copies of parameters and fields with a special value $\perp$ right before a given method starts. For example, if a field f is added to the current version, then we assume that an assignment command f=$\perp$; is executed right before a member method of the previous version is called. Similarly, an artificial formal parameter is assumed to be assigned $\perp$ before its corresponding method body is executed. Then, we can define the semantics of \latest(E) as in Figure 5. It is considered true only if its sub-expression E is evaluated into a non-$\perp$ value only at the post-state of the current version, (σ'_2, h'_2).

The semantics of \deprecated(E) is symmetrical as shown in Figure 5. It is considered true only if E is evaluated into a non-$\perp$ value only at the post-state of the previous version, (σ'_1, h'_1). As said earlier, \latest and \deprecated expressions can appear in ensures and signals clauses.

A \latest expression can also take as its input a method name. In such a case, true is returned only if only the current version has a method of a specified name. The previous version is assumed to have the corresponding side-effect-free method with the same signature whose parameters are assigned $\perp$ and whose return value, if any, is $\perp$.

3.5 Semantics of a change-specification case

We now present the semantics of a change-specification case consisting of method-specification clauses. Note that not all clauses need to be present in a change-specification case. When a certain type of clause is omitted, a default clause is used. For example, if there is no requires clause in a given specification case, we insert the default clause requires true;. The default clause of each clause type is shown in Table 1. We treat the empty contract separately as will be explained in Section 3.7.

For the sake of explanation, consider the following complete change-specification case for method m. In the below, greek letters mean predicates, and two subscripted Ts represent exception types (i.e., subtypes of java.lang.Exception). Lastly, variables x1 and x2 are scoped to θ and θ', respectively.

```
/*@ changed_behavior
  @  requires $\varphi$;
  @  when_ensured $\psi$;
  @  when_signaled ($T_1$ x1) $\theta$;
  @  ensures $\psi'$;
  @  signals ($T_2$ x2) $\theta'$;
  @*/
```

$$\frac{\{\neg(\varphi \wedge (wp(m_1, \psi) \vee \widehat{wp}(m_1, T, \theta)))\} m_1 \{\rho\}}{\{\neg(\varphi \wedge (wp(m_1, \psi) \vee \widehat{wp}(m_1, T, \theta)))\} m_2 \{\rho\}}$$

Figure 6: By-default-equal rule; wp and $\widehat{wp}$ represent the weakest precondition transformers for normal and abnormal terminations, respectively, and m_1 and m_2 represent the previous and the current version of method m.

The above specification should be read as follows: when started with a pre-state satisfying φ, if the previous version of m satisfies ψ at its normal termination and θ at its abnormal termination raising an exception of type T_1, respectively, then the current version of m should satisfy ψ' at its normal termination and θ' at its abnormal termination raising an exception of type T_2, respectively.

The following verification condition provides the meaning of the given change specification more formally:

$$\varphi \wedge (wp(m_1, \psi) \vee \widehat{wp}(m_1, T_1, \theta)) \Rightarrow wp(m_2, \psi') \vee \widehat{wp}(m_2, T_2, \theta')$$

In the above, the previous and the current versions of method m are distinguished as m_1 and m_2. We use two weakest-precondition notations $wp(m, \psi)$ for method m and its normal post-condition ψ, and $\widehat{wp}(m, T, \theta)$ for m's abnormal post-condition θ and exception type T. The latter makes it sure that a raised exception is of type T before asserting θ; i.e., $\widehat{wp}(m, T, \theta) \Leftrightarrow ((\texttt{x instanceof } T) \Rightarrow wp(m, \theta))$, where x refers to a raised exception. For the sake of simplicity, we assume methods m_1 and m_2 have only two exit points, one for normal termination and the other one for abnormal termination. As usual, all free variables appearing in the verification condition are assumed to be universally quantified. We disallow the verification condition to be vacuously true by reporting an alarm when the left-hand side of the verification condition, $\varphi \wedge (wp(m_1, \psi) \vee \widehat{wp}(m_1, T_1, \theta))$, becomes false.

