Automatic Contract Insertion with CCBot

Scott A. Carr*, Francesco Logozzo†, and Mathias Payer*
*Purdue University
†FaceBook, work completed while employed at Microsoft Research

Abstract—Existing static analysis tools require significant programmer effort. On large code bases, static analysis tools produce thousands of warnings. It is unrealistic to expect users to review such a massive list and to manually make changes for each warning. To address this issue we propose CCBot (short for **C**ode**C**ontracts**Bot**), a new tool that applies the results of static analysis to existing code through automatic code transformation. Specifically, CCBot instruments the code with method preconditions, postconditions, and object invariants which detect faults at runtime or statically using a static contract checker. The only configuration the programmer needs to perform is to give CCBot the file paths to code she wants instrumented. This allows the programmer to adopt contract-based static analysis with little effort. CCBot's instrumented version of the code is guaranteed to compile if the original code did. This guarantee means the programmer can deploy or test the instrumented code immediately without additional manual effort. The inserted contracts can detect common errors such as null pointer dereferences and out-of-bounds array accesses. CCBot is a robust large-scale tool with an open-source C# implementation. We have tested it on real world projects with tens of thousands of lines of code. We discuss several projects as case studies, highlighting undiscovered bugs found by CCBot, including 22 new contracts that were accepted by the project authors.

Index Terms—contract-based verification, automated patching, assertions, class invariants

1 Introduction

Static analysis tools help programmers improve their code, but for large projects, the number of annotations and warnings quickly gets overwhelming. A Stack Overflow question reads (emphasis ours) [1]:

> We have started using a static analyzer (Coverity) on our code base. We were promptly **stupefied by the sheer amount of warnings we received** (it[']s in the **hundreds of thousands**), it will take the entire team a few months to clear them all (obliviously [sic] impossible).

A static analysis tool dumping a huge list of warnings on the programmer is not an efficient approach. The programmer cannot understand a warning without seeing the surrounding code. For each error, she must look up the error location in her editor or IDE, resulting in many context switches between different windows. Some tools (e.g. Coverity[1]) have addressed this issue by presenting their own GUI, but the programmer must learn the new GUI and might prefer another interface.

One approach to reducing the number of warnings is adding annotations. In general, the annotations might disable certain messages, specify the intended behavior of the program, or enable/disable analysis on some piece of code. These manual annotations are manageable in small examples, but not for large code bases when starting from a completely unannotated code base.

Another approach to reducing the size of the list of warnings is calculating a confidence rating on each warning to heuristically filter out low confidence warnings. This can be based on internal metrics or historical bug data [2]. However, a heuristic based approach will always have a false alarm rate.

While presenting the results to the user is a major problem for every static analysis tool [3], this work focuses on contract-based static analysis. Contracts are method preconditions, postconditions, and object invariants.

Case studies have shown that annotation burden is a primary factor limiting the adoption of contract-based static analysis [4]. It is tedious to start using contracts from scratch on a large code base and wide coverage is required to get the most benefit.

If a piece of code does not have any contracts, i) a static contract checker will not find anything interesting and ii) at runtime the errors that would be caught by the contract checks will slip by. Hundreds of new contracts might be needed to give reasonable coverage to a large code base. Surveys have shown that 33% of program elements typically have contracts [5] and that assertions comprise up to 5% of all lines of code in design-by-contract software [6]. In our evaluation for FluentValidation, the maximum coverage was reached with the insertion of 642 new contracts in 3,000 lines of code.

To address this fundamental problem, we have developed CCBot, which makes contract-based static analysis programmer friendly by automatically inserting inferred contracts into existing code. For projects that do not already use contracts, CCBot adds many contracts quickly, but also supports incremental addition of contracts (considering existing contracts). By utilizing cccheck [7], which soundly infers and verifies contracts, CCBot can automatically mitigate null pointer dereferences, array out of bounds, integer overflows, and floating point precision mismatches. For example, the code in Listing 1 is a simplified version of a bug from our case studies. Without CCBot's new contract in the

1. https://scan.coverity.com/

constructor (highlighted), the method `PeekFirst` will later throw a null pointer exception if the `Queue` constructor's argument is null. This is confusing to the user of this API because the exception and the root cause are located in different methods. Adding the highlighted contract improves the robustness of the code in by:

1) documenting that the `pts` parameter should not be null;
2) adding a runtime check that "fails fast," i.e., as soon as the invalid parameter is given; and
3) providing the static checker with more information to prove/disprove other contracts.

```
public class Queue
{
  List<int> _items;
  public Queue(List<int> items) {
    Contract.Requires(items != null);
    _items = items;
  }
  public int PeekFirst() {
    _items.Sort();
    return items[0];
  }
}
```

Listing 1. An example CCBot transformation.

Automatically inserting the contracts, rather than presenting a list of warnings, has two usability benefits that reduce the effort required from the programmer. The programmer can i) view the contracts in place for context, using familiar tools and ii) accept or test the modified version of their code without any manual steps.

CCBot has annotated tens of thousands of lines of code from real projects including internal Microsoft projects and open source projects from GitHub. We have conducted several case studies to show CCBot is robust enough for real-world use and that it finds previously undiscovered bugs. Our contributions are:

1) design and implementation of an automatic contract insertion mechanism called CCBot; and
2) evaluation and cases studies of applying CCBot to large, real code bases.

2 Background

To make the description of the design and implementation of CCBot clear, we must first describe the existing tools CCBot makes use of, namely CodeContracts, cccheck, and Roslyn. These are not part of the contribution of this work, but they are needed to understand CCBot.

2.1 CodeContracts

CodeContracts is a language agnostic way of expressing preconditions, postconditions, and object invariants [8]. For the purposes of this paper, we focus on C#. CodeContracts provides the option of checking the contracts dynamically or statically. Some contract frameworks put contracts in comments [9], but CodeContracts are valid C# code. This allows C# IDEs and tools to interact with the contracts in the same manner as regular code. For example, in Visual Studio the user can right-click on a variable in a contract and jump to its definition. CodeContracts provides a class `Contracts` with methods `Ensures`, `Requires`, `Assume`, and `Invariant`. Each of these methods have the following parameters:

1) a Boolean predicate, i.e., the condition that should hold in the code;
2) (optionally) a message to print when the Boolean is false; and
3) (optionally) an exception to raise when the Boolean is false.

2.1.1 *Ensures, Requires, and Assume*

`Requires` and `Ensures` represent pre- and postconditions respectively. Listing 2 shows an example of a method with `Requires` and `Ensures` contracts. `Requires` and `Ensures` must appear at the start of a method body before any other statements while `Assume` can appear anywhere in the method body. An `Assume` is a fact that the static checker should assume will hold during program execution.

```
public static void ZeroAt(List<int> orig, int
    index) {
  Contract.Requires(orig != null):
  Contract.Requires(0 <= index);
  Contract.Requires(index < orig.Length);
  Contract.Ensures(orig[index] == 0);
  orig[index] = 0;
}
```

Listing 2. A simple Requires/Ensures example.

2.1.2 *Invariant*

`Invariant` contracts are properties that will always hold whenever an object is visible to the client. Dynamically the checks are made after the return of every public method (including the constructor). `Invariant` can only appear in a special method marked with the `ContractInvariantMethod` attribute. Each class can have at most one `ContractInvariantMethod`. As example class with an `ContractInvariantMethod` is show in Listing 3.

```
class Bucket {
  List<int> container;
  public Bucket(int n_slot) {
    container = new List<int>();
  }
  [ContractInvariantMethod]
  private static void
      BucketObjectInvariant() {
    Contract.Invariant(container != null);
  }
}
```

Listing 3. A simple ObjectInvariant example.

