Case Studies and Tools for Contract Specifications

Todd W. Schiller, Kellen Donohue, Forrest Coward, Michael D. Ernst
University of Washington
Seattle, WA, USA
{tws, kellend, fmc3, mernst}@cs.washington.edu

ABSTRACT

Contracts are a popular tool for specifying the functional behavior of software. This paper characterizes the contracts that developers write, the contracts that developers could write, and how a developer reacts when shown the difference.

This paper makes three research contributions based on an investigation of open-source projects' use of Code Contracts. First, we characterize Code Contract usage in practice. For example, approximately three-fourths of the Code Contracts are basic checks for the presence of data. We discuss similarities and differences in usage across the projects, and we identify annotation burden, tool support, and training as possible explanations based on developer interviews. Second, based on contracts automatically inferred for four of the projects, we find that developers underutilize contracts for expressing state updates, object state indicators, and conditional properties. Third, we performed user studies to learn how developers decide which contracts to enforce. The developers used contract suggestions to support their existing use cases with more expressive contracts. However, the suggestions did not lead them to experiment with other use cases for which contracts are better-suited.

In support of the research contributions, the paper presents two engineering contributions: (1) Celeriac, a tool for generating traces of .NET programs compatible with the Daikon invariant detection tool, and (2) Contract Inserter, a Visual Studio add-in for discovering and inserting likely invariants as Code Contracts.

Categories and Subject Descriptors: D.2.1 [Software Engineering]: Requirements/Specifications[languages, inference]
General Terms: Software Engineering
Keywords: Specifications, design by contract, invariant detection

1. INTRODUCTION

Contracts are a popular tool for formally specifying the functional behavior of software [30]. A method's contracts describe what must be true when the method is called (the method's *precondition*) and, given that the method is called correctly, what must be true when the method returns (the method's *postcondition*). Additionally for object-oriented languages, contracts can describe *object invariants*, properties that must hold for an object whenever it is visible.

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
ACM 978-1-4503-2756-5/14/05
http://dx.doi.org/10.1145/2568225.2568285

Contract-based specifications share many similarities with, and are complementary to, other development practices such as modeling and testing. In particular, the rising popularity of parameterized testing [46] and mocking frameworks has pushed testing toward specification; conversely, contracts provide a powerful semantic basis for test creation [9, 2, 27, 39]. Contracts can additionally augment or even automate refactoring [25, 11], debugging, program repair [14, 37, 28], and verification [26, 29].

To maximize benefit to developers, contract frameworks should enable developers to express semantically interesting properties with minimal annotation burden. Tools should be able to make use of the additional semantic information, yet still produce meaningful results without a full functional specification. A key observation in meeting these goals is that contract *semantics* are only partially determined by *syntax* — tooling design and assumptions (e.g., defaults) also contribute to contract semantics.

The aim of this paper is to guide the design of contract languages and tools by providing information about how developers use contract-style specifications. While this paper focuses on Microsoft Code Contracts (hereafter just Code Contracts), the ideas are also applicable to other contract languages and tools.

Contributions. This paper makes three research contributions based on an analysis of 90 open-source projects using Code Contracts and an in-depth investigation of four of the projects' use of Code Contracts:

- We identify that developers use simple contracts but underutilize expressive contracts for state update constraints, checking object state, and conditional properties (implications). For example, 75% of the projects' Code Contracts are basic checks for the presence of data (e.g., non-`null` checks), and another 3% of contracts (18% of all postconditions) repeat field assignments and return expressions from the code.
- We present evidence that annotation burden, tooling, and training are primary factors affecting the extent to which developers use contracts as specifications as opposed to argument validation/assertions.
- We performed two case studies of how developers react when shown what contracts they could write. The developers used the contract suggestions to capture more expressive contracts to support existing use cases. However, the suggestions did not lead the developers to explore new use cases for which contracts are well-suited. For example, one developer who had not previously written object invariants did not accept any of the suggested object invariants.

Based on the results, we recommend that contract language and tool designers take three complementary actions: (1) introduce tooling to reduce annotation burden, (2) make suggestions an in-

Table 1: Subject program summary. The "Static Checking" column indicates whether the project developers actively use cccheck, the static checker for Code Contracts. The "Dynamic Checking" column indicates whether the developers use Code Contracts for run-time checking in either debug or release builds. The "Other Tools" column lists the other specification, testing, and code quality tools the developers use for the project.

Project	Size (SLOC)	Downloads	Team Size	Code Contracts Introduced	Code Contract Use	Static Checking	Dynamic Checking	Other Tools
Labs Framework	11K	> 400	1	Spring 2012	Static checking	✓	✓	StyleCop[1]
Mishra Reader	19K	> 27K	1-5	Fall 2011	Debugging concurrent code		✓	JetBrains R#[2]
Sando	24K	> 500	3-6	Winter 2012	Early runtime error detection		✓	NUnit[3]
Quick Graph	32K	> 75K	1	2008	Documentation & testing		✓	Pex [45], MSTest

tegral part of tooling, and (3) curate best practices by establishing design/specification patterns. Reducing annotation burden is especially important to provide value to developers in the near-term — tools for static checking, refactoring, and testing with contracts are still relatively immature.

In support of the research contributions, this paper presents two engineering contributions: (1) Celeriac, an open-source tool for producing Daikon-compatible traces of .NET binary executions [17], and (2) an open-source Visual Studio add-in for inserting dynamically inferred contracts into C# software as Code Contracts. Inferring Code Contracts for .NET programs required the development of features not included in previous Daikon trace generators, as well as modifications to Daikon itself. These include a static analysis for determining expression comparability, support for multiple links between expressions (for hoisting inferred preconditions and postconditions to object invariants and interface contracts), and more fine-grained immutability tracking.

This paper proceeds as follows. Section 2 introduces four subject projects that will be referred to throughout the paper and describes each project's use of Microsoft Code Contracts as reported by their developers. Section 3 analyzes the developer-written Code Contracts in 90 programs, with a focus on the four subject programs. Section 4 characterizes the contracts that Celeriac and Daikon can infer for the subject programs, contrasting these to the developer-written contracts. Section 5 reports on two case studies in which the project developers added additional Code Contracts to their own software using a Visual Studio add-in that infers likely contracts from program traces. Section 6 discusses implications with respect to the design of contract languages and tools. Section 7 presents related work. Finally, Section 8 concludes.

2. SUBJECT PROGRAMS

We selected Code Contracts as a subject framework because it has a sizable user base: the extension has been downloaded over 49K times[4]. Code Contracts also has a low barrier to entry due to its integration with the popular C# language.

We automatically analyzed the 90 open-source C# projects listed on Ohloh[5] that use Code Contracts. These projects contain 3.5M source lines of code (SLOC). For context, Ohloh indexes 12M SLOC of Eiffel code across 331 projects; it indexes 568M SLOC of C# code across 44,440 projects.

We performed a more detailed analysis of four of the projects. We selected these projects because they were all actively developed, used, and employing contracts in a meaningful way. Additionally, they are diverse in both application domain and their reason for adopting Code Contracts. Table 1 overviews each project's use of Code Contracts.

