Detecting and Classifying Nonconformances in Code Contracts with CONTRACTOK

Alysson F. Milanez
Department of Engineering and Technology
Federal Rural University of the Semi-Arid
(UFERSA)
Pau dos Ferros, Brazil
alysson.milanez@ufersa.edu.br

Tiago L. Massoni
Department of Computing Systems
Federal University of Campina Grande
(UFCG)
Campina Grande, Brazil
massoni@dsc.ufcg.edu.br

Rohit Gheyi
Department of Computing Systems
Federal University of Campina Grande
(UFCG)
Campina Grande, Brazil
rohit@dsc.ufcg.edu.br

***Abstract*—Nonconformances, in the context of contract-based programs, must be detected and corrected. Classification may be useful in the process of nonconformances correction. Current approaches do not support any type of nonconformance classification. In this work, we present a dynamic approach (CONTRACTOK) for detecting and classifying nonconformances in the context of Code Contracts programs. The approach is based on random test generation for nonconformances detection and on heuristics for classification. We evaluate our approach in four real programs, summing up 82.8K lines of C# and Code Contracts, detecting and classifying 16 nonconformances.**

***Keywords*—contract-based programs, nonconformances, classification.**

I. INTRODUCTION

In contract-based programs (such as the Code Contracts [1]), early detection of nonconformances is desirable for providing a more reliable account of systems correctness [2]. Developers tend to apply automated, although incomplete, approaches as verification by formal proofs is hard to scale.

For Code Contracts, there is Clousot [3] a static approach for conformance checking. However, this approach does not present support for nonconformances classification. Nonconformances classification may be useful by giving a first step for the developer in the process of nonconformances correction [4].

In this paper, we present the initial version of CONTRACTOK tool, a dynamic approach for detecting and classifying nonconformances in C#/Code Contracts programs. The tool applies randomly-generated tests for nonconformances detection and a heuristics-based approach for classification (Section III). We performed a case study with four real C# with Code Contracts projects, summing up 109,6 KLOC and we were able to detect and classify 16 nonconformances (Section IV).

This work is structured as follows. Section II presents an example of a C#/Code Contracts program and section III describes CONTRACTOK tool and the heuristics for nonconformances classification. Then, section IV shows the case study performed for evaluating the tool. Next, section V presents the main related work. Finally, in section VI we summarize the main findings of this work and discuss prospects for future works.

II. MOTIVATING EXAMPLE

C# class `Division` (Listing 1) declares a constructor and two methods: one for contract purposes — visibility is omitted, for simplicity. Code Contracts method contracts are expressed by means of calls to static methods `Contract.Requires(...)` and `Contract.Ensures(...)`, specifying pre- and postconditions, respectively. The `Contract.Result<double>()` used in the postcondition refers to the return value of `div` method.

Listing 1. Class Division
```
class Division{
  Division() {
    this._count = 1;
  }
  double div(int x, int y) {
    Contract.Requires(y != 0);
    Contract.Ensures(Contract.Result<double>() * y == x);
    return x/y;
  }
}
```

The class is not in conformance with its contracts, as it presents one nonconformance that can be detected with a call to `div` method with a value for `x` not multiple of the value for `y` (like calling `div` with 1 and 6 as parameters). Nonconformances between contract and implementation may be subtle to detect.

Regardless of where the bug is located (contract or code, or both), the failure may only arise within specific values used as parameter for the method, as presented in Listing 2. Method `div` has a precondition and its body seems correct, so the likely cause suggested is *Strong Postcondition*. This problem may be solved by adding a tolerance value to the expression in the postcondition of `div`.

Listing 2. A test case for Division class
```
Division d = new Division();
d.div(1, 6);
```

III. CONTRACTOK

In this work, we present CONTRACTOK[1], an approach for detecting and classifying nonconformances in C# programs

[1]https://github.com/alyssonfm/contractok

with Code Contracts. Our approach follows the same structure presented by JMLOK2 [5]: tests are automatically generated – by using an adapted version of Randoop.NET [6] – and executed, comparing the test results with oracles (generated from the contracts).

The generated tests are composed of sequences of calls to methods and constructors under test, while the test oracles are assertions from the contracts, generated from Code Contracts. After test execution, a filter distinguishes faults from the returned failures – those faults make up the nonconformances subject to classification process.

Regarding nonconformance classification we use an adapted version of the set of heuristics from Milanez [4] that suggest likely causes for three types of nonconformances namely *precondition*, *postcondition*, and *invariant*. The context of a nonconformance is always a call to a method m within class C, as we only consider contracts whose assertions are checked before or after a method call.

The primary test for a *precondition* type is to verify whether any expression in the contract involves a parameter of m, or a field declared in C. If positive, the heuristic suggests *Strong Precondition*, as its boolean expressions includes tests to the method input. Such decision might evidently be inaccurate, as, even though the precondition tests a given parameter or field, those tests may really be relevant, and the program calling m may be failing to supply the correct state used as input to m.

