A Semantic Integrated Development Environment

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

Patrick Cousot, Radhia Cousot
ENS, CNRS, INRIA, NYU pcousot@cims.nyu.edu rcousot@ens.fr

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 simplifying 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-
STANCE, SIDE supports 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: x \neq \text{null}, a \cdot x > b \cdot 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, post-conditions 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 draws on a SQL database to cache the analysis results, so that unmodified code is not re-analyzed. CodeContracts has been publicly 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 unreachable. 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