For the convenience of users, multiple instances of the same type clause are allowed to be included in a change specification case. When multiple instances of the same type clause exist, we reduce them to its semantically equivalent form with a single clause in a standard way by basically conjoining predicates of the same type. For example, writing requires φ_1;requires φ_2;when_ensured ψ_1;when_ensured ψ_2;ensures ψ'_1;ensures ψ'_2; is equivalent to writing requires $\varphi_1 \wedge \varphi_2$;when_ensured $\psi_1 \wedge \psi_2$;ensures $\psi'_1 \wedge \psi'_2$;. Similarly, signals (T_1 x1) θ'_1;signals (T_2 x2) θ'_2; is equivalent to a single clause signals (Exception x) ((x instanceof T_1) $\Rightarrow \theta'_1$)$\wedge$((x instanceof T_2)$\Rightarrow \theta'_2$);. Multiple instances of when_signaled are reduced in the same way.

3.6 By-default-equal rule

Typically, users make only small changes across versions (applicable for a subset of inputs, i.e., the change domain), and change contracts explicitly express when those changes occur. For the rest of the inputs, it is reasonable to expect that the program behavior (and outputs) remain unchanged across versions. More specifically, if a method is given an input that is not included in the input domain for changes specified as a contract, then the post-condition of the previous and the current versions should be equal. Such "by-default equality" of behavior is captured in the inference rule of Figure 6. This Hoare-style rule forces the post-condition ρ satisfied at the end of m_1 (i.e., the previous version) to be satisfied at the end of m_2 (i.e., the current version) as well unless input given to the method is included in the change domain specified by a change contract. The degree of equality is decided by what kinds of ρ are used by a supporting tool. Currently, our tool checks the equality between the return values of two versions of a given method, and also between the values of fields of the enclosing class.

```
Set m(String s){
 if(/*complex predicate on s*/)
  return new HashSet();
 else
  return new TreeSet();
}
```

(a) Previous program

```
Set m(String s){
 if(/*complex predicate on s*/)
  return new HashSet();
 else
  return new TreeSet().add(s);
}
```

(b) Current program

```
/*@changed_behavior
  @ when_ensured \result
      instanceof TreeSet;
  @ ensures \result.size() ==
      \prev(\result).size() + 1;
  @*/
```

(c) Change contract

Figure 7: Using previous result in pre-condition

3.7 Empty contract

We earlier mentioned that regression errors can be checked by giving an empty contract. This is because when an empty contract is given, it is reasonable to assume that no behavioral change should be observed after code changes. This is the case where our by-default-equal assumption is exploited to the extreme.[6]

3.8 Well-formed change contracts

Contracts should be well-formed to have a valid meaning. In addition to being grammatically correct, they also have to follow well-formedness rules provided in the below.

(i) As explained earlier, we disallow verification conditions to be vacuously true; change contracts reduced to verification conditions that are vacuously true are not considered well-formed. Thus, specifying `requires b;when_ensured !b; ensures b;` for a final boolean field `b` is ill-formed.

(ii) Expressions used in a change contract, including method calls, must be side-effect and exception free. Also, their execution must terminate.

(iii) It is illegal to use `\prev`(exp).f or `\old`(exp).f for field f; a field access cannot be made to a `\prev` or `\old` expression. This is to avoid confusion about the value of f when the value of $exp.f$ changes across versions or during method execution. More concretely, consider the following method that switches a boolean value of `f`.

```
//@ ensures x.f != \old(x).f;
void m(final T x) {x.f = !x.f;}
```

JML reduces a given expression `\old(x).f` to v`.f` where v represents a value of `\old(x)`. This v is the same as the value of final variable `x`, and thus at the given post-state (i.e., `ensures`) context, v`.f` is reduced to the same value as `x.f`, resulting in making the predicate of the given `ensures` clause false. Note that the value of `\old(x.f)` is, in contrast, the same as `!x.f` at the post-state. Meanwhile, other specification languages such as Jass [4] interpret `\old(x).f` as `\old(x.f)`. In fact, JML manual [15] recommends to use `\old(x.f)` in the situation like the above. Similar confusion could arise from `\prev` expressions if field accesses to them were allowed.

(iv) By the same token, we do not allow method-call expressions such as `\prev`(exp).m($args$) or `\old`(exp).m($args$) for method m and arguments $args$.

(v) In addition, `\prev` and `\old` expressions can be used as method arguments *only if* their types are primitive such as `int` or immutable such as `String`. Therefore, in the above example code, it is illegal to use an expression such as `m(\old(x))`. If the above expression was allowed, confusion would arise as to which value of field `f` should be used inside the method body of m.