The following paragraphs describe each project's adoption of Code Contracts as reported by the project's developers via questionnaire (and the additional Skype interviews performed for Mishra Reader and Sando as part of developer studies in Section 5). Each paragraph additionally describes the project's use of other specification, testing, and code quality tools as they relate to the project's use of Code Contracts. Of particular significance is `cccheck`, the static contract checker packaged with the Microsoft Code Contracts framework. The checker uses a modular abstract-interpretation-based analysis to report unsatisfied contracts and to suggest additional contracts. To fully benefit from using `cccheck`, developers must add contracts to all the code being checked, as well as add contract stubs for method calls to external assemblies.

Labs Framework. The Labs Framework[6] is a framework for managing "experiments" demonstrating the behavior of an API or library. The static Code Contracts checker, `cccheck`, is enabled by default in the project. The project does not include any formal unit tests, instead relying on sample applications built with the framework.

Mishra Reader. Mishra Reader[7] is a Google Reader client. The lead developer introduced Code Contracts to the core library to help reduce bugs in multithreaded code. The developers add Code Contracts after the methods are implemented, to aid in debugging (as opposed to design by contract). At one point, the developer considered abandoning Code Contracts due to a lack of support for debugging with contracts in `async` and `await` constructs (the bytecode rewriter did not properly modify the debugging information); Microsoft has since added debugging support for these constructs. The team does not use `cccheck`, citing that it is slow and issues too many false positive warnings.

Sando. Sando[8] is a Lucene-based code search engine that includes a Visual Studio interface. Code Contracts were introduced to the project because one of main contributors had seen a webinar on Code Contracts and wanted to try them. The team primarily uses contracts in the core functionality. In particular, contracts are used in the Index component because placing bad data into the index can result in later errors. Contracts are typically written after a change is made but before running the unit test suite prior to check-in.

The developer we interviewed was not aware of the static checker for Code Contracts. The project does not use any other static analysis tools, in part because the team has limited build engineering resources. Code Contracts is seen as offering additional quality assurance without requiring additional build engineering, and likely makes the team less likely to try other quality assurance tools. The developer we interviewed feels that Code Contracts has sped the discovery of bugs and regressions, as well as increasing confidence in the quality of code containing contracts.

[1]https://stylecop.codeplex.com
[2]https://www.jetbrains.com/resharper
[3]http://www.nunit.org
[4]http://visualstudiogallery.msdn.microsoft.com/1ec7db13-3363-46c9-851f-1ce455f66970
[5]https://www.ohloh.net/
[6]https://labs.codeplex.com
[7]https://mishrareader.codeplex.com
[8]https://sando.codeplex.com

Quick Graph. Quick Graph[9] is a data structure and algorithm library. Code Contracts were introduced to the project to serve as documentation and for use in conjunction with the Microsoft's Pex white-box testing tool, which the Quick Graph developer also develops [45]. While the project has a single developer, a member of the Code Contracts team contributed to the project by fixing contracts that were malformed but were erroneously considered valid by older versions of the toolset; we included Quick Graph as an example of a well-annotated project. While, as anticipated, Code Contracts have led to the discovery of some bugs, the developer has also found that using contracts has forced a cleaner API and has exposed bugs in the Code Contracts and Pex tools themselves. The project does not use `cccheck` since it was not ready for use when the developer was adding contracts.

3. DEVELOPER-WRITTEN CONTRACTS

This section characterizes the types of specifications that the developers of the subject projects captured using Code Contracts. We aim to answer the following two questions:

RESEARCH QUESTION 3.1. *What properties do developers use Code Contracts to enforce (**semantics**)?*

RESEARCH QUESTION 3.2. *Are developers able to efficiently express these properties using Code Contracts (**syntax**)?*

The developers predominately use contracts to perform argument validation (consistent with Polikarpova et al.'s observations [39]). Approximately three-fourths of the contracts just check for the presence of data; an additional 3% of contracts (18% of all postconditions) repeat field assignment and return statements from the code.

3.1 Methodology

We divided contracts into three general categories: common-case, repetitive, and application-specific. *Common-case contracts* enforce expected (common) program properties: that data is present, strings aren't blank, collections aren't empty, indices are in-bounds, and methods don't modify unrelated variables. Common-case contracts often check for exceptional program behavior that produced a degenerate value (e.g., returning `null`) instead of throwing an `Exception`. *Repetitive contracts* repeat exact statements from the code: that a method returns a field, assigns a variable to a field, or returns a specific value (i.e., the contract repeats the return expression). *Application-specific contracts* enforce richer semantic properties: valid argument values, how state is modified, the relation between expressions, indicators of object state, and conditions under which properties hold (i.e., implications).

Common-case and repetitive contracts are good candidates for language/tool "optimizations" such as defaults and inference. Developer time is better spent writing expressive application-specific properties. Similarly, for developers concerned about code bloat, common-case and repetitive contracts can "crowd out" the semantically richer application-specific contracts.

We wrote a Roslyn[10] program to categorize each contract into the finer categories of Table 2. We ran the program on 90 C# projects (3.5M source lines of code) that use Code Contracts. The program (available on the paper website) categorizes each expression or top-level conjunct in a `Requires`, `Ensures`, and `Invariant` statement. We manually refined the categorization rules by spot-checking the results.

[9]https://quickgraph.codeplex.com

[10]http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx

The program looks only at the contract expressions themselves and errs on the side of categorizing a contract as application-specific. For example, the program categorizes the contract `idx >= 0` as a "Lower/Upper Bound" contract rather than a "Bounds Check" even if the variable `idx` is used as an index in the body of the method. This has the effect of making our assessment of the application-specific nature of developer-written contracts overly generous.

3.2 Results

Table 2 shows Code Contract content. The 90 projects had 43,823 top-level contract clauses across 3.5M source lines of code. Of those clauses, 29,770 (68%) were preconditions, 11,355 were postconditions (26%), and 2,698 (6%) were object invariants.

Out of all contract clauses, 32,072 (73%) just check for the presence of data (cf. rows "Nullness", "Null/Blank", and "Non-Empty"). For postconditions, 1,990 of the 11,355 clauses (18%) are "Getter/Setter" or "Return Value" specifications which are repetitive with the code (cf. the ENS columns).

The following paragraphs describe other characteristics of the contracts, including the use of special postcondition methods (e.g., `OldValue`) and object-oriented features (object invariant methods and contract classes). These methods and features gain their semantics from the bytecode rewriting process. Object invariants and contract classes are particularly interesting because they allow developers to capture application properties that are true at multiple points in the program without significantly increasing annotation burden — the contracts are automatically propagated by the bytecode rewriter.

Labs Framework. The Labs Framework uses indicator properties more than the other projects. Indicator properties describe type-state and/or which methods and properties can be called. Contracts using indicator properties (cf. the "Indicator" row in Table 2) refine the interface guarantee offered by the type system. For example, the Labs Framework uses checks for `IsEnabled` to specify methods that can only be called when a lab is active. Contracts over indicator properties convey rich semantic information with minimal annotation burden (syntax). Additionally, indicator properties provide a client method with a concise way to determine if/when it can call the object's methods.

Some contracts are lexically enclosed within `#if` preprocessor conditionals, impairing readability. Unlike other projects, the contracts for the project differ based on the target platform: checks for `IsNullOrWhitespace` are used when targeting the Windows Phone, and `IsNullOrEmpty` are used for other targets. Recent support in the Code Contracts framework for contract abbreviator methods would enable the developer to refactor this pattern as a method call.

