Scaling Testing of Refactoring Engines

Melina Mongiovi, Gustavo Mendes, Rohit Gheyi, Gustavo Soares
Federal University of Campina Grande
Campina Grande, Brasil
{melina,gugawag}@copin.ufcg.edu.br, {rohit,gsoares}@dsc.ufcg.edu.br

Márcio Ribeiro
Federal University of Alagoas
Maceió, Brazil
marcio@ic.ufal.br

***Abstract*—Proving refactoring sound with respect to a formal semantics is considered a challenge. In practice, developers write test cases to check their refactoring implementations. However, it is difficult and time consuming to have a good test suite since it requires complex inputs (programs) and an oracle to check whether it is possible to apply the transformation. If it is possible, the resulting program must preserve the observable behavior. There are some automated techniques for testing refactoring engines. Nevertheless, they may have limitations related to the program generator (exhaustiveness, setup, expressiveness), automation (types of oracles, bug categorization), time consumption or kinds of refactorings that can be tested. In this paper, we extend our previous technique to test refactoring engines. We improve expressiveness of the program generator for testing more kinds of refactorings, such as Extract Function. Moreover, developers just need to specify the input's structure in a declarative language. They may also set the technique to skip some consecutive test inputs to improve performance. We evaluate our technique in 18 refactoring implementations of Java (Eclipse and JRRT) and C (Eclipse). We identify 76 bugs (53 new bugs) related to compilation errors, behavioral changes, and overly strong conditions. We also compare the impact of the skip on the time consumption and bug detection in our technique. By using a skip of 25 in the program generator, it reduces in 96% the time to test the refactoring implementations while missing only 3.9% of the bugs. In a few seconds, it finds the first failure related to compilation error or behavioral change.**

I. Introduction

Defining and implementing refactorings is a nontrivial task since it is difficult to define all preconditions to guarantee that the transformation preserves the program behavior. In fact, proving refactoring correctness for entire languages such as Java and C constitutes a challenge [1]. As a result, refactoring engines may have bugs [2], [3]. In practice, developers of refactoring engines use tests to evaluate the refactoring implementations. However, testing refactoring engines is not trivial since it requires complex inputs, such as programs, and an oracle to define the correct resulting program or whether the transformation must be rejected. Manually writing test cases may be costly, and thus it may be difficult to create a good test suite considering all the language constructs.

Researchers have proposed a number of automated techniques for testing refactoring engines [4], [5], [3], [6]. They automate four major steps of the testing process: (i) generating test inputs; (ii) applying the refactoring implementation; (iii) checking the output correctness; (iv) and classifying the detected failures into distinct bugs. Although these techniques have detected a number of bugs in refactoring engines, it remains a question whether they scale to detect more bugs without considerable effort.

For example, to automate the test input generation, Gligoric et al. [5] propose UDITA (an extension of ASTGen [4]), a Java-like language to write program generators so that developers can generate programs as test inputs. They used UDITA to generate about 5,000 programs with up to 3 classes as test inputs [5]. However, writing some of these program generators demands a considerable effort [6]. Soares et al. [2], [3] propose a Java program generator called JDolly for exhaustively generating programs. By using JDolly, developers can specify the number of some Java constructs and constraints for the generated programs by using Alloy [7], a formal specification language. They used JDolly to generate more than 100,000 programs. Although JDolly can reduce the effort for generating Java programs, it only generates programs with simple method bodies (only one statement), which is not enough to test refactorings within method level. Additionally, exhaustively generating programs, even for a small number of Java constructs, can require a lot of time. To alleviate this problem, Jagannath et al. [8] propose the Sparse Test Generation technique (STG), which skips some test inputs.

Later, Gligoric et al. [6] propose to use real programs as test inputs, automatically applying the refactoring under test in every possible location of the program. They found 141 bugs related to compilation errors and engine crash in 8 real systems in Java and C. Although this approach can reduce the effort to create test inputs, testing refactoring engines in large programs may increase the costs of checking the output correctness. For example, to identify bugs related to behavioral changes, Soares et al. [3] use SafeRefactor, a tool that analyzes a transformation and generates tests to compare the program behavior before and after the transformation. SafeRefactor was useful for finding 63 bugs related to behavioral changes in the programs generated by JDolly. However, using SafeRefactor to evaluate transformations on large real programs would require a much higher time for analyzing the transformation and generating tests. Additionally, understand a failure in a large transformation demands more time. Gligoric et al. [6] take 1-60 minutes to analyze each failure in order to categorize them into distinct bugs. In summary, the previous approaches have limitations related to the program generator (exhaustiveness, setup, expressiveness), automation (types of oracles, bug categorization), time consumption or kinds of refactorings that can be tested.

In this paper, we extend our technique [3] to scale testing of refactoring engines. To improve the expressiveness of the program generation, we propose the DOLLY program generator. It can generate Java (JDOLLY [3]) and C programs (we propose CDOLLY in this paper). The programs generated by CDOLLY can have functions with a sequence of statements, which allow us to test refactorings applied within function level that is, refactorings that modify statements inside function body. For example, Extract Function refactoring extracts a code fragment of a function into a new function. By implementing CDOLLY, we also show evidence that our previous approach [3] can be used for testing refactoring engines for other languages than Java, such as C.

To reduce the time to test the refactoring implementations, we implement a technique to skip some consecutive test inputs [8]. Consecutive programs generated by DOLLY tends to be very similar, potentially detecting the same kind of bug. Thus, developers can set a parameter to skip some programs to reduce the time to test the refactoring implementations. By skipping these programs, we can reduce the Time to First Failure (TTFF), reducing the developer idle time [8]. Our technique uses a set of automated oracles to evaluate the correctness of the transformations related to compilation errors, behavioral changes, and overly strong conditions. After identifying the failures, the technique uses a set of automated bug categorizers to classify all failing transformations into distinct bugs. For simplification we use the term transformation to refer to a refactoring or a failing transformation.