4. WRITING CHANGE CONTRACTS

In this section, we focus on common styles of change contracts, as well as some example contracts for given example programs.

Change contracts mainly concern two aspects of program behavior – (i) under what conditions the program behavior changes (the pre-condition for the change), (ii) exactly how the program behavior changes (the post-condition after the change). We now discuss how these aspects of program changes are covered by writing of change contracts.

Specifying pre-conditions.

Usually when a method is changed, the behavior change is limited to only a sub-domain of its input space. We provide two ways to specify the input domains that contains behavior changes.

We can of course directly restrain the input domain of a method using its inputs (including parameters and fields). The keyword `requires` in JML serves this purpose. A change contract specification with `requires E` where `E` is an boolean expression on the method inputs means that we only focus on inputs that satisfy `E`. As mentioned in Section 3.5, a change contract comes equipped with pre-condition captured by a `requires` clause.

Apart from directly using `requires`, we can also specify the change contract pre-condition indirectly via `when_ensured` and `when_signaled` (the post-condition of the previous method). Suppose we have `when_ensured E1` in a change contract. It is equivalent to specifying `requires E2` where `E2` is the weakest pre-condition computed on the previous method with respect to `E1`. Thus, `when_ensured` and `when_signaled` can indirectly specify method pre-conditions. Figure 7 shows an example in which using `when_ensured` is more convenient (than using a `requires` clause) for specifying the pre-condition. In this example, if `requires` clause is used to specify the pre-condition, a complicated condition under which the `else` branch is executed has to be used. Instead, the complicated pre-condition can be simply specified using `when_ensured` clause on the post-state of previous method as shown in Figure 7.

Specifying behavior changes.

In general, there are two different styles to specify the behavior changes of a method. We can either specify the behavior of previous and current methods separately or the relation of their behaviors can be specified.

We first discuss the approach that the behavior of previous and current methods are separately specified. The behavior of the current method can be specified using `ensures` or `signals` JML clauses. Similarly, the behavior of previous method can be specified using `when_ensured` clause and `when_signaled` clauses. Apart from specifying the behavior separately, a relation of the current program behavior and previous program behavior can be used to specify how the current program behavior is different with

[6] Technically, ψ and θ of Figure 6 are assumed to be false.

```
// previous class    // current class                          // agent class
class C {            class C {                                 class A {
 int f;               int f;                                       int old_f, prev_f, mod_f;
                                                                   int old_x, prev_x, mod_x;
                                                                   int prev__res, mod__res;
                                                                   C old_this, prev_this, mod_this;
                                                                   Exception prev__expt, mode__expt;

                      //@changed_behavior                          //@normal_behavior
                      //@ requires \prev(f) > 0;                   //@ requires prev_f > 0;
                      //@ ensures \result == \prev(\result)+x;   //@ ensures mod__res == prev__res+mod_x;
                      //@ ensures \latest(x);                      //@ ensures true;
 public int m(){      public int m(int x){                         public static void test(){}
  return f;            return f+x;                                 public static void main(String[] a)
 }                    }                                            { /* First, init fields */ test(); }
}                    }                                         }
```

Figure 8: A generated agent class shown in the rightmost end; its fields and JML specification are translated from the user programs shown in its left-hand side.

respect to previous program behavior. This can be achieved in change contract language through a combination of `ensures` (or `signals`) and `\prev`.

Take the example in Figure 7. The change contract in Figure 7 is specified as a relation between the method return value of previous version and that of current version. Alternatively, the current method return value and previous method return value can be separately specified as follow

```
/*@ changed_behavior
  @ when_ensured \result instanceof TreeSet;
  @ when_ensured \result.size() == 0;
  @ ensures \result.size() == 1;
  @*/
```

Behavior-preserving changes.

Some program changes actually preserve program behavior. One common type of behavior-preserving change is code refactoring, which can used to increase program's manageability and extensibility. Programmers also make changes with functionality preserving goals such as increasing program performance, reducing memory consumption and so on. As far as the program's functional behavior is not changed, we consider it as behavior-preserving change in this paper. If a program's behavior is incidentally changed when behavior-preserving change is intended, a regression bug is introduced. Regression testing has been widely adopted to prevent regression errors. However, without knowing programmers' intention, it is difficult to classify a behavior change as a regression bug or an intended feature. Our notion of change contracts seeks to fill this gap. When behavior-preserving changes are made, no change contract needs to be written. By not providing any change contract, the default-equal assumption is activated. Any behavior change is clearly a regression bug, and it can be detected automatically by our checker - as evidenced by our experiments.

