Expressing and Checking Intended Changes via Software Change Contracts

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

ABSTRACT

Software errors often originate from incorrect changes, including incorrect program fixes, incorrect feature updates and so on. Capturing the intended program behavior explicitly via contracts is thus an attractive proposition. In our recent work, we had espoused the notion of "change contracts" to express the intended program behavior changes across program versions. Change contracts differ from program contracts in that they do not require the programmer to describe the intended behavior of program features which are unchanged across program versions. In this work, we present the formal semantics of our change contract language built on top of the Java Modeling Language (JML). Our change contract language can describe behavioral as well as structural changes. We evaluate the expressivity of the change contract language via a survey given to final year undergraduate students. The survey results enable us to understand the usability of our change contract language for purposes of writing contracts, comprehending written contracts, and modifying programs according to given change contracts. Finally, we discuss the tool support developed for our change contract language. The tool support enables (i) test generation to witness contract violation, as well as (ii) automated repair of certain tests which are broken due to program changes.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Programming by contract*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics, Syntax*

General Terms

Experimentation, Languages, Reliability

Keywords

Software changes, Dynamic checking, Test generation, Test repair

1. INTRODUCTION

Programmers often toil for hours or even days to find the root-cause of a single pernicious "bug" or observed error. What makes debugging so difficult? The difficulty in debugging primarily comes from the lack of capture of intended program behavior. Whenever a test case fails, it is due to an "unexpected" observable event — an unexpected output, or a program crash. Yet, what is "expected" from the program is hardly ever formally captured.

Program contracts, or Design by Contract programming [3,4,14] provide an alternative in this regard since they recommend writing contracts to express intended program behavior. Contracts may 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 puts the task of writing contracts squarely on the programmer. This typically leads to lack of widespread adoption of program contracts by programmers [18].

In our recent short paper [21], we have espoused the notion of "change contracts" where the intended behavior of program changes are expressed in a customized change contract language. Change contracts focus only on the program changes and their intended semantic effect. We believe this eases the task of writing contracts for several reasons. First of all, program behavior that is unchanged across versions does not need to be captured. Secondly, while contracts describing the intended behavior of a program typically capture the intended input-output relationship in a program, change contracts also retain the flexibility of describing the output-output relationship across program versions. Thus, it can describe properties like

$$\textit{whenever } in > 0 \textit{ holds, } out' == out + 1$$

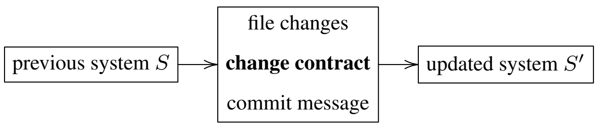
or even a property like

$$\textit{whenever } out > 0 \textit{ holds, } out' == out + 1$$

where in denotes input, out' denotes output of the updated program version and out denotes output of the previous version. As we show, such descriptions are likely to be more concise than a usual program contract of the form *whenever* $\varphi(in)$ *holds,* $out' == f(in)$ where $\varphi(in)$ is a constraint on the input, and the function application $f(in)$ captures the intended input-output relationship in the changed program version. Both $\varphi(in)$ and $f(in)$ can often be fairly complicated. The additional flexibility of relating the program outputs across program versions often leads to concise and intuitive change contract specifications.

In this paper, we study the expressivity/usability of our change contract language via a detailed user survey as well as by developing a change-contract based infra-structure to help locate failed/broken tests. The contributions in this paper are now stated in the following paragraphs.

Our change contract language is built on top of the Java Modeling Language (JML) [4]. Unlike the conventional program contract languages which typically provide pre/post condition of methods — we describe how the post-conditions of the same method in two consecutive versions relate to each other, under certain pre-conditions. Exceptional behavior, as well as structural changes



(a) The overview of a change contract configuration

Bug 51668 - <junitreport> broken on JDK 7 when a SecurityManager is set
Fails with: "Use of the extension element 'redirect' is not allowed when the secure processing feature is set to true." It turns out to apply to any environment in which there is a system security manager set. JDK 7's TransformerFactoryImpl constructor introduced:

```
    if (System.getSecurityManager() != null) {
        _isSecureMode = true; _isNotSecureProcessing = false;
    }
```

which conflicts with <redirect:write>.

(b) A sample Bugzilla report for software Ant

```
// file: XMLResultAggregator.jml
package org.apache.tools.ant.taskdefs.optional.junit;

public class XMLResultAggregator extends Task implements XMLConstants {
  /*@ changed_behavior
    @  requires System.getSecurityManager() != null &&
    @     System.getProperty("java.runtime.version").startsWith("1.7") &&
    @     getDestinationFile().exists() == false;
    @  when_signaled (BuildException e) e.getMessage().contains(
    @     "Use of the extension element 'redirect' is not allowed " +
    @     "when the secure processing feature is set to true.");
    @  signals (BuildException e) false;
    @  ensures getDestinationFile().exists();
    @*/
  public void execute() throws BuildException;
}
```

(c) A change contract corresponding to the bug report in (b)

```
// file: SourceTypeBinding.jml
package org.eclipse.jdt.internal.compiler.lookup;

class SourceTypeBinding extends ReferenceBinding {
    /*@ changed_behavior
      @  requires method.parameters.length > 0;
      @  when_ensured method.parameterNonNullness[0].booleanValue() ==>
      @             isNonNull(method.sourceMethod().arguments[0]) == false;
      @  ensures method.parameterNonNullness[0].booleanValue() ==>
      @             isNonNull(method.sourceMethod().arguments[0]) == true;
      @*/
    public MethodBinding resolveTypesFor(MethodBinding method);

    /*@ model boolean isNonNull(Argument arg) {
    return (arg.binding.tagBits & TagBits.AnnotationNonNull) != 0; } @*/
}
```

(d) A core-developer-level change contract

Figure 1: The overview and examples of change contracts

(such as introduction or removal of parameters/fields etc) and conditional refactoring (i.e., refactoring under a certain condition) are also supported. We present in § 3 our change contract language along with its syntax and formal semantics.

To evaluate possible field usage of change contracts, we conducted a survey of sixteen (16) final year undergraduate students in a senior year course at the National University of Singapore. The survey was administered as a mini-test with 20 questions lasting 60 minutes, accounting for 10% of grade in the course. The students participating in the survey had no prior background of program contracts or change contracts or JML. They were only provided one tutorial on these topics in a single week's lesson. The questions in the survey involved comprehending/writing change contracts and modifying code based on change contracts for small programs, as well as fragments of real-life programs. The results from the survey point to the possible ease of using our change contract language — with an overall correct answer rate of 92% from the respondents, in less than one hour for 20 questions.

Finally, we develop tool support to help find tests which lead to violations of a given change contract. We modify Randoop [17] to generate only tests that satisfy one of two necessary conditions for change contract violation; the other condition is checked while running those generated tests. We also provide tool support for repairing tests which are broken due to structural changes across program versions. We present experimental evaluation results summarizing the size of the change contracts, time to find the first test (if any), and whether change contract violation (if any) is detected. All the results are obtained from the well-known software project *Ant*. The experiments point to the efficacy of our tool support for checking violations of change contracts.