The project additionally makes use of special postcondition methods and the object-oriented features of Code Contracts. Of particular note is the use of `OldValue` to write frame conditions, contracts stating that a method does not modify a certain field or argument. These contracts are necessitated by the use of `cccheck` — the checker depends on frame conditions to reason modularly about method calls.

Mishra Reader. The Mishra Reader project primarily contains argument validation contracts (cf. the REQ column). The developers chose to include the exception type to throw (e.g., `ArgumentNullException`) with precondition contracts, making the contracts more informative. No contracts are written for private methods — since external input has already been validated, the developer feels that these contracts do not add enough value to justify code bloat and run-time overhead.

Interface contracts (i.e., contract classes) are provided for 10 interfaces, which primarily connect to external services (Google Reader, Facebook, and Twitter). However, the special quantification and

Table 2: Developer-written Code Contracts. The columns REQ(uires), ENS(ures), and INV(ariants) correspond to preconditions, postconditions, and object invariants, respectively. The contracts counted for each category (row) are mutually exclusive. The vast majority of preconditions written with Code Contracts simply check the presence of information; the majority of postconditions ensure that information is produced, or specify which information is produced. Section 4 characterizes the contracts that the developers could have written, as determined by contract inference.

Subject Program Contract Usage

Contract Property	Example	Labs Framework			Mishra Reader			Sando			Quick Graph			90 projects	
		REQ	ENS	INV	REQ	ENS	INV	REQ	ENS	INV	REQ	ENS	INV	Med.	Mean
Common-Case		82%	64%	82%	87%	71%	0%	91%	100%	-	81%	23%	27%	**80%**	**75%**
Nullness	`arg != null`	285	104	58	37	6		52	5		632	34	4	67%	66%
Null/Blank	`!string.IsNullOrEmpty(arg)`	33	11	6	17	4		10			13	1		4%	7%
Non-Empty	`list.Count() > 0`	3						12			1			0%	1%
Bounds Check	`idx < list.Count()`	11									7			0%	1%
Frame Condition	`this.fld == OldValue(this.fld)`		18											0%	0%
Repetitive with Code		0%	7%	0%	0%	0%	0%	0%	0%	-	0%	30%	0%	**0%**	**3%**
Getter/Setter	`this.fld == arg`		7									16		0%	2%
Return Value	`Result<T>() == this.fld > 0`		8									30		0%	0%
Application-Specific		18%	29%	18%	13%	29%	100%	9%	0%	-	19%	47%	73%	**18%**	**22%**
Constant	`this.fld == 3`										1			0%	0%
Lower/Upper Bound	`count >= 0`	21	1	1	5	1	2	1			51	4	5	3%	6%
State Update	`this.fld > OldValue(this.fld)`											14		0%	0%
Expr. Comparison	`!arg1.Equals(arg2)`	4	2								25	9	4	3%	5%
Membership	`list.Contains(elt)`										56	11		0%	1%
Indicator	`this.IsEnabled`	44	34	8	3							11		1%	5%
Implication	`arg == null || arg.Count > 0`	1	23	5		3		5			13	11	2	1%	2%
Other	`Func(arg1) == Func(arg2)`	2						1			10	12		1%	3%

postcondition methods provided by Code Contracts are not used, as highlighted by the fact that no contracts make use of quantification (e.g., `Contract.ForAll`). The lack of specifications for collection elements is consistent with the lack of contracts on private methods — validation of all the elements inserted into the collection partially implies the collection specification without incurring run-time overhead. However, as with private methods, neither runtime checks nor the static checker can enforce that all elements are indeed validated before insertion. Additionally, later modifications to the elements may violate the intended contracts for the collection.

Sando. The Sando developers use contracts as though they were standard argument validation and assertions (i.e., `Debug.Assert`). The project makes no use of contract classes or invariant methods. As with Mishra Reader, the project's contracts contain no use of Code Contracts's quantification expressions. However, in one location the C# `FindAll` method is used to check that a property does not hold for any of the elements (as opposed to `Contract.ForAll` or `Contract.Exists`). The lack of object-oriented and Code Contract-provided methods indicates a lack of familiarity with the contract framework's features, as was confirmed by the developer case study in Section 5.4.

Quick Graph. Compared to the other projects, Quick Graph includes a higher proportion of application-specific contracts. Many of these enforce algorithmic properties such as the color of a node during edge coloring. To express complex properties, contracts include helper method calls and lambda expressions (cf. the "Other" row). In conjunction with logic connectives and the heavy-weight syntax for special postcondition methods, these make many contracts inscrutable to the untrained eye, e.g.:

```
Contract.Ensures(
 !Contract.Result<bool>() ||
 (Contract.ValueAtReturn<IEnumerable<TEdge>>(out rslt) != null
   &&
   (typeof(TEdge).IsValueType ||
   Enumerable.All(
       Contract.ValueAtReturn<IEnumerable<TEdge>>(out rslt),
       e => e != null))
));
```

The developer could extract the logic into a separate method to eliminate the need for multiple special postcondition method calls.

As a data structure and algorithm library, the project relies heavily on interfaces for graphs, algorithms, and collections. 29 of these interfaces are annotated with contracts. However, the project contains relatively few object invariants — just 11 objects include invariant methods. These invariants are for collections classes (heaps) and the core graph abstractions. They predominately express basic facts about nullness and that countable properties (e.g., edges) are non-negative. More precise invariants are provided for the `BinaryHeap` and `BidirectionalGraph` classes, however these are excluded via preprocessor macro by default (since they are expensive "deep invariants"). These excluded invariants are not included in Table 2.

3.3 Discussion

There are material differences in contract usage across the projects. These relate to the different use cases that contracts were supporting: detecting one's own bugs vs. checking for ill-behaved clients, simple assertions vs. rich behavioral specifications, etc. One explanation for the differences in contract usage is that the developers using contracts for more than debugging (e.g., with `cccheck` or Pex) have greater incentive (or are forced) to write richer contracts. An alternative explanation is that the developers inclined to use the other tools are also inclined to use contracts more extensively. In either case, a developer who underutilizes the special postcondition methods and object-oriented features is missing out on exactly the features that make Code Contracts more powerful and more concise than standard argument validation/asserts. Conciseness and annotation burden (in addition to expressivity) is important because it affects whether or not developers use tools that require relatively complete specifications, such as `cccheck` [15].

The large number of nullness contracts relative to the other contract types suggests that nullness contracts may be "crowding out" application-specific contracts — that is, the developers' limited resources (time, lines of code, etc.) are being consumed by writing nullness contracts. Nullness contracts do provide value since they guarantee that types are inhabited, and therefore support the interface guarantees provided by the type system. However, since non-`null` is the common case [6], the annotation burden is difficult to justify.

4. CONTRACT INFERENCE

This section reports on what Code Contracts *could have* been written in the subject programs. We determined the potential contracts by running Daikon on a trace of the program's execution [17]; this methodology mimics the practice of a developer inferring the "contract" for a program by generalizing how they see the program behave. We use the results to explore two research questions.