Another frequent situation is that new fields are added in a class and only operations on the new fields are added. In this case, regression errors can also be prevented using the default-equal assumption. When new fields are added/removed, the default-equal assumption guarantees that the common fields and method result have the same value in the previous version and current version.

5. CHANGE CONTRACT CHECKING

Change contracts are checkable. As practiced in program contracts, various levels of checking are possible from lightweight run-time assertion checking (RAC) to heavyweight full static program verification (FSPV) and extended static checking (ESC) in between. Each level of checking has its own strength and weakness. In general, the degree of completeness of checking increases toward the FSPV side while the degree of easiness in usage and automation increases toward the RAC side. Currently, our tool supports RAC because RAC has been recognized as the most essential support for many well-known Design-by-Contract languages such as Eiffel [16] and JML [5]. As will be shown in Section 6, our change contract checker was used to detect incorrect changes that caused regression errors in various Apache software.

Recall that by default we assume an empty change contract. Even when no explicit change contract is given, we can check behavioral equivalence between the previous and the current versions. Thus, non-trivial behaviorial equivalence checking between program versions can be achieved with an empty change contract (whereas no checking can be achieved with empty program contracts!) In the remaining of this section, we explain how we support run-time assertion checking (RAC) of change contracts.

To automatically check a given change contract, we reduce the problem of change contract checking to the well-established problem of program contract checking. More specifically, our change contract checking is performed in three steps. (i) We first run the previous and the current versions of a program, and log program states at a few checkpoints. Those checkpoints consist of the entry of the method under investigation, and exits of the previous and the current versions of that method. (ii) We then generate an agent program annotated with program contracts (*i.e.*, ordinary JML specifications) translated from a given change contract. Figure 8 shows in its rightmost end an agent class `A` we generate to check behavioral changes occurring when previous-version class `C` shown in the leftmost end is changed to the current one shown in the middle of the figure. Notice that method `test` of the generated agent class `A` is annotated with an ordinary JML specification resembling the given change contract shown in its left-hand side. A prominent difference between the original change contract and the generated JML specification is that the latter uses generated fields instead of the original expressions (*e.g.*, `mod__res` instead of `\result`). Those generated fields represent the program states logged in the previous step. Detailed explanation about generated fields and rules for agent generation will be provided shortly. Such agent program generation is performed in a way that the agent program passes the checking for the translated program contract iff. code changes occurred in the target program passes the checking of a given change contract.

(iii) Finally, in the last step, we perform run-time assertion checking (RAC) on the generated agent program using a RAC facility of OpenJML [18], a JML tool-suite built on Oracle's OpenJDK.

Our tool automatically performs the above three steps. First, to log program states, we use AspectJ [14], a popular tool supporting aspect-oriented programming for Java. Using call pointcuts of AspectJ, our tool logs program states at the aforementioned designated checkpoints before and after method calls. Those program states include the states of receiver object, method arguments, and method results. Next, our extension of OpenJML parses a given change contract and generates an agent class following our generation rules described below. In the following description, we assume that method m of class C is annotated with a change contract, and an agent class A is generated.

- If C has a field f with type T, agent class A has three fields old_f, prev_f and mod_f of type T to respectively represent the value of f before m enters, after the previous version m exits, and after the current version m exists.
- Similarly, if method m of C has a parameter p with type T, A has three fields old_p, prev_p and mod_p of type T.
- To represent receiver states, A also has three fields old_this, prev_this, and mod_this of type C.
- If method m has a non-void return type T, A has two fields prev__res and mod__res of type T. Note that we need only two fields in this case because there is no return value at the entry of a method.
- Similarly, A has two fields prev__expt and mod__expt of type Exception to represent the exceptions thrown.
- Lastly, A has only two methods main and test.

Method main of the last item of the above list performs two tasks. It (1) first initializes all the fields described above using the program states logged at the previous step, and then (2) calls method test. Meanwhile, method test is annotated with a translated program contract made up of the fields described in the preceding. The only purpose of adding test is to execute its annotated program contract through a RAC facility, and hence its body is empty. To obtain such translated program contract of test, we use the translation rules of Table 2. In the table, the notation $\varphi[x \mapsto x']$ is used to denote that free variables x appearing in φ are replaced with x'. While most of translation rules are obvious, \latest and \deprecated expressions are transformed to either true or false depending on the comparison result of abstract syntax trees for the previous and the current versions of the method.