We evaluate 18 kinds of refactoring implementations of JastAdd Refactoring Tools (JRRT) [9], Eclipse JDT (Java) and Eclipse CDT (C). We found 76 (53 new bugs) bugs in a total of 49 bugs related to compilation errors, 17 bugs related to behavioral changes, and 10 bugs related to overly strong conditions. Among those bugs, we found 28 bugs in refactorings applied within function level. We also compare the impact of the skip on the time consumption and bug detection in our technique. By using a skip of 25 in the program generator, it reduces in 96% the time to test the refactoring implementations while missing only 3.9% of the bugs. Moreover, by using this same skip we find the first failure in general in a few seconds. So, the refactoring engine developer can find a bug in the refactoring implementation relatively quickly, fix it, run our technique again to find another bug, and so on. Before a release, tool developers can run the technique without skip to find some missed bugs.

II. TECHNIQUE

In this section, we explain the main steps of our technique (Figure 1). First, it automatically generates programs as test inputs for a refactoring using DOLLY, an automated program generator (Section II-A). Next, the refactoring under test is automatically applied to each generated program (Section II-B). To evaluate the correctness of the transformations, our technique uses a set of automated oracles (Section II-C). Finally, the detected failures are automatically categorized into distinct bugs (Section II-D).

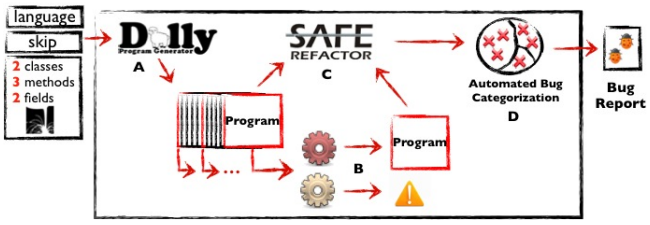

Fig. 1. A technique for scaling testing of refactoring engines.

A. Test Input Generation

We use DOLLY to automatically generate programs. It can generate Java and C programs. DOLLY uses JDOLLY, proposed by Soares et al. [2], [3], to generate Java programs. To generate C programs it uses CDOLLY (we propose CDOLLY in this paper in Section III). The refactoring engine developer passes as input an Alloy specification with the maximum number of elements that the generated programs may declare and additional constraints for guiding the program generation. DOLLY uses the Alloy Analyzer tool [10] that takes a specification and finds a finite set of all possible instances that satisfy the constraints within a scope. For each Alloy instance found by the Alloy Analyzer, DOLLY translates it to a program. Furthermore, developers can specify a skip number to jump some of the Alloy instances to avoid state explosion.

B. Refactoring Application

The second step of the technique is to automatically apply the refactoring under test to each program generated by DOLLY. For this purpose, we implemented a program that uses the engines API to apply the refactorings automatically. If there are more than one engine under test, it can perform differential testing to detect bugs related to overly strong conditions [2]. The engine checks a set of preconditions before applying the transformation. If they are satisfied, the transformation is applied. Otherwise, the transformation is rejected.

C. Automated Test Oracles

In this step, the technique uses a set of automated oracles to evaluate correctness of the transformations. Refactoring engines can have overly weak and strong conditions [3], [2]. Refactoring implementations with overly weak conditions allow applying transformations that change the program behavior. On the other hand, refactoring implementations with overly strong conditions reject applying transformations that preserve the program behavior.

We use a compiler to identify bugs related to compilation errors, and SAFEREFACTOR [11], [12] to identify behavioral changes. SAFEREFACTOR evaluates whether the transformation preserves the program behavior by automatically generating tests to the methods impacted by the change. By comparing two versions of a program, it identifies the common methods impacted by the change. Next, SAFEREFACTOR generates a test suite for the previously identified methods using an

automatic test suite generator. It executes the same test suite before and after the transformation. If the results are different, the tool reports a behavioral change, and yields the test cases that reveal it. Otherwise, we improve confidence that the transformation is behavior preserving.

In our previous work, we proposed a technique [2] to identify overly strong conditions in refactoring implementations based on differential testing [13]. It needs at least two engines to apply the same kind of refactoring. If an engine rejects a transformation, and the other one applies it and preserves behavior according to SAFEREFACTOR, the technique establishes that the former engine contains an overly strong condition.

D. Bug Categorizer

In the previous step, the automated oracles may detect a number of failures in the refactoring implementations. A single bug in the refactoring may cause several of those failures. Classifying them manually may be time consuming. Then, we automate this classification. Soares et al. [3] implemented a tool to categorize failures related to compilation errors and overly strong conditions. They use the technique proposed by Jagannath et al. [8] to automatically classify failures related to compilation errors into distinct bugs. It is based on splitting the failing tests based on messages from the test oracle. Soares et al. [3] used a similar approach to classify failures related to overly strong conditions. Moreover, they specified a systematic but manual approach to categorize failures related to behavioral changes based on structural characteristics of the transformation, such as overloading, overriding, and implicit cast. In this work, we automate the classification of failures related to behavioral changes based on their approach.

III. DOLLY

DOLLY is a program generator that exhaustively generates programs, up to a given scope. It has a parameter to skip some instances in order to reduce the set of generated programs. DOLLY generates programs for Java (JDOLLY) and C (CDOLLY). Soares et al. [3] proposed JDOLLY. In this section, we follow a similar approach and propose CDOLLY. It generates C programs from a C meta-model specification and additional constraints specified in Alloy. The main difference between JDOLLY and CDOLLY is that CDOLLY generates richer method bodies.

Next we provide an overview of Alloy (Section III-A). Section III-B presents the C meta-model used by CDOLLY. We then describe how to translate each Alloy instance to C and how to use CDOLLY for generating more specific C programs in Section III-C. Finally, Section III-D explains the technique to skip programs.