2.1.3 Contracts for Interfaces

Sometimes we would like to specify that all implementations of a certain interface should obey a contract. However, this is complicated because interfaces cannot contain code. CodeContracts' solution to this problem is to create an abstract class that holds the interface's contracts. To link the abstract class and the interface, there are two attributes, `ContractClass` and `ContractClassFor`. For example, if the original program (without any contracts) contains the code in Listing 4, we want to specify that all implementers of `IShape` should return a non-null value for `GetEdges()`. Otherwise, the function `CalculatePerimeter` would crash when it invokes `edges.Length()`.

```
public interface IShape {
  List<Edges> GetEdges();
}
static void CalculatePerimeter(IShape shape)
{
  var edges = shape.GetEdges();
  var n = edges.Length();
  ...
}
```

Listing 4. An interface to which we would like to add a contract.

This contract can be implemented by adding the abstract class with the contract and attributes as shown in Listing 5.

```
[ContractClass(typeof(IShapeContracts))]
public interface IShape {
  List<Edges> GetEdges();
}
static void CalculatePerimeter(IShape shape) {
  var edges = shape.GetEdges();
  var n = edges.Length();
  ...
}
[ContractClassFor(typeof(IShape))]
public abstract class IShapeContracts :
    IShape {
  List<Edges> GetEdges() {
    Contract.Ensures(
      Contract.Result<List<Edges>>() != null);
  }
}
```

Listing 5. This example specifies contracts for an interface.

2.2 Cccheck

Cccheck (short for **c**ode **c**ontracts **check**er) infers and verifies contracts statically [8]. Errors found by cccheck include null pointer deferences, array out of bounds, buffer overflows, integer overflows, and floating point precision mismatches. Cccheck uses abstract interpretation to determine facts about the program, and uses these facts to generate warnings and annotations. It uses assume/guarantee reasoning. When it is analyzing some method `foo`, it assumes the contracts of all methods other than `foo` hold and checks if `foo`'s contracts hold given those assumptions.

All the contracts inferred by cccheck are sound. In this context, "sound" means the new contracts never remove "good" executions, i.e., those that did not cause an exception. This means inserting the contracts is a so called verified repair [10] – one that will reduce the number of bad traces, but will increase or keep the number of good traces.

The input to cccheck is a Microsoft Common Intermediate Language (CIL) assembly and configuration options. Cccheck performs its static analysis over the assembly and outputs suggested contracts in addition to a list of warnings. Examples of warnings might include that cccheck can determine that a contract will never hold or that a certain Boolean predicate is always true or false leading to the untaken branch being dead code.

Cccheck uses the contracts in the code and the semantics of the CIL to build a partial specification of the program. The CIL gives implicit contracts. For example, dereferencing `foo` results in an implicit contract that `foo` should not be null. Cccheck infers *necessary* preconditions [11], i.e., if the precondition is false the method will fail, but the precondition holding does not guarantee the method will succeed. Similarly, cccheck can find necessary conditions on object invariants. These are conditions on object fields that when violated imply there exists a call trace (including object construction and potentially other methods) that leads to a contract violation [12].

2.3 Roslyn

Roslyn is Microsoft's open source C# and Visual Basic compiler framework. Roslyn may be integrated into other tools that manipulate not only the output assembly but the original code itself. Example use cases for Rosyln are IDE plugins, automatic refactoring tools, and diagnostic tools. Roslyn provides syntactic and semantic APIs. The syntactic API is for manipulating the actual text of the program. The semantic API provides a fully resolved and searchable model of all the symbols in the program. To use the semantic model, Roslyn does not work on individual C# files but entire projects and solutions. In MSBuild terminology, a project is a collection of source files and settings that produces a single assembly. A solution is a group of projects with inter-project dependencies and configurations.

3 Design

3.1 Automatic Contract Insertion

Our Automatic Contract Insertion tool (CCBot) directly modifies the code under analysis without any programmer intervention. This is in contrast to existing static analysis tools that merely output a list of warnings the programmer must manually address one-by-one. Our approach is more programmer friendly and scalable than a list of warnings. Removing the scalability and usability hurdles will have a positive effect on contract-based tool adoption and development.

Johnson et al. conducted a large (n = 20) detailed static analysis tool usability study [3]. They found that one of the biggest user frustrations is having to juggle multiple windows. For example the paper reads, "*For* [two developers surveyed], *the biggest downside to using Coverity is that it is not*

capable of being integrated into their coding environment, leading to a lot of clicking back and forth." Automatically modifying the code eliminates the switching problem.

The same survey also found that 14 out of 20 participants expressed poorly presented tool output has a negative effect. The participants felt that even if there were many warnings, it would help to present them in a more user-friendly and intuitive way. Again, directly modifying the code solves this issue because they can see everything in context and using their normal tools.

19 out of 20 participants said integrating the tool into their workflow was important. Directly modifying the code streamlines integration (viewing changes to code is part of every open source developer's work flow). Automatic Contract Insertion integrates well with Continuous Integration (CI) which has become very popular in open source development. CI is the practice of automatically executing an integration test suite on every code commit [13]. Projects might run their own CI server (e.g. Jenkins-CI [14]) or use a CI service (e.g. Travis-CI [15]). The test suite can run on both the original code and the modified code and report both results. This gives three chances to find each bug. First, the static analysis might find it. Second, the run of the CI test suite after inserting the new contracts might find it. Third, the CI run on the original code might find it. Having the modified code pass the CI test suite gives the user extra confidence that the automatic changes are correct.

The major practical hurdle is modifying the original sources in a semantically correct way, so that the new code retains all the valid executions of the original code. A purely syntactic tool (a tool that just manipulates text) cannot provide this guarantee. The tool could be fooled into inserting contracts in the incorrect place. For example, methods with the same name in different classes or even identical classes in different namespaces might confuse a text-based tool. Our tool must fully parse the code and resolve all references in order to build an accurate semantic model of the code. Another benefit of fully parsing the code is that it allows CCBot to recognize existing contracts. This enables CCBot to eliminate duplicate contracts, and is required to comply with the constraints CodeContracts requires on the ordering of contract types. Our design requires the tool to work equally well whether or not the code under analysis already had contracts.

Improving software robustness through automatically inserting contracts has limitations. The design space of fixes for a given bug can be large [16]. CCBot is not a code synthesis tool. It is limited to adding new contracts the analyzer can determine are correct. However, our case studies show that these simple additions are useful to real programmers. For example, if a class constructor receives an invalid parameter, it is better to "fail fast" and immediately report the error message than to let the invalid data propagate leading to a failure later on. In this case, a simple check with an appropriate error message is at least an improvement on the original code if not a fix. CCBot can create and insert these types of contracts. We do not contend that CCBot's new contracts can fix all bugs, but in large real world projects there is enough neglected code that finding and mitigating bugs automatically is feasible.

While CCBot's current implementation is closely tied to CodeContracts, the idea of a static analysis tool automatically modifying the code under analysis can be extended to other tools. Any type of analysis that suggests well defined code modifications could fit the automatic transformation model. Coupling CCBot with multiple (potentially more sophisticated) analyzers expands the space of potential fixes it can perform.

3.2 CCBot Design

At its core, CCBot is a source-to-source translation of C# code. CCBot's input is the code to be instrumented and the output is the original code plus contracts. CCBot can add all the types of contracts described in Section 2.1. The added contracts detect and mitigate null-pointer dereferences, buffer overflows, integer overflows, and floating point precision mismatches. CCBot integrates several existing tools: a static analyzer, a compiler, a version control system, and continuous integration testing, into a cohesive bug finding and code improvement tool that runs automatically.

3.2.1 Static Analysis

CCBot does not implement a static analysis itself, but uses cccheck. A key design requirement for CCBot is that the user trusts the tool, so they do not feel the need to scrutinize every individual contract. To meet this requirement, CCBot uses a sound static analysis to guarantee the additions never remove good executions from the original code. There is no technical reason that CCBot could not use an unsound static analysis, but we believe users would quickly abandon the tool if they ran into too many false positives. In the context of CCBot, a false positive would be a new contract that is too strict, i.e., a valid execution of the original program exists that violates the contract.