RESEARCH QUESTION 4.1. *To what extent are the contracts that developers could have written application-independent (**semantics**)?*

The results indicate that, in addition to writing numerous "common-case" and "repetitive" contracts, the developers of the subject programs could have written a higher proportion of application-specific contracts, particularly constraints on state updates, indicator expressions, and implications.

RESEARCH QUESTION 4.2. *What are the differences (qualitative and quantitative) between developer-written Code Contracts and the contracts that the developers could have written (**syntax** and **semantics**)?*

From the data in Section 3 (and our own experience), we hypothesized that developers disproportionately write basic contracts. Additionally, Polikarpova et al. note that developers are typically worse at writing postconditions than preconditions [39]. The results support these expectations.

4.1 Methodology

As a proxy for determining which contracts could be written for the subject programs, we used the Daikon invariant detector [17] to infer invariants. Daikon takes as input one or more execution traces and employs statistical methods (e.g., minimum support and confidence heuristics) to infer likely method preconditions, method postconditions, and object invariants. The contracts that Daikon infers are sound with respect to the observed executions — i.e., it does not infer any properties that are falsified by any traces.

For each program, we instrumented an assembly using the Celeriac trace generator (Section 4.2), and then ran the programs using tests or example inputs. For the Labs Framework and Quick Graph, we used the main assemblies. For Mishra Reader and Sando, we used the assemblies that are the subjects of the developer case studies in Section 5. We generated a trace for the Labs Framework by running the labs for the Rxx project[11], Mishra Reader by using the application normally, Sando by running its integration test suite, and Quick Graph by running a subset of the unit test-suite (excluding long-running tests). Celeriac's sampling feature was used for the Labs Framework, Sando, and Quick Graph, to reduce run time. Section 4.3.3 addresses the shortcomings of Daikon and Celeriac as they relate to this study.

4.2 Celeriac .NET Trace Generator

To infer likely invariants for .NET programs with Daikon, we built Celeriac, a tool that dynamically instruments .NET binaries to produce Daikon-compatible program traces. Celeriac uses the CCI Metadata IL rewriting library[12] to insert callbacks into managed C# code; the callback walks over data structures (i.e., fields) and performs pure method calls. Since Celeriac operates directly on a .NET binary, it can be used to generate traces without build integration. The following subsections describe three features to support the unique challenges encountered when tracing .NET programs, and the improvements we made to Daikon to support these features.

[11] https://rxx.codeplex.com

[12] http://ccimetadata.codeplex.com

4.2.1 Interface Inference and Behavioral Subtyping

Code Contracts enable a developer to strengthen the postconditions on a method implementing an interface or overriding another method. This is compatible with behavioral subtyping. (It is a limitation of Code Contracts that they do not allow the developer to modify the precondition, even though that would also be compatible with behavioral subtyping.)

Prior work has observed that Daikon produces invariants that violate behavioral subtyping by not incorporating inheritance information [12]. To address this problem, Celeriac links arguments and fields to the corresponding arguments and members of any supertype/interface. The link information causes Daikon to lift contracts that hold across all the implementations to the interface/supertype.

We modified Daikon to support multiple links per argument and field; Daikon previously used single links for encoding object invariant relationships. We added a post-processing step to Daikon that discards the non-lifted preconditions (the implementation-specific preconditions) from the implementation methods.

4.2.2 Comparability Analysis

By default, Daikon compares all values of the same primitive type and all references (including those which violate the language's typing rules). For example, in a program with `int` variables representing months, days, and years, Daikon will infer contracts comparing month values to year values. To prevent these spurious contracts from being inferred, Daikon supports "comparability sets" which identify groups of variables that can be compared.

Existing Daikon comparability analyses for Java and C/C++ programs are dynamic, recording variable interactions at run time [17]). For Celeriac, we opted to implement a conservative static comparability analysis using the CCI Code Model and AST API[13]. The analysis works in three steps:

1. For each method, calculate a comparability summary (comparability sets) of which expressions (fields, parameters, and return value) are used together in a binary operation or assignment statement.
2. Until a fixpoint is reached, for each call site, update the caller's comparability summary using the comparability summary of the method being called (the callee).
3. For each type, calculate the comparability summary by merging the comparability summaries of its methods.

For calls to external assemblies (i.e., methods that don't have a comparability summary), the analysis conservatively assumes that all method arguments with compatible types are in the same comparability set; two types are considered to be compatible if either either type is assignable to the other.

4.2.3 Read-only Variables

The .NET languages include a `readonly` keyword that specifies that a variable must be assigned in the constructor, or given a constant value; the equivalent in Java is the `final` keyword. For read-only variables, Daikon produces redundant postconditions stating that the variable has not been modified (e.g., for a variable `x`, `x == orig(x)`). We introduced an expression flag to Daikon to filter out these cases from the Daikon output.

The `readonly` keyword is shallow. For reference types, the keyword prevents the reference from being reassigned, but does not prevent the object from being modified through the reference. Therefore, when considering a composite expression (e.g., `this.foo.bar`), Celeriac cannot naively use the `readonly` attribute of the last field.

[13] https://cciast.codeplex.com

Table 3: Code Contracts inferred by Daikon from program traces produced with Celeriac (plus a summary of developer-written contracts for comparison). The header indicates the assembly used for each project and the size of the assembly in source lines of code (SLOC). The Mishra Reader View Models component and Sando Indexer component are the subjects of the developer case study in Section 5. The rows and columns are the same as in Table 2 with the addition of the Inf(erred) columns and Dev(eloper) columns that show the percentages of inferred and developer contracts for the project. Frame conditions, which are useful for static analysis but rarely written by developers in practice, make the proportion of common-case contracts appear comparable. However, when these contracts are excluded, it becomes more clear that developers write a higher proportion of common case contracts than what is inferred. Developers miss opportunities to write state update, indicator, and implication contracts.