A. Alloy Overview

Alloy is a formal specification language, based on first order logic that allows the user to specify software systems by abstracting their key characteristics [7]. An Alloy specification is a sequence of signatures and constraints paragraphs declarations. A signature introduces a type and can declare a set of relations. The Alloy relations have a multiplicity that is specified using qualifiers, such as `one` (exactly one), `lone` (zero or one), `set` (zero or more), and `seq` (sequence of elements). In Alloy, one signature can extend another, establishing that the extended signature is a subset of the parent signature. Next we specify a list of objects in Alloy. Each list (`List`) may have a sequence of objects (`Object`) in the relation `objs`.

```
sig Object {}
sig List {
  objs: seq Object
}
```

Facts are used to package formulas that always hold. The fact `listSize` specifies that all lists must have at most 10 elements.

```
fact listSize {
  all l: List | #l·objs < = 10
}
```

The keywords `all`, `some`, and `no` denote the universal, existential, and non-existential quantifiers, respectively. Predicates are used to package reusable formulas and specify operations. The predicate `noEmptyList` specifies that all lists are non empty. The relation `objs` yields the elements of the sequence and the relation `isEmpty` checks whether a sequence is empty.

```
pred noEmptyList[] {
  no l: List | l·objs·isEmpty
}
```

The Alloy Analyzer tool allows us to perform analysis on an Alloy specification [10]. A run command is applied to a predicate, specifying a scope for all declared signatures. For example, in the following command the Alloy Analyzer searches for an instance with at most three objects (scope) for `List` and `Object` satisfying all signature and fact constraints, in addition to the constraints specified in `noEmptyList`. The Alloy Analyzer has a feature to find all valid instances for a given scope.

```
run noEmptyList for 3
```

B. C Meta-Model

CDOLLY generates a subset of C programs. Next we present the C abstract syntax and well-formedness rules considered in CDOLLY.

1) Abstract Syntax: We specified in Alloy a subset of the C meta-model. A C program may declare some functions. We specified the signature `Function` representing the functions of a program. A function can have parameters, a sequence of statements, and one return type.

```
sig Function {
  param: lone LocalVar,
  stmt: seq Stmt,
  returnType: one Type
}
```

For simplicity, all functions contain at most one parameter and we consider only the primitive types `int` and `float` in the specification. A function return type can be `void` or a primitive type. The statements of a function can be variable attributions (`VarAttrib`), a return statement (`Return`), local variable declarations (`LocalVarDecl`), or #ifdef declaration (`IfDef`). We have also considered programs with global variables and some C preprocessor directives such as `#define`, `#ifdef`, and `#endif`. Figure 2 specifies the UML class diagram of CDOLLY's meta-model.

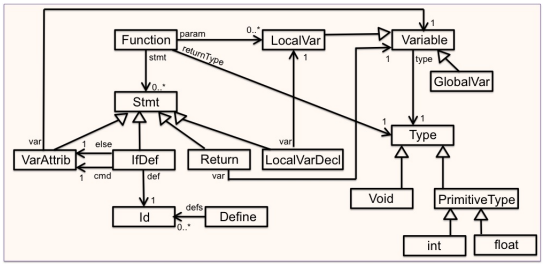

Fig. 2. UML class diagram of CDOLLY's meta-model.

2) Well-Formedness Rules: We specify well-formedness rules within Alloy facts. For example, the following fact specifies that if the return type of a function is not `void` the function must have a `Return` statement. The operator `&` denotes the set intersection operator.

```
fact WellFormednessRules {
  all f: Function |
    f·returnType ≠ Void ⇒
      #f·stmt·elems & Return = 1
  ...
}
```

We also specify some additional rules to cope with state explosion. For example, the predicate `optimization` does not allow functions with more than four statements. Moreover, all statements must be distinct. The relation `hasDups` yields whether there is some duplicate in the sequence.

```
pred optimization [] {
  all f: Function | #f·stmt < 5
  all f:Function | not f·stmt·hasDups
  ...
}
```

Similarly, we specified other elements of C's abstract syntax and other well-formedness rules. Notice that a sequence in Alloy may have a substantial impact in the number of Alloy instances generated by the Alloy Analyzer. Consider that a sequence may have at most `k` elements and `n` kinds of statements. The maximum number of valid sequences is $\sum_{i=0}^{k} n^i$. This number is multiplied by the number of all possible combinations of other elements of the specification.

C. Program Generation

We use this Alloy specification to generate C programs. Each Alloy instance found by the Alloy Analyzer is translated into a C program by CDOLLY. For example, on the left-hand side of Figure 3, the Alloy instance has one function containing a sequence of two statements (local variable declaration and variable assignment). CDOLLY translates this instance to the C program illustrated on the right-hand side of Figure 3. For variable assignments, CDOLLY randomly selects a value in a set of predefined values.

We can specify additional constraints to guide the program generation. For example, Listing 1 states that the generated programs must have at least one function (`func`) and one local variable (`varDecl`), which must be declared in a statement of `func`. The operator `in` denotes the set membership operator. We use a command to generate instances with at most two elements for each type declared in the C meta-model used by CDOLLY but `Stmt`, which can have up to three elements. Notice that the instances generated by the Alloy Analyzer may be used as input to test the Extract Function refactoring.

Listing 1. Specification to guide program generation.

```
one sig varDecl extends LocalVarDecl {}
one sig func extends Function {}
pred ExtractFunction {
  varDecl in func·stmt·elems
}
run ExtractFunction for 2 but 3 Stmt
```

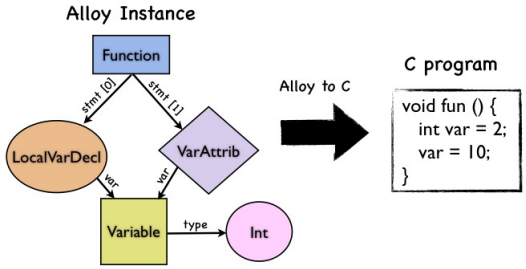

Fig. 3. Alloy instance and corresponding C program translated by CDOLLY.

D. Skipping Programs

By default, DOLLY exhaustively searches for all possible combinations yielded by the run command. Even for a small scope, DOLLY may generate thousands of programs specially when considering sequence of statements. However, the Alloy Analyzer may generate a number of similar consecutive instances [14]. Inspired on the STG technique [8], we allow developers to guide the program generation by skipping some instances. For a skip `n` greater than one, DOLLY generates one program from an Alloy instance, and jumps the following `n-1` Alloy instances. It follows this approach until the Alloy Analyzer does not generate more instances.