2. OVERVIEW

Figure 1(a) shows how change contracts are to be configured in the history of a software system. Change contracts are to be maintained along with file changes and commit messages in a version control system such as Git and Mercurial. While file changes represent actual code changes, change contracts capture the underlying intended changes.

Figure 1(c) shows an example of a change contract for the execute method of software Ant [1]. It almost looks like a typical JML annotation except that it uses a couple of extra keywords such as "changed_behavior" and "when_signaled". While the meaning of those keywords is described in § 3 in detail, changed_behavior indicates that its following contents are for a change contract, not for a program contract, and when_signaled is used to describe the output condition of the previous version method while signals can be used for the output condition of the updated version. While when_signaled and signals are for abnormal termination that signals an exception, output conditions for normal termination can be described with when_ensured and ensures. Meanwhile, to describe the shared input condition of the previous/updated versions, a requires clause is used.

Notice that a change contract is provided as a separate file, instead of annotating the program files. The change contract in Figure 1(c) is the contents of a contract file XMLResultAggregator.jml, and it describes behavioral changes between two consecutive versions of Java file XMLResultAggregator.java.

The change contract of Figure 1(c) is a counterpart of a verbal description given in a bug report of Figure 1(b). This bug report describes (i) an observed symptom (i.e., "Fails with: "Use of the extension ...") and (ii) necessary conditions to reproduce that symptom (i.e., "broken on JDK 7 when a SecurityManager is set"). A change contract expresses those descriptions programmatically. In our example, the above symptom is described with a when_signaled clause to specify that a behavior change is necessary when a BuildException is signaled in the previous version along with the error message described in that when_signaled clause.

Meanwhile, a requires clause is used to describe the necessary condition to reproduce the symptom. Its predicate expresses, using the standard methods of Java, the two conditions to reproduce the symptom, (i) a SecurityManager is set and (ii) JDK version is 7. In addition, it is also assumed that the destination XML file that is supposed to be generated after a successful run of the execute method (i.e., the target method of the above change contract) does not yet exist.

Once a symptom and reproduction conditions are recognized, one may wish to change the behavior in a specific way. In the case of the above example, it is obvious that the same exception should not be signaled in the updated version. Instead, (i) the execute method should terminate normally and (ii) the destination XML file should be successfully generated. Notice in the above change contract that these two intentions are expressed with the signals clause (by using false as a predicate) and ensures clause, respectively.

While the level of the intentions expressed in our first change contract example is close to the one of an end-user, lower level intentions made by core developers of software can also be expressed in a change contract. Figure 1(d) shows such a low-level change contract for the resolveTypesFor method of Eclipse JDT (Java Development Tools) [11]. This change contract equivalent to the JDT's Bugzilla report number 388281 expresses the intention to fix the mismatch between method.parameterNonNullness[0] (a boolean value) and method.sourceMethod().arguments[0] (a bitmask). The when_ensured clause of Figure 1(d) describes that the bitmask was not properly set in the previous version; the following ensures clause specifies that, in the updated version, the bitmask should be properly set instead.

We use a model method, isNonNull, in Figure 1(d) to improve the readability of a change contract. A model method is essentially an extra specification-purpose method whose execution does not alter the functional behavior of the program in a noticeable way. In JML upon which our change contract language is based, a model method is described between "/*@ model" and "@*/". It is often handy to define a model method and use it in a change contract as a predicate.

One may argue that existing program contract languages such as JML can already express the behavior described in the above examples. Indeed, one can write JML specifications corresponding to Figure 1(c) and Figure 1(d) without using change contract's when_signaled and when_ensured clauses. Instead, one can calculate the weakest pre-condition (viz., input condition) under which the observed symptom (viz., output condition) is bound to be reproduced, and write in a contract input-output relationship instead of writing output-output relationship of a change contract.

However, such specifications that solely rely on input-output relationship are, in general, not as intuitive as our change contracts for the following two reasons. First, while change contracts can clearly show the symptoms observed in the previous version such as throwing an exception, program contracts can hardly reveal those symptoms. After all, program contracts do not distinguish the previous version form the updated version. Second, it is often the case that output-output relationship is simpler and thus more comprehensible than its equivalent input-output relationship. For example, in Figure 1(c), imagine calculating the weakest pre-condition that induces at the method exit a BuildException along with the particular error message. Such a pre-condition can be quite long and complex depending on how complex the method body is and how specific the symptom is.

Our change contract can express not only behavioral changes but also structural changes such as adding a new method parameter. Such an example will be shown in § 3.3.

3. CHANGE CONTRACT LANGUAGE

To express intended program changes, we extend a subset of JML [4], the 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 as possible to lower the learning barrier, and our syntactic extension to JML is very limited. However, JML or any other specification languages, to the best of our knowledge, is not expressive enough to express program changes across two consecutive versions, and this requires us to propose non-trivial semantic extensions.

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*;
| **when_required** *pred*; | **preserves_when** *pred*;

exp ::= ... | \result | \old(*exp*) | **\prev**(*exp*)

param-modifier ::= ... | **new_param** | **old_param**

jml-modifier ::= ... | **new_field** | **old_field**

Figure 2: Change contract language as an extension to a JML subset; standard regular expression notation $*$ is used.

Notes on Expressivity. While the main objective of our change contract language is to specify behavioral changes that occur between two consecutive versions of a method, it is also possible to specify with this language accompanying structural changes such as adding/deleting method parameters or fields. While our change contract language captures the relationship among program variable values at the input/output points of the previous/updated program versions - it is not powerful enough to express temporal properties of changes in variable values, as in temporal logics. Lastly, as in JML, we are concerned only with sequential Java programs, and do not consider multi-threading.

A change contract is specified above the signature of a method m as an annotation between "/*@ changed_behavior" and "@*/". We call such a method m the *target method* of a given change contract. We require that expressions used in a change contract, including method calls, must be free of side effects and exceptions. Also, their execution must terminate. A change contract is maintained as a contract file (e.g., XXX.jml) separated from Java files.

3.1 Syntax

Figure 2 shows the syntax of our change contract language. The keywords in bold face are extensions to the standard JML. A change contract starts with the keyword "changed_behavior" followed by clauses that describe the pre/post conditions of a common target method of the previous/updated versions. To describe the pre-/post-conditions of an updated version, we use the existing JML clauses: a requires clause for a pre-condition and ensures/signals clauses for post-conditions; ensures expresses the post-condition at normal method termination (i.e., termination without throwing an exception), and signals the post-condition at abnormal method termination (i.e., termination with an exception thrown). Meanwhile, to describe the counterparts of the previous version, we introduce additional clauses: when_required, when_ensured, and when_signaled. For simplicity, we often use a shorthand notation $(\varphi, \psi, \theta; \varphi', \psi', \theta')$ to mean the following full change contract:

```
/*@ changed_behavior
  @ when_required φ; when_ensured ψ;  when_signaled (T1 x) θ;
  @  requires φ'; ensures ψ'; signals (T2 x) θ'; @*/
```

In the above, greek letters denote predicates, T_1 and T_2 represent exception types (i.e., subtypes of java.lang.Exception), and two instances of variable x are scoped to θ and θ', respectively. Note that not all clauses need to be present in a change contract. When a certain clause is omitted, a default predicate for that clause is used as will be explained later.