Contracts Inferred from Dynamic Traces

Contract Property	Labs Framework Labs.dll (7.4K)					Mishra Reader ViewModels.dll (4.4K)					Sando Indexer.dll (2.3K)					Quick Graph QuickGraph.dll (21K)					90 projects Written	
	REQ	ENS	INV	INF.	DEV.	REQ	ENS	INV	INF.	DEV.	REQ	ENS	INV	INF.	DEV.	REQ	ENS	INV	INF.	DEV.	Med.	Mean
Common-Case	68%	81%	73%	78%	*77%*	59%	73%	70%	70%	*82%*	48%	60%	50%	58%	*92%*	73%	80%	80%	79%	*71%*	*80%*	*75%*
Nullness	1031	1126	180	33%	*65%*	673	947	132	28%	*55%*	247	801	217	22%	*66%*	1280	1405	371	29%	*69%*	*67%*	*66%*
Null/Blank	132	159	28	5%	*7%*	4	12	5	0%	*27%*	20	33	13	1%	*12%*				0%	*1%*	*4%*	*7%*
Non-Empty	67	60	3	2%	*0%*	38	90	5	2%	*0%*	107	259	26	7%	*14%*	127	239	16	4%	*0%*	*0%*	*1%*
Bounds Check				0%	*2%*			1	0%	*0%*	4	7	11	0%	*0%*	15	16	3	0%	*1%*	*0%*	*1%*
Frame Condition		2669		38%	*3%*		2430		39%	*0%*		1586		27%	*0%*		4817		46%	*0%*	*0%*	*0%*
Repetitive with Code	0%	1%	0%	1%	*2%*	0%	4%	0%	3%	*0%*	0%	1%	0%	0%	*0%*	0%	2%	0%	1%	*5%*	*0%*	*3%*
Getter/Setter		45		1%	*1%*		132		2%	*0%*		12		0%	*0%*		82		1%	*2%*	*0%*	*2%*
Return Value		28		0%	*1%*		36		1%	*0%*		12		0%	*0%*		48		0%	*3%*	*0%*	*0%*
Application-Specific	32%	17%	27%	21%	*21%*	41%	24%	30%	27%	*18%*	52%	39%	50%	42%	*8%*	27%	18%	20%	20%	*24%*	*18%*	*22%*
Constant	224	324	24	8%	*0%*	299	555	25	14%	*0%*	210	541	100	15%	*0%*	316	617	37	9%	*0%*	*0%*	*0%*
Lower/Upper Bound	1			0%	*3%*	9	9	4	0%	*10%*	7	20	8	1%	*1%*	86	109	32	2%	*6%*	*3%*	*6%*
State Update		25		0%	*0%*		8		0%	*0%*		161		3%	*0%*		247		2%	*1%*	*0%*	*0%*
Expr. Comparison	1	7		0%	*1%*	12	31		1%	*0%*		1		0%	*0%*	11	22		0%	*4%*	*3%*	*5%*
Membership				0%	*0%*				0%	*0%*	1	2		0%	*0%*				0%	*7%*	*0%*	*1%*
Indicator	288	351	40	10%	*12%*	23	63	20	2%	*4%*	115	413	101	11%	*0%*	90	155	10	2%	*1%*	*1%*	*5%*
Implication		93		1%	*4%*		269		4%	*4%*		502		9%	*6%*		273		3%	*3%*	*1%*	*2%*
Other	52	56	16	2%	*0%*	151	215	11	6%	*0%*	76	115	56	4%	*1%*	27	59	16	1%	*2%*	*1%*	*3%*

To compute whether an expression should be flagged as `is_readonly`, Celeriac starts at the root (e.g., `this`, in the the case of `this.foo.bar`) of the composite expression and propagates reference immutability and value immutability information [48]. Reference immutability is propagated via the `readonly` fields (`this` is reference-immutable). Value immutability is propagated / introduced for `readonly` fields that have an immutable type. Celeriac considers a type to be immutable if, and only if, it is composed of `readonly` fields with immutable types.

4.3 Results

Table 3 shows the inferred contracts across the projects using the categories from Section 3.1. Table 3 differs from Table 2 in two ways. First, the results are for a single assembly within the subject project, as indicated in the table. Second, implications that "guard" a possibly null expression (e.g. `x != null implies x.f != null`) are categorized using the type of property that is guarded, as opposed to being categorized as an implication. This adjustment has the effect of categorizing some contracts as "common-case" or "repetitive" that would be categorized as an implication (and therefore "application-specific") in Table 2. Therefore, the categorization would tend to make the inferred contracts appear less application-specific than the developer-written contracts.

4.3.1 Application-Independence

Nullness and frame conditions dominate the inferred contracts, accounting for 49% – 75% of the contracts for each project (cf. the "Nullness" and "Frame Condition" rows). This is expected as programs operate on data, and methods generally only modify a portion of the program state. As reported in Section 3, developers rarely write frame conditions. When frame conditions (cf. the "Frame Condition" row) are excluded from the inferred contracts, the results reveal a higher proportion of application-specific contracts than what developers write (34% – 57% of the contracts for each project).

While frame conditions are important to specify what a method does not *not* do in the presence of mutable data types, the number of reported conditions could be reduced by making two changes to the toolset: (1) Daikon could report method purity concisely, rather than listing every side effect that a method does *not* have. This would be similar to Daikon's treatment of the `modifies` clause when it creates JML output. (2) Celeriac could perform static analysis to infer which properties return the same value when called twice (`Date.Now` is an example that does not). This would allow Celeriac and Daikon to treat more expressions as read-only (see Section 4.2.3).

The "Constant" row shows that many contracts specify a constant value. These contracts (and some in the "Other" category that specify the set of values an expression can have) are a product of the sample executions/tests not achieving value coverage for methods. For example, we used only a single username and password when running Mishra Reader.

Every inferred implication was a postcondition for a method call that returns a Boolean value. In order to infer other implications, Daikon requires a user-supplied list of predicate expressions. Celeriac lacks this feature, and consequently, no implications were inferred for preconditions (cf. the REQ columns) or object invariants (cf. the INV columns).

4.3.2 Relationship to Developer-Written Contracts

There are two notable differences between the contracts that developers have written, and the contracts that the developers could have written: (1) developers write relatively fewer postconditions, and (2) developers write relatively less-expressive contracts.

Postconditions. Based on previous research [39] and our own experience, we expected that, relatively speaking, developers would write fewer postconditions (ensures contracts) than they could have as indicated by Daikon. The results confirmed our expectation: for every project and nearly every contract type, Daikon infers more postconditions than preconditions (cf. the REQ and ENS columns).

This suggests that the strong developer bias toward preconditions (68% of written contracts are preconditions) cannot be attributed to an absence of potential postconditions. In particular, there are opportunities to capture how a method updates program state. This can be done by relating pre-state and post-state values (cf. the "State Update" row in Table 3) or by setting indicator properties (cf. the "Indicator" row). For example, the Sando developers could add postconditions to specify which methods increment counts, e.g.:

```
this.Reader.GetRefCount() >=
    Contract.OldValue(this.Reader.GetRefCount())
```

Additionally, there are opportunities to use implication (the "Implication" row) to capture richer method return and state update behavior.

Expressivity. Daikon includes templates for a wide variety of contracts, many of which are non-trivial to express efficiently using Code Contracts (e.g., that all elements in a collection are distinct). Therefore, it was not surprising that inference discovered more expressive contracts than the developers wrote for the Labs Framework, Mishra Reader, and Sando. For Quick Graph, Daikon was able to infer interesting application-specific contracts, but they were, in many cases, less expressive than what the developers wrote. One reason is that Daikon does not infer contracts that contain calls to helper methods that take arguments.

The inferred implications generally describe type-state. For example, for Sando, Daikon infers that an `Indexer` is either "usable" or "disposed":

```
(this._disposed == true) == (this.IsUsable == false)
```

Daikon additionally infers the properties associated with each object state, e.g.:

```
(this.IsUsable == false) ==
      (this.Reader.TermPositions == null)
```

In cases where an indicator property/variable is not present, a non-`null` value for a field serves as a proxy for type-state (particularly for simple initialization).

There are downsides to a developer writing the more expressive/complex of these properties in the Code Contracts framework. First, they are verbose, especially when involving pre-state or return values. For commonly-used contracts, though, helper methods and contract abbreviator methods can be introduced to mitigate the verbosity problem. Second, they cannot be checked statically, resulting in a warning that the contract is unproven. While static checking can be disabled for a contract via annotation, disabling static checking would require the developer to additionally write simpler invariants (implied by the more complex invariant) or otherwise write assumptions at the client sites. This developer doesn't receive additional compile-time benefits for writing the semantically expressive contract, and instead incurs extra annotation burden and the burden of figuring out why the static checker cannot prove the contract. Third, they cannot be controlled readily via the Code Contracts configuration. For example, run-time checks for the built-in Code Contract quantification methods (e.g., `ForAll`) can be toggled via the configuration, but checks utilizing the more expressive LINQ methods cannot.