IV. EVALUATION

We evaluate our technique in 18 refactoring implementations of Java (Eclipse and JRRT) and C (Eclipse) with respect to time-consumption and bug detection. First, we present the experiment definition (Section IV-A) and planning

(Section IV-B). Then, Sections IV-C and IV-D present and discuss the results, respectively. Section IV-E describes some threats to validity and Section IV-F summarizes the main findings. Finally, we also compare our technique to other ones to test refactoring engines with respect to automation, oracles, exhaustiveness, setup and other characteristics in Section IV-G.

A. Definition

The goal of this experiment is to analyze our technique for the purpose of evaluating with respect to time consumption, bug detection, and kinds of refactorings that can be tested from the point of view of the developers of refactoring engines in the context of refactoring implementations for Java and C. In particular, we address the following research questions:

- **Q1** In what kinds of refactorings the proposed technique can detect bugs?
  We measure the number of bugs related to compilation errors, behavioral changes, and overly strong conditions for each kind of refactoring implementation.
- **Q2** What is the rate of time reduction and undetected bugs using skips in the technique?
  We measure the number of detected bugs and the total time to test the refactoring implementations without skip and using skips of 10 and 25 to generate programs. The total time includes the time to generate the programs, apply the transformations, and execute the automated oracles and bug categorizers.
- **Q3** What is the impact of using skip to generate programs on the time to find the first failure?
  We measure the time to find the first failure related to compilation error or behavioral change without skip and using skips of 10 and 25 to generate programs.

B. Planning

In this section, we describe the subjects used in the experiment and its instrumentation.

1) Selection of subjects: We tested 18 refactoring implementations of Java (Eclipse and JRRT [9]) and C (Eclipse). Eclipse is a widely used refactoring engine in practice and JRRT was proposed to improve the correctness of refactorings by using formal techniques. The evaluated refactorings focus on a representative set of program structures. We also evaluate all refactoring implementations of Eclipse CDT (C) but one: toggle function. This refactoring needs more than one C file to apply the refactoring, which is not supported by the current version of CDOLLY. Table I summarizes all evaluated refactorings for Java and C.

2) Instrumentation: We ran the experiment on a Desktop 3.0 GHz core i5 with 8 GB RAM running Ubuntu 12.04. We tested the refactoring implementations of Eclipse JDT 4.3, Eclipse CDT 8.1, and JRRT (03/feb/2013). We use SAFEREFACTOR [11] with the change impact analysis parameter activated to evaluate whether a transformation preserves the program behavior. This version of SAFEREFACTOR generates tests only for the methods impacted by the changes. We use SAFEREFACTOR for C to detect behavioral changes in C refactoring implementations. The time limit used by SAFEREFACTOR generates tests is 0.3 second. This time limit is enough to test transformations applied to small programs [11]. Finally, we use JDOLLY 0.2 and CDOLLY 0.1 to generate the programs.

We use the same Alloy specifications proposed before [3] as input parameter of JDOLLY to test the Java refactoring implementations. We use the scope of two packages, three classes, and at most two fields and three methods to JDOLLY generate the programs. We use the scope of two functions, two variables, two defines, and three statements to CDOLLY generate the programs. We define some additional constraints for guiding DOLLY to generate programs with certain characteristics needed to apply the refactoring. They prevent the generation of programs to which the refactoring under test is not applicable. For example, to test the Pull Up Field refactoring, the program must have at least two classes in a hierarchy, which a subclass contains a field that its super class does not contain. Another example, Listing 1 is used to generate programs to test Extract Function. Finally, the oracles save the results in files, which are used by the bug categorization module.

C. Results

JDOLLY and CDOLLY generated 96,129 compilable programs to evaluate all refactorings without skip. We use skips of 10 and 25 to reduce the set of generated programs. DOLLY generated 9,371 and 3,932 compilable programs to those skips, respectively (see Table I). The technique detects a total of 76 bugs (53 new bugs), which 49 bugs are related to compilation errors, 17 to behavioral changes, and 10 to overly strong conditions. When using skips of 10 and 25 the technique misses 3 bugs related to compilation error, behavioral change, and overly strong conditions in the Move Method refactoring.

Since we test newer versions of Eclipse JDT and JRRT comparing with our previous works [3], [2], we detect some new bugs that our previous works did not detect. But, we also detect some bugs that we have already detected (those bugs have not been fixed yet in the engines). In this work we show that the technique can be instantiated to test refactoring implementations of another language, such as C. Then, all bugs that we have detected in Eclipse CDT we reported as new bugs. We also reported the new bugs in the Java refactoring engines. Tables II, III, and IV summarize the results of the detected bugs related to compilation errors, behavioral changes, and overly strong conditions, respectively.

The total time to evaluate the 18 refactoring implementations without skip to generate programs is 61.61h. When using skips of 10 and 25, the technique takes 5.89h (90% of time reduction) and 2.34h (96% of time reduction), respectively. By using both skips the technique misses only 3.9% of the bugs. Table I summarizes the results related to the time to test the refactoring implementations.

We measure the TTFF in the refactoring implementations under test. The technique takes in general a few seconds to find the first failure, which can be related to compilation error or

TABLE I
SUMMARY OF THE NUMBER OF GENERATED PROGRAMS AND THE TIME TO EVALUATE THE REFACTORING IMPLEMENTATIONS.

Refactoring		Generated Programs			Time: Behavioral Change and Compilation Errors (h)						Time: Overly Strong Conditions (h)		
					Eclipse			JRRT					
		No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25
Java	Rename Method	8,634	809	361	4.63	0.45	0.18	4.19	0.42	0.15	2.53	0.25	0.09
	Push Down Field	9,117	870	390	4.32	0.42	0.17	1.75	0.17	0.05	0	0	0
	Pull Up Field	8,712	867	345	5.9	0.58	0.24	3.01	0.28	0.10	0	0	0
	Encapsulate Field	1,831	175	68	1.55	0.14	0.05	1.30	0.12	0.06	0.46	0.04	0.01
	Move Method	14,814	1,293	624	8.6	0.72	0.29	6.16	0.63	0.27	5.08	0.58	0.23
C	Rename Global Variable	2,040	198	86	0.61	0.06	0.02	-	-	-	-	-	-
	Rename Local Variable	7,359	786	314	2.15	0.17	0.07	-	-	-	-	-	-
	Rename Define	1,034	101	38	0.23	0.04	0.01	-	-	-	-	-	-
	Rename Function	13,188	1,327	531	3,56	0.22	0.09	-	-	-	-	-	-
	Rename Parameter	5,964	587	233	1.54	0.13	0.06	-	-	-	-	-	-
	Extract Function	7,812	786	314	2.10	0.12	0.05	-	-	-	-	-	-
	Extract Local Variable	7,812	786	314	2.95	0.18	0.07	-	-	-	-	-	-
	Extract Constant	7,812	786	314	2.55	0.17	0.08	-	-	-	-	-	-
Total		**96,129**	**9,371**	**3,932**	**37.13**	**3.40**	**1.38**	**16.41**	**1.62**	**0.63**	**8.07**	**0.87**	**0.33**

