A Semantic Integrated Development Environment

Francesco Logozzo, Michael Barnett,
Manuel Fändrich
Microsoft Research
{logozzo, mbarnett, maf}@microsoft.com

Patrick Cousot
ENS, CNRS, INRIA, NYU
pcousot@cims.nyu.edu

Radhia Cousot
CNRS, ENS, INRIA
rcousot@ens.fr

Abstract

We present SIDE, a Semantic Integrated Development Environment. SIDE uses static analysis to enrich existing IDE features and also adds new features. It augments the way existing compilers find syntactic errors — in real time, as the programmer is writing code without execution — by also finding semantic errors, *e.g.*, arithmetic expressions that may overflow. If it finds an error, it suggests a repair in the form of code — *e.g.*, providing an equivalent yet non-overflowing expression. Repairs are correct by construction. SIDE also enhances code refactoring (by suggesting precise yet general contracts), code review (by answering what-if questions), and code searching (by answering questions like "*find all the callers where* $\mathtt{x} < \mathtt{y}$").

SIDE is built on the top of CodeContracts and the Roslyn CTP. CodeContracts provide a lightweight and programmer-friendly specification language. SIDE uses the abstract interpretation-based CodeContracts static checker (`cccheck/Clousot`) to obtain a deep semantic understanding of what the program does.

Categories and Subject Descriptors D. Software [*D.3 Programmimg Languages*]: D.3.3 Language Constructs and Features; F. Theory of Computation [*F.3 Logics and meanings of Programs*]: F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.2 Semantics of Programming Languages; I. Computing Methodologies [*I.2 Artificial Intelligence*]: I.2.2 Automatic Programming

General Terms Design, Documentation, Experimentation, Human Factors, Languages, Reliability, Verification.

Keywords Abstract interpretation, Design by contract, Integrated Development Enviroment, Method extraction, Program Repair, Program transformation, Refactoring, Static analysis

1. Introduction

Integrated Development Environments (IDEs) provide a cohesive view of the software development environment in which many tools are unified under a common and uniform user interface. The ultimate goal of an IDE is to assist and improve programmer productivity by simplyfing and rationalizing program development. Routinely, IDEs include a source editor, build automation tools, debuggers and profilers. Modern IDEs, like Eclipse or Visual Studio, provide additional functionalities like real-time compilation, type checking, IntelliSense, refactoring, class browsers, quick fixes for *compile-time* errors, *etc.* Existing IDEs have only a very partial and *syntactical* understanding of the program. We believe that in order to provide further value to the programmer the IDEs should get a deeper, more *semantic* understanding of what the program does. In the demo we show a working prototype of a Semantic Integrated Development Environment (SIDE).

2. SIDE

SIDE is a smart programmer assistant. It statically analyzes the program in real time, while the programmer is developing it. Unlike similar program verification tools, our static analysis infers loop invariants, significantly reducing the annotation burden. The information gathered by the static analysis is used to verify the absence of common *runtime* errors (*e.g.*, division by zero, arithmetic overflows, null pointer exceptions, and buffer overruns) as well as user-provided assertions and contracts [1].

If SIDE detects a potential runtime error, it suggests a fix in the form of code. The suggested fix is valid in that it guarantees that no good execution is removed: only bad ones are [7]. Since the fix is based on a static analysis, SIDE can suggest fixes for partial or even syntactically incorrect programs. No test runs are needed. Examples of fixes include object and constant initializations, arithmetic overflows, array indexing, wrong guards, missing contracts — *e.g.*, preconditions [4].

SIDE helps the programmer in other common tasks, such as refactoring. For instance, when the programmer extracts a method, SIDE proposes a contract (precondition, postcondi-

SPLASH'12, October 19–26, 2012, Tucson, Arizona, USA.
ACM 978-1-4503-1563-0/12/10.

tion) for the extracted method [5]. The proposed contract is valid, safe, complete, and general. In particular, completeness implies that the contract is precise (strong) enough to carry on the proof in the method from which the code was extracted. Generality guarantees that the contract can be called from other calling contexts *i.e.*, it does not just project of the state of the analyzer, which encodes the local context of the extracted method.