For example, consider writing contracts for a method that adds an entry to a dictionary/map. Daikon generates postconditions that the size of the dictionary has increased by one and that the keys and values are supersets of the pre-state versions. The static checker does not reason about individual collection elements, so the contract that the original elements are retained would have to be left as an expensive dynamic check.

4.3.3 Threats to Validity

The key threat to external validity is that we give detailed information about a single assembly from each of four C# projects. Though we chose them to be diverse, the results might not generalize.

The key threat to internal validity is that if Daikon does not do a good job inferring contracts, then comparing developer-written contracts to Daikon's output may not be informative.

Nimmer and Ernst [32] reported that contracts inferred from even small test suites were relatively precise, with less than 10% on average being incorrect (from a verification perspective). Polikarpova et al. [38] later reported that one third of contracts inferred by Daikon were incorrect or irrelevant. However, most of the uninteresting contracts that they report are caused by limitations in their Citadel tool, which does not output comparability, inheritance, or constant information.

Use of a different contract inference tool (e.g., DySy [13]) could produce different results; however, this would not diminish the primary result that developers omit many interesting application-specific contacts. Improvements to Daikon's recall (say, by adding new contract templates or by improving Celeriac's method purity or comparability analysis) would demonstrate an even larger gap between what contracts developers write and the contracts they could write.

5. DEVELOPER CASE STUDY

Sections 3 and 4 characterized the difference between the contracts that developers write and the contracts that developers could write. This section presents two case studies of how developers react when shown the difference.

RESEARCH QUESTION 5.1. *How do developers decide which properties to record and enforce (**semantics**)?*

A developer from the Mishra Reader project and a developer from the Sando project inserted inferred contracts into their project. They used the Contract Inserter (Section 5.1), a Visual Studio interface for viewing, inserting, and documenting contracts discovered with Celeriac and Daikon. We interviewed each developer about their experience and decision-making process. When suggesting improvements to the Contract Inserter, both developers stressed IDE integration.

When the tool identified new/different properties that the developer could capture using contracts, the developers wrote these in similar locations, and used them for similar purposes as they had used contracts in the past. The tool suggestions did not lead the two developers to adopt new use cases, even when those use cases were more powerful or a better match for Code Contracts and the developers' needs. For example, the tool suggestions did not lead the Mishra Reader developer to expand from annotating only public methods (to catch misbehaving clients), to annotating private ones as well (to find bugs in the developers' own code). Nor did the tool suggestions lead the two developers to expand from using Code Contracts as assertions and argument validation, to more comprehensive behavioral specifications. And the two developers did not transition from annotating methods one at a time to annotating data structures with object invariants.

5.1 Contract Inserter Add-in

To support C# developers in specifying their programs with Code Contracts, we developed the Contract Inserter, a Visual Studio add-in that inserts inferred contracts as Code Contracts or as documentation. This section describes the design of the interface (Figure 1).

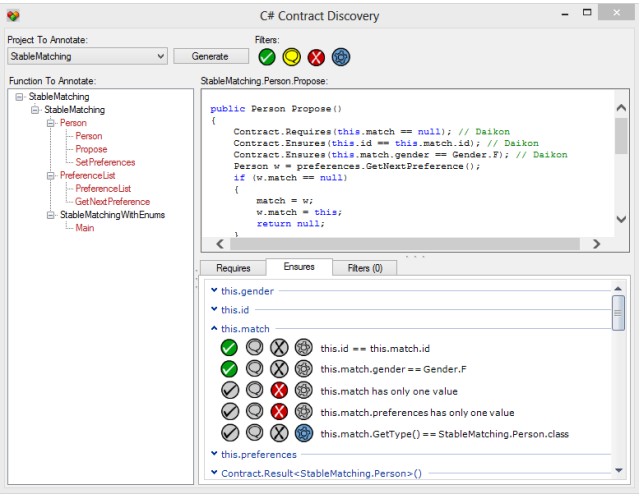

Figure 1: The Contract Inserter Add-In (Section 5.1). Left pane: the class and method tree. Top right pane: source code with preview of inserted contracts and documentation. Bottom right pane: inferred contracts, grouped by variable. The developer can insert or ignore each contract by clicking an icon. The developer can introduce filters by right-clicking on a contract and selecting a filter from the context menu (e.g., filter all contracts that compare two variables).

Contract Actions. For each method and object, the user interface lists likely contracts grouped by variable. The developer can toggle between viewing the contracts as a Code Contract or in Daikon's concise English-like output format. For each contract, the developer can take one of four actions:

- Insert as Code Contract
- Insert as documentation
- Ignore, because the contract is not true
- Ignore, because the property is an implementation detail

The developer performs the action by clicking on the associated icon next to the contract in the display.

Inserting a contract as documentation adds an XML-documentation clause in the same format as that generated by the `ccdoc` tool packaged with Code Contracts. The contract is written using Daikon's concise English-like output.

A developer can hide ignored contracts, so that the Contract Inserter displays contracts that are inserted and work-to-be-done. Because the Daikon output can be noisy, developers can also hide contracts by contract type or the variables involved.

To aid readability and to support Daikon's more complex contracts (e.g., greatest common denominator), the tool is packaged with a C# library for expressing inferred contracts concisely. For example, the `OneOf` extension method obviates the need for chained or-expressions, e.g., `x.OneOf(1,2,3)` instead of `x == 1 || x == 2 || x == 3`.

5.2 Case Study Methodology

For each project, we selected an assembly (sub-project) for the developer to annotate with Code Contracts. To shorten the developers' time commitment, we generated a trace prior to the study (as described in Section 4.1). We provided each developer with the Contract Inserter installer, the program trace, and written instructions. We instructed each developer to use the Contract Inserter for approximately one hour to add Code Contracts to their project. Immediately following each developer's use of the tool, we performed a semi-structured interview with the developer via Skype. The study materials are available on the paper website.

5.3 Mishra Reader Case Study

The Mishra Reader developer annotated the View Model assembly, the model subcomponent of the application's Model-View-Control architecture. Initially, the assembly contained just 10 preconditions and a single object invariant across 80 classes and 300 methods.

Determining Which Properties are Valid. The developer cited his intuition and knowledge of the project to decide when a contract reported by the tool was valid. However, for contracts involving instance variables, he stated that a lack of context about the instance fields of the object was a barrier to determining the validity of a contract. In these cases, he used the standard Visual Studio interface to review the source for the entire class.

The developer made two tool suggestions. First, the developer suggested that the add-in be more tightly integrated into the Visual Studio editor, possibly as an add-in to the popular ReSharper tool[14]. Second, the developer suggested that the tool should provide information (e.g., values) explaining *why* the tool reported a Code Contract. The Contract Inserter could provide concrete example values from the trace.

Determining Which Properties to Enforce. During the session, the developer inserted 13 contracts and marked 28 contracts as false. Of the contracts the developer inserted, 11 were nullness checks, 1 stated that a return value was non-negative, and 1 stated that a method set the class `is_loaded` flag. Of the contracts the developer marked as false, 10 were nullness checks, 10 were contracts for the run-time type of an expression, and 8 were checks that an expression was constant-valued. The Contract Inserter now hides contracts for the run-time type of an expression by default because they are implementation details (C#'s type system enforces static type correctness; behavioral subtyping implies that run-time type does not affect program behavior). These contracts are not included in the results for Section 4, either.