A requires clause often is shared between the previous/updated versions as a common pre-condition. A when_required clause is used only when it is necessary to distinguish the pre-conditions between the previous version and the updated version. For example, if the pre-condition of the updated version depends on a newly added method parameter, then the same pre-condition cannot be used for the previous version. In such a case, the pre-condition of the previous version can be separately expressed with a when_required clause.

The keyword \prev constructs a "prev" expression that accesses the previous-version value from an updated-version context. For example, one can write "ensures x==\prev(x)+1;" to express the intention that the value of x at the post-state of the updated version should be greater by one than the value of x at the post-state of the previous version. Readers familiar with JML could find the similarity between \prev and the \old of JML. While \old makes a value of a pre-state available at a post-state, \prev makes a value of the previous version available at the updated version.

It is not unusual in some programs to modify a method signature or a field when changing (or preserving) the behavior of a method. Our change contract language can handle common structural changes such as adding or deleting a method parameter or a field by using the keywords new_param, old_param, new_field, and old_field.

3.2 Semantics

3.2.1. Execution Model. It is convenient to conceptually assume that two versions of a program are run in parallel when considering the semantics of a change contract between two versions of a program. Recall that a change contract concerns currently only sequential programs as JML does, and the introduced parallelism is not intended to interfere with Java's multi-threading. The overall semantic rule shown in Figure 3(b) clarifies such a parallel execution model. Given two commands, c_1 and c_2, that represent the method bodies of the previous and the updated versions respectively, we assume that they are run in parallel as denoted with $c_1 \mid\mid c_2$.

Nonetheless, not all parallel executions $c_1 \mid\mid c_2$ are interesting to the users of a change contract. For example, given a change contract, ensures \result==\prev(\result)+1, of a method m(int x), one would expect the increase of the return value only when the same integer value for parameter x is given to both versions. Roughly speaking, input equality between the two versions needs to be assumed when considering a change contract. However, naive input equality is not enough for two reasons. First, the above parameter x may not be of a primitive type but of a subtype of Object. If that is the case, simple reference comparison is inappropriate. Second, the method signatures of the two versions are not necessarily the same. For example, a user may want to add a new parameter to a method.

To address the first issue, we compare object graphs instead of object references. Conventionally, two graphs are considered isomorphic if there is a unique one-to-one correspondence between the vertexes and edges of the two graphs. If, in addition, all the one-to-one corresponding vertexes that represent primitive values of the two object graphs contain the same values, the two object graphs are considered isomorphic. We extend this notion of isomorphism to the program state level. A program state consists of a store σ and a heap h, and two program states, (σ_1, h_1) and (σ_2, h_2), are considered isomorphic if for all variables x that commonly exist

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

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

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

(a) Semantic domains

$$\frac{\begin{array}{c}(\sigma_1, h_1) \approx (\sigma_2, h_2) \qquad h_1 \perp h_2 \\ \langle c_1, (\sigma_1, h_1)\rangle \Downarrow_c (\sigma_1', h_1') \qquad \langle c_2, (\sigma_2, h_2)\rangle \Downarrow_c (\sigma_2', h_2')\end{array}}{\langle c_1 \mid\mid c_2, (\sigma_1, h_1, \sigma_2, h_2)\rangle \Downarrow_c (\sigma_1', h_1', \sigma_2', h_2')}$$

(b) An overall semantic rule that describes our parallel execution model; for explanation, refer to §3.2.1.

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

$$\frac{\langle c_1 \mid\mid c_2, (\sigma_1, h_1, \sigma_2, h_2)\rangle \Downarrow_c (\sigma_1', h_1', \sigma_2', h_2') \qquad \langle E, (\sigma_1, h_1)\rangle \Downarrow_e v}{\text{requires} \vdash \langle \backslash\text{prev}(E), \sigma_1', h_1', \sigma_2', h_2'\rangle \Downarrow_e v}$$

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

$$\frac{\langle c_1 \mid\mid c_2, (\sigma_1, h_1, \sigma_2, h_2)\rangle \Downarrow_c (\sigma_1', h_1', \sigma_2', h_2') \qquad \langle E, (\sigma_1, h_1)\rangle \Downarrow_e v}{\text{ensures} \vdash \langle \backslash\text{old}(\backslash\text{prev}(E)), (\sigma_1', h_1', \sigma_2', h_2')\rangle \Downarrow_e v}$$

(c) Semantic rules for \prev(E) expressions; for explanation, refer to §3.2.2.

$$\frac{\begin{array}{c}\langle c_1 \mid\mid c_2, (\sigma_1, h_1, \sigma_2, h_2)\rangle \Downarrow_c (\sigma_1', h_1', \sigma_2', h_2') \\ (\sigma_1, h_1) \vdash \varphi \qquad (\sigma_1', h_1') \vdash \psi \vee (\sigma_1', h_1') \vdash \theta \\ (\sigma_2, h_2) \vdash \varphi' \qquad (\sigma_2', h_2') \vdash \psi' \wedge (\sigma_2', h_2') \vdash \theta'\end{array}}{\langle c_1 \mid\mid c_2, (\sigma_1, h_1, \sigma_2, h_2)\rangle \vdash (\varphi, \psi, \theta; \varphi', \psi', \theta')}$$

(d) An inference rule for a change contract $(\varphi, \psi, \theta; \varphi', \psi', \theta')$; the second and the third lines correspond to the update condition and the change condition, respectively; for explanation, refer to §3.2.3.

Figure 3: Semantic rules for given two method bodies, c_1 and c_2, of the previous and the updated version, respectively; $\Downarrow_e$ and $\Downarrow_c$ represent reduction relations of big-step operational semantics for expressions and commands, respectively.

in the domain of σ_1 and σ_2, the two object graphs that $\sigma_1(x)$ and $\sigma_2(x)$ respectively refer to are isomorphic to each other. Note that heaps, h_1 and h_2, are consulted if necessary when constructing object graphs. Also note that, as usual, the receiver of an object (i.e., this) is considered an implicit parameter of a non-static method.

To address the second issue concerning structural changes, we exclude during comparison of object graphs method parameters and fields that are not in common between two versions. For example, if a method parameter is added to an updated version, then that added parameter is not compared between the two versions. Similarly, fields that are newly added to the updated version or deleted from the previous version do not contribute to the construction of object graphs.

