A

Software Change Contracts

Jooyong Yi, Dawei Qi, Shin Hwei Tan, Abhik Roychoudhury, National University of Singapore

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 develop both dynamic and static checkers for change contracts, and show how they can be used in maintaining software changes. We use our dynamic checker to automatically suggest tests that manifest violations of change contracts. Meanwhile, we use our static checker to verify that a program is changed as specified in its change contract. Apart from verification, our static checker also performs various other software engineering tasks, such as localizing the buggy method, detecting/debugging regression errors, and classifying the cause for a test failure as either error in production code or error in test code.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Programming by contract*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Languages, Reliability, Theory, Verification

Additional Key Words and Phrases: Software changes, dynamic checking, static checking

ACM Reference Format:
Jooyong Yi, Dawei Qi, Shin Hwei Tan, and Abhik Roychoudhury, 2014. Software Change Contracts. *ACM Trans. Softw. Eng. Methodol.* V, N, Article A (January YYYY), 44 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

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.

An earlier version of this article [Yi et al. 2013] was published in ISSTA 2013. Section 7 of this article is an original extension of this article. This work is partially supported by Singapore Ministry of Education research grant MOE2010-T2-2-073 and T1 251RES1314. Author's addresses: School of Computing, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117417; email: {jooyong,dawei,shinhwei,abhik}@comp.nus.edu.sg.

DOI: http://dx.doi.org/10.1145/0000000.0000000

Program contracts, or Design by Contract programming [Meyer 1997; Burdy et al. 2005; Barnett et al. 2004] 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 [Parnas 2011].

In our previous paper [Qi et al. 2012], 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 throughout this article, such descriptions are likely to be more concise than a usual program contract of the following form:

$$\textit{whenever } \varphi(in) \textit{ holds, } out' == f(in)$$

where $\varphi(in)$ is a constraint on the input, and the function application $f(in)$ expresses the intended output of the changed program version as a function of input in. Unlike in our change contract where the outputs of two versions are compared to each other, a program contract does not reveal changes explicitly. Also, in the above program contract, 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 article, we study the expressivity/usability of our change contract language via a detailed user survey as well as by developing change-contract based infrastructures to help debug change-related errors or verify the absence of such errors. The contributions in this article are now stated in the following paragraphs.

Our change contract language is built on top of the Java Modeling Language (JML) [Burdy et al. 2005]. 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 preconditions. Exceptional behavior, as well as structural changes (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 Section 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 checkers for change contracts and show how they can be used in maintaining software changes. We develop both dynamic and static checkers in this article. Our dynamic checker monitors executables, whereas our static checker analyzes source code to check change contracts. As usual in program analysis, both are complementary to each other.

We use our dynamic checker to suggest tests, each of whose execution leads to a violation of a given change contract. To do so, we first modify Randoop [Pacheco and Ernst 2007] and apply it to a previous-version program to generate tests, each of whose execution leads to a program state required to be changed according to a given change contract. Afterwards, we run those generated tests against the updated version to monitor whether the updated version behaves as specified in a given change contract. We also provide tool support for repairing tests which are broken due to structural changes across program versions (e.g., a new method parameter can be added in an updated version). We present experimental evaluation results summarizing the size of the change contracts, time taken to generate tests, and whether change contract violation (if any) is detected. All the results are obtained from the well-known software project *Ant* [1], a Java library to build Java applications. The experiments point to the efficacy of our dynamic checker in detecting the violations of change contracts.

Meanwhile, we use our static checker to verify that a program is changed as intended (i.e., as specified in its change contracts). To do so, we customize the existing automated program verification technique such as ESC/Java [Flanagan et al. 2002]. For scalability, we support modular checking – i.e., when encountered with a method call, our modular checker interprets the change contract of that callee without looking into its body (callees that do not change across versions are deemed to have implicit change contracts that specify no change). Although modular checking is the norm in program verification, it is *not* trivial to support modular checking with change contracts. Interpreting a change contract is significantly different from interpreting a program contract. The conventional *simple modular rule* to interpret a program contract – asserting the precondition of a callee followed by assuming the postcondition of a callee – cannot be used for a change contract. To see this, consider again the change relationship, "whenever $out > 0$ holds, $out' == out+1$," as the change contract of a callee. Depending on whether out of the previous version is positive or not, out' of the updated version should either increase by one (if $out > 0$) or remain the same as before. This additional version-related context calls for an *alternative modular rule* that can interpret a change contract correctly. We introduce in this article an alternative modular rule for change contracts. We also show how we enforce that alternative rule in our static checker.

Our static checker is an extension of OpenJML [Cok 2014]. We present experimental evaluation results summarizing the size of the change contracts, time taken to verify change contracts, and whether change contract can be verified. All the results are obtained from *Joda-Time*,[2] an open-source date/time library for Java. The experiments point to the efficacy of our static checker in verifying change contracts at reasonable cost – verification takes on average 7.4 seconds, and we wrote on average, 2.7 lines of change contracts for methods that change across versions.

[1] http://ant.apache.org/
[2] http://www.joda.org/joda-time/

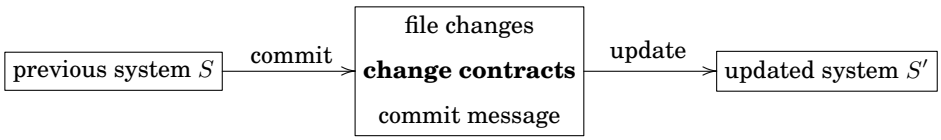

Fig. 1: Change contracts are to be maintained in a version control system along with file changes and commit message.

Finally, we also report various usage of our static checker. Using our static checker, we perform not only verification, but also other tasks, such as (1) localizing the buggy method, (2) detecting/debugging a regression error, and (3) classifying the cause for a test failure as either error in production code or error in test code. The change contracts we used to evaluate our dynamic/static checker are available at the following web site: http://www.comp.nus.edu.sg/~abhik/CC-survey/SCC.htm.

2. OVERVIEW

Figure 1 shows how change contracts are to be configured in the history of a software system. Change contracts are to be maintained in a version control system (VCS) such as Git or Mercurial. When a user commits changes to a VCS, not only file changes and commit messages are stored in VCS as usual, but also change contracts can be stored in a VCS at the same time. While file changes represent actual code changes, change contracts capture the underlying intended changes.

Figure 2(b) shows an example of a change contract for the execute method of software Ant. 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 Section3 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 2(b) is the contents of a contract file XMLResultAggregator.scc, and it describes behavioral changes between two consecutive versions of Java file XMLResultAggregator.java.

The change contract of Figure 2(b) is a counterpart of a verbal description given in a bug report of Figure 2(a). 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

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>.

(a) A sample Bugzilla report for software Ant

```
// file: XMLResultAggregator.scc
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;
}
```

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

```
// file: SourceTypeBinding.scc
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);

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

(c) A core-developer-level change contract

Fig. 2: The examples of change contracts

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

```
1  public class DirectoryScanner implements FileScanner {
2
3    private /*@ new_field @*/ int mode;
4
5    // If !cs at the entry of the method, the behavior of the method changes.
6    // If cs at the entry of the method, the behavior of the method is preserved.
7    /*@ changed_behavior
8      @  when_required true;
9      @  requires !cs;
10     @  ensures /* omitted: description about behavioral changes */;
11     @  preserves_when cs;
12     @*/
13   File findFile(File base, String path, /*@ old_param @*/ int mode, /*@ new_param @*/ boolean cs);
14 }
```

Fig. 3: DirectoryScanner.scc: a change contract involving structural changes such as adding/removing a parameter/field

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 2(c) shows such a low-level change contract for the resolveTypesFor method of Eclipse JDT (Java Development Tools) [3]. 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 2(c) 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 pure model method, isNonNull, in Figure 2(c) to improve the readability of a change contract. A pure 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 pure model method is described between "/*@ pure model" and "@*/". It is often handy to define a pure 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 2(b) and Figure 2(c) 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 2(b), 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

[3] http://www.eclipse.org/jdt/

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/removing a new method parameter/field. Figure 3 shows such an example. In line 13, the removal of a parameter mode and the addition of a parameter cs are described with modifiers /*@ old_param @*/ and /*@ new_param @*/, respectively. Similarly, the /*@ new_field @*/ modifier in line 3 describes the addition of a new field. Also, the "preserves_when cs" clause in line 11 expresses the expectation that, when cs is true, the updated version of findFile should find and return the same file as in the previous version, given the same base and path. Meanwhile, the behavioral changes expected to be made when cs is false (i.e., requires !cs in line 9) is also described in the ensures clause in line 10. Lastly, the when_required clause is used in line 8 to describe the input condition of the previous version, because the input condition of the updated version – requires !cs – cannot be shared in the previous version; notice that cs does not exist in the previous version.

3. CHANGE CONTRACT LANGUAGE

To express intended program changes, we extend a subset of JML [Burdy et al. 2005], 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.

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.scc) separated from Java files.

3.1. Syntax

Figure 4(a) 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 precondition 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.

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 (*reference-type* [*ident*]) *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**

(a) The grammar of our change contract language, which is an extension of a JML subset; standard regular expression notation $*$ is used.

```
// the full change contract, (φ, ψ, θ; φ′, ψ′, θ′)
/*@ changed_behavior
  @  when_required φ;  when_ensured ψ;  when_signaled (T1 x) θ;
  @  requires φ′;      ensures ψ′;      signals (T2 x) θ′;
  @*/
```

(b) A boilerplate for the full change contract; the greek letters denote predicates, and T_1 and T_2 represent exception types.

Fig. 4: Our change contract language

For simplicity, we often use a shorthand notation $(\varphi, \psi, \theta; \varphi', \psi', \theta')$ to mean the full change contract shown in Figure 4(b).

In the above, greek letters (i.e., φ, ψ, θ, and their primed variants) denote predicates, and T_1 and T_2 represent exception types (i.e., subtypes of java.lang.Exception). Also, variable x, which refers to the exception of type T_1 (or T_2) signaled when the previous (or updated) version of the method exits, can appear in θ (or θ'). One can consider x as a quantified variable associated with quantifier when_signals (or signals). Thus, variable capture should be avoided in θ and θ'. 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 detailed in Section 3.2.4.

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.

Our change contract language can also express structural changes such as addition/removal of parameters/fields. As shown in Figure 3, a new parameter can be recognized by the /*@ new_param @*/ modifier. Conversely, a removed parameter is annotated with the /*@ old_param @*/ modifier.

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.

Lastly, clause "preserves_when φ" is a syntactic sugar for the following combination of clauses that dictate that if φ holds at the entry of the updated version, then the output should be preserved across versions:

```
/*@ changed_behavior
  @  when_required true;
  @  requires φ;
  @  ensure out == \prev(out);
  @  signals (Exception e) \typeof(e)==\typeof(\prev(e)) && out == \prev(out);
  @*/
```

In the above, out denotes method output such as the return value of the method.

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 5(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, there may be structural changes such as addition of a method parameter or a field.

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 as follows. Note that a program state consists of a store σ and a heap h.

DEFINITION 1 (ISOMORPHIC PROGRAM STATES). *Two program states,* (σ_1, h_1) *and* (σ_2, h_2)*, are considered* isomorphic *to each other, if for all variables* x *that commonly exist 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. We denote the fact that two program states,* (σ_1, h_1) *and* (σ_2, h_2)*, are isomorphic to each other with notation* $(\sigma_1, h_1) \approx (\sigma_2, h_2)$*. As usual in Java programs (and other object-oriented programs), the receiver of an object (i.e.,* this*) is considered an implicit parameter of a non-static method, and thus* this *is in the domain of* σ_1 *and* σ_2*.*

Note that in Definition 1, heaps (h_1 and h_2) are consulted if necessary when constructing object graphs. As in variables, only the fields that commonly exist in h_1 and h_2 are compared to each other. This resolves the second issue about structural changes; when comparing object graphs, we exclude method parameters and fields that are not in common between two versions. Notice that our overall semantic rule in Figure 5(b) has $(\sigma_1, h_1) \approx (\sigma_2, h_2)$ in its premise to force isomorphic inputs.

Our execution model based on isomorphic program states imposes one restriction. A change contract should not contain an expression such as \prev(this)==this that compares the reference of the previous version with the reference of the updated version, because the reference value of a non-primitive variable will be different at each version. For the same reason, an expression such as \prev(o.hashCode()) == o.hashCode() should be avoided. In fact, programmers usually do not expect reference values to be preserved across versions, when making changes to their programs.

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 5(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 5(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

$$c \in \mathbf{Cmd} \quad v \in \mathbf{Value} \stackrel{\text{def}}{=} \mathbf{Location} \cup \ldots$$
$$\sigma \in \mathbf{Store} \stackrel{\text{def}}{=} \mathbf{Variable} \stackrel{\text{fin}}{\to} \mathbf{Value} \qquad h \in \mathbf{Heap} \stackrel{\text{def}}{=} \mathbf{Location} \stackrel{\text{fin}}{\to} (\mathbf{Field} \stackrel{\text{fin}}{\to} \mathbf{Value})$$

(a) Semantic domains

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

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

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

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

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

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

(c) Semantic rules for $\backslash\mathsf{prev}(E)$ expressions; for explanation, refer to Section 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 Section 3.2.3.

Fig. 5: 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.

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/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 5(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 inference 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 con-

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

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

dition 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.

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. All clauses of a change contract do not have to be specified, as mentioned in Section 3.1. Default predicates are used for omitted clauses, following the rule described in Table I.

If an ensures (or a signals) clause is omitted, ensures true (or signals true) is used by default. To understand why true is used as a default clause for an ensures clause, recall that given a full change contract $(\mathsf{true}, \psi, \theta; \mathsf{true}, \psi', \theta')$,[4] our change contract inference rule in Figure 5(d) checks whether change condition $\psi' \wedge \theta'$ holds at the end of the updated version, whenever update condition $\psi \vee \theta$ holds at the end of the previous

[4]For simplicity of the description, we give true to the when_required and requires clauses.

version. When omitting the predicate ψ' of the ensures clause, and only the predicate θ' of the signals clause is used, the change condition we want to check is true $\wedge\, \theta'$. Thus, the omitted predicate ψ' should be true. Similarly, the default predicate of an omitted signals clause should be true.

Meanwhile, when omitting the predicate ψ of the when_ensured clause, the default value of ψ seems to be false at first, considering the update condition $\psi \vee \theta$. However, the problem is one can omit both when_ensured and when_signaled clauses simultaneously, as in the following change contract: ensures \result==\prev(\result)+1, which describes the expectation that the return value of the updated version should be one larger than the return value of the previous version. If this is the case, the use of false as a default predicate makes the update condition false, and as a result, the change condition is not checked. To avoid such situations, we assign a default predicate differently depending on the context. If it is the case that only either of the when_ensured or when_signaled clause is omitted, we use false as a default predicate for the omitted clause. However, if both when_ensured and when_signaled clauses are omitted, we use true as a default predicate instead.

Lastly, let us explain the case for when_required. 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.

3.3. Discussion

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, inequality between two output states indicates a regression bug. As an example, consider the following change contract that specifies NullPointerException signaled in the previous version should disappear in the updated version:

```
/*@ changed_behavior
  @  when_signaled (NullPointerException e) true;
  @  signals (NullPointerException e) false;
  @*/
   int m(int p);
```

If the previous version method m does not signal a NullPointerException under the input state S_{in}, and instead terminates normally with an integer return value r, then the above change contract implicitly specifies that the updated version should return the same return value r when the same input state S_{in} is given. Two different return values returned from two versions under the same input state indicate that there is a regression bug. Without a change contract, it is difficult to distinguish a regression bug from software progression even if inequality between two output states is found.

Structural changes often involve conditional refactoring – the behavior of a method should be preserved under a certain condition. For example, Figure 3 describes that the behavior of method findFile should be preserved if the newly added parameter cs has value true when the method is called, as described with preserves_when cs.

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.

Consider the following LazyMethodGen constructor.

```
1  public LazyMethodGen(Method m, LazyClassGen enclosingClass ) {
2   ...
3  }
```

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.

```
1  public class InterTypeMethodBinding extends MethodBinding {
2    private MethodBinding syntheticMethod;
3    public MethodBinding getAccessMethod () {
4      return syntheticMethod;
5    }
6    /* the rest of the code is omitted */
7  }
```

```
1  private MethodBinding /*@ new_field @*/ postDispatchMethod;
2  /*@ changed_behavior
3    @ preserves_when !staticRef;
4    @*/
5  public MethodBinding getAccessMethod(/*@ new_param @*/ boolean staticRef);
```

```
1  public class InterTypeMethodBinding extends MethodBinding {
2    public MethodBinding getAccessMethod(boolean staticRef) {
3      if (staticRef) return postDispatchMethod;
4      else return [          ] ;
5    }
6  }
```

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

Fig. 6: 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.

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 6 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.

(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 6(a)), fill in a blank with a program statement or a program expression (e.g., **Q2** of Figure 6(b)), and write down a verbal description of a change contract (e.g., **Q1** of Figure 6(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 an open-book 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.

4.4. Survey Results

Table II 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:

Table II: 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%

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

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%).

4.5. Threats to Validity

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 number 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.

5. FORMULATION OF CHANGE CONTRACT CHECKING (CCC)

Our change contract, as a formal description of software changes, is checkable – in an automatic way. If there is any discrepancy between a given change contract and actual code changes, we report a violation of the given change contract. Furthermore, we provide an explanation about why such a violation happens. Such an explanation can be a test case that enables a user to observe a change contract violation. We can also more directly show a counterexample path (sequence of statements) that leads to a change contract violation.

Change contract checking can be performed either dynamically – by running executables – or statically – by analyzing source code. We in this article describe the both. As well-known, dynamic and static checking have their own advantages and disadvantages. In general, static checking can guarantee the absence of contract violations with higher confidence than dynamic checking, when no contract violation is found.

Meanwhile, dynamic checking seldom sets off a false alarm, whereas a false alarm is one of the key problems of static checking. Those advantages and disadvantages of dynamic/static checking are also inherited when checking a change contract.

However, we can mitigate the disadvantage of each analysis by exploiting the fact that we deal with two versions of software, as will be described in detail in the following subsections. Beforehand, we first formally describe the problem of change contract checking.

Problem Definition. Before we develop dynamic/static checking for change contracts, we first formally define the problem of change contract checking (CCC) for the following full-blown (i.e. without omitted clauses) change contract, $(\varphi, \psi, \theta; \varphi', \psi', \theta')$:

```
1  when_required φ; when_ensured ψ;  when_signaled (T x) θ;
2  requires φ'; ensures ψ'; signals (T' x) θ';
```

The meaning of each clause will be described shortly through the definition of CCC. Since we are primarily interested in behavioral changes, we first define the behavior of a deterministic method:

DEFINITION 2 (BEHAVIOR). *Given a deterministic method m whose method body is a command c, we define the behavior of m (notated with $B[m]$) as the following possibly infinite set of relations between an input state S_{in} and an output state S_{out} :*

$$B[m] = \{(S_{in}, S_{out}) \mid \langle c, S_{in}\rangle \Downarrow_c S_{out}\}$$

In the above, $\langle c, S_{in}\rangle \Downarrow_c S_{out}$ refers to a semantic reduction that indicates that the method body c reduces to the output state S_{out} when it starts with the input state S_{in}.

We will also use the following notations. $S \models \varphi$ means that predicate φ is satisfied at state S. Using dedicated predicate ex to denote an exception, $S_{out} \models ex$ and $S_{out} \models \neg ex$ mean that an exception is thrown and not thrown in state S_{out}, respectively. Also, $S_{out} \models ((\neg ex \Rightarrow \psi) \vee (ex \Rightarrow \theta))$ means $(S_{out} \models \neg ex \Rightarrow \psi) \vee (S_{out} \models ex \Rightarrow \theta)$. Finally, we use m_v1 and m_v2, respectively, to refer to the method m at the previous (v1) and the updated version (v2). We define CCC as follows:

DEFINITION 3 (CCC). *Given a full-blown change contract $(\varphi, \psi, \theta; \varphi', \psi', \theta')$ of a method m, we say that CCC succeeds in m iff the following two properties hold. For all $(S_{in}, S_{out}) \in B[m\_v1]$ and $(S'_{in}, S'_{out}) \in B[m\_v2]$,*

$$\begin{aligned}(P1)\ & S_{in} \approx S'_{in} \wedge (S_{in} \models \varphi \wedge S_{out} \models ((\neg ex \Rightarrow \psi) \vee (ex \Rightarrow \theta)))\\ & \quad \Rightarrow (S'_{in} \models \varphi' \Rightarrow S'_{out} \models ((\neg ex \Rightarrow \psi') \wedge (ex \Rightarrow \theta')))\\ (P2)\ & S_{in} \approx S'_{in} \wedge \neg(S_{in} \models \varphi \wedge S_{out} \models ((\neg ex \Rightarrow \psi) \vee (ex \Rightarrow \theta)))\\ & \quad \Rightarrow S_{out} \approx S'_{out}\end{aligned}$$

CCC essentially compares the output states, S_{out} and S'_{out}, whenever their corresponding input states, S_{in} and S'_{in}, are isomorphic to each other (i.e., $S_{in} \approx S'_{in}$).[5] There can be two possibilities; the behavior of a method either changes (P1), or remains the same (P2). The premise of P1 describes the condition in which the method should change its behavior. In particular, its second conjunct describes which pattern of the behavior of m_v1 triggers behavioral changes in m_v2. That is, m_v2 should have a different behavior only if (1) m_v1 satisfies the when_required clause at its entry (i.e., $S_{in} \models \varphi$),

[5]The definition of isomorphic program states is provide in Definition 1.

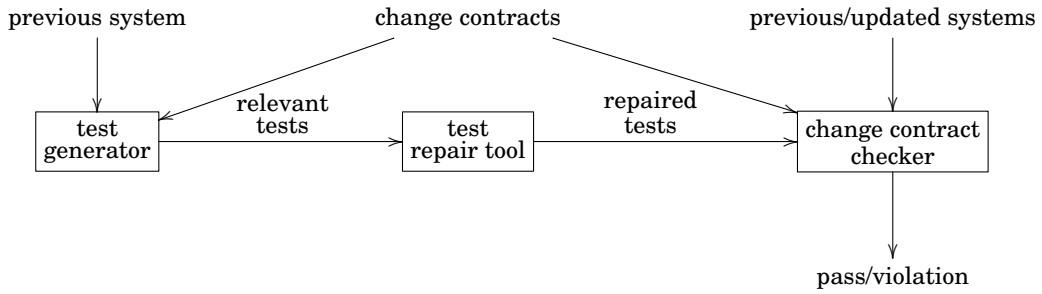

Fig. 7: The workflow of our dynamic CCC

and (2) m_v1 also satisfies either the when_ensured clause or the when_signaled clause at its exit (i.e., $S_{out} \models ((\neg ex \Rightarrow \psi) \vee (ex \Rightarrow \theta))$). When these two conditions are evaluated true, we say that the *update condition* of a change contract holds.

In case the same input is used at both versions and the update condition holds, m_v2 should satisfy the condition described in the conclusion of P1; that is, if the requires clause is satisfied at its entry (i.e., $S'_{in} \models \varphi'$), then the ensures (signals) clause should also be satisfied at its normal (abnormal) exit (i.e., $S'_{out} \models ((\neg ex \Rightarrow \psi') \wedge (ex \Rightarrow \theta'))$). When this is true, we say that the *change condition* of a change contract holds.

When P1 cannot be applied, P2 should hold instead. Notice in the premise of P2 that the update condition is negated. As mentioned, P2 describes the behavioral preservation of a method. Thus, the conclusion of P2 is $S_{out} \approx S'_{out}$.

We note that when_required and requires clauses usually have the same predicate (i.e., $\varphi = \varphi'$), considering that $S_{in} \approx S'_{in}$. It is only in some special cases (e.g., some parameters exist only in one version) that one needs to constrain the input differently depending on the version. In the remaining sections, we use only requires clause – the omitted when_required clause is assumed to have the same predicate as the requires clause.

6. DYNAMIC CHANGE CONTRACT CHECKING (DYNAMIC CCC)

Figure 7 shows the workflow of our dynamic CCC. Our dynamic checker runs a set of tests generated for the purpose of checking change contracts, and monitors 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 Section 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. Note that the tests we use to run the program are generated at the unit (method) level. These tests run the methods, the behavioral changes of which are described in given change contracts.

Our dynamic CCC starts with generating tests satisfying the following two conditions: (i) a test executes the target method and (ii) when the target method exits, the update condition of a given change contract holds true (recall that if the update condition holds, the target method is expected to change its behavior). We call such a test that satisfies the above two conditions a *relevant test*. We provide a test generator in our dynamic CCC toolset that can collect only relevant tests efficiently. Recall that the update condition of a change contract involves only the states of the previous ver-

sion. 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.

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

6.1. Change Contract Checking

To support dynamic checking of change contracts, we use our custom compiler, an extension of OpenJML [Cok 2014]. When we compile a Java source file, say C.java, its corresponding change contract file, C.scc, 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 [6]. Such XML graphs can be viewed as object graphs, the data format we assumed for input isomorphism in Section 3.2. Those two XML graphs of the previous and the updated versions are compared to check input isomorphism. We used XMLUnit [7] 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.

6.2. Test Generation

We extended a popular random test generator, Randoop [Pacheco and Ernst 2007], 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

[6] http://xstream.codehaus.org/
[7] http://xmlunit.sourceforge.net/

Table III: The subject changes of software Ant for our experiment; we extract change contracts from these changes.

Change		Bug #	Contract size	
Old	New		Core	Extra
0632cd	b6c725	51668	4	0
c39b90	2f95b7	50515	2	10
32e664	f0e466	49271	4	4
a84f2e	1de96b	46172	3	0
cbda11	9a0689	N/A	2	0
dfa59d	de3f32	N/A	5	4
5bee9d	1532f4	N/A	3	0
1de7b3	626f28c	N/A	2	0
3a1518	aef2f7	N/A	3	0
f87075	d17d1f	N/A	3	0

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.

6.3. 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.

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 a new parameter value should be possible, if the type of that new parameter is primitive. For a non-primitive type parameter, it should be possible to use Randoop again and select a value that satisfies the requires clause of a given change contract. Currently, in our tool, only the test transformation is automated while the values for new parameters and new fields are given by a user.

Table IV: The experiment results for our dynamic CCC

Change		Randoop	Test generation		Test repair		Contract checking	
Old	New	T_{first} (s)	T_{first} (s)	# of tests/m	# of errors	# of fixes	# of passes	# of violations
0632cd	b6c725	290	5	17	0	0	17	0
c39b90	2f95b7	0.4	0.4	1	0	0	0	0
32e66f	f0e466	62	9	4	0	0	4	0
a84f2e	1de96b	32	0.9	58	0	0	6	0
cbda11	9a0689	> 300	0.2	252	0	0	0	250
dfa59d	de3f32	> 300	1	79	0	0	0	79
5bee9d	1532f4	1	0.3	762	1239	1239	172	506
1de7b3	626f28c	5	1	183	263	263	0	183
3a1518	aef2f7	0.3	0.2	1209	1832	1832	1209	0
f87075	d17d1f	0.2	0.2	955	2	2	955	0

6.4. Experiments and Evaluation of Dynamic CCC

We perform our experiments for our dynamic CCC 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 [8], 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.

6.4.1. Three Sources of Change Contracts. Table III shows ten version changes from which we extract 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 (Section 2). In fact, the first row of Table III corresponds to the example we used in Section 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 dynamic CCC 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 Section 6.4.4) 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 [Qi et al. 2012]. 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.

6.4.2. Contract Size. In each of the ten cases, only one change contract file is used and its size is shown under the "Contract size" column of Table III. 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 2(c) for the example of a model method).

6.4.3. 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

[8] http://ant.apache.org/

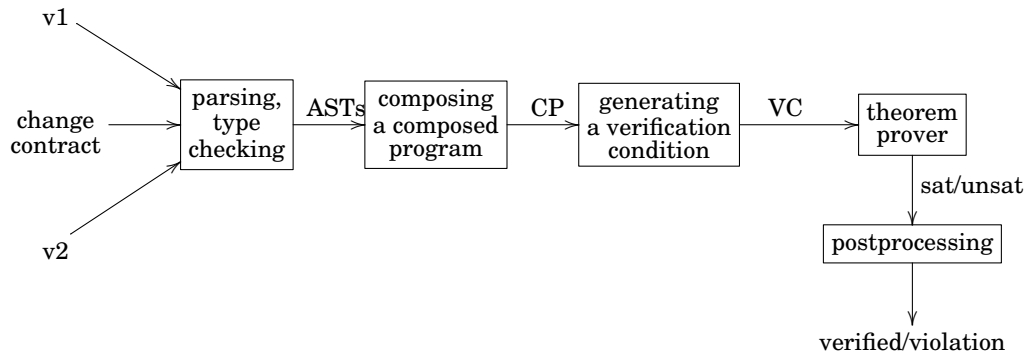

Fig. 8: The workflow of our static CCC

modified Randoop's. Table IV 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.

6.4.4. Threats to Validity. Due to the randomness of Randoop, the numbers in 3rd-to-last columns of Table IV 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 limitations 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 experimental 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. STATIC CHANGE CONTRACT CHECKING (STATIC CCC)

To perform static CCC, we customize the standard approach of automated program verification. Figure 8 shows the overall flow of our approach. Given two versions of source code we want to compare – we denote them with v1 and v2 – and their change contract, we compose a composed program (CP) by manipulating the ASTs (Abstract

Syntax Trees) of the given source code and change contract. A CP implements the core logic of CCC. More specifically, it interprets both v1 and v2 and compares their output to check whether observed behavioral changes coincide with the changes specified in a given change contract.

Conceptually, this composed program CP is interpreted "symbolically" with symbolic input to v1 and v2. We perform such symbolic interpretation of a CP via a theorem prover. To achieve that, we transform a CP into a verification condition (VC), which is a logical formula a theorem prover can understand. If a theorem prover finds that this VC is *invalid*, then this means that there exists an input to the program under consideration that leads to a change contract violation. On the contrary, if the VC is *valid*, we conclude that program changes are verified against a given change contract.

In the rest of this section, we describe each step of the workflow in more detail. We focus on the unique features of static CCC, leaving out the standard procedures such as parsing and type checking. Throughout this section, we assume there is no structural changes such as method name changes – we touch on structural changes at the last part of this section.

7.1. Programming Language

For efficiency of description, we describe our static CCC on the following minimal programming language. Note that our static CCC supports Java programs, and we later describe Java-specific issues.

$$z \in \mathbb{Z} \quad x \in \textbf{Variable} \quad E \in \textbf{Expression} \quad \textbf{Stmt} \in \textbf{Statement} \quad p \in \textbf{Procedure-Name}$$

$$E ::= z \mid x \mid E^b$$

$$E^b ::= \mathsf{true} \mid \mathsf{false} \mid E{=}{=}E \mid E > E \mid !E^b \mid E^b \;\&\&\; E^b \mid E^b \mid\mid E^b \mid \mathsf{call}\ p(E)$$

$$\textbf{Stmt} ::= x{=}E \mid \mathsf{return}\ x \mid \textbf{Stmt}; \textbf{Stmt} \mid \mathsf{if}\ E^b\ \mathsf{then}\ \textbf{Stmt}\ \mathsf{else}\ \textbf{Stmt} \mid \mathsf{while}\ E^b\ \mathsf{do}\ \textbf{Stmt}$$

Our minimal language is a typical imperative procedural language that can manipulate integers and booleans. Pointers are not part of our minimal language. However, our language supports a procedure call with a call expression, call $p(E)$. Expression call $p(E)$ invokes procedure p with call-by-value semantics, and returns a value of the ending return statement of p.

To further simplify discourse, we assume that (1) a procedure takes only one argument, (2) a procedure always returns a return value, (3) there are no global variables, and (4) a procedure call is deterministic and side-effect-free. These artificial assumptions are only for simplicity, and they are unnecessary for supporting static CCC. In Section 7.7, we describe how we handle the extended features of the Java programming language such as throwing exceptions and modifying fields.

7.2. Composing a CP (Composed Program)

To relate the two versions of a program (v1 and v2) to their change contract, we compose a composed program (CP) that incorporates v1 and v2 and their change contract. We express a CP using an extended programming language that additionally has, for example, assume/assert statements.

Figure 9(b) shows the high-level structure of a composed program (CP), given two versions v1, v2, and their change contract (see Figure 9(a)). We assume that the body of procedure p changes from $body_1$ to $body_2$. The given change contract says whenever the input of procedure p satisfies φ and the output of p satisfies ψ at v1, v2's output is expected to satisfy ψ'.