Finally, our tool compiles a generated agent class with the RAC option of OpenJML turned on. Running the compiled code effectively checks the translated JML specification in the agent program, and its failure amounts to detecting a mismatch between actual code changes and the intended change expressed via the change contract. When the test input does not match any of the specification case in change contracts, the default-equal assumption is checked by checking whether all post-states of fields and method return values are the same in the previous version and current version.

6. EVALUATION

In this section, we evaluate change contract and our change contract language in two different aspects. We first focus on the expressiveness and usability of change contract language. Following this, we evaluate the efficacy of change contract in detecting incorrect program changes.

change contract	program contract
changed_behavior	normal_behavior
\old(x)	old_x
\prev(x)	prev_x
x	mod_x
\prev(\result)	prev__res
\result	mod__res
\latest(x)	true or false
\deprecated(x)	true or false
when_ensured φ	requires φ
when_signaled (T x) φ	requires (prev__expt instanceof T) && φ[x $\mapsto$ prev__expt]
signals (T x) φ	ensures (mod__expt instanceof T) && φ[x $\mapsto$ mod__expt]

Table 2: Translation rules

Three open source Java programs — Ant, JMeter and log4j are used in our evaluation. 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.

We evaluate the following two research questions (RQ).

RQ1: Can change contracts describe real-life changes.

We have conducted user studies to answer the above research questions. Two users participated in this user study. Both users are second-year Master's students majoring in computer science. Before the user study, they both have no knowledge on program contract and JML. The users are asked to first understand the programs as well as the changes across different versions. Based on their full understanding of the changes, they write change contracts. Note that the change contracts are written based on real changes rather than the intention of these changes. Another experiment in which change contracts are written based on the programmer's intention to prevent incorrect changes is presented later in this section.

We select changes from the Bugzilla database of each Java project. Only entries with patch files are selected. Each selected entry contains a set of discussions and some patch files containing the program changes. Note that other types of changes also exist in the Bugzilla database apart from bug-fixes. For example, new feature requests constantly appear in the Bugzilla database. We select changes in this way because the developers' comments and discussions in Bugzilla provide great help for understanding the changes. As these programs and changes are not written by the users, these detailed discussion logs are indeed very important for the users to understand the changes correctly.

The user study results are summarized in Table 3. The "Add/Delete" column denotes changes that involve adding or deleting

Subject prog.	Changes	Applicable Changes				Not Applicable	
		Refactoring	Behavior diff	Add/Delete	Not understood	Not concerned	Non-code
Ant	43	4	13	15	3	3	5
JMeter	17	1	5	6	1	4	0
log4j	20	2	6	7	1	0	4

Table 3: User study results on expressiveness and useability of change contract language

fields, methods or parameters. There are some changes involving library calls that are not open-source. Without the source code of the libraries, the users are not able to fully understand the effect of the changes. The amount of these changes is given in the "Not understood" column. The last two columns show the changes that are not applicable for this user study. The "Not concerned" column contains changes that are not concerned by change contract, such as changes in synchronization in multi-threaded programs. The "Non-code" column shows changes that are not inside Java source code files. For example, a change in XML file is considered as non-code change. In total, 52 change contracts were written for the changes in column "Behavior diff" and "Add/Delete". No change contract needs to be written for re-factoring changes as it is covered by the default equivalence assumption. In the process of writing change contracts, the users did not observe any case where changes cannot be expressed using change contracts.

Feedback from users.

We got the following feedback from users in this study.

- It is difficult to write change contract when the change happens on local variables that have long dependence chain from inputs and outputs. The users have to manually follow the program dependence chain to figure out under which condition the change is executed and how the change affects output. The users also suggested that program dependence tracking (such as the dependency analysis performed by a slicing tool) could reduce this manual effort.
- More time is spent on understanding the programs and changes than writing change contracts. This is however partly because the users did not write these programs themselves.
- Common changes seen in the user study are bug-fixes (typically fixing unexpected exceptions) and adding new features by adding new fields and methods. Changes in method signature and deletion of fields/methods are infrequent.

RQ 2: How effective are change contracts in terms of detecting incorrect changes.