Overall, we say that two states are isomorphic modulo structural changes if these two states are isomorphic when ignoring variables and fields that are not in common between the two versions. We denote such isomorphic states modulo structural changes with notation $(\sigma_1, h_1) \approx (\sigma_2, h_2)$. Notice that our overall semantic rule in Figure 3(b) has $(\sigma_1, h_1) \approx (\sigma_2, h_2)$ in its premise to force isomorphic inputs. For simplicity, we will simply say that two states are isomorphic omitting "modulo structural changes".

We impose one more restriction on our parallel semantics. Executing two versions of a method in parallel should not interfere with each other. Recall that we use parallelism only for the purpose of analyzing behavioral changes across versions. To guarantee non-interference, we maintain a disjoint heap for each version of a method. More precisely, the domains of the two heaps, h_1 and h_2, are forced to be disjoint, and we denote such a constraint with $h_1 \perp h_2$ as shown in the premise of our overall semantic rule (i.e., Figure 3(b)).

Once two input states, (σ_1, h_1) and (σ_2, h_2), satisfy the isomorphism condition (i.e., $(\sigma_1, h_1) \approx (\sigma_2, h_2)$) and the heap disjointness condition (i.e., $h_1 \perp h_2$), the two versions are run in parallel in an obvious way without interfering with each other. As a result, we obtain the reduction relation appearing in the conclusion part of the rule: $\langle c_1 \mid\mid c_2, (\sigma_1, h_1, \sigma_2, h_2)\rangle \Downarrow_c (\sigma_1', h_1', \sigma_2', h_2')$. Recall that c_1 and c_2 amount to the method body of the previous and the updated versions, respectively. Accordingly, input states (σ_1, h_1) and (σ_2, h_2) amount to pre-states of the previous version and the updated version, respectively, and output states (σ_1', h_1') and (σ_2', h_2'), the post-states of the previous and the updated versions, respectively.

3.2.2. \prev Expression. Our prev expressions can be used in a change contract to refer to the value of the previous version from the context of the updated version. The value of \prev(E) is decided depending on where this prev expression appears. If \prev(E) appears in an ensures clause or a signals clause (i.e., the post-condition of the updated version), E should be evaluated in the post-state of the previous version (i.e., (σ_1', h_1')). Meanwhile, if it appears in a requires clause (i.e., the pre-condition of the updated version), E should be evaluated in the pre-state of the previous version (i.e., (σ_1, h_1)). Such a difference is captured in the two topmost rules in Figure 3(c) where notations "ensures $\vdash$" and "requires $\vdash$" designate the clause in which a prev expression appears. The cases for the signals clause are omitted because they can be treated identically to the cases for the ensures clause.

Notice that a prev expression, regardless of where it appears, makes a context switch from the updated version to the previous version. Such a context switch over a program version made by a prev expression is orthogonal to the old expression's context switch from a post-state to a pre-state.

3.2.3. Update Condition, Change Condition and Inference Rule. Given a change contract $(\varphi, \psi, \theta; \varphi', \psi', \theta')$ and two versions of a program that satisfy $\langle c_1 \mid\mid c_2, (\sigma_1, h_1, \sigma_2, h_2)\rangle \Downarrow_c (\sigma_1', h_1', \sigma_2', h_2')$, we check if the given change contract is satisfied using the inference rule shown in Figure 3(d). We write $\langle c_1 \mid\mid c_2, (\sigma_1, h_1, \sigma_2, h_2)\rangle \vdash (\varphi, \psi, \theta; \varphi', \psi', \theta')$ in the conclusion part of the rule to mean that change contract $(\varphi, \psi, \theta; \varphi', \psi', \theta')$ is satisfied in the context of configuration $\langle c_1 \mid\mid c_2, (\sigma_1, h_1, \sigma_2, h_2)\rangle$.

In order for a change contract to be satisfied, the pre-condition of the previous version must be satisfied beforehand at the pre-state of the previous version. Such a condition is expressed in the premise part of the rule as $(\sigma_1, h_1) \vdash \varphi$; we write $(\sigma_1, h_1) \vdash \varphi$ if predicate φ is satisfied at state (σ_1, h_1).

In addition, one of post-conditions of the previous version (recall that there are two kinds of post-conditions depending on whether the target method terminates normally or not) must also be satisfied at the post-state of the previous version. Such a condition is denoted in the inference rule as $(\sigma_1', h_1') \vdash \psi \vee (\sigma_1', h_1') \vdash \theta$. We say that the *update condition* is satisfied if the above two conditions hold true as described in the second line of the inference rule. If the update condition holds, it means that a given input state (σ_1, h_1) triggers in the previous version an execution whose behavior is intended to be changed in the updated version.

Table 1: Rules to fill in omitted clauses with default predicates; for explanation, refer to §3.2.4.

Omitted clause	Context	Default predicate
when_ensured	∃when_signaled	false
	∄when_signaled	true
when_signaled	∃when_ensured	false
	∄when_ensured	true
when_required	∃requires φ'	φ'
	∄requires	true
requires	always	true
ensures	always	true
signals	always	true

Once the update condition holds, we next check another condition we call the *change condition* to see if the behavior of the execution of interest changes as intended. The change condition is described in the third line of the inference rule. To see if the change condition is satisfied, we check the following two conditions. First, we check if the pre-condition of the updated version is satisfied at the pre-state of the updated version (i.e., $(\sigma_2, h_2) \vdash \varphi'$ of the rule). Note that we can assume that (σ_2, h_2) is isomorphic to (σ_1, h_1) because that is implied by $\langle c_1 \mid\mid c_2, (\sigma_1, h_1, \sigma_2, h_2)\rangle \Downarrow_c (\sigma_1', h_1', \sigma_2', h_2')$ in the premise of the inference rule. Next, we check all the post-conditions of the updated version are satisfied at the post-state of the updated version (i.e., $(\sigma_2', h_2') \vdash \psi' \wedge (\sigma_2', h_2') \vdash \theta'$). We assume that prev expressions appearing in ψ' or θ' are replaced with their values obtained using their semantic rules explained earlier.

If both the update and the change conditions hold, we conclude that a given change contract is satisfied under the given input states of the two versions of a program. Meanwhile, we report a change-contract violation only if the last condition of the inference rule does not hold (i.e., $\neg((\sigma_2', h_2') \vdash \psi' \wedge (\sigma_2', h_2') \vdash \theta'))$ while the preceding conditions hold.

3.2.4. Default Predicates for Omitted Clauses. We earlier mentioned that if a certain clause is omitted, then its default predicate is used. For example, if there does not appear a when_ensured clause in a change contract while there appears a when_signaled clause, an omitted when_ensured clause is implicitly assigned false as its default predicate because we are in this case interested only in abnormal termination of the previous-version method. Imagine a situation where one wants to deal with only an unexpected exception (e.g., Figure 1(c)). What if there is neither a when_ensured nor when_signaled clause in a change contract, and only an ensures clause containing a prev expression appears as the following?