CCBot's static analyzer gathers a partial specification of the program's behavior through existing contracts, if any, and the semantics of the CIL. The semantics of the CIL give implicit contracts. For example, the statement `foo.bar();` gives the implicit contract `foo != null` (or else the program would crash). Implicit contracts also arise from array accesses. If the static analyzer finds a contract whose negation is satisfiable, that is a bug in the program. The analyzer emits a warning when it cannot prove a contract is valid.

3.2.2 Compiler

The design of CCBot is to improve the code rather than just notifying the programmer there is a bug. Accordingly, CCBot uses a compiler framework to modify the buggy code. CCBot's modifications are restricted to new contracts that check for the buggy behavior both statically and dynamically. These new contracts do not necessarily match the fix the programmer would make but they mitigate the errors by checking for invalid state early and failing fast.

An engineering benefit of using a compiler is that CCBot can perform a sanity check. It can use the compiler to check the modified code for compiler errors and revert any modifications that lead to new errors. The compiler gives the line number that contains the error so the contract that was inserted at that line is deleted from the list of new contracts and CCBot starts over from scratch with a smaller list. In a perfect implementation the modifications would

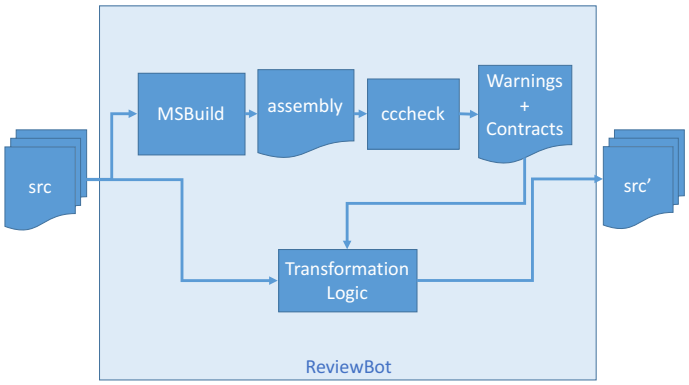


Fig. 1. Overview of CCBot's Transformation

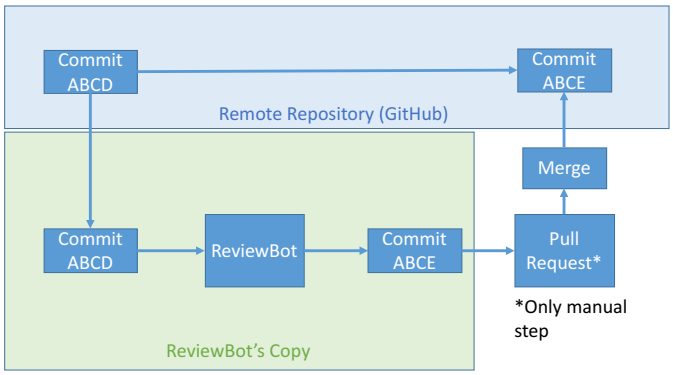


Fig. 2. High-level flow chart of the CCBot mechanism.

never cause errors, but experience shows bugs can appear in implementations of theoretically correct designs.

3.2.3 *Version Control System*

Since CCBot directly modifies the code, it needs some way of saving the results that allows the user to easily merge or revert the changes it made. It uses Git to facilitate this.

CCBot creates its own branch which effectively gives it its own copy of the code to modify. Then it builds, analyzes, and instruments the code locally. With the changes saved to disk, it commits the code to its local repository. A high level diagram of CCBot's transformation is shown in Figure 1. Since CCBot only inserts new statements and classes into the code, Git can merge CCBot's changes back into the main repository with a single command. This workflow is depicted in Figure 2.

3.3 Continuous Integration

Continuous Integration testing starts when CCBot's changes are committed. The Continuous Integration test run will have contracts enabled. Dynamically checking that the contracts hold at runtime gives another way of finding new bugs, in addition to what the static analyzer can find.

4 Contract Insertion Challenges

Previously, programmers could use the above mentioned tools separately, but CCBot combines them in a novel way to provide valuable end-to-end guarantees. These guarantees are that CCBot's augmented version of the code:

1) always compiles if the original program did; and
2) never removes a valid program execution of the old code.

CCBot provides these guarantees using the Roslyn C# compiler to always insert the contracts found by cccheck in the semantically correct location. The new contracts are necessary preconditions for correct execution of the program, so adding the new contracts never removes valid execution traces from the program.

One might think that the problem is simple, and that a tool could copy-paste text from the cccheck output into the code. However, cccheck's analysis is over the CIL. It does not examine the program's source code, so the mapping from a cccheck suggestion to the resulting code with the contracts inserted is far from trivial. Inserting the contracts correctly requires the capability of searching and modifying the code using fully resolved symbols.

A simple tool, like a code style checker, could provide the exact lines and characters to be changed, but cccheck's analysis and suggestions are much more sophisticated. The output of cccheck gives the text of the contract to be inserted, but CCBot must determine where the contract should be inserted. Finding the correct location is crucial. If CCBot could mistakenly make a change in the wrong location, it would destroy any guarantees cccheck provides. For this reason CCBot's design does not trust cccheck to output accurate line numbers, but instead uses the fully qualified name for every class, field, method, interface, and namespace. While two classes can have the same name, they cannot have the same fully qualified name, i.e., two different namespaces can have a class with the same name, but a single namespace has to have unique class names. The static analyzer finds where the bug occurs, but cannot know exactly where the contract should go in the source code.

For example, cccheck will find interface contract violations in the methods of classes which implement the interface, but the new contracts should go in a `ContractClassFor` associated with the interface. However, since cccheck operates over CIL, it does not know if the `ContractClassFor` exists. For a second example, the violations of object invariants occur in the member functions, but the invariant contracts go in the `ContractInvariantMethod`. Again, cccheck does not know if the `ContractInvariantMethod` exists (or if it exists in one of many partial class definitions). In these cases, CCBot searches for the location where the new contract should be inserted based on the location of the violation using the symbol names.

5 Implementation

CCBot's implementation is over 7,500 lines of C# code. The lines of code count is not huge because CCBot makes efficient use of existing tools such as cccheck and Roslyn. We have open sourced CCBot (MIT licensed) on Github[2]. We hope to incorporate improvements and feedback from the community.

2. https://github.com/scottcarr/ccbot

CCBot's main tasks are parsing the output of cccheck and using Roslyn to find where the contracts should go and insert them. CCBot uses Roslyn's semantic and syntactic APIs for manipulating C# source code. Opening the entire MSBuild solution and selected project lets Roslyn resolve references to other libraries and projects within the solution. This gives CCBot the fully qualified name of every type, method, field, and namespace even if it is defined externally. The fully qualified name of each element is guaranteed to be unique for a given project and configuration. This allows the precise matching of cccheck's suggestions (which come from an analysis of bytecode) to symbols in the source code. Roslyn's syntactic API allows CCBot to insert contracts into the code, while preserving whitespace, formatting, and comments. If CCBot did not keep consistent formatting the user would have to fix it manually, breaking the scalability of our approach.

5.1 Contract Verification and Inference

CCBot builds the project using MSBuild and starts cccheck's analysis of it. The output of cccheck is a large XML file with a list of all the suggested contracts and warnings for the entire project.

5.2 Contract Insertion

CCBot parses the XML file to find all the new contracts for each method. To insert the contracts, CCBot prepends statements to some method's body. The complications (explained in detail in the subsequent sections) are ordering rules, duplicate contracts, and contracts associated with an interface or object rather than a specific method.

After inserting is done, CCBot uses Roslyn to check for errors and reverts any new contracts that might have caused the errors. When no new errors are returned by Roslyn, then all the files are written to disk.