In the previous study, users write change contracts based on their understanding of the real program changes. Thus, if users do not make any mistake either in understanding the program changes or in witting change contracts, the programs should always be consistent with the written change contracts. However, change contracts, as designed, should reflect the intention of program changes. Only when change contracts contain the intention of program changes, incorrect changes are possible to be detected by change contracts.

We use the following approach to find incorrect program changes and the intended changes from real-life software repositories. Similar approach has been used in existing research to find incorrect bug-fixes in operating systems [25]. We start with a bug-fix in the repository. Let `v3` be the version where a bug is fixed. We search backward in the repository to find where the bug fixed in `v3` is introduced. If the bug resides in method `m`, we only need to focus on the changes that touched method `m`. Suppose we find that a change from version `v1` to version `v2` introduced the bug. The change from `v1` to `v2` is clearly an incorrect change. When the programmer made changes in `v1`, the intended resultant program should be the bug-free program `v3`. The programmer's intention when changing version `v1` to `v2` is then captured by the differences between version `v1` and `v3`.

Subject prog.	Changes	Detected	Undetected
Ant	5	5	0
JMeter	3	3	0
log4j	2	2	0

Table 4: Checking of change contracts on incorrect changes

Incorrect changes and their corresponding intended changes are found in the three open source Java programs using the aforementioned method. Change contracts are then written based on the intention of changes instead of the real program changes. Original incorrect program changes are checked against the written change contracts. A test case stressing the incorrect change is required in the Runtime checking method mentioned in Section 5. We write unit tests with the goal of stressing these incorrect changes. We have also tried Randoop [19], which is a random unit test generation tool. However, Randoop was not able to generate test cases that meet our criteria for most of the changes. This is because we need test cases that satisfy certain pre-condition of the changed method.

Results from using change contracts to detect incorrect changes are shown in Table 4. We studied 10 incorrect changes in the repositories of Ant, JMeter and log4j. All incorrect changes are detected by the written change contracts.

7. RELATED WORK

Program contracts.

Design by contract [17] was proposed by Bertrand Meyer and first realized in Eiffel programming language [16]. Apart from Eiffel, Spec# [3] also incorporates design-by-contract into its core language. Different from Eiffel and Spec#, JML provides program contracts for the existing Java language. In the simplest form of design by contract, each method has its contract in the form of pre-condition and post-condition. A method has to guarantee its own post-condition whenever its pre-condition is satisfied. When a method invocation happens, it is then the caller's responsibility to guarantee the callee's pre-condition. As mentioned, change contracts differ fundamentally from program contracts. Program contracts capture the intended behavior of a single program whereas change contracts capture the intended behavior change between two program versions.

Inspired by the concept of Design by Contract, several research projects have focused on checking program code w.r.t. program contracts. Extended Static Checking [2, 8–10] aims at automated program contract checking at compile time. In extended static checking, verification conditions are generated from program code and the program contracts; these verification conditions are dispensed via automated theorem provers. Runtime assertion checking of contracts has also been studied [7]. In a typical runtime assertion checking system, program contracts are translated into checkable assertions and compiled into the associated program. By executing these assertions, the original program contracts are checked. The recently proposed hybrid checking approach [24] combines the power of static and dynamic checking of contracts. In this approach, a runtime assertion corresponding to a program contract is inserted into the compiled program and checked at runtime, only when it cannot be statically proved to be true.

Regression testing and debugging.

Regression errors constitute an important class of errors which can be automatically detected via default change contracts (which merely specify that the program output remains unchanged across program versions by default). In prior research, regression testing (finding test cases which expose regression errors), and regression debugging (methods to explain the failed behavior of tests exposing regression errors) have been studied. The work of [6, 11, 23] focus on the test selection and prioritization problem - choosing tests from a large test suite for exposing regressions. Thus, when a program is slightly modified, only a small number of regression test cases need be executed.

Recent research [13] by Jin et al. advances regression testing by automatically generating test cases to stress program changes. When a program is changed, a set of unit test cases concentrating on the changed portion of the program is automatically generated. These test cases are executed on both the unchanged and changed programs. Any observed behavior difference between the two versions is analyzed and presented to the user. Without a specification of whether a behavior difference is intended, programmers have to manually go through all the reported differences across program versions. When change contract is used together with regression testing, we can not only automatically differentiate regression errors from intended program behavior changes, but also detect incorrect new features and imperfect bug-fixes.