"ensures \result==\prev(\result)+1;"

If this is the case, we do not assign the same false predicate to the omitted when_ensured clause because by doing so the \prev(\result) expression cannot have any value. To address this problem, we assign a default predicate differently depending on the context following the rules of Table 1. Most rules are straightforward. Let us explain the case for when_required though. If there is no structural changes, it is most likely that writers of a change contract would want to assume the same pre-condition for the previous and the updated versions. We accordingly assign the predicate of a given requires clause to the omitted when_required clause.

```
public class DirectoryScanner implements FileScanner {
  /*@ changed_behavior
    @  when_required true;
    @  requires !cs;
    @  ensures \result != null &&
    @     \result.equals(\old(\prev(findFileCaseInsensitive(base, path))));
    @  preserves_when cs;  @*/
  File findFile(File base, String path, /*@ new_param @*/ boolean cs);
}
```

Figure 4: DirectoryScanner.jml: a change contract involving structural changes

3.3 Structural Changes

Consider an example of Figure 4 where a method findFile of the previous version changes its signature by adding an boolean parameter cs (standing for "Case Sensitivity") at the end. Such a method signature change can also be accommodated into a change contract; in this example, the new parameter cs is annotated with /*@ new_param @*/. Similarly, the parameter of the previous version removed from the updated version, if any, is annotated with /*@ old_param @*/

When reading the method signature of a change contract, one can get the signature of the previous version by including parameters annotated with /*@ old_param @*/ and non-annotated parameters while excluding parameters annotated with /*@ new_param @*/. The signature of the updated version can also be obtained in the opposite way. Notice that the order of the parameters and parameter names are preserved in a change contract.

Similarly to parameter changes, field addition and removal are annotated with /*@ new_field @*/ and /*@ old_field @*/ modifiers. For example, if one wants to add a new private int-type field f to the above DirectoryScanner class, then she can add the following inside the class declaration of DirectoryScanner of the DirectoryScanner.jml file: "private /*@ new_field @*/ int f;"

As a side note, Figure 4 also shows an interesting combination of \old and \prev expressions. Notice that in the ensures clause expression \old(\prev(findFileCaseInsensitive(base, path))) is used. Using that expression, a side-effect-free method findFileCaseInsensitive can be executed in the context of the pre-state (due to \old) of the previous version (due to \prev).

3.4 Refactoring

Structural changes are often accompanied by refactoring. Consider the example of Figure 4 where method findFile takes an additional boolean parameter cs to indicate the case-sensitiveness. While the shown change contract describes that the result of findFile should coincide with the result of findFileCaseInsensitive of the previous version when cs is false, the expected behavior of findFile when cs is true is not explicitly described with ensures or signals clauses. While it is possible to do so, the example uses instead a simpler clause "preserves_when cs;" to mean the output isomorphism, which is defined similarly to input isomorphism, between the previous and updated versions. A preserves_when clause is to describe the condition in which behaviors should be preserved.

3.5 Regression Bugs

While intended changes can be expressed and checked through a change contract, it is also of interest to developers to check if there is a regression bug. To find a regression bug, one can compare the output states obtained when isomorphic inputs are given to the two versions. If the update condition of a given change contract does not hold, disequality between two output states indicates a regression bug. Without a change contract, it is difficult to distinguish a regression bug from software progression even if disequality between two output states is found.

3.6 Change Contract Composition

In program verification, it is common to reuse the program contracts of the callees (instead of the bodies of them) to verify the contract of the caller. Such a compositional approach is difficult to be applied to change contracts for the following two reasons. First, we do not force the users to write change contracts for every method that is changed. For instance, in our first change contract example (i.e., Figure 1(c)), method execute's callees that are changed across versions do not have to be assigned change contracts. User's intention about changes are expressed at the level of the execute method without constraining the changes of lower level methods. Second, even if there is a change contract of a callee available, the extent of its usefulness is limited by innate partial information of a change contract. A change contract itself does not show how the program state should change if changes across versions are not intended at that program state. In such a case, the method body of the previous version might as well be looked up.

Meanwhile, multiple change contracts can be composed in a different way from the conventional composition of program contracts. In the following, we compose (i) one change contract cc_1 that expresses changes of method m between v_1 (i.e., version 1) and v_2 and (ii) the other change contract cc_2 of m for changes between v_2 and v_3.

$$\begin{tikzcd} m \arrow[r, "cc_1"] \arrow[rr, bend right, "cc_1 \circ cc_2"'] & m' \arrow[r, "cc_2"] & m'' \end{tikzcd}$$

In theory, one can synthesize a single change contract from a series of change contracts by applying composition rules of change contracts. Also, one can detect conflicts between two changes in history through change contract composition. Two changes conflict with each other if the change condition of the composed change contract is $false$ while its update condition is not. Such conflict detection can be useful when developers unconsciously try to make changes that are in conflict with previous changes. We do not have a complete set of change contract composition rules yet, and leave it as future work.

4. USER STUDY

To evaluate possible field usage of change contracts, we conducted a survey of sixteen (16) final year undergraduate students in a senior year course (formal verification of embedded software) at the National University of Singapore in 2012.

4.1 Demographics

We asked seven (7) demographic questions. Almost all respondents responded that they have experience in programming in Java for certain projects. Only two respondents responded that they had equivalent experience with another programming language, one with C++ and the other with Python. Meanwhile, all respondents responded that they had used neither JML nor any other program contract languages before. Overall, our participants can be considered equivalent to entry-level developers who have no background of program specification.

4.2 Survey Questionnaire

Figure 5 shows two sample questions from our survey questionnaire that encompass diverse question types we describe in this section. Each of our questions falls under primarily one of the following three types of questions:

(i). Read-Modify (RM) type questions. In this type of questions, we show a program and its change contract and then ask respondents to modify the program in a way to reflect the given change contract. This type of question measures how easy it is to comprehend change contracts.

Consider the following LazyMethodGen constructor.

```
public LazyMethodGen(Method m, LazyClassGen enclosingClass ) {
  ...
}
```

This constructor raises a RuntimeException if method m (i.e., the first formal parameter of the constructor) does not have its associated code for its body when this method is expected to have a body. Otherwise, an object should be created successfully. Remember that a Java method does not have its body only when it is declared as either an abstract method or a native method.

The problem of the above LazyMethodGen constructor is that a RuntimeException is raised even when the given first parameter m represents a native method. Such behavior of the constructor is buggy because a native method does not have to have body code. Thus, instead of raising a RuntimeException, the constructor should create an object successfully.

Q. Based on the above description, write a change contract for the above constructor.

(a) A question categorized as type W, AspectJ, and B

Consider the following program changes where the previous version at the top is changed to the new version at the bottom according to the change contract in the middle.

```
public class InterTypeMethodBinding extends MethodBinding { ... }
```

```
private MethodBinding /*@ new_field @*/ postDispatchMethod;
/*@ changed_behavior preserves_when !staticRef @*/
public MethodBinding getAccessMethod(/*@ new_param @*/ boolean staticRef);
```

```
public class InterTypeMethodBinding extends MethodBinding {
  public MethodBinding getAccessMethod(boolean staticRef) {
    if (staticRef) return postDispatchMethod; else return [          ] ;
  }
}
```

Q1. Explain in English what the above change contract means.
Q2. Also, fill in the blank of the new version.