5.2.1 Ensures and Requires

`Ensures` and `Requires` contracts are added to existing methods, so CCBot finds the method and inserts the contract. Cccheck gives the file which contains the method and the fully qualified name, which CCBot uses to locate the method. With the method found, CCBot then parses the recommended contract, checks for duplicates, and inserts the new annotation subject to the ordering constraints. For `Requires` and `Ensures` the ordering constraints are:

- All `Requires` and `Ensures` come before any normal body statements
- All `Requires` come before all `Ensures`

The ordering means that CCBot cannot simply copy cccheck's suggested contracts to the beginning of the method. It must parse the existing contracts and combine them with the newly suggested ones in sorted order by type. Even this seemingly simple ordering step cannot be done syntactically while still preserving CCBot's correctness guarantee. For example, consider the case of inserting a new `Requires` contract into method `Foo`. CCBot must find the first non-`Requires` statement and insert the new contract before that statement. To accomplish this, we need to classify an arbitrary statement into one of the CodeContracts types or a non-CodeContracts statement. Consider the text in Listing 6. It is impossible to syntactically determine if this is a call to `System.Diagnostics.Contracts.Contract.Requires` or any arbitrary static method named `Requires` belonging to an arbitrary class named `Contract`.

```
Contract.Requires(arg0 != null);
```

Listing 6. We cannot syntactically determine if this call is to `System.Diagnostics.Contracts.Contract.Requires`.

Even if we could forbid C# programmers from naming classes `Contracts` or methods `Requires` (totally unrealistic), we can write the same contract in arbitrarily many ways. All of the contracts in Listing 7 are equivalent.

```
using Foo =
    System.Diagnostics.Contracts.Contract;
using System.Diagnostics.Contracts;
...

Class C {
  public static F() (Object arg0)  {
    System.Contracts.Contract.Requires(arg0
        != null);
    Contract.Requires(arg0 != null);
    Foo.Requires(arg0 != null);
  }
  ...
}
```

Listing 7. Examples of syntactically different but semantically equivalent contracts.

Note that this same problem applies to `Ensures` equally as `Requires`.

Checking for duplicate contracts is necessary because the same annotation may be suggested multiple times by cccheck. An example of this is interface annotations. There may be multiple implementations of the interface, so when cccheck analyzes the methods of each implementation, it might infer the same annotation. Inserting the same annotation more than once would be syntactically valid as long as it obeys the ordering constraints, but adds clutter, so CCBot drops duplicate annotations. CCBot also checks for an existing contract that is the same as the newly suggested contract. When identifying duplicate contracts, CCBot determines the type of contract and contract location semantically, but the Boolean predicates of the contracts are matched syntactically. This means that two contracts are considered duplicates if:

- both contracts should be inserted at the same location,
- both contracts are the same type (`Requires`, `Ensures`, `Assume`, or `Invariant`), and
- the text of the predicate of both contracts matches exactly.

This rule is effective for contracts suggested multiple times by cccheck, because it generates contract predicates in a consistent manner.

5.2.2 Assumes

Since `Assume` can appear anywhere in the body of a function and is not subject to any ordering constraints, it is more straightforward than `Ensures` and `Requires`. CCBot simply inserts the location suggested by cccheck using the line numbers provided by Roslyn's syntactic API and checks for duplicates.

5.2.3 Object Invariants

Inserting contracts for object invariants is complicated by a few factors. Since cccheck operates on the CIL, it just suggests that an invariant be added to the class that contains a particular method. It does not know if a `ContractInvariantMethod` exists for the class, where it is defined, or what the name of the `ContractInvariantMethod` is. CCBot must determine this.

The class in which the new invariants should be added might already have a `ContractInvariantMethod`. This is even further complicated by partial classes. In C#, the partial keyword can be applied to a class definition which allows the definition to be split between multiple files. CCBot must search the entire project to find all partial definitions of the class to determine if any of them contain a `ContractInvariantMethod`. CCBot gathers all the (partial) definition(s) of the class and checks the attributes of each method in the definition for the `ContractInvariantMethodAttribute` using the fully qualified symbol. If no `ContractInvariantMethod` is found, CCBot creates one and inserts the contracts, otherwise the contracts go in the existing method. In either case, duplicates are removed as before.

Note that locating the `ContractInvariantMethod` cannot be done correctly syntactically because of all of the following:

- The attribute symbol must be fully resolved to avoid confusion with other similarly named types.
- The `ContractInvariantMethod` may not be in the same file that contained the method in which cccheck found the contract violation.
- The `ContractInvariantMethod` is not required to have any particular name.

CCBot follows the convention of naming the method `$(class_name)ContractInvariantMethod` when creating a new method, but users may name their method arbitrarily and CCBot will find and insert contracts into the correct method.

5.2.4 Interface Contracts

When cccheck finds contract violations for an interface, it finds them in methods of classes that implement the interface. CCBot needs to locate the interface to determine if the interface has an existing `ContractClassFor` or not. CCBot searches the project for the interface and checks the interface for the `ContractClass` attribute. If the attribute exists, then the `ContractClassFor` abstract class (which contains the contracts for this interface) already exists and the contracts can be inserted in the same manner as the contracts described previously.

If the `ContractClass` attribute is missing from the interface, CCBot creates a new abstract class to hold the contracts in the same file as the interface. CCBot ensures that the new class is uniquely named and implements each of the interface's methods. The Roslyn API has a method to create the class with stub methods. CCBot inserts the contract into the appropriate stub method and adds the attributes to the class and interface.

If the interface already has the `ContractClass` attribute, then a `ContractClassFor` already exists for this interface. The class type that is the `ContractClassFor` is a parameter of the `ContractClass` attribute. CCBot retrieves the type and searches the project for the type. Again, this must be done semantically, as CCBot does, and not syntactically, in order to guarantee correctness. Using the resolved type symbol guarantees CCBot finds the correct `ContractClassFor`. The `ContractClass` attribute is typically written by C# programmers using the unqualified name of the `ContractClassFor` class type. A syntactic tool would likely confuse the class name with that of another class.

Generic interfaces raise an additional complication for interface annotations because generic interfaces can have the same name as long as they have a different number of type parameters. An example of a generic interface is shown in Listing 8. Each distinct parameterization is its own interface, so it needs a separate `ContractClassFor` abstract class with the same number of parameterized types. CCBot solves this problem by obtaining the fully qualified interface names from Roslyn's semantic model plus the parameterized types, but any approach that does not fully parse the original code will almost certainly fail with interfaces like these. CCBot is careful when inventing the name of the new abstract class of a generic interface to avoid conflicts. CCBot invents distinct names for each `ContractClassFor` by appending the identifiers used to denote the generalized type(s) to the class name. In the example in Listing 8, the `ContractClassFor` name would be IDictionaryTKeyTValueContracts.

```
public interface IDictionary<TKey, TValue> {
  public void Add(TKey key, TValue value);
  ...
}
```

Listing 8. An example of a genereric interface.

6 Evaluation

In addition to our case studies, we have run CCBot on several large, popular open source projects including NetwonSoft.Json, NuGet, FluentValidation, and other internal Microsoft projects. The ability to handle large code bases written by others demonstrates the robustness of CCBot.

The open source applications were chosen from GitHub based on popularity (number of stars). The internal Microsoft applications were selected based on their project members being interested in contracts and source code availability.