Although a more complex heuristic could speculate on such scenarios, our main purpose is to provide a suggestion to direct experts to a limited scope of error — lacking an automatic suggestion could mean more disparate answers from them. Otherwise, if a parameter or field is untested in the precondition, we suggest *Weak Postcondition*, assuming that the program calling m called a previous method k that incorrectly set the state that is not accepted by m's precondition. The postcondition considered weak is then from k.

For the *postcondition* type, there may be a default precondition, or at least one field modified by m; in either case, the likely cause suggested is *Weak precondition*. It is assumed that any input is allowed, which might cause problems by the end of the method's execution – m can be unable to produce the desired result. Else, *Strong Postcondition* is suggested. For the *invariant* type, from a default precondition, or at least one field modified within m, the likely cause is *Weak Precondition* – nothing was checked in the contract against potential changes to the fields. If none of the previous cases is observed, *Strong Invariant* is suggested.

IV. Case Study

In this section, we present a case study performed in order to evaluate our approach. This case study was used as a proof of concept of ContractOk tool capabilities of detecting and classifying nonconformances in C#/Code Contracts programs. More details are available online.[2]

[2]https://sites.google.com/a/copin.ufcg.edu.br/contractok/case-study

TABLE I
Experimental units' summary. Column KLOC shows the code size of each experimental unit in terms of code lines. Column #CC presents the total of contract clauses of each experimental unit.

Experimental Unit	**LOC**	**#CC**
AutoDiff	1.3	157
Boogie	67	5,214
DBExecutor	2	12
Mishra Reader	12.5	20
Total	**82.8**	**5,403**

A. Programs

The C# - Code Contracts systems collected for this study were taken from Microsoft Research's site.[3] They amount to 82.8 K lines of C# and 5.4 Code Contracts clauses (#CC henceforth): `AutoDiff`, `Boogie`, `DbExecutor`, and `Mishra Reader`.

While `AutoDiff` [7] is a library for automatic differentiation of mathematical functions and `Boogie` [8] is an Intermediate Verification Language (IVL) for proof obligations solved by reasoning engines, `DBExecutor`[4] is a simple and Lightweight Database Executor for .NET 4 Client Profile and all ADO.NET DbProviders(SQL Server, SQLCE, SQLite, SQL Azure, Entity SQL, MySql, Oracle, etc...), and `Mishra Reader`[5] is a Google Reader client developed using WPF with focus on ergonomy and smooth animations.

These systems are listed in Table I. The systems are characterized in terms of KLOC - thousands of code lines and contract clauses (#CC).

B. Results

ContractOk was able to detect 16 nonconformances in four real projects: one nonconformance in `AutoDiff` and 15 in `Boogie`. The others experimental units (`DbExecutor` and `Mishra Reader`) have not presented nonconformances, considering a time limit for tests generation in range [10, 100] seconds. Table II presents the detected nonconformances with their classification.

C. Discussion

As we can see in Table II, ContractOk was able to detect and classify 16 nonconformances in real projects: one nonconformance in `AutoDiff` and 15 in `Boogie`. This result might be related to the kind of contracts that the experimental units have and/or due to the random-based approach for tests generation (Randoop.NET [6]) used by ContractOk.

The nonconformance discovered in `AutoDiff` occurs because the `Sum` method (from `TermBuilder` class) has several checks on the list received as parameter.

Following our set of heuristics (Section III), our tool classifies this nonconformance as *Strong Precondition* because the expressions in the contract involve a parameter of the

[3]http://research.microsoft.com/en-us/projects/contracts/
[4]https://archive.codeplex.com/?p=dbexecutor
[5]https://archive.codeplex.com/?p=mishrareader

TABLE II
NONCONFORMANCES DETECTED AND CLASSIFIED IN C# SYSTEMS.

Unit	Nonconformances			
	Class	**Method**	**Type**	**Likely Cause**
AutoDiff	TermBuilder	Sum	precondition	Strong Precondition
Boogie	CommandLineOptions	AddZ3Option	invariant	Strong Invariant
		set_LogPrefix	invariant	Strong Invariant
		Usage	invariant	Strong Invariant
	Expr	StoreTok	precondition	Strong Precondition
	GenKillWeight	Constructor	invariant	Weak Precondition
	HelperFuns	BoogieFunction	precondition	Strong Precondition
	VCExprBinary	Constructor	precondition	Strong Precondition
	VCExprBvExtractOp	Constructor	precondition	Strong Precondition
	VCExpressionGenerator	Exists	precondition	Strong Precondition
		Forall	precondition	Strong Precondition
	VCExprMultiAry	Constructor	precondition	Strong Precondition
	VCExprNullary	Constructor	precondition	Strong Precondition
	VCExprSelectOp	Constructor	precondition	Strong Precondition
	VCExprStoreOp	Constructor	precondition	Strong Precondition
	VCExprUnary	Constructor	precondition	Strong Precondition

method. Concerning to `Boogie`, CONTRACTOK detected 11 *precondition* problems and four problems of *invariant*.