(b) A question categorized as type RD (Q1), RM (Q2), AspectJ, and S

Figure 5: Survey question samples. W, RD and RM stand for Write, Read-Describe and Read-Modify, respectively. Also, B and S stand for behavioral changes and structural changes, respectively. For more details, refer to § 4.2.

(ii). Read-Describe (RD) type questions. Here, we first show a program and its change contract. We then ask respondents to describe the change contract in plain English. This type of question double-checks the comprehensibility of change contracts.

(iii). Write (W) type questions. In this type of questions, we ask respondents to write a proper change contract that they think can reflect a given verbal description of desired changes. This type of question measures how easy it is to write change contracts.

We asked thirteen (13) questions in total (excluding seven demographic questions). We asked multiple questions for each type of questions, i.e., 3 for the RM type, 5 for the RD type, and 5 for the W type. All of these questions were constructed as open questions, not as multiple-choice questions; respondents were asked, depending on the type of a question, to write down a change contract (e.g., Figure 5(a)), fill in a blank with a program statement or a program expression (e.g., **Q2** of Figure 5(b)), and write down a verbal description of a change contract (e.g., **Q1** of Figure 5(b)).

Each of these thirteen questions shows a fragment of a subject Java program. We used in total eight distinct Java program fragments; some fragments were re-used for multiple questions.

About the two third of these program fragments (i.e., 5 fragments) were carefully designed by us for this survey. Those fragments include a buggy version of a singly linked list and its extension to a doubly linked list. To measure the effectiveness to real-life programs, we also used three fragments of AspectJ that changed over consecutive versions. We asked four questions using these AspectJ fragments.

Recall that our change contract language can deal with not only behavioral changes (B-type changes) but also structural changes (S-type changes). We distributed both kinds of changes evenly throughout the questions (i.e., 6 for B-type and 7 for S-type).

Our survey questionnaire can be downloaded at the following website: `http://www.comp.nus.edu.sg/~abhik/CC-survey/SCC.htm`. In addition, the responses of the participants and a sample answer can be downloaded from the same website.

4.3 Survey Administration

We offered a single tutorial session about change contract to the survey participants before they took a mini-test two weeks later (the education materials we used for this tutorial can also be downloaded from the aforementioned website). During the test, we measured the time each student spent filling in the questionnaire. To encourage the students, we allocated 10% of credit points of the course for this survey.

While grading the answers to the RD type questions, we occasionally gave a half point when the answered verbal description about a change contract is neither entirely correct nor entirely incorrect. No partial points were given for the other types of questions.

Table 2: Distribution of *correct answer rates* depending on the criterion used to categorize questions.

Three Categorization Criteria						
Question Type			Program Source		Change Kind	
RM	RD	W	Artificial	AspectJ	B	S
100%	86%	93%	92%	92%	85%	97%

4.4 Survey Results

Table 2 shows the results of our survey with the correct answer rate for each type of questions. For the correct answer rate of question type T, we use the following formula:

$$\frac{\text{(the total sum of scores of the } T \text{ type questions)}}{\text{(the total number of the } T \text{ type questions)} \times \text{(the total number of students)}}$$

The correct answer rate is high throughout all categories, forming the overall correct answer rate at 92% – calculated using the formula, (the total sum of scores of all questions) / (13 $\times$ 16). Meanwhile, the participants spent on average 53 minutes to answer a total of 20 questions with the standard deviation being about 3 minutes. To answer each question, it took on average 2 minutes and 40 seconds. Note that we did not inform the participants that we were measuring the time.

Overall, our survey results indicate that the participants easily learned and used change contracts. In our study, the correct answer rate was not affected by whether a subject program is artificially made or extracted from a real-life program (i.e., AspectJ). Also, structural changes were more easily handled than behavioral changes were (97% vs 85%).

5. TOOL SUPPORT AND EVALUATION

Overview of Our Toolset. Figure 6 shows the workflow of our change contract checker. We check change contracts at runtime. We run a set of tests generated for the purpose of checking change contracts, and monitor the executions of the two versions of a program to see if there is any change contract violation. The two versions of a program are instrumented appropriately to support such monitoring.

Instead of running the two versions of a program in parallel as described in § 3.2, we run them in a sequential order, i.e., first the previous version, and the next the updated version while collecting information necessary to simulate the parallel-execution model of our change-contract semantics.

Our dynamic checker requires proper tests that (i) execute the target method and (ii) satisfy the update condition (see § 3.2 for its definition) of a given change contract. We call such a test that satisfies the above two conditions a *relevant test*. We provide a test generator in our toolset that can collect only relevant tests efficiently. Recall that the update condition of a change contract involves only the states of the previous version. Accordingly, our relevant-test generator considers only the previous system while ignoring the updated system.

Some of such tests generated based on the previous system may fail to be compiled in the context of the updated system if structural changes such as adding a new method parameter are made to the updated system. If this happens, those broken tests must be repaired. We, thus, provide a test repair tool in our toolset that can repair those tests using the information in a change contract.

As compared to our previous work [21], test generator and repairer are newly added. We also redeveloped our change-contract checker so that not only program states but also any legal Java expression values can be monitored. This makes it possible to check not only "ensures field==\prev(field)+1;" but also "ensures \result.equals(\old(\prev(findFileCaseInsensitive())));" where the value of findFileCaseInsensitive() should be monitored.

We now elaborate each of the three components of our toolset (i.e., dynamic change-contract checking, relevant-test generation, and test repair), and then report the experimental results.

5.1 Change Contract Checking

To support dynamic checking of change contracts, we use our custom compiler, an extension of OpenJML [16]. When we compile a Java source file, say C.java, its corresponding change contract file, C.jml, is also looked up. If that change contract exists, the resulting class file C.class is instrumented with that change contract. Recall that a change contract is satisfied if the previous and the updated versions satisfy, respectively, the update condition and the change condition of that change contract. Accordingly, we instrument the previous and the updated versions differently. For example, only at the previous version we need to store in the disk the boolean value of the update condition of a given change contract.

To align isomorphic inputs between the two versions, the two instrumented systems, when encountered with the target method during the run, convert input states (i.e., the states of parameters and the receiver) into XML graphs using XStream [24]. Such XML graphs can be viewed as object graphs, the data format we assumed for input isomorphism in § 3.2. Those two XML graphs of the previous and the updated versions are compared to check input isomorphism. We used XMLUnit [23] for this comparison.

In addition to input states, the values of \prev expressions are also stored in the disk while running the instrumented system of the previous version. Afterwards, the instrumented system of the updated version uses these pre-stored values to replace \prev expressions.