```
// the previous version (v1)
int p(int x) {
  body_1
}
```

```
/*@ changed_behavior
  @  requires φ;
  @  when_ensured ψ;
  @  ensures ψ';
  @*/
int p(int x);
```

```
// the updated version (v2)
int p(int x) {
  body_2
}
```

(a) The two versions of procedure p and their change contract in the middle

```
/***** Part I: assume (1) isomorphic input and (2) the requires clause *****/
assume x_v1 == x_v2; // parameters should be isomorphic
boolean requires_clause = [[φ]]; // store the value of the requires clause

/***** Part II: interpret v1 to see if the update condition is true *****/
boolean update_condition = false; // the update condition is initially false.
int result_v1; // the variable to hold the return value of m at v1
result_v1 = [[body_1]]; // interpret body_1 and store the return value at result_v1

// set the update condition true if the when_ensured clause is true.
boolean when_ensured_clause = [[ψ]];
if(requires_clause && when_ensured_clause) {
  update_condition = true;
}

/***** Part III: interpret v2 to see if there is any change contract violation *****/
int result_v2; // the variable to hold the return value of m at v2
result_v2 = [[body_2]]; // interpret body_2 and store the return value at result_v2

if(update_condition) {
  // we expect the ensures clause to be true
  boolean ensures_clause = [[ψ']];
  assert ensures_clause;
} else {
  // we expect no change
  assert result_v1==result_v2;
}
```

(b) The composed program (CP) for the two versions of a procedure and their change contract shown in (a).

Fig. 9: The high-level structure of a composed program (CP)

A CP consists of three parts. Part I establishes the basic assumption of CCC: the inputs to v1 and v2 are isomorphic to each other. To force the procedure parameter x to have the same value at v1 and v2, a CP has a statement "assume x_v1==x_v2". We use suffixes _v1 and _v2 to distinguish between the variables of different versions. At part I, we also evaluate the requires clause φ of the given change contract – the common input condition for v1 and v2 –, and store the evaluated value before the program state changes at subsequent parts.

Afterwards, part II and part III interpret v1 and v2, respectively. In the figure, high-level notations $[\![body_1]\!]$ and $[\![body_2]\!]$ denote the interpretation of $body_1$ and $body_2$, respectively. The main task of part II is to check whether the update condition of a given change contract holds. Recall that we expect behavioral changes between the two versions, only when the update condition holds. In our running example, the update condition holds when the requires clause φ and the when_ensured clause ψ are satisfied.

Meanwhile, part III compares the outputs of v1 and v2. If the update condition holds, we check whether the expected new output condition – the ensures clause ψ' in our example – holds true, using an assert statement. Conversely, if the update condition does not hold – that is, this is the case no behavioral change is expected –, we check the equivalence of the return values of v1 and v2.

To interpret $body_1$ and $body_2$ in a CP, we transform each statement in those bodies following the standard procedure used in automated program verification. One exception is procedure calls, because they require significantly different handling than in conventional program verification, due to the differences between change contracts and program contracts. In the subsequent subsection, we explain those differences and describe how we handle procedure calls.

Assuming that $body_1$ and $body_2$ are interpreted correctly, our CP is sound in the following sense:

THEOREM 7.1 (SOUNDNESS OF CP). *If our composed program CP is correct (i.e., no assertion error is possible), then CCC (see Definition 3) succeeds.*

PROOF. Part I of CP establishes $S_{in} \approx S'_{in}$. Continuously, part II establishes either (1) $S_{in} \models \varphi \wedge S_{out} \models \psi$ – in which case, update_condition is true– or (2) its negation. Therefore, the premises of (P1) and (P2) of CCC are established by interpreting part I and part II. (Full-fledged premises as in Definition 3 can be obtained by using the refined CP of Figure 14(a).) Subsequently, part III asserts two different conditions, depending on the value of update_condition. If update_condition is true, which corresponds to (P1) of CCC, then our CP asserts ψ', which matches $S'_{out} \models \psi'$ in the conclusion of (P1). Note that $S'_{in} \models \varphi'$ of (P1) also holds because in our CP, we assume φ and φ' are identical with each other. Meanwhile, If update_condition is false, which corresponds to (P2) of CCC, then our CP checks whether the outputs of both versions are identical with each other, which matches the conclusion of (P2), $S_{out} \approx S'_{out}$. □

7.3. Modular Handling of Procedure Calls via Change Contracts

When encountered with a procedure call, modern contract checkers espouse modular checking since looking into the body of a callee can be costly. Modular checking interprets the contract attached to a callee without looking into the body of a callee. For example, if a callee p has a program contract consisting of a precondition φ and a postcondition ψ, one can treat the call of p with the following simple Hoare triple:

$$\{\varphi\} \text{ call p(x) } \{\psi\}$$

If precondition φ is satisfied before calling p, one can assume that postcondition ψ is satisfied after calling p, assuming that the program contract of p is correct (this can be verified separately, hence modular checking). Such modular treatment of procedure calls is proven to be critical in the literature for scalable and systematic analysis [Flanagan et al. 2002; Müller 2002; Leavens 1991; Berdine et al. 2006]. Following this trend, we also support modular checking.

However, handling a procedure call with its change contract is significantly different than in program contract. The same simple rule shown above cannot be applied when a change contract is attached to a procedure, because a change contract does not describe an *absolute* input-output relationship unlike in a program contract. Instead, a change contract describes a *relative* relationship between the two versions of a procedure. Figure 10 describes such relationship as two axiomatic rules where p_v1 and p_v2 refer to the procedure p of version v1 and v2, respectively. We assume that procedure p has a change contract shown in the middle of Figure 9(a).

Notice in the first rule (i.e., [CHANGE-RULE]) that the premise contains a Hoare triple $\{\varphi\}$ call p_v1(x) $\{\psi\}$, which denotes the fact that the update condition of a given change contract is satisfied (i.e., the requires and when_ensured clauses are satisfied). Recall that if the update condition is satisfied, we expect the procedure to change its behavior at the next version. Therefore, the conclusion of this first rule is $\{\varphi\}$ call p_v2(y) $\{\psi'\}$. That is, ψ' instead of ψ is assumed to be satisfied, provided that

$$[\text{CHANGE-RULE}]\ \frac{\texttt{x == y} \qquad \{\varphi\}\ \texttt{call p\_v1(x)}\ \{\psi\}}{\{\varphi\}\ \texttt{call p\_v2(y)}\ \{\psi'\}}$$

$$[\text{PRESERVE-RULE}]\ \frac{\texttt{x == y} \qquad \neg(\{\varphi\}\ \texttt{call p\_v1(x)}\ \{\psi\})}{\texttt{call p\_v1(x) == call p\_v2(y)}}$$

Fig. 10: The axiomatic rules to interpret modularly a procedure call expression call p(x); p_v1 and p_v2 refer to the procedure p of version v1 and v2, respectively. The conclusion of [PRESERVE-RULE] means the return values of p_v1(x) and p_v2(y) are the same to each other.

procedures p_v1 and p_v2 are called with the identical input (i.e., x==y in our simple language). Meanwhile, the second rule ((i.e., [PRESERVE-RULE])) is for the remaining case where the update condition of a given change contract is not assumed. In this case, the output of the two versions of a procedure are assumed to be identical – i.e., in our simple language, the two return values are identical to each other, as we denote with call p_v1(x) == call p_v2(y) in the conclusion of [PRESERVE-RULE]. We describe how to handle procedure calls that have side effects in Section 7.7.2.

Overall, our two modularity rules are sound in the following sense:

THEOREM 7.2 (SOUNDNESS OF MODULARITY RULES). *Our two modularity rules are sound with respect to the CCC defined in Definition 3. That is, if the premise of the rule is valid, then its conclusion is also valid.*

PROOF. Our modularity rules are used under the assumption that the callee p satisfies its change contract. If the premise of [CHANGE-RULE] is valid, then the premise of (P1) of CCC is satisfied. Following the conclusion of (P1), the conclusion of [CHANGE-RULE] is also valid. Recall our current assumption that φ' is identical with φ, and ex is false. The soundness of [PRESERVE-RULE] is proved in a similar way. □

Our two modularity rules essentially describe how to interpret a call of procedure p_v2 – procedure p used in v2 –, based on how a call of procedure p_v1 – p used in v1 – is assumed to be interpreted. In [CHANGE-RULE], p_v1 is assumed to satisfy the update condition of the change contract of p, while in [PRESERVATION-RULE], p_v1 is assumed not to satisfy the update condition. While performing static CCC, we conservatively consider the both situations separately because, in our modular reasoning framework where the body of p is not looked into, we cannot know whether the update condition is satisfied or not. If p has no change contract, however, we only consider [PRESERVATION-RULE], because no behavioral change is expected at all. Technically, one can consider ψ of the rules as false, and as a result, only [PRESERVATION-RULE] can be activated.

When p_v1 and/or p_v2 are called multiple times, it is necessary to properly align each p_v2 with its matching p_v1. Consider the following two versions of a program in which procedure p is called twice at each version.

```
1  // previous version (v1)
2  int x = in;
3  int r1 = call p_v1(x);
4  int y = -x;
5  int r2 = call p_v1(y);
```

```
1  // updated version (v2)
2  int x = -in;
3  int r1 = call p_v2(x);
4  int y = -x;
5  int r2 = call p_v2(y);
```

In the above, variable in refers to the input of the program. Recall that CCC is performed with the same input to both versions. In this example, p_v1(x) in line 3 (left) should be aligned with p_v2(y) in line 5 (right), because at both sites, the procedures are called with a parameter whose value is the same as in. For a similar reason, p_v1(y)

```
1  // Part I: interpret p(x) of v1 (i.e., p_v1(x)) modularly (used at v1 and v2)
2  boolean update_condition = false;
3  int result_v1;
4  assume result_v1 == '(call p_v1(x)); // treat p_v1(x) as an uninterpreted function
5  boolean requires_clause = [[φ]]; // store the value of the requires clause
6  if(requires_clause) {
7    if(*) { // nondeterministic choice
8      // assume the when_ensured clause; [[ψ]] uses result_v1
9      assume [[ψ]];
10     // {φ}call p_v1(x){ψ} is established
11     update_condition = true;
12   } else {
13     // assume the negation of the when_ensured clause; [[¬ψ]] uses result_v1
14     assume [[¬ψ]];
15   }
16 }
17
18 // Part II: interpret p(x) of v2 (i.e., p_v2(x)) modularly (used at v2)
19 if(version == 2) { // reached only at v2
20   int result_v2;
21   assume result_v2 == '(call p_v2(x)); // treat p_v2(x) as an uninterpreted function
22   if(update_condition) {
23     // assume the ensures clause; [[ψ']] uses result_v2
24     assume [[ψ']];
25     // {φ}call p_v2(x){ψ'} is concluded
26   } else {
27     // assume the output equivalence
28     assume result_v1 == result_v2;
29   }
30 }
```

Fig. 11: The CP fragment for a procedure call expression call p(x)

in line 5 (left) should be aligned with p_v2(x) in line 3 (right). The general rule about procedure call alignment is to align two procedure calls residing in two different versions, when they take the same input (in the case of our current simple language, their parameters). This is why the above two rules – [CHANGE-RULE] and [PRESERVATION-RULE] – have an equation x == y in their premises. The description about how we enforce the above input-based procedure call alignment is provided in Section 7.4.

When there are aligned callees between the two versions, our modularity rules can be applied to constrain the behavior of the two versions. If there is no p_v1 call aligned with a p_v2 call, however, that p_v2 call is left unconstrained, which can lead to a spurious change contract violation. In practice, modularity rules are particularly handy when reasoning about how the behavioral changes of one procedure is propagated to the callers of the changed procedure. Even if a caller does not change its procedure body, its behavior would change according to the changes made to its callee.

7.4. Enforcing Modular Handling of Procedure Calls

We enforce our modularity rules of Figure 10 in our CP (composed program). More specifically, we transform each procedure call into the CP fragment shown in Figure 11.

We earlier noted that both [CHANGE-RULE] and [PRESERVATION-RULE] should be considered for each procedure call. To check both rules, our CP fragment uses a nondeterministic branch (see if(*) in line 7). If the then branch is chosen nondeterministically, the update condition $\{\varphi\}$ call p_v1(x) $\{\psi\}$ is established by assume $[\![\psi]\!]$[9] (φ is already established beforehand in line 6). Thus, at the aligned procedure call at version v2,

[9] $[\![\psi]\!]$ represents the interpretation of ψ.

[CHANGE-RULE] is enforced. Conversely, if the else branch is taken, [PRESERVATION-RULE] is enforced instead.

We earlier also noted that procedure calls should be aligned semantically based on the parameter values of procedure calls. To consider that, let us revisit the following versions of a procedure.