The developer cited three reasons for not inserting valid contracts: (1) the developer believed contracts should only be written at module boundaries, i.e., public methods, (2) the developer did not want to introduce run-time overhead, and (3) the developer wanted to avoid code bloat. When asked if he would add private method contracts and/or potentially expensive contracts if they could be disabled at run-time (which the Code Contract rewriter supports), he reconfirmed his decision to exclude the contracts. The developer did not choose to insert any contracts as documentation, citing that properties that are important enough to be documented should be enforced.

Overall, these decisions appear to be inconsistent with the developer's stated goal of using contracts as tool for debugging multi-threaded code. Contracts on private methods help to localize errors which are caused by the interaction between threads (and therefore would not be detected by argument validation on public methods).

5.4 Sando Case Study

The Sando developer annotated the Indexer subcomponent of the application. Initially, the Indexer assembly contained 17 contracts across 34 classes and 182 methods.

Determining Which Properties to Enforce. During the session, the developer inserted 35 contracts, did not insert any contracts as documentation, and did not mark any contracts as false or as an implementation detail. Of the contracts the developer inserted, 25 were nullness checks, 3 were blank string checks, 2 were for non-empty collections, 1 was that a return value was not `NaN` (not-

[14]https://www.jetbrains.com/resharper/

a-number), 2 were indicator property checks, and 1 was that a line-number property was non-negative. Additionally, the developer incorrectly inserted, and did not remove, a method precondition stating that a string expression was constant. While the developer did insert one object invariant that a field was non-`null`, he later removed it. Using the Contract Inserter did not cause the developer to begin using object invariants in the project (cf. Table 2).

The developer chose to enforce properties he deemed interesting. For preconditions, this meant asking the question "How could other people mess up if they're calling this method?" For postconditions, the developer asked the question "How could my code or future versions of my code blow up?" The developer's approach is a mix of client-centric and code-centric strategies. For preconditions, his approach is client-centric. If he were instead following a code-centric approach, he might try determining the necessary preconditions for the method [10]. For postconditions, he focuses on the current and future versions of the method rather than determining which properties the clients of the method must be able to expect.

The developer added no contracts as documentation, as he disables contract checking for release builds and therefore the contracts do not affect speed. The codebase does not contain documentation.

The process of inserting contracts did not reveal any bugs, however it did reveal places clients of the library could misuse classes. The process of inserting contracts also revealed a place where a method name was misleading: the `AddField` method was adding a body. Additionally, the developer believed that bugs might be uncovered by running the other system/project tests with the new contracts inserted (he did not try this, though).

Based on the above usage, the developer made three suggestions for improving the tool. First, the developer suggested that the Contract Inserter be better integrated into the Visual Studio editor to enable navigation and to gain context regarding how methods are called. Second, the developer suggested to have contracts ranked by how relevant the contracts likely are. For example, inferred preconditions on fields not used by a method are likely just artifacts of the test suite. Third, the developer suggested support for a worklist view of classes, possibly ranked by which classes clients use most.

5.5 Threats to Validity

Our case studies were with two developers and two C# projects using Code Contracts, so the results might not generalize to other developers, projects, languages, or implementations of contracts.

While the two developers were not students and they added contracts to their own software, the studies were artificial in other senses. First, developers typically write contracts during development or debugging, not as a separate activity. Second, this was the developers' first use of the tool. With practice, the developers would likely become more proficient with the Contract Inserter.

At the time the studies were performed, the Contract Inserter had limitations that affected the contracts shown to the developers. The Contract Inserter redundantly displayed inferred object invariants as both object invariants and method invariants. Additionally, due to a quirk in how Daikon handles pairwise equality, the Contract Inserter did not suggest any implications aside from "guarded" expressions (e.g., `x != null implies x.f >= 0`). These limitations may have hindered the participants in finding and inserting valid, especially application-specific, contracts.

6. RECOMMENDATIONS

This section discusses our results and presents three action items for contract language and tool designers: (1) employ tooling to reduce annotation burden, (2) make suggestions an integral part of tooling, and (3) curate best practices by establishing design patterns.

While there is a risk that the subject projects are not representative, the information gathered from the project's developers can provide context for where the results are applicable. Additionally, since the subject projects are all open-source, we encourage the reader to download and explore the projects; all data and tools are available at `http://homes.cs.washington.edu/~tws/code-contracts/`.

Annotation Burden. The expected marginal benefit of writing a contract is low, particularly if the developer only uses Code Contracts for run-time checking. Using tools that make use of contracts to improve results, such as `cccheck` or Pex, improves the immediate benefit of contracts. However, as discussed in Section 4.3.2, some tools model contract semantics incompletely, making it less attractive to write expressive contracts.

As a wider array of tools take advantage of contracts, it will become easier for developers to derive immediate benefit from each additional contract. As a result, annotation burden may no longer be a deal-breaker. In the near-term, though, contract language and tool designers should aim to reduce annotation burden both when writing and when reading contracts. The Microsoft Code Contracts team has reduced the burden of writing contracts with contract abbreviator methods and code snippets. However, these features (especially code snippets) do not alleviate the burden caused by the large number of common-case contracts (e.g., non-`null` checks).

Based on our experience with pluggable types [15], we believe that defaults could reduce the burden of common-case properties. For example, tools could assume by default that each formal parameter is non-null, and this could be overridden by the programmer in the minority of cases where `null` is permitted [6, 36]. Eiffel's type system supports types that are non-null ("attached") by default [31].

Contract Suggestions. Contract suggestions, like those provided by Daikon and the Contract Inserter, serve to both reduce annotation burden and to educate users. Currently, the only Code Contracts tool that provides suggestions is `cccheck`, the static verifier. Since not all developers want to do static verification (especially when first using contracts), suggestions should be incorporated into other tools as well. The `cccheck` analysis could be retooled to offer suggestions during activities such as debugging and refactoring.

How to most effectively incorporate suggestions into an IDE is an open problem — intrusive or invalid suggestions can drive away users due to annoyance or distrust [24]. One possible solution is to conservatively exclude potentially-invalid information from the suggestions. For the developers using the Contract Inserter in the case studies, even simple suggestions such as using `Contract.ForAll`, object invariant, and interface contracts would help them to better achieve their goals in using contracts.

Curating Best Practices. While individual contract suggestions can be helpful, effective use of contracts is intrinsically tied to program design and structure. For example, in order to encode the type-state pattern [49], a developer introduces properties that indicate object state to clients and adds implications to the object invariant to describe the states. These steps require a combined understanding of code refactoring, object invariants, and logic (implication). Therefore, just as there has been extensive work on object-oriented design patterns, contract language designers should establish design pattern for development with contracts.