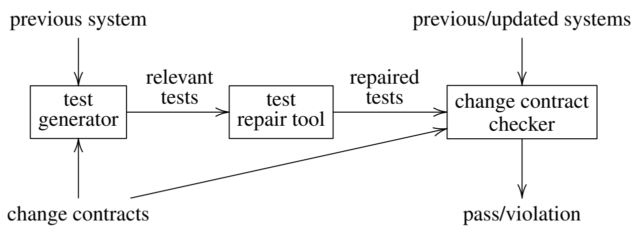


Figure 6: The workflow of our change contract checking

5.2 Test Generation

We extended a popular random test generator, Randoop [17], to collect only relevant tests. Note that whether a test is relevant is decided at runtime while Randoop is generating tests. In our initial experiment, it took too long (almost five minutes in some instance) for Randoop to start generating relevant tests. We made a couple of simple changes to Randoop to alleviate the problem.

First, our test generator selects the seed method with a 50% chance from specified target methods unlike the original Randoop that selects the seed method from all legal methods that are in the scope of the tool. As target methods, we used either (i) the target method m of a change contract if m is public or (ii) public callers of m if m is not public. Such target-method specification can be automated with the help of static analysis. The reason for assigning a 50% chance to the target methods (as opposed to assigning 100% chance) is that otherwise Randoop does not consider other method calls that may be necessary for constituting a relevant test.

The second change we made to Randoop is to address the following problem we found in our initial experiments. It took particularly long for Randoop to generate relevant tests in a case where the update condition of a change contract is satisfied only if void-type methods are called to change the program state properly before the target method is called. For example, if a target method is m2(int i), then the unmodified Randoop opts for generating a sequence that ends with "m2(var2);" preceded by a sequence of statements that ends with a statement to assign a value to variable var2, e.g., "var2=m1(var1);". This statement is again preceded by another statement to assign a value to var1. Such a style of Randoop's sequence generation tends to exclude void-type-method calls in the middle of a sequence.

To address the above issue, we intersperse a statement sequence with random void-type-method calls. We also transform statements like "var1.m1(); var2.m2();" into "var2.m1(); var2.m2();" to merge the receivers. We let such a transformation take place with an 80% chance in our experiments.

Note that generally there is no guarantee that executing a relevant test in the updated system will execute the target method with isomorphic input because only the previous version was considered when constructing relevant tests. Obviously, by considering the updated system as well, this problem can be avoided in exchange for spending more time generating each test. We make a trade-off between the time cost and the effectiveness of generated tests.

5.3 Test Repair

Consider a change contract whose target method m has different parameters in the updated version as shown in the following change contract fragment: public void m(/*@ old_param @*/ int i, /*@ new_param @*/ boolean b). Since only the previous system is looked up when generating relevant tests, those tests fail to be compiled in the updated system complaining about method signature mismatches. Our test repair tool repairs such broken tests using a change contract.

Table 3: The experiment results for our toolset

Change		Bug #	Contract size		Randoop	Test generation		Test repair		Contract checking	
Old	New		Core	Extra	T_{first} (s)	T_{first} (s)	# of tests/m	# of errors	# of fixes	# of passes	# of violations
0632cd	b6c725	51668	4	0	290	5	17	0	0	17	0
c39b90	2f95b7	50515	2	10	0.4	0.4	1	0	0	0	0
32e664	f0e466	49271	4	4	62	9	4	0	0	4	0
a84f2e	1de96b	46172	3	0	32	0.9	58	0	0	6	0
cbda11	9a0689	N/A	2	0	> 300	0.2	252	0	0	0	250
dfa59d	de3f32	N/A	5	4	> 300	1	79	0	0	0	79
5bee9d	1532f4	N/A	3	0	1	0.3	762	1239	1239	172	506
1de7b3	626f28c	N/A	2	0	5	1	183	263	263	0	183
3a1518	aef2f7	N/A	3	0	0.3	0.2	1209	1832	1832	1209	0
f87075	d17d1f	N/A	3	0	0.2	0.2	955	2	2	955	0

First, it is easy to deal with old parameters; they can simply be removed. Meanwhile, new parameters should be assigned proper values in repaired tests. Such values can be obtained from the requires clauses of change contracts. In the above example, provided that the change contract contains "requires !b;", one can infer that the value of the new parameter b should be false. In general, by using automated theorem provers, automatic inference of new parameter values should be possible in many practical cases. Currently, in our tool, only the test transformation is automated while the values for new parameters and new fields are given by a user.

5.4 Experiments and Evaluation

Table 3 shows the results of our experiments for our toolset conducted on an Intel Core i5 CPU 650 (3.2GHz $\times$ 4) processor, 4GB RAM, running Ubuntu 12.04 (32-bit) Linux. Our subject program was Ant [1], a popular tool for building Java-based systems. We chose Ant mainly because it is one of popular real-life open-source programs, and also we had basic understanding of it. The second reason is important because if one wants to write a change contract, the intended change must be understood beforehand.

Three Sources of Change Contracts. We prepared change contracts from three different sources. **(I).** First, to reflect user intentions as faithfully as possible, we transformed bug reports to change contracts as we did in the overview section (§ 2). In fact, the first row of Table 3 corresponds to the example we used in § 2. Notice the same bug number (i.e., 51668) shown in the third column. Meanwhile, the first and the second columns show the first six Git snapshot IDs of the previous and the updated systems, respectively. While the the first four rows of the table are collected by transforming bug reports, they are only partially effective in testing our toolset. Although relevant tests are successfully generated in all four cases, those tests are either passed or abandoned (isomorphic input is not found sometimes due to the limit of our tool; see § 6.2) without reporting a change contract violation. **(II).** To see the efficacy of our toolset in detecting change-contract violations, we used incorrect program changes of Ant found in our previous study [21]. These four defective cases are shown between 5th and 8th rows of the table. **(III).** Lastly, to see the efficacy of our test repair tool, we additionally collected two structural changes (method parameter additions) from Ant. The two last rows of the table correspond to these cases. Note that structural changes were also found in two defective cases.

Contract Size. In each of all the ten cases, only one change contract file is used and its size is shown under the "Contract size" column of the table. The "Core" sub-column shows the number of total clauses used in change contracts (e.g., the use of one requires clause and one when_ensured clause are counted as two), and the "Extra" sub-column the number of primitive statements used in optional auxiliary model methods (see Figure 1(d) for the example of a model method).

Results. To see the efficiency of our modified Randoop in generating relevant tests, we compare the time elapsed until the first relevant test is found during test generation (we use the notation T_{first} for this in the table) in the original and the modified Randoop's. Table 3 shows the T_{first} information in the unit of seconds (the tenths place value is also shown when the time is less than 1 second) — the first T_{first} column for the original Randoop and the next one for our modified Randoop. In all cases, our modified Randoop generated the first relevant test 1–1500 times faster than the original Randoop. In fact, in two cases, the original Randoop failed to find a relevant test within 5 minutes.

When using Randoop, its Java method pool was mainly provided through Randoop's "--classlist" option; the class for which a change contract was given was used as the main source Randoop can use to compose tests. In eight cases, we also provided one or two idiomatic statements (e.g., creating Java's SecurityManager or a sequence of statements to execute an Ant script provided in a Bugzilla report) as additional sources Randoop can use for test generation. We occasionally (in three cases) informed Randoop about a constant to use in generating tests (e.g., a string appearing in a change contract). We always used the same method/constant pools for the original and our modified Randoop.