6.1 Fixed Point Experiment

Given the ability to automatically infer contracts and automatically insert contracts, a natural question arises. Can we

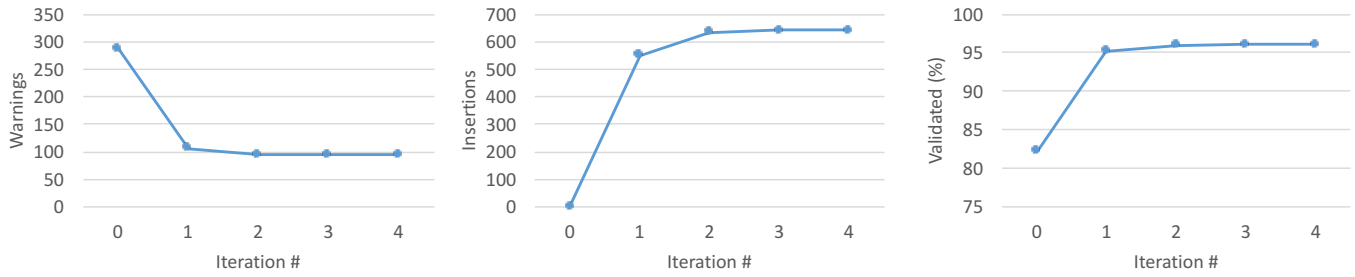


Fig. 3. Fixed point experiment: Insertions, Warnings, and Validated % over iterations for FluentValidation.

keep inferring and inserting new contracts in a loop until we do not infer any new contracts, i.e., can we reach a fixed point? In theory, there is no guarantee that a fixed point will be found, because there is no guarantee adding any particular contract will decrease the number of warnings. For example, adding a contract in a method can cause new warnings to pop up in all its callers. In this manner, contracts can have a cascading effect that can be witnessed when inserting contracts manually. This is different from the compiler warnings a programmer is used to addressing. Typically, when a programmer fixes a compiler warning, the length of her list of compiler warnings decreases by (at least) one. Proof-based static analyzers do not behave the same way. Adding an annotation to the callee might simply shift the burden of proof to the caller.

However, we hypothesized that in practice eventually all methods (and their callee/callers) will be annotated and a fixed point will be reached. To evaluate this we selected an open source C# project, FluentValidation, as our test case. The high-level experiment is relatively simple. We do the following in a loop:

- run cccheck,
- add the annotations with CCBot, and
- count the total number of annotations in the code.

Then we check for the stopping condition, i.e., if the total number of annotations is the same as the previous iteration, we stop.

We empirically measured a fixed point after three iterations. The fixed point version had 642 annotations and the warnings from the initial version to the fixed point reduced from 287 to 94. FluentValidation is around 3,000 lines of code. Figure 3 shows the warnings, annotations, and percentage of validated contracts over the iterations. In Figure 3, Iteration 0 is after running cccheck the first time, but before inserting any new contracts. As the graph shows, adding new contracts gives ccccheck more information, thus it can prove more contracts correct. The percentage of provably correct contracts begins at 82.2% and levels off at 96.1% when the fixed point is reached after three iterations.

6.2 Performance

Compared to the cost of performing more complex static analyses, the runtime of automatically applying the annotations with CCBot is short. Many static analysis tools

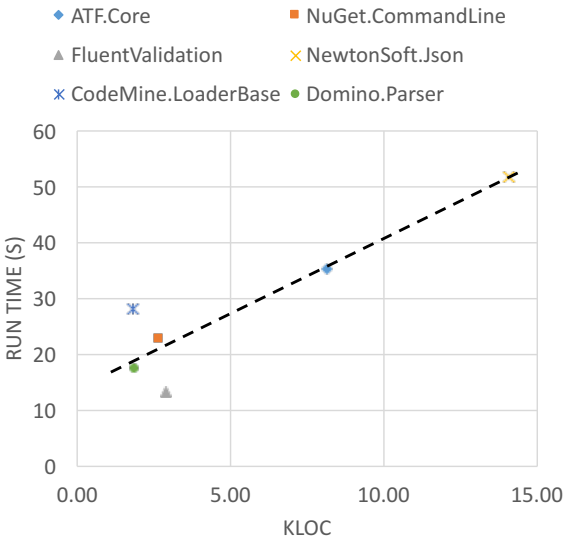


Fig. 4. CCBot's runtime versus code size.

terminate either when they have reached a fixed point or when a timer expires. The runtime of annotating a codebase is linear in the lines of code as shown by the trend line in Figure 4. The most costly operation is searching through a given syntax tree for a method, type, or field. This can be accomplished naively with a linear scan of the code. The runtime of CCBot's transformation on several codebases is shown in Table 1. From the table, the runtime of CCBot (measured in seconds) is dwarfed by the runtime of cccheck (measured in minutes).

7 Case Studies

To conduct our case studies we began by sorting the open source C# projects on GitHub by most stars. GitHub users can star projects they like so star rating is a rough measure of popularity. From this list we manually filtered for activity (measured in time since last commit), projects that worked with Roslyn, and projects that did not require custom build scripts. Roslyn is under active development and we were unable to fully open and resolve all references for some popular projects. CCBot requires the full resolution of all symbols and references to accurately insert the annotations.

	ATF	NuGet	FV	NewtonSoft.Json	Domino	CodeMine
LOC	8128	2661	2912	14081	1867	1831
Insertions	2130	467	552	2901	377	494
cccheck warnings	1150	345	287	1564	407	274
Run time (s)	35.29	22.74	13.31	51.74	17.54	28.27
cccheck time (min:s)	14:55	2:58	0:44	27:17	0:35	2:52

TABLE 1
Runtime performance of CCBot and cccheck.

Name	LOC total	LOC main project	Requires	Ensures	Assume	Invariant	Warnings	Submitted Insertions	Accepted
Scriptcs	8236	1800	67	258	85	23	290	16	16
Ninject	5436	2642	143	664	167	9	418	6	6
MongoDB	88200	7174	35	3776	187	24	483	3	0
Nancy	11123	11123	447	2675	208	44	1262	2	0

TABLE 2
Case Study Statistics

With our selected set of projects identified, we created our own Git forks for CCBot to modify. Once the fork is cloned onto our local machine, the only configuration CCBot needs are the project and solution file paths. CCBot automatically builds the selected solution and runs cccheck on the assembly created by the project. It inserts the contracts, commits the changes to the local repository, and pushes them to GitHub. In GitHub, we create a pull request that notifies the project authors we would like to make changes to their code. The author can accept and merge the changes with one click. Alternatively, if the author wants to review the changes she has two main options. She can see the `diff` in GitHub or pull the changes to her local machine. On her local machine she can use all the features of Visual Studio (or other tools) to investigate all the variables the contracts reference. These alternatives are much more useful than just getting a flat list of suggestions which is what existing static analysis tools provide.

For the purposes of our case studies, we assume the authors will only accept a pull request that made a small number of changes. Maintainers are hesitant to accept changes from a bot. They might not even review a large pull request that does not meet their contribution guidelines. To mitigate this issue we manually split the pull requests into digestible chunks and did not submit them all at once. While this method does not perfectly match a new users first experience with CCBot, it allows us to demonstrate that at least some subset of the contracts inserted with CCBot are useful to practitioners. There is a social aspect to contributing to open-source software. We received no response to our pull request for MongoDB and we can only guess as to why. However, we were successful in submitting small easily-reviewed pull requests to other projects.

We submitted changes that looked like unmistakably missing checks. For example, if a constructor checks two out of three of its parameters for null but not the other, the programmer really meant to check all three but forgot. If the authors truly wanted to adopt CodeContracts they would accept all the changes and on subsequent runs CCBot would make far fewer changes as shown by our fixed point experiment. Detailed statistics for our case studies are show in Table 2. The first column is the total lines of code in the entire repository. We only annotated the main project from each repository. The `Requires`, `Ensures`, and `Assume` columns are the total number of new contracts of each type inserted into the code. Note that for evaluation we run cccheck with its most strict settings. In practice, cccheck has options the user could set to reduce the size of cccheck's output based on heuristics. The "Warnings" column is the number of warnings that cccheck produced. The warnings are not necessarily bugs but suggest actions the programmer could take to give cccheck more information. The "Submitted Insertions" column is the number of new lines of code we submitted to the project authors in our pull requests. The "Accepted" column is how many of those new lines were merged into the main repository.

7.1 Scriptcs

Scriptcs is an editor and read-eval-print-loop for C# built on top of Roslyn. It allows interactive editing and execution of C# code. The project is a little more than two years old, but still under active development. Scriptcs is the at the time of writing the 30th most popular C# repository on GitHub. In total, the repository contains over 8,000 lines of code. The project maintainers are interested in using static analysis to improve the quality of their code. They use Coverity, StyleCop, and Visual Studio's Code Analysis. Even though the maintainers have used these tools CCBot found (and inserted) missing contracts.

