Practical Specification and Verification with CodeContracts

Francesco Logozzo
Microsoft Research
One Microsoft Way, Redmond, WA, USA
logozzo@microsoft.com

ABSTRACT

In this tutorial I will introduce CodeContracts, the .NET solution for contract specifications. CodeContracts consist of a language and compiler-agnostic API to express contracts, and of a set of tools to automatically generate the documentation and to perform dynamic and static verification. The CodeContracts API is part of .NET since v4, the tools are available for download on the Visual Studio Gallery. To date, they have been downloaded more than 100,000 times.

Categories and Subject Descriptors

D [**Software**]: Miscellaneous; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers,Programming by contract*

General Terms

Contracts,Verification

Keywords

Abstract Interpretation,Contracts,Inference,Program Verification

1. INTRODUCTION

Contracts are a popular software design methodology [15]. They are based on the idea that software modules should expose a well-defined interface, clearly stating the properties each module expects on the input and ensures on the output values. In object-oriented languages, contracts take the form of an invariant attached to the object and of preconditions and postconditions attached to the methods.

Traditionally a programmer who wants to use contracts has two choices. The first one is to adopt a programming language which has first class support for contracts, as for instance Eiffel [15] or Spark [1]. The second one is to keep the language she is used to, but to use a different compiler opportunely extended to support contracts, *e.g.*, JML [12] or Spec# [2] respectively extending Java and C#. The advantages of the first choice are the evident beauty and uniformity provided by having language support for contracts. The main disadvantages is the requirement to learn a new programming language and new libraries. The second choice mitigates those problems, but it has the (big) practical the drawback of asking the programmer to trust a non-standard compiler.

CodeContracts [3] provide a third option, which overcomes the problems above by using a library-based approach. With a library, there is no need to adopt a new language or a new compiler. Writing contracts is as difficult as invoking as function.

HILT 2013 Nov 10-14 2013, Pittsburgh, PA, USA
ACM ACM 978-1-4503-2466-3/13/11
http://dx.doi.org/10.1145/2527269.2527282 .

2. CODECONTRACTS

The main insight of CodeContracts is that code can be specified with code. We have developed a contract library and a set of tools that consume those contracts. CodeContracts originated from the Spec# project, where the language-based approach to contracts was replaced with a library-based one and the deductive verification-based static analysis tool was replaced by an abstract interpretation-based one.

2.1 API

Since .NET v4, the static class `System.Diagnostic.Contracts.Contract` contains the definitions for CodeContracts. The class contains methods to express preconditions, postconditions, object invariants, assertions, assumptions, and legacy requires. It also defines: (i) a few helper methods used as placeholder for the method return value and the old value of some element of the prestate; (ii) the attributes to help the tools find the contracts for abstract methods and interfaces. All members of the `Contract` class are conditionally defined.

2.2 Dynamic verification

We have developed `ccrewrite`, a tool to perform the runtime checking of contracts. The tool runs as a post-build step. It performs rewriting of the binary. It takes care of basic things as inheriting contracts and inserting postconditions and object invariants checkings at the right spots. But it has also some more advanced options, to provide: (i) customizable behavior for contract violation — *e.g.*, throw an exception, behave as an assertion violation, user-defined ...; (ii) fine-grain tuning of the contract checking — *e.g.*,

check preconditions at call site, skip quantifier checks, check only public surface contracts ....

2.3 Static verification

Unlike most of existing solutions, we use abstract interpretation [5] for static verification. Abstract interpretation: (i) provides a high level of automation —*e.g.*, no need to specify loop invariants, which are automatically inferred; (ii) enables a fine tuning of the analyzer towards the properties to prove; (iii) guarantees scalability. Our static verification tool, `cccheck` [9], analyzes each method in isolation, infers the properties of interest [11, 10, 14, 7], and it uses those properties to prove user-provided contracts (*e.g.*, preconditions, postconditions) and language-induced contracts(*e.g.*, absence of null-pointers, buffer overruns). When it fails to prove a contract, it suggest a verified code repair [13]. It infers preconditions [6], postconditions, and object invariants [4] which are propagated to the callers. It uses a database to cache the analysis results, ensuring scalability and sharing of results among team members. It uses advanced heuristics to prioritize warnings and to eliminate false positives.

2.4 Other tools

We have developed tools to automatically generate the documentation from CodeContracts and to enhance Visual Studio to show contracts while typing. Furthermore, we integrated all our tools in Visual Studio [8], so that: (i) they can be configured from within it; (ii) the errors and the alarms are presented in the programmer's usual environment.

2.5 Adoption

CodeContracts are increasingly adopted inside and outside Microsoft. They are available for download from the Visual Studio Gallery [16]. We frequently interact with our external customers via an external forum. The interaction with the customers enables us to improve the quality of the tools, fix bugs, add new features, and answer common and tricky questions on contracts.

3. REFERENCES

[1] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[2] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[3] M. Barnett, M. Fähndrich, and F. Logozzo. Embedded contract languages. In *SAC'10*. ACM Press, 2010.

[4] M. Bouaziz, L. Logozzo, and M. Fähndrich. Inference of necessary field conditions with abstract interpretation. In *APLAS*, 2012.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*. ACM Press, Jan. 1977.

[6] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *VMCAI*, pages 128–148, 2013.

[7] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL 2011*. ACM Press, Jan. 2011.

[8] M. Fahndrich, M. Barnett, D. Leijen, and F. Logozzo. Integrating a set of contract checking tools into visual studio. In *TOPI*. IEEE, 2012.

[9] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, 2010.

[10] P. Ferrara, F. Logozzo, and M. Fähndrich. Safer unsafe code in .NET. In *OOPSLA'08*. ACM Press, 2008.

[11] V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI '09*, 2009.

[12] G. T. Leavens, J. R. Kiniry, and E. Poll. A jml tutorial: Modular specification and verification of functional behavior for java. In *CAV*, 2007.

[13] F. Logozzo and T. Ball. Modular and verified automatic program repair. In *OOPSLA*. ACM, 2012.

[14] F. Logozzo and M. Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *SAC*. ACM, 2008.

[15] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[16] Microsoft. Codecontracts tools. `http://aka.ms/codecontracts/vsgallery`.