```
1  // previous version (v1)
2  int x = in;
3  int r1 = call p_v1(x);
4  int y = -x;
5  int r2 = call p_v1(y);
```

```
1  // updated version (v2)
2  int x = -in;
3  int r1 = call p_v2(x);
4  int y = -x;
5  int r2 = call p_v2(y);
```

Suppose that the change contract of procedure p has the following when_ensured clause:

```
when_ensured \result > 0;
```

In the above, \result refers to the return value of the procedure. We explicitly represent the return value of callee p_v1 with an uniterpreted function p_v1(x). The quote expression '(call p_v1(x)) in Figure 11 denotes that uniterpreted function. Suppose that the when_ensured clause of p is nondeterministically assumed to be satisfied in line 3 of v1, and dissatisfied in line 5 of v1. Then, we have the following constraint:

$$\text{p\_v1(in)} > 0 \wedge \neg(\text{p\_v1(-in)} > 0)$$

Let us move on to the updated version. At line 3, the procedure is called with parameter -in. Since the current constraint entails $\neg(\text{p\_v1(-in)} > 0)$, [PRESERVATION-RULE] should be applied. How do we enforce this? Our solution is that whenever interpreting p_v2, we also (modularly) interpret p_v1. Note that in Figure 11, p_v1 is interpreted at part I before interpreting p_v2 at part II. While interpreting p_v1, only the else branch of if(*) can be considered. If the then branch is taken, then the assumption established there (i.e, $\text{p\_v1(-in)} > 0$) conflicts with the already-established constraint, $\neg(\text{p\_v1(-in)} > 0)$. As a result, the update_condition flag is not turned on, and [PRESERVATION-RULE] is enforced at part II. Conversely, [CHANGE-RULE] is applied at line 5 of version v2.

THEOREM 7.3 (SOUNDNESS OF THE CP FRAGMENT FOR A PROCEDURE CALL). *The interpretation of our CP fragment for a procedure call (see Figure 11) correctly enforces our two modularity rules (see Figure 10).*

PROOF. Consider two (semantically) aligned procedure calls p_v1(x) and p_v2(y) where x == y. Then, at part I of Figure 11, the following three execution paths are possible when interpreting p_v1(x).

Case1. requires_clause at line 6 is true, and the then branch is taken at line 7. As a result, variable update_condition becomes true, and $\{\varphi\}$ call p_v1(x) $\{\psi\}$ holds true.

Case2. requires_clause at line 6 is true, and the else branch is taken at line 7. As a result, variable update_condition remains false, and $\neg(\{\varphi\}$ call p_v1(x) $\{\psi\})$ holds true.

Case3. requires_clause at line 6 is false. As a result, variable update_condition remains false, and $\neg(\{\varphi\}$ call p_v1(x) $\{\psi\})$ holds true.

When p_v1(y) is interpreted later on, the same execution path is taken at part I because of the reason we explained earlier. Subsequently at part II, $\{\varphi\}$ call p_v2(y) $\{\psi'\}$ is established only when update_condition is true (also, $\{\varphi\}$ call p_v1(x) $\{\psi\}$ holds true). In the other cases (when $\neg(\{\varphi\}$ call p_v1(x) $\{\psi\})$), call p_v1(x) == call p_v2(y) holds true. □

```
1  // v1                                    // v2
2  int sum(int k) {                         int sum(int k) {
3    int s = 0;                               int s = 0;
4
5    //@ set s = call sum_loop(k);            //@ set s = call sum_loop(k);
6    for (int i=1; i < k; i++) {              for (int i=1; i <= k; i++) { // operator changes
7      s = s+i;                                 s = s+i;
8    }                                        }
9
10   return s;                                return s;
11 }                                        }
```

(a) v1 and v2 of procedure sum.

```
1  /*@ changed_behavior                      /*@ changed_behavior
2    @ requires k >= 1;                       @ requires k >= 1;
3    @ ensures \result==\prev(\result)+k;     @ ensures \result==\prev(\result)+k;
4    @*/                                      @*/
5  int sum_loop(int k);                     int sum(int k);
```

(b) The change contracts of p and loop.

Fig. 12: An example to describe how loops can be handled modularly.

7.5. Modular Handling of Loops by Means of Procedure Calls

In the literature of program contracts (e.g., [Flanagan et al. 2002; Barnett et al. 2006b; Ahrendt et al. 2004]), either of the following two approaches are taken to handle loops: (1) Loop unrolling – the behavior of each loop is under-approximated by unrolling that loop a finite number of times, or (2) a modular approach using loop invariants – each loop is associated with its loop invariant that takes a role of the contract for that loop.

When using change contracts instead, both approaches can be taken again with adjustments. First, loop unrolling is straightforward. One simply needs to unroll each loop of both versions the same number of times.

Meanwhile, substantial adjustment is necessary for the second modular approach. Note that a loop invariant is the *program contract* for the corresponding loop, in a sense that it describes the behavior of an individual loop. However, what we need is the *change contract* of a loop that describes how the behavior of that loop changes across versions. We use a different specification than a loop invariant to describe the changes of a loop, for the same reason we use a change contract instead of a program contract to describe the changes of a procedure.

Figure 12 shows how we specify the behavioral changes of a loop. First, Figure 12(a) shows the two versions of procedure sum. The only difference between them is the operators used for the loop exit conditions (i.e., $\mathsf{i} < \mathsf{k} \rightarrow \mathsf{i} <= \mathsf{k}$). As a result, the sum of v1 adds the numbers from 1 to $\mathsf{k}-1$, while its counterpart of v2 adds the numbers from 1 to k. The value of k is given as a parameter of sum.

Notice that we annotate those loops with "//@ set s=sum_loop(k);". Apparently, this annotation does not express a loop invariant. It is instead an assignment statement. [10] This assignment is used by our static checker, and is not executed at runtime. The left-hand side of this assignment is variable s whose value changes over the loop. The right-hand side expression "call sum_loop(k)" calls an auxiliary specification-purpose procedure sum_loop. This procedure sum_loop does not exist in the original source code.

We use these new-style specifications of loops in interpreting loops in a modular way. Our static checker skips over loops. Instead, it uses the specifications of loops. Since

[10] "//@ set" is a JML notation to designate a specification-only assignment.

each of those specifications calls a procedure, we can reuse our modular handling of procedure calls.

More specifically in our example, we assign a new procedure sum_loop a change contract that describes the behavioral differences between the loops of two versions. The left-hand side of Figure 12(b) shows the change contract of procedure sum_loop. As long as variable k is greater than equal to one, the return value at v2 (i.e., \result) is k more than the the return value at v1 (i.e., \prev(\result)). This matches the fact that the loop in v2 iterates one more time than the one in v1 as long as k >= 1 holds, and as a result, the final value of variable s – the value s has when the loop exits – is greater in v2 than in v1 by the last added value, i.e., k.

Given the above loop annotations and the change contracts of sum_loop, our checker recognizes that the final value of variable s is k more in v2 than in v1. As a result, our checker concludes that the change contract of procedure sum – the procedure that contains the loop – is satisfied (the change contract of sum is shown in the right-hand side of Figure 12(b)).

In summary, we annotate a loop with a procedure call, and express behavioral changes of a loop as a change contract of the procedure used in that loop annotation. This way, we reduce the problem of modular handing of loops into the problem of modular handling of procedures, which we already addressed.

Comparison with Loop Invariants. In our example, the loop invariant of the first loop (the loop of v1) is s==k(k-1)/2, whereas the one of the second loop is s==k(k+1)/2. Subtraction of k(k-1)/2 from k(k+1)/2 is indeed k, as we describe with a change contract in Figure 12(b) (i.e., \result == \prev(\result) + k).

In this comparison, we observe again the following difference between program contracts and change contracts. If one is only interested in the difference across versions, one can directly specify that difference while omitting unnecessary details. It is usually easier to know the difference between two similar loops than the loop invariants of those loops.

7.6. Generating a VC (Verification Condition)

A composed program CP described earlier leads to a change contract violation when one of the assertions in CP is violated. In the CP shown in Figure 9, we use two assertions, both of which appear in part III where outputs of versions v1 and v2 are compared to each other. One assertion checks whether the output of v2 changes as expected following a given change contract. The other assertion checks whether output is preserved across versions when no behavioral change is expected, according to a given change contract.

If one can find an input to a CP that leads to the violation of one of the assertions in that CP, then that input witnesses the violation of a given change contract. Otherwise, it can be concluded that the actual program changes respect a given change contract. It is well-known that the problem of finding such a violation-inducing input can be reduced to the problem of satisfiability.

We use the standard approach based on a verification condition (VC) that is automatically generated from a given program. A VC is a first-order logic predicate that can be valid only when the program cannot reach an error state. In our context, the validity of the VC implies that there is no input that leads to the violation of the assertions in the source CP. The validity of a VC can be checked by querying the satisfiability of the negation of that VC.

We customize OpenJML [Cok 2014] to generate a VC. Given a composed program CP, our customized OpenJML generates a VC in the format of SMT2 [Barrett et al.

```
*** The previous version
.../pre-bug/.../ZonedChronology.java:8:
  int o = getOffset(i);
      VALUE: i === 1139
      VALUE: o === 1143
.../pre-bug/.../ZonedChronology.java:9:
  r = iField.roundFloor(i+o);
      VALUE: i+o === 2282
      VALUE: r === 1143
.../pre-bug/.../ZonedChronology.java:10:
  return iTime?(r-o):toUTC(r);
      VALUE: iTime === true
      VALUE: r === 1143
      VALUE: o === 1143
      VALUE: \result === 0
.../pre-bug/.../ZonedChronology.scc:5:
  when_ensured getOffset(r)==getOffset(r-getOffset(r));
      VALUE: r === 1143
      VALUE: getOffset(r) === 1142
      VALUE: getOffset(r-getOffset(r)) === 1142
      VALUE: getOffset(r)==getOffset(r-getOffset(r)) === true


*** The updated version
.../post-bug/.../ZonedChronology.java:8:
  long t = toLocal(i);
.../post-bug/.../ZonedChronology.java:8:
  ensures \result == i+getOffset(i);
      VALUE: i === 1139
      VALUE: getOffset(i) === 1143
      VALUE: \result === 2282
.../post-bug/.../ZonedChronology.java:9:
  r = iField.roundFloor(t);
      VALUE: t === 2282
      VALUE: r === 1143
.../post-bug/.../ZonedChronology.java:10:
  return toUTC(r);
      VALUE: r === 1143
      VALUE: \result === 1
.../ZonedChronology.scc:6:
  ensures \result==\prev(\result);
      VALUE: \result ===  1
      VALUE: \prev(\result) === 0
      VALUE: \result==\prev(\result) === false
```

Fig. 13: A counter-example path witnessing a change contract violation

2012]. Then, an automated theorem prover such as Z3 [Moura and Bjørner 2008] is used to check the validity of a VC.

If a VC is proven to be invalid, Z3 can generate a witness for a change contract violation. Using this information, our static checker generates counter-example report. Figure 13 shows such a counter-example report. As shown in the figure, a counter-example report describes an execution path that leads to a change contract violation, and the values of variables and expressions that appear in that execution path. In Figure 13, the upper part describes the previous version, and the bottom part the updated version. This counter-example corresponds to a regression error between Joda-Time version 1.4 and 1.5. Indeed, the last line of the figure – "\result == \prev(\result) === false" – shows that the return values of the two versions are different from each other (Recall that \result and \prev(\result) represent the return value of the previous and the updated version, respectively).