Past research on debugging of evolving programs [22, 26] have mainly focused on regression errors. Due to the lack of formal software requirements, other types of incorrect changes (such as incorrect implementation of a new feature and incorrect bug-fixes) have not been studied by these works. The proposed notion of change contracts can fill this gap.

Program difference summarization.

Semantic program difference summarization presents users with a clear view of semantic behavior changes introduced by syntactic program changes. Jackson and Ladd [12] propose to identify changes in input-output dependence chains in programs. However, differences at the input-output dependence level are too coarse-grained in certain cases. A change may alter the program's input-output dependence relation without changing the program's actual behavior. Differential symbolic execution (DSE) [20], on the other hand, computes the symbolic summary of program differences using symbolic execution. Unchanged program portions are abstracted using uninterpreted functions to achieve efficiency. The results from program difference summarization approaches (such as DSE) capture real program changes whereas our change contracts are designed to specify intended program changes.

8. DISCUSSION

In this paper, we propose 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 behavior differences across program versions, they can be easier to write than program contracts. In particular, owing to the default-equal assumption, regression errors can be detected without the need for writing any change contracts. Based on JML, we have designed a full annotation language for specifying intended changes in Java programs. We present the precise formal semantics of our annotation language for specifying change contracts. Several concrete examples are given to illustrate the usage of change contracts. We have also proposed a runtime checking method for change contract and implemented it based on the runtime assertion checker of JML. Through a user study on three Java open source projects (Ant, JMeter, log4j), we find that our change contract language is expressive and usable. In addition, all 10 incorrect changes found in our experiments are detected by their change contracts.

Over and above our technical contributions, we believe that the concept of change contract takes us one-step closer to the overarching goal of writing quality software. We conjecture that change contracts can be used in (at least) the following scenarios.

- *Early detection of incorrect program changes*. We have discussed this scenario in this paper. Either the programmer or the tester writes change contracts to make sure that program changes are correct.

- *Serving as program change requirement*. Change contracts can be written (potentially by programmers) prior to making changes in code. In this case, the change contracts serve as formal requirements for program changes.

- *Providing formal change logs*. Programmers often maintain change logs in natural language, to document the changes being made to programs. Sometimes, the change logs are inconsistent with real program changes. The inconsistency is hard to discover and causes serious confusion for other colleague programmers. This problem can be solved if checkable change contracts are used in change logs.

- *Change contract and previous program version jointly form the oracle for testing the current program*. In case the intended program behavior is not changed, we can use the output from previous program version for the purpose of testing the current program version. However, if the intended program behavior changes, the expected output of the current program can be found from the change contract as well as the previous program version's output.

- *Change contracts can help in test suite augmentation.* Current test suite augmentation approaches often focus on syntactic changes (*e.g.* [21]) to generate a test case which executes a syntactic change and propagates it to the output. However, there may exist many possible dependency chains across which a change's effect can be propagated only some of which violate the intended change. Capturing the intended change as change contracts can thus help in more accurate test suite augmentation.

In terms of future work, note that we have focused on runtime checking of change contracts in this paper. We plan to study static checking and hybrid checking of change contracts in the future.

9. REFERENCES

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

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[3] M. Barnett, K. Leino, and W. Schulte. The Spec# programming system: An overview. *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69, 2005.

[4] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103–117, October 2001.

[5] 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. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

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

[7] Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, pages 322–328. CSREA Press, 2002.

[8] D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. *In Proceedings of Internationl Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 108–128, 2005.

[9] D. Detlefs, K. Leino, G. Nelson, and J. Saxe. Extended static checking. Technical report, Compaq Systems Research Center, 1998.

[10] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.

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

[12] D. Jackson and D. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM*, pages 243–252, 1994.

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

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. Knudsen, editor, *ECOOP*, volume 2072 of *LNCS*, pages 327–354, 2001.

[15] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. R. Kiniry, and P. Chalin. *JML Reference Manual*. Available at `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html`.

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

[17] B. Meyer. Applying “Design by Contract". *IEEE Computer*, 25:40–51, 1992.

[18] OpenJML. OpenJML. `http://sourceforge.net/apps/trac/jmlspecs/wiki/OpenJml`, 2012.

[19] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA*, pages 815–816, 2007.

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

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

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

[23] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[24] J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In *SEFM*, pages 382–398, 2011.

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

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