TABLE II
SUMMARY OF THE DETECTED BUGS RELATED TO COMPILATION ERRORS.

Refactoring		Eclipse						JRRT					
		Failures			Bugs			Failures			Bugs		
		No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25
Java	Rename Method	595	68	19	1	1	1	0	0	0	0	0	0
	Push Down Field	342	37	15	2	2	2	0	0	0	0	0	0
	Pull Up Field	518	59	26	1	1	1	0	0	0	0	0	0
	Encapsulate Field	238	22	12	1	1	1	0	0	0	0	0	0
	Move Method	214	22	9	2	2	2	3	0	0	1	0	0
C	Rename Global Variable	907	91	33	4	4	4	-	-	-	-	-	-
	Rename Local Variable	3,274	326	129	4	4	4	-	-	-	-	-	-
	Rename Define	391	41	21	4	4	4	-	-	-	-	-	-
	Rename Function	6,165	627	245	5	5	5	-	-	-	-	-	-
	Rename Parameter	2,681	262	108	4	4	4	-	-	-	-	-	-
	Extract Function	5,745	583	231	7	7	7	-	-	-	-	-	-
	Extract Local Variable	3,880	393	157	5	5	5	-	-	-	-	-	-
	Extract Constant	4,168	415	158	8	8	8	-	-	-	-	-	-
Total		**29,118**	**2,946**	**1,163**	**48**	**48**	**48**	**3**	**0**	**0**	**1**	**0**	**0**

behavioral change (see Table V). In some refactorings, such as Push Down Field and Encapsulate Field, it takes some minutes to find the first failure without skip. In case of no failure, we show "n/a."

TABLE V
SUMMARY OF THE TIME TO FIND THE FIRST FAILURE (TTFF).

Refactoring		Eclipse			JRRT		
		TTFF (min)			TTFF (min)		
		No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25
Java	Rename Method	1.00	0.15	0.68	3.54	2.10	1.74
	Push Down Field	6.95	1.03	0.65	n/a	n/a	n/a
	Pull Up Field	0.13	1.05	0.51	n/a	n/a	n/a
	Encapsulate Field	2.60	1.51	1.03	n/a	n/a	n/a
	Move Method	0.17	0.62	0.61	0.25	1.05	0.58
C	Rename Global Var.	0.10	0.05	0.02	-	-	-
	Rename Local Var.	0.02	0.12	0.05	-	-	-
	Rename Define	0.18	0.2	0.01	-	-	-
	Rename Function	0.06	0.01	0.08	-	-	-
	Rename Parameter	0.20	0.05	0.11	-	-	-
	Extract Function	0.09	0.02	0.01	-	-	-
	Extract Local Var.	0.12	0.04	0.02	-	-	-
	Extract Constant	0.03	0.14	0.01	-	-	-

D. Discussion

Now we discuss issues of the detected bugs related to compilation errors, behavioral changes, and overly strong conditions, the time to test the refactoring implementations, and the refactorings applied within function level.

1) Compilation Errors: A total of 29,118 transformations applied by Eclipse failed due to compilation errors without skip. Those failures were categorized in 48 bugs (42 new bugs). Among the new bugs, the C refactoring implementations have 41 and the Java ones have 1 new bug. We missed no bugs in the C refactoring implementations using the skips of 10 and 25. In this context, there is a high number of failure transformations regarding compilation errors in such implementations. For example, the Eclipse CDT does not check whether the new names are keywords and, then introduces compilation errors when applying the transformation. JRRT applied only three transformations (Move Method) with compilation errors. These failures were categorized in one bug.

2) Behavioral Changes: JRRT applied 6,320 behavioral changes transformations. We categorized them in three distinct

TABLE III
SUMMARY OF THE DETECTED BUGS RELATED TO BEHAVIORAL CHANGES.

Refactoring		Eclipse						JRRT					
		Failures			Bugs			Failures			Bugs		
		No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25
Java	Rename Method	0	0	0	0	0	0	286	36	8	1	1	1
	Push Down Field	71	5	4	1	1	1	0	0	0	0	0	0
	Pull Up Field	690	64	29	4	4	4	0	0	0	0	0	0
	Encapsulate Field	0	0	0	0	0	0	0	0	0	0	0	0
	Move Method	2,983	295	126	3	2	2	6,034	592	250	2	2	2
C	Rename Global Variable	107	12	4	1	1	1	-	-	-	-	-	-
	Rename Local Variable	228	48	11	1	1	1	-	-	-	-	-	-
	Rename Define	0	0	0	0	0	0	-	-	-	-	-	-
	Rename Function	0	0	0	0	0	0	-	-	-	-	-	-
	Rename Parameter	133	17	3	1	1	1	-	-	-	-	-	-
	Extract Function	659	71	28	1	1	1	-	-	-	-	-	-
	Extract Local Variable	701	66	41	1	1	1	-	-	-	-	-	-
	Extract Constant	597	63	25	1	1	1	-	-	-	-	-	-
Total		**6,169**	**641**	**271**	**14**	**13**	**13**	**6,320**	**628**	**258**	**3**	**3**	**3**

TABLE IV
SUMMARY OF THE DETECTED BUGS RELATED TO OVERLY STRONG CONDITIONS.