SIDE exploits the inferred semantic information to answer non-trivial queries on the program execution. For instance, SIDE supports *what-if* scenarios: The programmer adds extra-assumptions on the program state at some points and then she asks, *e.g.*, if some program point is reachable, or a certain property holds. The assumption and the queries are arbitrary Boolean expressions in the target language. SIDE enables semantic search, too. The programmer can ask if a certain method is invoked in a certain state. Examples of semantic searches are callers such that: $\texttt{x} \neq \texttt{null}$, $\texttt{a.f} > \texttt{b.c} + 1$, or a Boolean combination thereof. Overall, the semantic queries targets common scenarios in the code-reviewing phases.

3. The Architecture

Our target language is C# or VB, the two most popular .NET languages. We implemented SIDE on the top of the Roslyn CTP and of CodeContracts. The Roslyn CTP exposes the VB and C# compilers as services. We leverage Roslyn for the user interaction, *e.g.*, the squiggles for warnings and the previews for applying fixes, as well as to get basic services as “standard” refactoring. We use the CodeContracts API as the specification language for the preconditions, postconditions and object invariants. The CodeContracts API is a standard part of .NET. The CodeContracts static checker (`cccheck` [6]) is the underlying semantic inference and reasoning engine for SIDE. `cccheck` is a static analyzer based on abstract interpretation [3]. To enable real-time analysis, `cccheck` drawes on a SQL database to cache the analysis results, so that unmodified code is not re-analyzed. CodeContracts has been publically available for 3 years and has been downloaded more than 60,000 times.

4. The Demo

We show how SIDE acts as a smart programmer assistant, quickly catching tricky bugs, explaining them, and proposing fixes. In particular we show how the interaction is very natural for the user, despite the complex analyses and reasoning performed underneath.

In the first part of the demo, we code an `Insert` method, which inserts an element into a list represented as an array. SIDE points out several errors in a trivial implementation (a buffer overrun and a null dereference) and it proposes some preconditions to fix them. Then we add some code to resize the array when an insertion into a full array occurs. SIDE points out that the new code is unreached. Once the bug is fixed, it finds some other weaknesses in the code: an arithmetic overflow and a buffer overrun. In both cases it suggests a code repair — actually more than one: we will see and discuss in the demo that there are several different ways of fixing a program. In the case of the buffer overrun, we use the query system of SIDE to understand the origin of the warning (“*what happens when* ...”). Then we apply one of the (non-trivial) fixes proposed by SIDE. Finally, we realize that the code for resizing is more general than the usage made in the `Insert` body. Therefore we decide to refactor it into a new method. SIDE generates a new method, `Resize`, and the corresponding contracts. In particular: (i) the inferred precondition is more general than the simple projection of the original abstract state, enabling more calling contexts; (ii) the inferred postcondition is strong enough to ensure the safety in the refactored `Insert` method, *i.e.*, no imprecision is introduced by the assume/guarantee reasoning. We conclude this part of the demo by asking SIDE some semantic queries (*e.g.*, “*which callers insert an empty string into the list?*”).

In the second part of the demo, we consider a slightly more complicated example, a buggy implementation of the binary search algorithm. Discovering the bug(s) and presenting the fixes require the analysis to perform complex reasoning, *e.g.*, inferring a complex loop invariant. However, we will show how all this machinery is totally transparent to the user. For instance we show how SIDE naturally suggests a (verified!) repair for the famous Java arithmetic overflow bug [2].

5. Presenters

F. Logozzo is a researcher in the RiSE group at MSR Redmond. He is the co-author of the CodeContracts static checker and of SIDE. His main interests are abstract interpretation, program analysis, optimization, and verification.

References.

[1] M. Barnett, M. Fähndrich, and F. Logozzo. Embedded contract languages. In *SAC'10*, pages 2103–2110. ACM, 2010.

[2] J. Bloch. Nearly all binary searches and mergesorts are broken, 2008. `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[4] P. Cousot, R. Cousot, and F. Logozzo. Contract precondition inference from intermittent assertions on collections. In *VMCAI*, pages 150–168, 2011.

[5] P. Cousot, R. Cousot, F. Logozzo, and M. Barnett. An abstract interpretation framework for refactoring with application to extract methods with contracts. In *OOPSLA*, 2012.

[6] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, pages 10–30, 2010.

[7] F. Logozzo and T. Ball. Modular and verified repairs. In *OOPSLA*, 2012.