All *precondition* problems occurred as a result of the several checks on the methods' parameters, so our heuristics suggests *Strong Precondition* as likely cause; and the *invariant* problems occurred due to the *Strong Invariant* of the classes and one case (class `GenKillWeight`) due to the *Weak Precondition* of the constructor.

D. Threats to validity

The randomness promoted by the use of an automatic test generator (Randoop [6]) by the tools is an internal threat to the validity of our study; so, we ran each system 10 times for each time limit (varying from 10 to 100 seconds) for confidence. In each execution, tests are generated independently of the previous run, which may show different contexts revealing the same nonconformances.

In the context of external validity, our results are only valid in the experimental units considered; in others units these results may change considerably. In addition, a Randoop-based [6] approach lacks repeatability, in terms of machine setting or operating system.

V. RELATED WORK

For DBC, a related approach proposes auto tests [9], in which contracts are used as oracles to outputs, with automatic test generation. AutoTest is an implementation of conformance checking to the Eiffel language [10]. This tool is similar to CONTRACTOK both aim at conformance checking, and use randomly-guided tests generation. However, AutoTest supports mixing manual and automated tests, while our approach focuses on complete automation.

Concerning to Java/JML programs, Burdy et al. [11] list tools for detecting nonconformances in those kind of programs. Moreover, CONTRACTOK applies ideas from JMLOK2 [5]: both tools aim to detect nonconformances in contract-based programs. It is intricate to compare CONTRACTOK with similar JML-based tools; JML [12] offers a much more complete foundation for formal proofs and detailed analysis, when compared to Code Contracts.

Regarding Code Contracts, Clousot [3] performs static analysis over contract-based programs. CONTRACTOK is based on dynamic analysis; they could definitely be combined for a unified approach to fill the gaps of each tool alone (false negatives in dynamic analysis and false positives in static analysis).

VI. CONCLUSIONS

In this work, we presented an approach for detecting and classifying nonconformances in C# – Code Contracts programs. In our experimental study CONTRACTOK detected 16 nonconformances in two real systems: one nonconformance was detected in `AutoDiff` and 15 nonconformances in `Boogie`.

As future work, we intend to perform new experimental studies for evaluating our approach with other experimental units, in addition we plan to perform a comparison between CONTRACTOK and Clousot [3]. Other interest for future work is to perform a study concerning the suggestion of fixes to the detected nonconformances.

We also plan to improve the set of heuristics by means of approaches such as Data Flow [13] or Control Flow Analysis [14]. We believe some static technique might improve the heuristics and give better results for the classification, and more comprehensive experiments can be performed.

ACKNOWLEDGMENT

This work was supported by the National Institute of Science and Technology for Software Engineering (INES[6]), funded by CNPq, grant 573964/2008-4.

REFERENCES

[1] M. Barnett, M. Fahndrich, and F. Logozzo, "Embedded contract languages," in *ACM SAC - OOPS*. ACM, March 2010.
[2] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall, 1997.
[3] M. Fahndrich and F. Logozzo, "Clousot: Static contract checking with Abstract Interpretation," in *FoVeOOS 2010*. Springer Verlag, 2010.
[4] A. F. Milanez, "Enhancing Conformance Checking for Contract-Based Programs," Master's thesis, Federal University of Campina Grande, 2014.
[5] A. Milanez, D. Sousa, T. Massoni, and R. Gheyi, "JMLOK2: A tool for detecting and categorizing nonconformances," in *CBSoft (Tools session)*, 2014.
[6] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 75–84.
[7] A. Shtof, A. Agathos, Y. Gingold, A. Shamir, and D. Cohen-Or, "Geosemantic Snapping for Sketch-Based Modeling," *Computer Graphics Forum*, vol. 32, no. 2, pp. 245–253, 2013.

[6]www.ines.org.br

[8] M. Barnett, B. Chang, R. DeLine, B. Jacobs, and R. Leino, “Boogie: A Modular Reusable Verifier for Object-Oriented Programs,” in *Formal Methods for Components and Objects*. Springer Berlin Heidelberg, 2006, vol. 4111, pp. 364–387.

[9] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf, “Programs That Test Themselves,” *IEEE Computer*, pp. 46–55, 2009.

[10] B. Meyer, “Design by Contract,” in *Advances in Object-Oriented Software Engineering*. Prentice Hall, 1991, pp. 1–50.

[11] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, R. Leino, and E. Poll, “An overview of JML tools and applications,” *STTT*, pp. 212–232, 2005.

[12] G. Leavens, A. Baker, and C. Ruby, “JML: A Notation for Detailed Design,” in *Behavioral Specifications for Businesses and Systems*. Springer US, 1999, pp. 175–188.

[13] M. Hecht, *Flow Analysis of Computer Programs*. Elsevier Science Inc., 1977.

[14] O. Shivers, “Control Flow Analysis in Scheme,” in *PLDI*. ACM, 1988.