```
1  /***** Part II: interpret v1 to see if the update condition is true *****/
2  boolean update_condition = false; // the update condition is initially false.
3  int result_v1; // the variable to hold the return value of m at v1
4  Exception exception_v1; // an exception, if any, thrown at v1
5  try {
6    result_v1 = [[body_1]]; // interpret body_1 and store the return value at result_v1
7  } finally {
8    if (exception_v1 == null) { // if no exception is thrown at v1
9      // set the update condition true if the when_ensured clause is true.
10     boolean when_ensured_clause = [[ψ]];
11     if(requires_clause && when_ensured_clause) {
12       update_condition = true;
13     }
14   } else { // if an exception is thrown at v1
15     // set the update condition true if the when_signaled clause is true.
16     boolean when_signaled_clause = [[θ]]; // θ refers to the given when_signaled clause.
17     if(requires_clause && when_signaled_clause) {
18       update_condition = true;
19     }
20   }
21 }
```

(a) A (simplified) refined CP of Figure 9(b); it can deal with abnormal termination of a (Java) method; part I and III are omitted.

```
1  // Part I: interpret p(x) of v1 (i.e., p_v1(x)) modularly (used at v1 and v2)
2  assume result_v1 == '(call p_v1(x));
3  assume exception_v1 == '(call p_v1_abnormal(x));
4
5  boolean requires_clause = [[φ]]; // store the value of the requires clause
6  if(requires_clause) {
7    if (exception_v1 == null) { // if assumed that an exception is not thrown
8       // nondeterministically assume either the when_ensured clause or its negation
9    } else {
10     // nondeterministically assume either the when_signaled clause or its negation
11     if (caller_version == 1) {
12       throw exception_v1;
13     }
14   }
15 }
```

(b) A (simplified) refined CP fragment of Figure 11; it can modularly reason about abnormal termination of a callee (as well as normal termination); part II is omitted.

Fig. 14: Refinements of the basic CP shown earlier to deal with abnormal termination.

7.7. Java-specific and Miscellaneous Issues

In this section, we address Java-specific issues such as handling exceptions and fields. We use the terms procedure and method interchangeably in this section. We also mention a few miscellaneous issues worth mentioning in this section.

7.7.1. Handling Exceptions. A Java method can terminate not only normally but also abnormally by throwing an exception. In fact, many fixes of Java programs are related to handling such abnormal termination. For example, an exception thrown unexpectedly in the previous version should disappear in the updated version. As described earlier, our change contract language can handle abnormal termination as well as normal termination.

To handle abnormal termination, we refine the basic CP shown earlier. Figure 14 shows our refinements. Our refined CP distinguishes an abnormal termination of a procedure from a normal termination. In Figure 14(a), the exception thrown at $body_1$ – the body of the previous version procedure – is stored in variable exception_v1, and thus, by checking whether exception_v1 is null, we can distinguish whether $body_1$ terminates

normally or abnormally. In case $body_1$ terminates abnormally, we check whether the given when_signaled clause is satisfied, in the same way as we check the when_ensured clause for the normal termination case. Likewise, $body_2$ – the body of the updated version procedure – can be handled in a similar way (we omit to describe this refined handling of $body_2$ in Figure 14(a)).

We also refine our modular handling of callees. As a simple example, consider a case where a NullPointerException is unexpectedly thrown from a callee method m. In that case, a user can assign callee m the following change contract:

```
/*@ changed_behavior
  @  when_signaled (NullPointerException) true; // whenever NPE is thrown at v1
  @  signals (NullPointerException) false; // v2 should not throw NPE
  @*/
 int m(int x);
```

As before, our modular checker interprets the above change contract instead of looking into the body of callee m. The behavior of m at v2 changes when a NullPointerException is thrown from v1. However, when an exception which is not a NullPointerException is thrown from v1 given a certain input – this makes the above when_signaled clause unsatisfiable –, the behavior of m is preserved across versions. Similarly, when m terminates normally at v1 without throwing an exception, the behavior is preserved again.

Similar to how we represent a return value of a method with an uninterpreted function, we also represent a (potential) exception of a method with another uninterpreted function. Notice in Figure 14(b) that we use an uninterpreted function p_v1_abnormal(val) to represent an exception thrown from procedure p of version v1 when value val is passed to p as its parameter. Similar to normal termination, this uninterpreted function is further constrained by a given when_signaled clause. For example, in case the when_signaled clause of our running example is assumed to be true, the type of p_v1_abnormal(val) is constrained to be NullPointerException.

7.7.2. Callees that Read/Write Fields. Java methods are not necessarily side-effect free. They can update the values of fields. Consider the following change contract involving a field value change:

```
/*@ changed_behavior
  @  when_ensured this.name == null; // whenever name has null at the method exit in v1,
  @  ensures this.name.equals(""); // name should have an empty string "" instead in v2.
  @*/
 int p(int x);
```

The above change contract describes the change of field name. As we represent the return values of the two versions of a procedure p with uninterpreted functions p_v1(x) and p_v2(x), we represent the field values via another uninterpreted functions, p_v1_field_value(x) and p_v2_field_value(x). These two new uninterpreted functions can be constrained by the given when_ensured clause and ensures clause, respectively. In the above example, our static checker can maintain a constraint, p_v1_field_value(x)==null, to consider the case field name has null at the method exit in v1.

Recall that to align callees called in different versions, we compare the input of callees. We earlier showed how we align two versions of callees called with the same parameter values. In the presence of fields, we extend our alignment mechanism to accommodate the fields read by a callee. More specifically, we extend uninterpreted functions such as p_v1(x) into p_v1(this_v1, x, f) where f refers to a field read by method p and this_v1 the implicit receiver of a method call.

The fields that are read/written by a callee can be specified with a JML's accessible/assignable clause, and our prototype tool consults accessible/assignable clauses when constructing uninterpreted functions. Automatic inference of those clauses is also pos-

sible through side-effect analysis [Sălcianu and Rinard 2005], while our prototype tool currently does not contain it.

7.7.3. Field Updates. Consider the following two simple versions of a Java program.

```
// previous version (v1)
x.f = x.f + 1;
```

```
// updated version (v2)
x.f = x.f + 2;
```

The change between the above two versions can be described with:

```
ensures x.f == \prev(x.f) + 1;
```

However, special care is necessary to ensure the above simple change contract. The essence of the problem is that we compose two versions of a program into a single program CP (Composed Program), and interprets v1 and v2 sequentially. As a result, the field updates that occurs at v1 can affect the field values at v2, unless special care is taken.

We address this problem by customizing the conventional VC (Verification Condition) generation method. In a VC, a field is represented with an array. For example, a field access expression, x.f, is encoded as $f[x]$ where f is an array corresponding to field f, and x is an variable corresponding to x. What about x.f = x.f+1 of the above example? The standard way to encode a field update is to update the array representing the field. When encountered with a field update x.f = x.f+1, the original array f is updated into f' as follows:

$$f'[r] = \begin{cases} f[r] + 1 & \text{if } r \text{ equals } x, \\ f[r] & \text{otherwise} \end{cases}$$

We customize the above standard encoding in two ways. First, to enforce input equivalence at the entries of versions v1 and v2, we use the same array f at both versions to access the initial value of field f. Second, we confine the scope of an array update only to the version where an update takes place. In other words, even after array f is updated into f' at version v1, this array update is not propagated into version v2. By updating field arrays separately in each version this way, we prevent the update of a field at one version interfering with the interpretation of the other version.

7.7.4. Handling \prev Expressions. To check the change contract of the above example, we also need to be able to handle a \prev expression. For this, we make use of our customized VC described above. We obtain the value of \prev(x.f) through the last field array for f defined at version v1.

7.7.5. Structural Changes. Even in the presence of signature changes across versions – e.g., class/method names may change and method parameters may be added/deleted –, CCC can still be performed. One additional task in this case is to match a method at v1 with a method at v2 (this is trivial when there is no structural changes). We perform this match based on the information available in change contracts. Recall that one can describe in a change contract how a class/method name changes, and which parameters/fields are added or removed across versions.

7.7.6. Multiple Change Cases. A change contract can express multiple behavioral changes of a method. For example, the change contract in Figure 15(a) describes that the behavior of the updated version changes differently depending on how the previous version method terminates. The first case corresponds to the situation where the previous version method (v1) terminates abnormally, throwing an NullPointerException, as described in line 3. If this is the case, line 4 dictates that a NullPointerException should not be thrown in the updated version (v2) when the same input is given. What if v1 ter-

```
1   /*@ changed_behavior
2     @ // case 1. abnormal termination case
3     @  when_signaled (NullPointerException) true; // whenever NPE is thrown at v1
4     @  signals (NullPointerException) false; // v2 should not throw NPE
5     @ also
6     @ // case 2. normal termination case
7     @  when_ensured this.name == null; // whenever name has null at v1,
8     @  ensures this.name.equals(""); // it should be an empty string "" instead at v2.
9     @*/
10    int m(int x);
```

(a) Multiple cases of changes are separated by keyword "also"