We let our test generator collect relevant tests for one minute (the number of collected tests are shown under the "# of tests/m" column), and used those tests in checking change contracts. In all four defective cases (i.e., the 5th to 8th rows), change contract violations were successfully detected as indicated with the last column. Also, all the syntactically broken tests (i.e., the last four rows) were successfully fixed.

Our tool could generate relevant tests successfully for all 10 cases of our experiment, and, by running those tests (after repairing them if necessary), it detected change contract violations in all 4 cases of incorrect actual program changes of Ant.

6. THREATS TO VALIDITY

In this section, we outline the threats to validity of our user study and our experiment results.

6.1 User Study

As mentioned earlier, our survey was conducted with only one group of students taking a particular course of a particular university. We, however, also mentioned that our survey participants were final-year undergraduate students majoring computer science who can be considered entry-level developers.

Our survey fulfilled its purpose of gauging initial response to our change contract language; our students easily learned and used our change contract language. However, given the size of participants, a larger-scale study is necessary to confirm our results. In particular, more sophisticated study is required to see the validity of several interesting initial observations such as higher correctness rates in structural changes than in behavioral changes and little difference between the correctness rates for artificial programs and real-life programs.

6.2 Experiments

Due to the randomness of Randoop, the numbers between 6th and 12th columns of Table 3 can be varied each time an experiment is performed although in our experience the gap was not significant. In addition, those numbers are also affected by the limitation of our tool. For example, we found that XMLUnit, a tool we used to check the isomorphism between inputs, occasionally categorized isomorphic inputs as non-isomorphic due to the order-sensitiveness of the tool in comparing object graphs. Lastly, our experiment results are confined to a single subject Ant, and we need to conduct experiments with more subjects to generalize the results we obtained to other cases.

7. RELATED WORK

Design by Contract. Design by contract (DbC) [15] influenced the design of many program-level specification languages such as Eiffel [14], JML [4] and Spec# [3]. In DbC, each method has its contract typically in the form of pre and post conditions. And the contract in DbC roughly means the following two things. First, a method has to guarantee its own post-condition whenever its precondition is satisfied. Second, when a method is called, it is the caller's responsibility to guarantee the callee's pre-condition. Such a concept of a contract is different from the concept of a change contract. With a change contract, we want to capture the intended behavioral/structural changes between two program versions rather than the behavioral contracts within a single program.

Program contracts are typically checked either by extended static checking (ESC) [2, 7, 8] or runtime assertion checking (RAC) [6]. ESC checks program contracts at compile time. It first generates verification conditions from program code and accompanying program contracts. Afterwards, these verification conditions are discharged via automated theorem provers. Meanwhile, RAC checks program contracts at run time. It translates program contracts into executable assertions and weave those assertions into the program to obtain an instrumented program. Then, by running that instrumented program, violation of program contracts can be reported if one of those assertions fails during the run.

The method we used to check change contracts is on the side of RAC. While RAC is effective in reporting a contract violation without false alarms, ESC is also attractive because it can cover more error cases than RAC can without having to run a program. In future, we plan to pursue ESC as well.

Regression Testing and Debugging. Regression errors constitute an important class of errors. Traditionally, it has been interesting to select and prioritize tests from a large test suite to expose regression errors efficiently without having to test the entire test suite [5,9,22]. More recently, Jin et al. proposed a method that, given program changes, automatically generates tests that stress those program changes [12]. These tests are executed on both the previous and the updated systems, and afterwards all the observed behavioral differences between the two versions are analyzed and presented to the user. Without a specification about intended changes, however, users have to manually go through all the reported differences across program versions to validate those differences. We envision that, by combining change contracts and regression testing, those manual efforts can be significantly reduced.

Even if a regression error is found, one has to understand why that regression error took place before fixing it. In this regard, there have been efforts to debug regression errors [20, 25]. The lack of formal specifications, however, has hampered extending those research results beyond debugging regression errors. We believe that change contracts can enable debugging other types of errors related to software evolution, such as incorrect implementation of a new feature and incorrect bug fixes.

Semantic Difference Summarization. While change contracts capture intended behavioral/structural changes across program versions, there has been work to capture actual changes of program behaviors (i.e., semantic differences) given two program versions. Jackson and Ladd [10] suggested a tool that summarizes the comparison of the two sets of dependence relations between the input and output of a C program procedure of the previous version and the updated version, respectively. For example, if variable x depends on only itself in the previous version whereas it depends on another variable y in the updated version, one can guess that program behavior around x would be different between those two versions. More recently, Person et al. [19] exploited symbolic execution to compare program behaviors of the two versions, and as a result could provide more accurate functional input-output relations of each version than mere dependence relations. SymDiff [13] can also do the same, but under the hood, it generates verification conditions and passes them to an SMT solver.

We believe that comparing these two kinds of changes, i.e., (i) actual program changes provided by the aforementioned tools and (ii) intended program changes provided through change contracts can help with debugging evolving programs.

8. DISCUSSION AND FUTURE WORK

In this paper, we have followed the thesis that program changes can be easily expressed through change contracts. Writing such change contracts is often easier and also more intuitive than writing program contracts not only because one can directly focus on changes, but also because of the additional flexibility of change contracts to express intended output-output relationship across program versions as well as conventional input-output relationship. Our user study also indicates positively that change contracts can be easily learned and used by entry-level developers.

We have also presented that change contracts can be checked through targeted test generation and runtime checking of instrumented programs. We also showed the possibility of using change contracts for test repair which we plan to refine in the future.

9. ACKNOWLEDGEMENTS

This work was partially supported by Singapore Ministry of Education research grant MOE2010-T2-2-073.

10. REFERENCES

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

[2] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects*, pages 364–387, 2006.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of the 2004 Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69, 2004.

[4] 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.

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

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

[7] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Proceedings of the 2004 International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 108–128, 2004.

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

[9] R. Gupta, M. Harrold, and M. Soffa. An approach to regression testing using slicing. In *Proceedings of the 1992 Conference on Software Maintenance*, pages 299–308, 1992.

[10] D. Jackson and D. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the '94 International Conference on Software Maintenance*, pages 243–252, 1994.

[11] Eclipse JDT. `http://www.eclipse.org/jdt/`.

[12] W. Jin, A. Orso, and T. Xie. Automated behavioral regression testing. In *Proceedings of 2010 International Conference on Software Testing, Verification and Validation*, pages 137–146, 2010.

[13] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *CAV*, pages 712–717, 2012.

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

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

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

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

[18] D. Parnas. Precise documentation: The key to better software. In *The Future of Software Engineering*, pages 125–148. Springer, 2011.

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

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

[21] D. Qi, J. Yi, and A. Roychoudhury. Software change contracts. In *FSE*, pages 22:1–22:4, 2012.

[22] 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.

[23] XMLUnit. `http://xmlunit.sourceforge.net/`.

[24] XStream. `http://xstream.codehaus.org/`.

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