Refactoring		Eclipse						JRRT					
		Rejected Behavior Pres. Transf.			Overly Strong Conditions			Rejected Behavior Pres. Transf.			Overly Strong Conditions		
		No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25	No skip	Skip 10	Skip 25
Java	Rename Method	3,590	362	140	3	3	3	0	0	0	0	0	0
	Push Down Field	0	0	0	0	0	0	0	0	0	0	0	0
	Pull Up Field	0	0	0	0	0	0	0	0	0	0	0	0
	Encapsulate Field	642	68	24	1	1	1	0	0	0	0	0	0
	Move Method	2,056	226	94	4	4	4	90	9	2	2	1	1
Total		**6,288**	**656**	**258**	**8**	**8**	**8**	**90**	**9**	**2**	**2**	**1**	**1**

bugs. Eclipse JDT applied 3,744 transformations that change the program behavior while Eclipse CDT applied 2,425 ones. We found 14 distinct bugs related to behavioral changes in the refactorings of Eclipse. Among those bugs, we found six new bugs in Eclipse CDT. For example, we can extract the assign of a local variable to a function with a name starting with * in Eclipse CDT. This introduces a behavioral change. We found a new bug in the Pull Up Field of Eclipse JDT. The transformation enables a field to hide another field, which changes the program behavior.

3) Overly Strong Conditions: We found eight bugs (two new ones) related to overly strong conditions in Eclipse JDT and two new bugs in JRRT. We found new bugs in the Move Method refactorings of Eclipse and JRRT. Listing 2 shows the original program generated by JDOLLY and Listing 3 illustrates the resulting program after the Move Method refactoring of Eclipse JDT. The transformation moved the method `m(int)` from class `B` to class `A`. This transformation does not change the program behavior. All methods of the program result the same value before and after the transformations. JRRT rejects applying this transformation and reports the following warning: `cannot adjust accessibilities`.

4) Time: Our technique takes 61.61 hours to evaluate all refactoring implementations under test without skips. DOLLY generates a number of similar programs that may increase the time for testing refactoring engines and potentially detect the same kind of bug. We have observed that similar programs tend to be generated consecutively by DOLLY. Then, to alleviate this problem we implemented a feature that allows skipping consecutive test inputs.

Listing 2. Before Refactoring.
```
class A {}
class B extends A {
  A f = null;
  long m(int a) {
    return 0;
  }
}
class C extends B {
  long m(int a) {
    return 1;
  }
  long k() {
    return super.m(2);
  }
}
```

Listing 3. After a Move Method Refactoring.
```
class A {
  long m(int a) {
    return 0;
  }
}
class B extends A {
  A f = null;
}
class C extends B {
  long m(int a) {
    return 1;
  }
  long k() {
    return super.m(2);
  }
}
```

When using skips, the refactoring engine developer can detect a number of bugs in a few hours. For example, the technique evaluated 18 refactoring implementations and detects 73 bugs in 2.34 hours using a skip of 25 to generate programs. The developer can run the technique again without skipping while fixing the detected bugs in order to find some missed bugs. Moreover, we can reduce even more the idle time of the developer. The technique finds the first failure in

Listing 4. Before Refactoring.

```
void func() {
  int var = 1; ...
}
```

Listing 5. After a Extract Constant Refactoring.

```
static const int one = 1;
void func() {
  int var = one; ...
}
```

the refactoring implementations in a few seconds using a skip of 10 or 25. When there are many failures transformations in a refactoring implementation, the TTFF is similar even varying the skip to generate programs. So, the developer can find a bug in a few seconds, fix the bug, run it again to find another bug, and so on. By using this strategy, the bug categorization step is no longer needed since there is only one failure in each execution. Before a new release, the developer can run the technique without skip to improve confidence that the implementation is correct.

5) Refactorings Applied within Function Level: The statements variable assignment and local variable declaration, specified in the C meta-model of CDOLLY, enable testing refactorings applied within function level, such as Extract Constant, Extract Function, Extract Local Variable, and Rename Local Variable. For example, Figure IV-D5 illustrates an Extract Constant refactoring applied within a variable assignment statement.

E. Threats to Validity

Next we present the threats to validity of our evaluation.

1) Construct Validity: We reported all bugs found by our technique. Developers accepted some of them and marked others as duplicate or new bugs.

2) Internal Validity: We specify some additional constraints in Alloy for guiding the program generation to each kind of refactoring. Those constraints aim to generate programs with certain characteristics needed to apply the refactoring. It also avoids a state explosion of Alloy instances. However, the additional constraints may hide possibly detectable bugs. We categorize the bugs related to compilation errors and overly strong conditions by splitting the failing tests based on messages from the engine. We classify behavioral changes based on the program's structure. However, we can miss some bugs if the engine reports the same message to different kinds of bugs. Developers can mitigate this threat when they execute the technique a number of times after fixing the bugs.

3) External Validity: We have only considered a subset of Java and C and a small scope to generate programs. Some of the generated programs may be artificial. We cannot assert that all bugs actually happen in practice. Nevertheless, the technique is useful to warn developers about some overly weak and strong preconditions in the refactoring implementations.

F. Answer to the Research Questions

We now answer our research questions.

- **Q1** In what kinds of refactorings the proposed technique can detect bugs?
  The technique detects a total of 76 bugs related to compilation errors, behavioral changes, and overly strong conditions in 18 refactorings applied above method/function level and within function level. Among those bugs, it detects 28 bugs in the refactorings applied within function level. We improve our previous technique [3] in order to test more kinds of refactoring implementations, such as refactorings applied within function level. Yet, to apply some kinds of refactorings the program must have some language constructs currently not supported by the program generator.
- **Q2** What is the rate of time reduction and undetected bugs using skips in the technique?
  When using skips of 10 and 25, the technique takes 5.89h (90% of time reduction) and 2.34h (96% of time reduction), respectively. By using both skips the technique misses only 3.9% of the bugs.
- **Q3** What is the impact of using skip to generate programs on the time to find the first failure?
  The technique reduces on average 47% and 60% (it takes a few seconds) the time to find the first failure using skips of 10 and 25, respectively.

G. Comparison with other techniques

Next, we compare our technique with other related techniques to test refactoring engines.