The most common missing check (that these other tools presumably missed) was a null parameter passed to a constructor when the parameter was not dereferenced in the constructor itself, but it was in one of the class's methods. Listing 9 shows an example bug with the new contracts highlighted in red. The yellow highlighted lines show the bug. The parameters `fileSystem` and `logger` are stored in fields and then later dereferenced without any null check. The inserted contracts will cause the code to "fail fast" at runtime and will display a more informative error message or cccheck might be able to prove that the parameters will be null in some case and statically determine the contract is not satisfied.

In total, we used CCBot to insert 16 new checks for Scriptcs that were merged into the main repository by the project authors.

```
public class FilePreProcessor :
    IFilePreProcessor
{
  ...
  public FilePreProcessor(IFileSystem
      fileSystem, ILog logger,
      IEnumerable<ILineProcessor>
      lineProcessors)
  {
    Contract.Requires(fileSystem != null);
    Contract.Requires(logger != null);
    _fileSystem = fileSystem;
    _logger = logger;
    _lineProcessors = lineProcessors;
  }
  public virtual FilePreProcessorResult
      ProcessScript(string script)
  {
    var scriptLines =
    _fileSystem.SplitLines(script).ToList();
    return Process(context =>
        ParseScript(scriptLines, context));
  }
  protected virtual FilePreProcessorResult
      Process(Action<FileParserContext>
      parseAction)
  {
    Guard.AgainstNullArgument("parseAction",
        parseAction);
    var context = new FileParserContext();
    _logger.DebugFormat("Starting pre-processing");
    ...
  }
  ...
}
```

Listing 9. An example bug from Scriptcs

7.2 Ninject

Ninject is a dependency injector for .NET applications and is the 32nd most starred project on GitHub. It is over 5,000 lines of code in total and the project started in 2009. It is still an active project. There were commits in January 2015, but the rate of commits peaked in 2012 and has since slowed down. Similar to Scriptcs, there were several cases where a public constructor took a parameter, assigned it to a member, and later dereferenced the member without a null check in a method. There was also a bug where a parameter was immediately dereferenced without any null check. The project authors accepted and merged six additions across four different files.

7.3 MongoDB C# Driver

The C# driver for MongoDB is a very mature, almost five years old, project that is very active with many contributors (24 listed on the project website). It is the 40th most starred project on GitHub. The contributors are a combination of MongoDB employees and volunteers. In total the project is around 80,000 lines of C# code. As we might expect for such a mature project, cccheck gave relatively few suggestions for such a large code base. It found missing null checks in two `ClusterBuilder` constructors. The constructors take a delegate (C#'s version of a function pointer) and immediately invoke the delegate. `ClusterBuilder` has multiple constructors. The others have a contract to check for null on the same parameter, so CCBot's inserted contract fixes the bug and makes all the constructors have consistent behavior. At the time of writing the authors have not yet responded to the pull request.

7.4 Nancy

Nancy is a framework for building HTTP services in C#. It is a 6 year old project with over 10,000 lines of code. Contributions peaked around 2013, but it is still an active project with over 20 commits in April 2015. Most of the Nancy code does parameter validation, but CCBot inserted missing checks for two parameters in a public constructor. These parameters were stored in members and later dereferenced without null checks in other methods. At the time of writing the authors have not responded to the pull request.

8 Discussions

Project authors accepting the new contracts indicated that the new contracts improve the quality of the code. The underlying problems could have been found and fixed by programmers, but they were not. Programmer time is the most precious resource. The files that CCBot modified in the case studies typically had not been touched by the authors for years, so we hypothesize the bugs were there only because no one had the time to look.

CCBot's current implementation inserts CodeContracts, but some projects use other forms of contracts. For these projects, we transformed all the contracts with a simple string replacement. For example, cccheck suggests the contract `Contract.Requires(foo != null);` but Scriptcs uses contracts of the form `Guard.AgainstNullArgument(foo, ''foo'');`. A more automated solution supporting a wider range of contract transformations is feasible with little additional effort.

8.1 Contract Usage in Practice

We found checking for null to be the most common contract written by users and inserted by CCBot. This is largely because dereferencing a pointer is an extremely common operation in C#.

Other contract types, while fully supported by CCBot, occur less frequently. For example, CCBot can also insert contracts that check array bounds. However, in practice, it is much more common for cccheck to find potential null pointer dereferences than array out of bounds violations. With CCBot and cccheck the new version of the code (with new contracts) is guaranteed to have as many "good" executions as the original code. We need to be absolutely sure that the new contract holds for all possible executions that do not crash. Dereferencing a null pointer without a check always leads to a crash, and is very common, so we insert many of these contracts.

An array bounds contract could be inserted when a function's parameters include an array and an index, and in the original function body the array is indexed (using

the index parameter) without a check. This is not as common in C# as simply dereferencing a pointer, and without more information the contract we can safely insert is $0 \leq index \leq array.Length - 1$. Since CCBot's inserted contracts never remove valid executions, in order to insert a contract like $0 \leq index \leq 5$, cccheck has to be able to determine that the length of the array is never 6 or greater for any execution. It is common to not have a static upper bound on a given array.

Further, instead of inserting the bounds check contract in the function that takes the array and index as parameters, the better approach is to have the function which calculated the index have a post-condition $0 \leq result \leq array.Length - 1$. That way, the user does not have to write the bounds contract in all callers. However, in C# the common functions that search an array and return an index are builtin framework functions so the contract should be inserted into the framework itself, not the code under analysis. There is a version of the .NET Framework with contracts available to CodeContracts users.

9 Related Work

There is significant related work in the areas of code review, design by contraction, automatic repair and specification generation.

9.1 Review and Test Tools

CCBot, was originally conceived as an automatic code review tool. Previous work has examined the effectiveness of combining static analysis with code reviews [17]. CCBot was designed to prevent the problem the studies identified of the programmers removing or ignoring many of the warnings.

Khasiana [18] is an online portal that provides static analysis tools as a service. It eliminates the need for each user to download and install the tools. However, the output is a report the user must read and manually address.

Tricorder [19] is a program analysis platform that integrates multiple program analyses with an emphasis on usage metrics and scalability.

A key differentiator between CCBot and these platforms is that since CCBot makes the modifications to the code directly, the user can use whatever tool she chooses to review the changes. The Tricorder user tracking found that users would often bounce back to their own editor to apply fixes. Even when these tools offer fixes, they do not provide CCBot's strong guarantees. CCBot guarantees that the fixes it applies never remove valid executions of the original code.

Commercial analyzers such as Coverity and Klockwork[3] offer their own IDE plugins, which can mitigate the problem of switching back and forth between the analysis results and the code. However, these plugins require the use of a specific IDE while CCBot can integrate into any workflow. Further, these IDE plugins usually provide a list of problem locations, but do not modify the code itself. CCBot automatically applies the suggestions from its analyzer with correctness guarantees.

Previous work has identified Continuous Integration test execution as an opportunity to do additional analysis and testing. Continuous Test Generation (CTG) uses a Continuous Integration run to automatically generate new tests for a given project [20].

3. http://www.klocwork.com/

9.2 Design By Contract

There are many existing tools that facilitate design-by-contract. CodeContracts [8] and Spec# [21] add embedded contracts in .NET languages. Some approaches such as JML [9] for Java do not embed the contracts in the language itself, but in comments. This makes rich tooling more difficult. Eiffel [22] is a language with design by contract at its core. Though Eiffel's contracts are dynamically checked, AutoProof is a research prototype tool that brings static contract analysis to Eiffel [23].