```
1  boolean /*@ model @*/ ex; // an unconstrained specification-only field
2
3   /*@ changed_behavior
4     @  when_ensured !ex && this.name == null;
5     @  when_signaled (NullPointerException) ex;
6     @  ensures !ex ==> this.name.equals("");
7     @  signals (NullPointerException) ex ==> false;
8     @*/
9     int m(int x);
```

(b) An alternative change contract expressing the same multiple behavioral changes

Fig. 15: Change contracts expressing multiple behavioral changes

minates normally, without throwing an exception? Depending on which input is given to v1, v1 may terminate either normally or abnormally. The second case of the change contract corresponds to the normal termination case where field name has null as its value, as described in line 7. If this is the case, line 8 dictates that name should be an empty string instead at the end of v2.

Our checker supports such multiple change cases. To handle multiple change cases, we refine the CP shown in this section in a way that the information about which case is under consideration is maintained in a CP. Such extension is straightforward, and we omit to describe details.

As a side note, the same multiple behavioral changes described in the above can also be expressed with a single case, as shown in Figure 15(b). In the figure, ex is an unconstrained specification-only variable (field) that is supposed to indicate whether an exception is thrown. Only when ex is randomly chosen to be true, the given when_signaled and signals clauses are activated. Similarly, the when_ensured and ensures clauses are activated only when ex is chosen to be false.

7.8. Experience with Our Static Checker

In this section, we report our experience of using our checker we implemented on top of OpenJML [Cok 2014]. We applied our checker to 18 change instances extracted from various versions of Joda-Time [11], an open-source date / time library for Java. Table V shows the overall results we obtained after running our checker on our system – Ubuntu 12.04 (32-bit) Linux; Intel Core i5 CPU 650 (3.2GHz $\times$ 4) processor; 4GB RAM.

In Table V, we group the 18 change instances into 4 different groups depending on what our checker is used for. We used our checker not only for verifying program changes (usage V), but also for localizing the buggy method (usage L), detect-

[11] http://www.joda.org/joda-time/

Table V: Experimental results; pre-fix/post-fix indicates the previous/updated revision provided through the iBUGS dataset; in the first column, V stands for Verification, L Localization, R Regression, and C Classification; each usage is detailed in each subsection.

Usage	Bug #	Revision		Diff		Contract Size (lines)		Kind		Time (s)		Verified
		Previous	Updated	-	+	CC (lines/mthds)	JML	B	S	Total	Z3	
V	1788282	pre-fix (iBUGS)	post-fix	98	82	3/1	2	✔	✘	7.7	1.4 (18%)	✔
	1877843			62	81	3/1	23	✔	✘	8.1	1.9 (23%)	✔
	2111763			9	14	2/1	3	✔	✘	6.7	7.5 (4%)	✔
	2487417			25	28	2/1	5	✔	✘	6.2	4.7 (7%)	✔
	2783325			2	14	(1+1)/1	0	✔	✔	6.2	2.6 (4%)	✔
	2903029			78	45	2/2	4	✔	✘	6.5 6.5	1.0 (16%) 0.6 (10%)	✔ ✘
L	2025928	pre-fix (iBUGS)	post-fix	8	6	22/7	6	✔	✘	7.6 8.5 7.0 8.5 9.5 8.0	1.0 (14%) 1.5 (18%) 1.4 (21%) 1.7 (20%) 3.2 (35%) 0.9 (11%)	✔ ✔ ✔ ✔ ✔ ✔
R	1887104	7755b 7755b	c41ef a478f	95 1417	222 3524	2/1	10	✔	✘	8.4 6.7	1.0 (12%) 0.9 (15%)	✘ ✔
C	–	7b179	7b179′ 7b179′′ 1c524	2038	962	(8+3)/3	4	✔	✔	7.9 7.1 6.7	2.3 (30%) 1.9 (28%) 1.8 (27%)	✘ ✘ ✔

ing/debugging regression errors (usage R), and classifying the causes for test failures (usage C). The Usage column of Table V shows these four different usages.

We collected the majority of change instances from the Joda-Time dataset of iBUGS [Dallmeier and Zimmermann 2007]. This dataset is organized by bug numbers (shown in the second column of the table); each bug number is linked to its bug report and the source code of the pre-fix and post-fix revisions. We wrote change contracts based on the provided bug reports. We also described in change contracts structural changes if they occur. We provided our static checker with these change contracts along with a pair of the source code for pre-fix and post-fix revisions available through the iBUGS dataset.

We also collected some change instances directly from the Joda-Time repository [12] to experiment with change instances that are not available in the iBUGS dataset. For these non-iBUGS cases, we mark, in the Previous and Updated columns, the first five digits of Git snapshot IDs of the previous and updated revision, respectively.

The size of lexical changes made across revisions is shown in the Diff column, where the number of deleted (-) and inserted (+) lines are marked. Meanwhile, the size (i.e., the number of lines) of contracts is shown in the Contracts column, where the size of change contracts (CC) is distinguished from the size of program contracts (JML) used to remove false alarms; we did not count the header line "changed_behavior", and the library of JML contracts, e.g., the program contract for Object.equals. In the majority of cases, it was enough to write a change contract for one method. However, we occasionally wrote change contracts for more than one method. To inform the average size of a change contract per method, we mark in the CC column (the total number

[12] https://github.com/JodaOrg/joda-time.git

of lines of change contracts)/(the number of methods assigned a change contract). For example, 22/7 means that 22 lines were used for the change contracts of 7 methods. On average, we wrote 2.7 lines of change contracts for each method. Sometimes, we also described structural changes in change contracts (e.g., when refactoring was involved). To distinguish the portion of a change contract used to describe structural changes from the rest, we mark e.g., (8+3)/3, which means that in 3 methods, 8 and 3 lines were used to describe behavioral and structural changes, respectively. The Kind column more explicitly shows the kind of changes – among behavioral (B) and structural (S) changes – that were described in change contracts.

To finish each CCC session, it took on average 7.4 seconds (s), as shown in the Total column. A theorem prover (i.e., Z3 [Moura and Bjørner 2008]) consumed on average 27% (i.e. 2s) at the last phase of checking (the Z3 column shows its breakdown) – more time was consumed to parse and type-check source code. Lastly, the Verified column shows the result of checking: either verified (✔) or failed (✘). In the following subsections, we explain how to interpret those results in relation to four usages of our checker.

7.8.1. Verifying Intended Program Changes. The most basic usage of our checker is to verify that a program is changed as intended (i.e., as specified in change contracts). Our checker successfully verified program changes except in one case, where Z3 failed to handle a $\forall$-quantified expression used in a contract. As a result, our checker issued a false alarm. In other words, our checker is incomplete. In fact, it also inherits the unsoundness of its underlying platform, OpenJML; some errors – e.g., overflow of arithmetic expressions – can be missing. The sources of unsoundness and incompleteness of OpenJML can be found in [Cok 2014]. However, this soundness/completeness issue is orthogonal to the problem of CCC. In general, the techniques to improve soundness/completeness in checking program contracts can also benefit CCC.

7.8.2. Localizing the Buggy Method. The method that manifests an error is not necessarily buggy. Rather, it is often one of its callees (or a callee of a callee) that is buggy. For example, one bug report of Joda-Time (bug 2025928) reports that method print does not behave as expected (i.e., nothing is output when "0" should be output). However, in fact, it turns out that print itself is not buggy. Instead, another method getFieldValue is found to be buggy – print eventually calls getFieldValue before it returns, and the wrong return value of getFieldValue propagates to print, where an error is manifested. In such a case where the method that manifests an error (e.g., print) is not buggy itself, one first needs to localize the buggy method (e.g., getFieldValue).

We found that our checker can help localize the buggy method. We first started with writing a change contract of print, reflecting our intention to fix the manifested error. Our initial trial of verification failed. By looking at the generated counterexample, we were able to find that one of the callees (i.e., printTo), should change its behavior to satisfy the given change contract. Once we assigned a proper change contract to this callee, CCC succeeded. That is, the change contract of print was successfully verified, assuming that the change contract of printTo is correct. To see if the assumption we made is true, we tried to verify printTo. Again, our initial verification trial failed, and we repeated to look for suspicious method calls in a counterexample to assign proper change contracts to them. We repeated this procedure until we reached the buggy getFieldValue method whose change contract was successfully verified, without having to assign change contracts to callees. The L section of the table shows the experimental data obtained through this repeated procedure, with the top row corresponding to print (where an error is manifested), the next row to a callee of print, and so on, and finally the bottom row to getFieldValue, the buggy method.

```
/*@ changed_behavior
  @  when_ensured true; // passes in the previous version
  @  signals (AssertionFailedError e) false; // no failure
  @*/
 public void testXXX(); // test method
```

Fig. 16: The predefined change contract for a test method

7.8.3. Detecting/Debugging Regression Errors. We earlier showed in Figure 13 a counter-example that witnesses a regression error. That regression error takes place between revision 7755b and c41ef of Joda-Time where code changes are made to fix a problem about DST (Daylight Saving Time) cutover. We write a change contract corresponding to that intention - fixing a bug about DST cutover – and feed that change contract to our static checker along with the two revisions 7755b and c41ef.

Our checker was able to report a regression error along with a counter-example of Figure 13. Notice the verification failure mark (✘) at the end of the first row of the R section of Table V. Meanwhile, the next row shows the result when we replace a buggy-fix version c41ef with a correct-fix version a478f. In this case, our checker successfully verifies program changes against a given change contract.

7.8.4. Classifying the Cause for a Test Failure. As mentioned in Section 1, a test failure can be caused by the error in product code or test code. Classifying this cause for a test failure is on its own a research problem [Hao et al. 2013]. We found that our checker can help distinguish the cause for a test failure. The idea is to assign a change contract to a test. The change contract of Figure 16 expresses the intention that the test should pass in the updated version whenever it passes in the previous version. This change contract can be predefined and applied to any test method.

We applied this change contract to a test in Joda-Time revision 7b179 (i.e., testConstructor_long_DurationType1). This test in its body calls several methods. Among them, two methods change both their names and behaviors at the next revision (1c524). We assigned these two methods change contracts describing behavior / structural changes. Given these change contracts along with a pair of source code of the previous (7b179) and updated revision (1c524), our checker successfully completed verification (see the last row of the table) – indicating that a test was correctly modified.

Meanwhile, to check the efficacy of our checker in detecting the obsoleteness of a test, we prepared two variations of the previous-version (7b179) test; they served as obsolete tests in our experiment. In the first variation (7b179′), we changed the names of the callees correctly – assuming that renaming is trivial –, but did not update the oracles affected by the behavioral changes of the product code. In the second variation (7b179″), we additionally updated the oracle affected by the first callee, but did not do the same for the second callee. Our checker successfully detected the obsoleteness of these two tests. As expected, it failed at verification (see the first two rows of the C section of the table) – indicating that a test began to fail in the updated version, given changes of the methods under testing. Also, a generated counterexample shows which oracle fails.

What if a checker issues no warning while a test fails when actually run? This can happen because modular checking interprets method calls based on their contracts, not on their actual bodies. For example, the actual behavior of a callee under testing may be different from the intended behavior specified in its change contract. The conformity of a callee to its change contract should be checked separately. If this is the case, it is evident that a callee does not conform to its intended changes. Thus, one can conclude that a test fails because of an error in the production code.

7.8.5. Discussion. As mentioned, we often used not only change contracts but also program contracts to remove false alarms. We found that the size of these program contracts varies depending on a change instance, whereas the size of change contracts is more or less the same (i.e., 2.7 lines/method). However, in many cases, those program contracts tended to be simple and similar to each other. For example, we used the common program contract, "signals (UnsupportedOperationException) false", for 14 out of 23 program contracts used at bug 1877843 (for the purpose of removing false alarms, specifying partial behaviors is sufficient).

Our experience is restricted to a single subject (Joda-Time), and more experiments are desirable to validate some of our observations such as an average size of change contracts.

8. RELATED WORK

8.1. Design by Contract

Design by contract (DbC) [Meyer 1992] influenced the design of many program-level specification languages such as Eiffel [Meyer 1997], JML [Burdy et al. 2005] and Spec# [Barnett et al. 2004]. In DbC, each method has its contract typically in the form of pre and post conditions. And the contract in DbC (i.e., program contract) roughly means the following two things. First, a method has to guarantee its own post-condition whenever its pre-condition 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 significantly different from the concept of a change contract. A change contract captures the intended behavioral/structural changes between two program versions rather than the behavioral contracts within a single program. Unlike a program contract that makes an input-output relation, a change contract makes an output-output relation. In other words, an updated-version method has to guarantee its post-condition ψ', whenever its previous-version counterpart satisfies its own post-condition ψ. Meanwhile, when a method m is called in the updated version, the caller does not have to guarantee ψ, i.e., the post-condition of m's previous version (Contrast this with a program contract where the caller should guarantee the callee's pre-condition). Instead, if ψ does not hold, then m should produce the same output across versions.

Program contracts are typically checked either by extended static checking (ESC) [Flanagan et al. 2002; Cok and Kiniry 2004; Barnett et al. 2006a] or runtime assertion checking (RAC) [Cheon and Leavens 2002]. 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.

Both RAC and ESC have been explored in this article. Our dynamic checker corresponds to RAC, and static checker to ESC. Our both checkers are significantly different from those for program contracts, due to the facts that (1) the semantics of a change contract is different from the semantics of a program contract, and (2) two versions of a program are analyzed at the same time.

8.2. 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

```
1   // v1 (previous buggy version)
2   int m(int a[],int i,int s){
3     if (i <= MAX) { // buggy
4       assert Valid(i);
5       return a[i]+s;
6     } else return s;
7   }
```

```
// v2 (updated fixed version)
int m(int a[],int i,int s){
  if (i < MAX) { // fixed: use < instead of <=
    assert Valid(i);
    return a[i]+s;
  } else return s;
}
```

(a) An example for DAC (differential assertion checking)

```
// if there is no assertion violation in v1,
when_ensured true;
// there should be no assertion violation in v2 as well.
signals (AssertionError) false;
```