Daniel et al. [4] proposed an approach for automated testing refactoring engines. The technique is based on ASTGEN, a Java program generator, and a set of programmatic oracles. However, their oracles can only syntactically compare the programs to detect behavioral changes. So, they found only one bug related to behavioral change in Eclipse JDT. The other bugs are related to compilation errors and engine crashes. We found 17 bugs related to behavioral change in 18 refactoring implementations of JRRT and Eclipse.

Jagannath et at. [8] presented the STG technique to reduce the costs of bounded-exhaustive testing by skipping some test inputs. They randomly select a skip up to 20 after generating each program. They evaluated it using ASTGEN and found that the technique took some seconds to find the first failure related to compilation error or engine crash in the refactoring implementations using STG. We also included the skip parameter in DOLLY to reduce the time to test the refactoring implementations and to find the first failure, which can be related to compilation error or behavioral change. Different from them, we use a fixed skip that is set by the user. As the results are deterministic, we can execute the tests again using the same skip to evaluate whether we have already fixed the bugs. Moreover, we can execute using a different skip to find some missed bugs. Finally, they do not measure the rate of missed bugs using skips to generate programs different from our work.

Later, Gligoric et al. [5] proposed UDITA, a Java-like language that extends ASTGEN allowing users to describe properties in UDITA using any desired mix of filtering and generating style in opposed to ASTGEN that uses a purely

generating style. UDITA evolved ASTGEN to be more expressive and easier to use, usually resulting in faster program generation as well. They found four new bugs related to compilation errors in Eclipse in a few minutes. However, the technique requires substantial manual effort for writing test generators [6], since they are specified in a Java-like language. Soares et al. [3] found that UDITA does not generate some programs that JDOLLY generates using the same scope and without skipping.

More recently Gligoric et at. [6] used real systems to reduce the effort for writing test generators using the same oracles [5]. They found 141 bugs related to compilation errors in refactoring implementations for Java and C in 285 hours. However, the technique may be costly to apply the refactoring in large systems and to minimize the failure into a small program to categorize the bugs. Moreover, evaluating transformations on large real programs is time consuming, and it would produce less accurate results. We can use SAFEREFACTOR to automatically detect behavioral changes in their technique. SAFEREFACTOR detected behavioral transformations applied on real systems that even a well defined manual inspection conducted by experts did not detect [15], [11].

In our previous work [2], [3] we proposed a technique to test refactoring engines by detecting bugs related to compilation errors, behavioral changes, and overly strong conditions. It is based on JDOLLY, an exhaustive program generator, a set of automated oracles, such as SAFEREFACTOR [12], and differential testing to identify overly strong conditions. As opposed to ASTGEN and UDITA that use a Java-like language, JDOLLY only needs to declaratively specify the structures of the programs. However, it may be costly to evaluate all test inputs. It took a total of 590 hours to detect 106 bugs related to compilation errors and behavioral changes in 39 refactoring implementations. Moreover, the technique does not test refactorings applied within method level. In this work, we optimize the technique to reduce the costs of testing. For example, using a skip of 25 in the program generator, it reduces in 96% the time to test the refactoring implementations while missing only 3.9% of the bugs. We also extend the technique to improve expressiveness of the program generator and show that it is also useful to test refactoring implementations in C language. We found 28 bugs in refactorings applied within function level. Our current technique may allow developers to run it more often during the refactoring implementation.

Table VI summarizes the comparison among our technique and the main techniques to test refactoring engines. The column Automation indicates whether the technique is automated in generating test inputs and categorizing the bugs. In the column Oracle we use *yes* whether the technique implements the oracle and *no* otherwise. In the BC (behavioral change) column we also differentiate the oracles by the kind of approach used to detect behavioral change, which can be *syntactic* or *dynamic*. The column Refactoring represents the kinds of refactorings supported by the techniques. Finally, the column Test Input indicates whether the generation or selection of the test inputs are exhaustive, and the effort to setup the program generator or to select the test inputs.

V. RELATED WORK

Opdyke [16] coined the term refactoring. He proposed a number of refactorings for C++ and specified conditions to guarantee behavior preservation. However, there was no formal proof of the correctness and minimality of these conditions. Later, Tokuda and Batory [17] showed that Opdyke's notion was not sufficient to ensure preservation of behavior.

Garrido and Johnson [18], [19] proposed CRefactory, a refactoring engine for C. They specified a set of refactoring preconditions that support programs in the presence of conditional compilation directives and implemented the refactorings. However, they did not prove them sound. We can use our technique to test their refactoring implementations.

Steimann and Thies [20] proposed a constraint-based approach to specify Java accessibility. It is useful for detecting bugs regarding accessibility-related properties. Schäfer et al. [9] implemented a number of Java refactoring implementations in JRRT. They aim to improve correctness of the refactoring implementations of Eclipse. They included some of the results presented by Steimann and Thies [20]. We evaluated five JRRT implementations and found some bugs.

Borba et al. [21] proposed a set of refactorings for a subset of Java with copy semantics. They proved them to sound with respect to a formal semantics. Later, Silva et al. [22] extended their work to consider an OO language with reference semantics. However, they have not considered all Java constructs, such as overloading and field hiding. Developers may use their templates to implement the refactorings and use our technique to test the refactoring implementations.

Li and Thompson [23] proposed a technique to test refactorings using a tool, called Quvid QuickCheck, for Erlang. They evaluated a number of implementations of the Wrangler refactoring engine. For each refactoring, they state a number of properties that it must satisfy. If a refactoring applies a transformation but does not satisfy a property, they indicate a bug in the implementation. They found four bugs. We use SAFEREFACTOR to evaluate behavior preservation. Our technique uses a similar approach for testing refactorings for Java and C. Their approach applies refactorings to a number of real case studies and toy examples. In contrast, we apply refactorings to a number of programs generated by DOLLY.

Vakilian and Johnson [24] presented a technique to detect usability problems in refactoring engines. It is based on refactoring alternate paths. They adapt critical incident technique to refactoring tools and show that alternate refactoring paths are indicators of the usability problems of refactoring tools. Their technique manually found two usability problems related to overly strong conditions. We use SAFEREFACTOR to evaluates whether the applied transformation is behavior preserving. Our technique automatically found 10 bugs related to overly strong conditions in Eclipse JDT and JRRT.