Contract Inserter (CI) [4] is an IDE plugin that allows the developer to go through a list of contracts and accept or reject them one-by-one. Putting a human in the loop makes CI fundamentally different than CCBot. CI requires manual effort for every individual contract which is not scalable. Even more importantly, CI is built on top of Daikon [24] and is not compatible with cccheck. Daikon has some major drawbacks relative to cccheck for the automatically applied static analysis use case. Daikon needs to observe a program execution and infers **likely** invariants. This means the contracts suggested by Daikon are only likely to be correct. CI has to rely on the user to filter out incorrect contracts. It is a strong requirement to have to execute the program we want to analyze. Daikon cannot analyze libraries on their own. CCBot does not need to run the program or rely on the user to filter incorrect contracts because cccheck is a static analysis and infers **sound** invariants. The soundness property is required to achieve CCBot's level of robustness. In the CI case studies, both users said they wanted more context and better navigation – a more Visual Studio-like experience. Even though CI is integrated into the IDE it still pops up as a separate window and suffers from the "*too much switching back and forth*" problem [3]. Since CCBot inserts the contracts automatically into the source, the user can simply open the Visual Studio project and have all their usual tools at their disposal. Houdini [25] is a tool similar to Contract Inserter for ESC/Java [26] that presents potential contracts to the user in a specialized GUI. CCBot is able insert contracts into generic interfaces, but CI cannot.

Surveys have measured how contracts are used in practice by analysing a sample of large projects [6], [5]. To summarize their findings, when programmers write code in design-by-contract languages, contracts comprise a significant portion of the code – about 33% of program elements (classes, methods, etc.) and about 5% of all lines of code. These results show that automated tools for handling contracts in bulk will aid contract based verification.

9.3 Automatic Repair

CCBot is an extension of previous work that presented the concept of verified repair at the source code level [10], but previous work integrated the repairs into the Visual Studio IDE. CCBot is the first implementation of totally automatically carrying out this type of program repair.

Automatic program repair is a well-researched topic. One approach is to use Evolutionary Computation (EC) to

find a repaired version of the program that meets some fitness criteria. These approaches modify the binary or source and require: i) input data, ii) execution trace of the program, and iii) a test suite or fitness metric that successfully differentiates versions of the program with the bug from those without the bug. An example of this type of tool is ClearView [27]. This approach has even been proposed to using multiple cooperating embedded devices [28]. GenProg is an EC based approach that works at the source level through patches [29]. While EC based approaches can construct much more complex fixes than CCBot, they run into a fundamental problem that their fixes only optimize for the fitness criteria and not other factors that are important to programmers. The neglected factors include performance, memory usage, readability, and maintainability. Some approaches propose on-the-fly patching of programs [30].

AutoFix-E [31] is an automatic bug fixer for Eiffel. It relies upon an automated random testing framework AutoTest [32] to discover a trace of the program that triggers a bug first before creating the fix.

ReAssert [33] is a tool for automatically repairing unit tests. The use case is that the developer makes a (correct) change to her application, but the change breaks a test case. In this case she must fix the test case, and ReAssert is designed to do that automatically.

Previous work has attempted automatic repairs of client-server programs using differential repairs [34]. The assumption is that the client and server should have the same functionality so they act as a specification for each.

Works on recommended refactorings [35] [36] are similar to CCBot in that they generate new code to be added. CCBot is different from these approaches in that the only information CCBot needs is the code and its goals are different. Recommended refactorings typically i) rely on feedback from the programmer to rank recommendations or a training set of existing refactorings and ii) try to improve code organization rather than correct specific bugs. Similarly, statistical bug finders try to use a historical database of bugs to find new ones.

9.4 Bug Finding

Though CCBot is coupled with cccheck [8] many automatic bug finding tools (both static and dynamic) exist. There are many subcategories such as model checkers and symbolic execution (e.g. Symbolic PathFinder [37]), automated test generators (e.g. EvoSuite [38]), and statistical or pattern matching bug finders (e.g. FindBugs [39]).

An industry study found that dynamic tools are less widely used than static tools (like CCBot) [40].

9.5 Specification Generation

Cccheck tries to infer a specification (invariants, pre- and post-conditions, etc.) statically. Many approaches try to infer such a specification by observing a program execution [24] [41], by symbol elimination [42] and by many other static and dynamic techniques.

Another proposed approach to finding preconditions is mining software repositories [43]. This approach requires a corpus of libraries and clients. The assumption is that the clients' aggregate behavior should define the library's preconditions.

10 Future Work

A standardized format for warnings and annotations would greatly increase the flexibility and utility of CCBot. CCBot could take the output of any sound static analysis and automatically rewrite the analyzed code.

Along the same lines, generalizing CCBot to other languages is another direction for future work. In particular, Java with JML [9] is an interesting candidate. The main interface requirements for CCBot are i) starting the analysis, ii) gathering the results from the analysis and iii) applying the results to the source code. For C# and CodeContracts, the contracts themselves are valid C# code. CCBot starts the analysis (cccheck) by providing an assembly with the compiled code, including contracts. Then cccheck reports back the results in XML and CCBot uses the Roslyn C# compiler to apply the suggestions. Extending CCBot to support JML would be significantly different, as JML contracts are written as specially formatted comments. A different method would be needed to extract and insert the contracts. In order to be robust, the extension would require tooling to parse JML contracts and match the program elements to the suggestions from the analyser. It would be tempting to simply copy-and-paste the suggestions from the analyser into the program, since the JML contracts are just comments and would never break the compilation. However, in our experience, analyzers report where the error is, not which lines of code should change. The problem is worsened when the analyzer works over some other representation of the program than the source code (e.g., abstract syntax tree, intermediate representation, etc.). For these reasons, we envision that the JML extension would need some method of identifying program elements (classes, methods, interfaces, etc) rather than doing simple text manipulation. Fortunately, Java has excellent tooling and many analyzers exist for JML that the extension could use. The main hurdle would be that the representation of the contracts in CodeContracts and JML is significantly different.

Unfortunately there are multiple tools which compete with CodeContracts that implement design-by-contract for C#. CCBot could be extended to insert user contracts in a user-specified format. Some projects even create their own stripped-down contract framework. Even though the solutions end up being semantically equivalent, they use different method and class names so CCBot should be extended to support other implementations.

Cccheck cannot always generate a fix for each bug it finds. In this case it produces a warning and CCBot cannot insert that warning because it is not C# code. The Microsoft internal version of CCBot uses a code review tool called CodeFlow to present the warnings to the user, but in future work we could develop a tool-agnostic way of presenting warnings.

11 Conclusions

In this work we have proposed a mechanism for automatic contract insertion and our implementation of it, CCBot.

CCBot alleviates the static analysis tool "*too many annotations*" problem by automatically inserting annotations at the semantically correct location. The programmer receives an instrumented version of her original code that can be accepted into the main repository or tested with almost no effort. CCBot can instrument a large existing codebase with minimal programmer effort, but it works just as well incrementally with codebases that already have some contracts. Because CCBot uses contracts inferred by cccheck all the inserted contracts are sound. We have demonstrated CCBot's efficiency and robustness by running it on large, real codebases. This shows CCBot is a usable tool for real projects and programmers and not simply a research prototype. Our success with mitigating bugs in open source projects and getting 22 new contracts merged into the main repositories sets CCBot apart from other tools that merely find bugs.

12 Acknowledgements

We thank Xiangyu Zhang, Stephen McCamant, Michael Pradel, and the anonymous reviewers for their invaluable feedback on this work. The technical assistance from HeeJae Chang, Birendra Acharya, Jack Tilford, and Mike Barnett from Microsoft is greatly appreciated. This work was sponsored, in part, by NSF CNS-1464155.

References

[1] "StackOverFlow: how to deal with a static analyzer output," 2016, http://stackoverflow.com/questions/2070397/how-to-deal-with-a-static-analyzer-output.