```
// if there is an assertion violation in v1,
when_signaled (AssertionError) true;
// that assertion violation should be fixed in v2.
signals (AssertionError) false;
```

(b) The change contract equivalent to DAC (left), and another (predefined) change contract more faithful to the intention of the change (right)

Fig. 17: Comparing DAC to CCC

errors efficiently without having to test the entire test suite [Rothermel et al. 2001; Chen et al. 1994; Gupta et al. 1992]. More recently, Jin et al. proposed a method that, given program changes, automatically generates tests that stress those program changes [Jin et al. 2010]. 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 [Qi et al. 2009; Zeller 1999]. 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.

8.3. Regression Verification and Relative Verification

Regression verification (RV) [Godlin and Strichman 2009; Godlin and Strichman 2013] and other similar approaches [Böhme et al. 2013; Korel and Al-Yami 1998] compare two versions of a program in search of regression errors. In essence, it is equivalence between two programs that is checked there (regression is a counterexample for equivalence). Meanwhile, our checker assures not only intended equivalence (against the implicit assumption of behavioral preservation), but also intended differences (against the explicit specification of change contracts). In this sense, CCC (Change Contract Checking) subsumes RV.

Differential assertion checking (DAC) [Lahiri et al. 2013] is a technique that checks whether v2 (the updated version) is as safe as v1 (the previous version). In other words, it checks whether v2 is safe relative to its previous version v1. Unlike in RV, behavioral preservation does not have to be guaranteed across versions. Even if v2 behaves differently from v1, relative safeness can be proved if no assertion violation is found in v2. DAC proves that by checking whether all the assertions appearing in v2 are satisfied,

provided that all the assertions appearing in v1 are assumed to be satisfied. Consider Figure 17(a) as an example paraphrased from [Lahiri et al. 2013]. While version v1 is buggy because an illegal array access a[MAX] is possible there, this problem is fixed at version v2. DAC succeeds in this example because: (1) DAC assumes that v1 passes all the instances of "assert Valid(i)" where i $= 1, 2, ...,$ MAX. (2) the same assertion appearing v2 must also be true in all instances considering that i can be $1, 2, ...,$ MAX-1.

DAC can be viewed as one instance of CCC. The change contract shown in the left-hand side of Figure 17(b) amounts to the intention of DAC; if no assertion is violated at v1 (i.e., v1 terminates normally as specified as "when_ensured true"), then AssertionError, which is thrown when the assertion of an assert statement is violated, should not be thrown at v2 as described in the signals clause. However, CCC can perform more than DAC by using a different change contract. For example, one can use the change contract in the right-hand side of Figure 17(b) to be more faithful to the intention of the change; i.e., fixing a bug manifested by AssertionError. When this alternative contract is applied, our checker reports a warning, reflecting the fact that v2 is not a complete fix; when using a[i], the length of array a should be guaranteed to be greater than i.

In summary, CCC subsumes RV and DAC. For methods that do not have change contracts, CCC performs RV. We can also easily change this default action to DAC by enforcing the predefined change contract for DAC when no change contract is given. Furthermore, the use of a few lines of change contract pushes the checking scope of CCC beyond its default action – to the extent that arbitrary program changes can be verified. This makes an interesting parallel to model checking [Clarke et al. 1999]; while model checking can by default check the absence of deadlock, other properties can also be checked when a few lines of specifications – e.g., temporal logic formulas – are provided.

8.4. Specifying/Checking Changes

Hawblitzel et al. [2013] also independently introduced specifications for program changes named mutual summaries – which can be viewed as change contracts for Boogie [Barnett et al. 2006b] programs. Boogie, as a low-level programming language, is significantly simpler than Java. Accordingly, mutual summaries are simpler than change contracts – e.g., no explicit consideration of abnormal termination and no implicit assumption of behavioral preservation. This simplicity of programming/contract languages makes the problem of contract checking simpler. Instead, potential impact on mainstream programmers is less immediate. On the contrary, our change contract language is designed to be used by Java programmers with little additional effort. For better user-friendliness, our change contract language has constructs such as when_ensured and when_signaled that are absent in mutual summaries. As a result, a programmer can write "when_ensured ψ; ensures ψ';" instead of having to write "ensures \prev(ψ) ==> ψ'" – the latter is akin to a mutual summary. While both change contracts express the same behavioral changes, the former more clearly shows the expected differences between two versions – i.e., when ψ is ensured in the previous version, a programmer needs to ensure a new behavior ψ' in the updated version.

Hawblitzel et al. [2013] also presented modular static checking of mutual summaries. To support modular checking, they directly manipulate the verification condition by adding to it an axiom whose essence can be paraphrased as: $\forall \bar{x} : f\_v1(\bar{x}) \wedge f\_v2(\bar{x}) \Rightarrow f\_v1\_v2(\bar{x})$. That is, whenever $f\_v1(\bar{x})$ called in v1 is aligned with $f\_v2(\bar{x})$ called in v2, their mutual summary (i.e., $f\_v1\_v2(\bar{x})$) is enforced. Note that we do not use quantifiers to support modular checking. While it is too early to tell which approach is advantageous, it is well known that the use of quantifiers often causes the incomplete verification result – i.e., the verification condition can be neither confirmed nor refuted. In addition, the use of quantifiers tends to increase the time cost. As de Moura (the key developer of

Z3) said, “as a rule of thumb, we should avoid quantifiers whenever possible.” [Moura 2012]

Differential assertion checking (DAC) [Lahiri et al. 2013] described in Section 8.3 uses mutual summaries under the hood to specify relative safeness. DAC is performed in the form of modular static checking. Unlike in [Hawblitzel et al. 2013], however, a forall quantifier is not used to align callees of two different versions. Instead, static checking is performed with each of all possible combinations of paris between a call expression of procedure p_v1 (a procedure p at version v1) and a call expression of p_v2 (p at version v2). As a result, if p is called twice at both v1 and v2, as in the example shown in Section 7.4, then 2×2 different combinations are included in the composed program. On the contrary, we align callees using uninterpreted functions, without explicitly enumerating all possible combinations – those combinations are tried implicitly inside a theorem prover if necessary. Note that modern theorem provers such as Z3 are generally quite efficient in dealing with combinations when quantifiers are not involved.

8.5. Specifying/Checking Intended Changes vs. Summarizing Actual Code Changes

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 [1994] 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. [2008] 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 [Lahiri et al. 2012] 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.

9. CONCLUSIONS

In this article, 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. This is not only because one can directly focus on changes, but also because one can conveniently express output-output relationship between program versions with a change contract. Our user study also indicates positively that change contracts can be easily learned and used by entry-level developers.

We have also presented two kinds of checkers for change contracts – a dynamic checker and a static checker. We have shown the efficacy of our dynamic checker in generating tests that manifest the violation of change contracts. Also, the efficacy of our static checker in verifying program changes against change contracts has been shown. Apart from verification, we also successfully used our static checker for various software engineering tasks such as localizing the buggy method, detecting/debugging a regression error, and classifying the cause for a test failure to blame either product code or test code.

ACKNOWLEDGMENTS

We thank David Cok (supported by NSF grants ACI-1314674, CNS1228930) for helping us use OpenJML and jSMTLIB. This work is partially supported by Singapore Ministry of Education research grant MOE2010-T2-2-073 and T1 251RES1314.

REFERENCES

Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. 2004. The KeY tool. *Software and Systems Modeling* 4, 1 (April 2004), 32–54.

Mike Barnett, B. Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006a. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects*. 364–387.

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006b. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proceedings of Symposium on Formal Methods for Components and Objects*. 364–387.

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. 2004. The Spec# Programming System: An Overview. In *Proceedings of the 2004 Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. 49–69.

Clark Barrett, Aaron Stump, and Cesare Tinelli. 2012. *The SMT-LIB Standard Version 2.0*. Technical Report. SMT-LIB.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2006. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO*. 115–137.

Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. 2013. Partition-based regression verification. In *ICSE*. 302–311.

Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. 2005. An overview of JML tools and applications. *STTT* 7, 3 (2005), 212–232.

Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. 1994. TestTube: A system for selective regression testing. In *ICSE*. 211–220.

Yoonsik Cheon and Gary T. Leavens. 2002. A runtime assertion checker for the Java Modeling Language (JML). In *Proceedings of the 2002 International Conference on Software Engineering Research and Practice*. 322–328.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. The MIT Press.

David R. Cok. 2014. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In *Proceedings 1st Workshop on Formal Integrated Development Environment*. 79–92.

David R. Cok and Joseph Kiniry. 2004. 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*. 108–128.

Valentin Dallmeier and Thomas Zimmermann. 2007. Extraction of Bug Localization Benchmarks from History. In *ASE*. 433–436.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended static checking for Java. In *PLDI*. 234–245.

Benny Godlin and Ofer Strichman. 2009. Regression verification. In *DAC*. 466–471.

Benny Godlin and Ofer Strichman. 2013. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability* 23, 3 (2013), 241–258.

Rajiv Gupta, Mary Jean, Mary Jean Harrold, and Mary Lou Soffa. 1992. An approach to regression testing using slicing. In *Proceedings of the 1992 Conference on Software Maintenance*. 299–308.

Dan Hao, Tian Lan, Hongyu Zhang, Chao Guo, and Lu Zhang. 2013. Is This a Bug or an Obsolete Test?. In *ECOOP*. 602–628.

Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. 2013. Towards Modularly Comparing Programs Using Automated Theorem Provers. In *CADE*. 282–299.

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

Wei Jin, Alessandro Orso, and Tao Xie. 2010. Automated behavioral regression testing. In *Proceedings of 2010 International Conference on Software Testing, Verification and Validation*. 137–146.

Bogdan Korel and Ali M. Al-Yami. 1998. Automated regression test generation. In *ISSTA*. 143–152.

Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SymDiff: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *CAV*. 712–717.

Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential assertion checking. In *ESEC/FSE*. 345–355.

Gary T. Leavens. 1991. Modular Specification and Verification of Object-Oriented Programs. *IEEE Software* 8, 4 (1991), 72–80.

Bertrand Meyer. 1992. Applying "Design by Contract". *IEEE Computer* 25 (1992), 40–51. Issue 10.

Bertrand Meyer. 1997. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall.

Leonardo de Moura. 2012. Answer for the question titled "Quantifier Vs Non-Quantifier". http://stackoverflow.com/questions/10011478/quantifier-vs-non-quantifier. (2012).

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS*. 337–340.

Peter Müller. 2002. *Modular Specification and Verification of Object-Oriented Programs*. Lecture Notes in Computer Science, Vol. 2262. Springer.

Carlos Pacheco and Michael D. Ernst. 2007. Randoop: feedback-directed random testing for Java. In *OOPSLA*. 815–816.

David Lorge Parnas. 2011. Precise documentation: The key to better software. In *The Future of Software Engineering*. Springer, 125–148.

Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina Păsăreanu. 2008. Differential symbolic execution. In *FSE*. 226–237.

Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. 2009. DARWIN: An Approach for Debugging Evolving Programs. In *ESEC-FSE*. 33–42.

Dawei Qi, Jooyong Yi, and Abhik Roychoudhury. 2012. Software Change Contracts. In *FSE*. 22:1–22:4.

Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 27, 10 (2001), 929–948.

Alexandru Sălcianu and Martin Rinard. 2005. Purity and Side Effect Analysis for Java Programs. In *VMCAI*. 199–215.

Jooyong Yi, Dawei Qi, Shin Hwei Tan, and Abhik Roychoudhury. 2013. Expressing and checking intended changes via software change contracts. In *ISSTA*. 1–11.

Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why?. In *ESEC/FSE*. 253–267.