Rachatasumrit and Kim [25] found that refactorings are not well tested in real systems. The existing regression test suites

TABLE VI
COMPARISON BETWEEN TECHNIQUES TO TEST REFACTORING ENGINES. CE = COMPILATION ERRORS; BC = BEHAVIORAL CHANGES; OS = OVERLY STRONG CONDITIONS.

Technique	Automation				Oracle			Refactorings	Test Input (program)	
	Test Inputs	Bug Categorization								
		CE	BC	OS	CE	BC	OS		Exhaustiveness	Setup Effort
Gligoric et al. [ICSE 2010]	Yes	Yes	No	No	Yes	Yes, syntactic	No	Above/within met. level	Yes	High
Soares et al. [TSE 2013]	Yes	Yes	No	Yes	Yes	Yes, dynamic	Yes	Above met. level	Yes	Medium
Gligoric et al. [ECOOP 2013]	No	Yes	No	No	Yes	Yes, syntactic	No	All	No	Medium
Our proposed technique	Yes	Yes	Yes	Yes	Yes	Yes, dynamic	Yes	Above/within function level	Yes	Medium

may not cover the impacted entities, and a number of test cases may not be relevant for testing the refactorings. In our previous work, we evolved SAFEREFACTOR to generate tests only for the methods impacted by the change [11]. It generates only relevant tests that exercise at least one entity impacted by the change. The change impact analysis performed by SAFEREFACTOR also reduced the time to test the refactoring implementations comparing with our previous work [3].

VI. CONCLUSIONS

In this work we propose a technique to scale testing of refactoring engines. We extend our previous technique [3] by improving expressiveness of the program generator to enable testing refactorings applied within function level. We also optimize it to reduce the time to test the refactoring implementations by skipping some consecutive test inputs. We evaluate it on testing 18 kinds of refactorings implemented by Eclipse JDT, Eclipse CDT, and JRRT. We found 76 (53 new bugs) bugs in a total of 49 bugs related to compilation errors, 17 bugs related to behavioral changes, and 10 bugs related to overly strong conditions. Among those bugs, we found 28 bugs in refactorings applied within function level. Moreover, using a skip of 25 in the program generator, it reduces in 96% the time to test the refactoring implementations while missing only 3.9% of the bugs.

In our study, the technique finds the first failure in a few seconds. The refactoring engine developer can fix the bug and run it again while implementing the transformation. Since it takes less than an hour to test each implementation using a skip of 10, developers may also run it a couple of times to find the bugs missed using a different skip. Before a release, developers may run the technique without skipping instances to improve confidence that the implementations are correct.

As a future work we aim at proposing some automated oracles to detect other kinds of bugs in refactoring implementations. We also intend to improve the skip mechanisms to reduce the rate of missed bugs. Finally, we aim at specifying more language constructs in DOLLY and testing other kinds of refactoring implementations.

REFERENCES

[1] M. Schäfer, T. Ekman, and O. de Moor, "Challenge proposal: verification of refactorings," in *PLPV*, 2008, pp. 67–72.

[2] G. Soares, M. Mongiovi, and R. Gheyi, "Identifying overly strong conditions in refactoring implementations," in *ICSM*, 2011, pp. 173–182.

[3] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Transactions on Software Engineering*, vol. 39, pp. 147–162, 2013.

[4] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *FSE*, 2007, pp. 185–194.

[5] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *ICSE*, 2010, pp. 225–234.

[6] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov, "Systematic testing of refactoring engines on real software projects," in *ECOOP*, 2013, pp. 629–653.

[7] D. Jackson, *Software Abstractions: Logic, Language, and Analysis. Revised edition*. The MIT Press, 2012.

[8] V. Jagannath, Y. Lee, B. Daniel, and D. Marinov, "Reducing the costs of bounded-exhaustive testing," in *FASE*, 2009, pp. 171–185.

[9] M. Schäfer and O. Moor, "Specifying and implementing refactorings," in *OOPSLA*, 2010, pp. 286–301.

[10] D. Jackson, I. Schechter, and H. Shlyahter, "Alcoa: the Alloy constraint analyzer," in *ICSE*, 2000, pp. 730–733.

[11] M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, and P. Borba, "Making refactoring safer through impact analysis," *SCP*, 2014, In press.

[12] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making program refactoring safer," *IEEE Software*, vol. 27, pp. 52–57, 2010.

[13] W. Mckeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[14] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *TACAS*. Wiley, 2007, pp. 632–647.

[15] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson, "Comparing Approaches to Analyze Refactoring Activity on Software Repositories," *JSS*, pp. 1006–1022, 2013.

[16] W. Opdyke, "Refactoring Object-Oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[17] L. Tokuda and D. Batory, "Evolving object-oriented designs with refactorings," *ASE*, vol. 8, pp. 89–120, 2001.

[18] A. Garrido and R. Johnson, "Refactoring C with conditional compilation," in *ASE*, 2003, pp. 323–326.

[19] A. Garrido and R. E. Johnson, "Analyzing multiple configurations of a C program," in *ICSM*, 2005, pp. 379–388.

[20] F. Steimann and A. Thies, "From public to private to absent: Refactoring Java programs under constrained accessibility," in *ECOOP*, 2009, pp. 419–443.

[21] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio, "Algebraic reasoning for Object-Oriented programming," *SCP*, vol. 52, pp. 53–100, 2004.

[22] L. Silva, A. Sampaio, and Z. Liu, "Laws of Object-Orientation with reference semantics," in *SEFM*, 2008, pp. 217–226.

[23] H. Li and S. Thompson, "Testing Erlang Refactorings with QuickCheck," in *IFL*, 2008, pp. 19–36.

[24] M. Vakilian and R. E. Johnson, "Alternate refactoring paths reveal usability problems," in *ICSE*, 2014, pp. 1–11.

[25] N. Rachatasumrit and M. Kim, "An empirical investigation into the impact of refactoring on regression testing," in *ICSM*, 2012, pp. 357–366.