[2] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu, "Comparing Static Bug Finders and Statistical Prediction," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 424–434. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568269

[3] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 672–681. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486877

[4] T. W. Schiller, K. Donohue, F. Coward, and M. D. Ernst, "Case Studies and Tools for Contract Specifications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 596–607. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568285

[5] H.-C. Estler, C. Furia, M. Nordio, M. Piccioni, and B. Meyer, "Contracts in Practice," in *FM 2014: Formal Methods*, ser. Lecture Notes in Computer Science, C. Jones, P. Pihlajasaari, and J. Sun, Eds. Springer International Publishing, 2014, vol. 8442, pp. 230–246. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-06410-9_17

[6] P. Chalin, "Rigorous Development of Complex Fault-Tolerant Systems," M. Butler, C. B. Jones, A. Romanovsky, and E. Troubitsyna, Eds. Berlin, Heidelberg: Springer-Verlag, 2006, ch. Are Practitioners Writing Contracts?, pp. 100–113. [Online]. Available: http://dl.acm.org/citation.cfm?id=2167981.2167987

[7] M. Fähndrich and F. Logozzo, "Static Contract Checking with Abstract Interpretation," in *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software*, ser. FoVeOOS'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 10–30. [Online]. Available: http://dl.acm.org/citation.cfm?id=1949303.1949305

[8] F. Logozzo, "Practical Specification and Verification with Code Contracts," in *Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '13. New York, NY, USA: ACM, 2013, pp. 7–8. [Online]. Available: http://doi.acm.org/10.1145/2527269.2534188

[9] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An Overview of JML Tools and Applications," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 3, pp. 212–232, Jun. 2005. [Online]. Available: http://dx.doi.org/10.1007/s10009-004-0167-4

[10] F. Logozzo and T. Ball, "Modular and Verified Automatic Program Repair," *SIGPLAN Not.*, vol. 47, no. 10, pp. 133–146, Oct. 2012. [Online]. Available: http://doi.acm.org/10.1145/2398857.2384626

[11] P. Cousot, R. Cousot, M. Fahndrich, and F. Logozzo, "Automatic Inference of Necessary Preconditions," in *in Proceedings of the 14th Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'13)*. Springer Verlag, January 2013. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=174239

[12] M. Bouaziz, F. Logozzo, and M. Fahndrich, "Inference of Necessary Field Conditions with Abstract Interpretation," in *10th Asian Symposium on Programming Languages and Systems (APLAS 2012)*. Springer, December 2012. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=172534

[13] M. Fowler, "Continuous Integration," 2016, http://www.martinfowler.com/articles/continuousIntegration.html.

[14] "Jenkins CI," 2016, https://jenkins-ci.org/.

[15] "Travis CI," 2016, https://travis-ci.org/.

[16] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The Design Space of Bug Fixes and How Developers Navigate It," *IEEE Transactions on Software Engineering*, vol. 41, no. 1, pp. 65–81, January 2015.

[17] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, March 2015, pp. 161–170.

[18] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran, "Making Defect-finding Tools Work for You," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 99–108. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810310

[19] C. Sadowski, J. van Gogh, C. Jaspan, E. Soederberg, and C. Winter, "Tricorder: Building a Program Analysis Ecosystem," in *International Conference on Software Engineering (ICSE)*, 2015.

[20] J. Campos, A. Arcuri, G. Fraser, and R. Abreu, "Continuous Test Generation: Enhancing Continuous Integration with Automated Test Generation," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 55–66. [Online]. Available: http://doi.acm.org/10.1145/2642937.2643002

[21] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview," in *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, ser. CASSIS'04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 49–69. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30569-9_3

[22] B. Meyer, *Eiffel: The Language*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992.

[23] J. Tschannen, C. Furia, M. Nordio, and B. Meyer, "Automatic Verification of Advanced Object-Oriented Features: The AutoProof Approach," in *Tools for Practical Software Verification*, ser. Lecture Notes in Computer Science, B. Meyer and M. Nordio, Eds. Springer Berlin Heidelberg, 2012, vol. 7682, pp. 133–155. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35746-6_5

[24] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 213–224. [Online]. Available: http://doi.acm.org/10.1145/302405.302467

[25] C. Flanagan and K. R. M. Leino, "Houdini, an Annotation Assistant for ESC/Java," in *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, ser. FME '01. London, UK, UK: Springer-Verlag, 2001, pp. 500–517. [Online]. Available: http://dl.acm.org/citation.cfm?id=647540.730008

[26] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI '02. New

York, NY, USA: ACM, 2002, pp. 234–245. [Online]. Available: http://doi.acm.org/10.1145/512529.512558

[27] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically Patching Errors in Deployed Software," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 87–102. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629585

[28] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest, "Automated Repair of Binary and Assembly Programs for Cooperating Embedded Devices," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 317–328, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2490301.2451151

[29] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 3–13. [Online]. Available: http://dl.acm.org/coitation.cfm?id=2337223.2337225

[30] M. T. Azim, I. Neamtiu, and L. M. Marvel, "Towards Self-healing Smartphone Software via Automated Patching," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 623–628. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642955

[31] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated Fixing of Programs with Contracts," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831716

[32] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf, "Programs That Test Themselves," *Computer*, vol. 42, no. 9, pp. 46–55, Sept 2009.

[33] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "ReAssert: Suggesting Repairs for Broken Unit Tests," in *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, Nov 2009, pp. 433–444.

[34] M. Alkhalaf, A. Aydin, and T. Bultan, "Semantic Differential Repair for Input Validation and Sanitization," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 225–236. [Online]. Available: http://doi.acm.org/10.1145/2610384.2610401

[35] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation System for Software Refactoring Using Innovization and Interactive Dynamic Optimization," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 331–336. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642965

[36] G. Bavota, S. Panichella, N. Tsantalis, M. Di Penta, R. Oliveto, and G. Canfora, "Recommending Refactorings Based on Team Co-maintenance Patterns," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 337–342. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642948

[37] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: Symbolic Execution of Java Bytecode," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 179–180. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859035

[38] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated Unit Test Generation for Classes with Environment Dependencies," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 79–90. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642986

[39] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens, "Improving Your Software Using Static Analysis to Find Bugs," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 673–674. [Online]. Available: http://doi.acm.org/10.1145/1176617.1176667

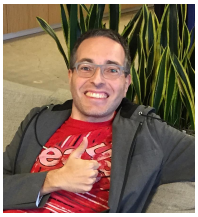
[40] R. D. Venkatasubramanyam and G. R. Sowmya, "Why is Dynamic Analysis Not Used As Extensively As Static Analysis: An Industrial Study," in *Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices*, ser. SER & IPs 2014. New York, NY, USA: ACM, 2014, pp. 24–33. [Online]. Available: http://doi.acm.org/10.1145/2593850.2593855

[41] M. Pradel and T. R. Gross, "Automatic Generation of Object Usage Specifications from Large Method Traces," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 371–382. [Online]. Available: http://dx.doi.org/10.1109/ASE.2009.60

[42] K. Hoder, L. Kovács, and A. Voronkov, "Invariant Generation in Vampire," in *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ser. TACAS'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 60–64. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987389.1987398

[43] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining Preconditions of APIs in Large-scale Code Corpus," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 166–177. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635924

Scott A. Carr is a PhD candidate in the Purdue University Department of Computer Science. Compiler-based techniques in software engineering and software security are his research interests. In software security, his work concerns the confidentiality and integrity of sensitive data in systems software and control-flow integrity protections. For more information, please visit http://www.scottandrewcarr.com.



Francesco Logozzo loves static program analysis. He has been designing and implementing widely used static analysis tools. He published papers in the most important research conferences and gave talks at main industrial conferences as e.g., Build.

Mathias Payer is a security researcher and an assistant professor in computer science at Purdue university leading the HexHive group. His research focuses on protecting applications even in the presence of vulnerabilities, with a focus on memory corruption. He is interested in system security, binary exploitation, user-space software-based fault isolation, binary translation/recompilation, and (application) virtualization.