7. RELATED WORK

Specifications. Our goal in this paper was to help guide the design of contract frameworks. Each contract framework chooses a different trade-off between expressivity, verbosity, and tooling. The Java Modeling Language (JML) aims at expressing full behavioral specifications for both runtime checking and static verifica-

tion [4, 26]. Statically verifiable specifications make heavy use of JML's modeling functionality [21], however support for dynamically checking these specifications is nascent [8]. JML's lack of language integration has translated into a lack of modern tool support [4]. Microsoft's Code Contracts are expressible in all .NET framework languages as syntactically-valid statements without changing the existing languages. The trade-off is that their syntax is often verbose. A bytecode rewriter [19] enables dynamic checking, even for features such as prestate values. Microsoft also provides a static verifier `cccheck` [20]. The verifier performs partial verification, opting not to model properties such as the values of individual array elements. Eiffel was the first language to have support for contracts built into both its language and toolset [30, 39], but Eiffel is not a mainstream language [43]. Eiffel's contracts are designed for dynamic checking — the only static checkers are research tools (e.g., [47]). As a result, like Code Contracts, Eiffel lacks many verification-centric features such as ghost variables. Indeed, support for modeling in contract languages such as JML and Spec# [3] were primarily introduced to provide a basis for formal verification. Recent work by Polikarpova et al., though, demonstrates how using a model-based pattern for Eiffel and Code Contracts results in specifications that improve the efficacy of testing [39].

There are two related empirical studies of contract usage. Chalin studied 5 proprietary Eiffel projects, 79 open-source Eiffel projects, and the EiffelStudio libraries and samples [5]. Concurrently with our work, Estler et al. studied contract usage across revisions (i.e., over time) for 21 open-source projects using JML, Eiffel, and Code Contracts [18]. Unlike our work, these studies did not semantically categorize the contracts. Estler et al.'s results support our finding that nullness contracts dominate the contracts that developers write, however Chalin found that Eiffel programs contained a lower overall proportion of nullness contracts (35% for Eiffel vs. 66% for C# programs). (Earlier work estimated the incidence of non-null values in Java to be 50% at contract locations [7].) Both report more equitable distributions of preconditions and postconditions than what we report. However these results are not necessarily inconsistent with ours. Chalin compares *lines* of contracts as opposed to the number of clauses. In Estler et al.'s study, 5 of the 7 C# projects favor preconditions. Indeed, for each finding, there are exceptions that can be attributed to project, development language, and project lifecycle (e.g., periods of refactoring). These exceptions demonstrate a need for establishing context in empirical studies of contract usage.

Inference. Our tools and experiments use Daikon [16, 17] to infer possible contracts. We do not have space to review the a large body of work for specifying and inferring data, temporal, and other properties; a *partial* survey of inference techniques is provided in [40], though it oddly excludes tools such as Daikon. Tools ("front-ends") for generating Daikon compatible traces exist for other programming languages including Java [17], C/C++ [22, 41], and Eiffel [38]. An important part of a Daikon front end is a comparability analysis [23] that determines which variables should be compared (e.g., do not report `age < bank_account_number`). We implemented a simple, scalable, conservative static comparability analysis. Ajax [34] does a similar task more precisely but less scalably; like our analysis, it is inspired by static type inference of abstract types [1, 35]. Daikon's Java and C/C++ front-end perform comparability analysis dynamically. The Eiffel front-end does not perform comparability analysis. Unlike the Java and C/C++ front-ends, which require a pure method whitelist, the Eiffel front-end unsoundly assumes that all nullary methods are pure; users can supply a blacklist of impure methods.

Polikarpova et al. used the Eiffel front-end to study the differences between human-written and Daikon-inferred contracts. Their 25 subject classes (7 KLOC) are Eiffel base classes and student assignments, both of which are likely to be more heavily documented than typical code. They conclude that Daikon infers useful contracts not written by developers. They also conclude that Daikon misses many contracts written by developers [38], but this may stem from their Citadel tool giving low-quality input to Daikon. They measured the numeric recall and precision of inferred contracts with respect to what programmers wrote, but not with respect to what is true or useful. By contrast, we examined more code, qualitatively described the aggregate distribution of contract behaviors, and interviewed developers.

Inference Human Factors. Prior work suggests that inferred contracts are an important complement to human-originated contracts. Nimmer and Ernst found that including that inferred contracts (i.e., inferred by Daikon or Houdini) made users significantly more effective at writing verified specifications [33]. Polikarpova et al. [38] additionally reported that Daikon inferred meaningful contracts that humans had missed.

Despite the large body of literature on inferring contracts and specifications, there has been little direct study of the human factors affecting their use. Anecdotally, some users writing verified specifications in our previous work [42] reported that the included inferred contracts were distracting or even harmful. Directly studying contract comprehension is a difficult problem. Staats et al. suggest that users face difficulty determining whether or not postconditions inferred by Daikon are true [44]. However, they found no dominating factors (human or contract characteristics) associated with contract difficulty. A major threat to the validity of the study is that method preconditions were elided since preconditions require knowledge of the intended behavior. As such, it would be difficult for participants to infer the possible values of parameters and fields when a method was called, making determining the correctness of postconditions difficult.

8. CONCLUSION

In 90 open-source C# programs, the lion's share of contracts that developers write are basic checks enforcing common cases such as the presence of data. The tendency toward basic contracts can be explained by annotation burden, tooling, and training. In 2 case studies, contract suggestions inferred from run-time behavior aided developers in capturing more expressive contracts to support existing use cases. However, the suggestions did not lead the developers to new use cases for contracts. Therefore, contract language and tool developers should not only introduce tools to reduce annotation burden and suggest contracts, but also curate best practices by establishing design patterns. All data and tools from this paper are publicly available at `http://homes.cs.washington.edu/~tws/code-contracts/`.

9. ACKNOWLEDGMENTS

We thank Mike Barnett, Manuel Fähndrich, and Sai Zhang for their comments on a draft of this paper. We thank H.-Christian Estler, Carlo A. Furia, Martin Nordio, Marco Piccioni, and Bertrand Meyer for sharing a draft of their paper [18] with us. This material is based upon research sponsored by a National Science Foundation Graduate Research Fellowship under Grant No. DGE-0718124 and by DARPA under agreement number FA8750-12-2-0107. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

10. REFERENCES

[1] H. Baker. Unify and conquer (garbage, updating, aliasing, ...). In *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, pages 218–226, Nice, France, June 27–29, 1990.

[2] M. Barnett, M. Fahndrich, P. de Halleux, F. Logozzo, and N. Tillmann. Exploiting the synergy between automated-test-generation and programming-by-contract. In *ICSE'09, Proceedings of the 31st International Conference on Software Engineering*, May 2009.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69, Marseille, France, March 10–13, 2004.

[4] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[5] P. Chalin. Are practitioners writing contracts? In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 100–113. Springer-Verlag, 2006.

[6] P. Chalin and P. R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP 2007 — Object-Oriented Programming, 21st European Conference*, pages 227–247, Berlin, Germany, August 1–3, 2007.

[7] P. Chalin and F. Rioux. Non-null references by default in the Java Modeling Language. In *SAVCBS 2005: Specification and Verification of Component-Based Systems*, Lisbon, Portugal, September 5–6, 2005.

[8] P. Chalin and F. Rioux. JML runtime assertion checking: Improved error reporting and efficiency using strong validity. In *FM 2008: Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 246–261. 2008.

[9] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP'02, pages 231–255, London, UK, UK, 2002. Springer-Verlag.

[10] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *Verification, Model Checking, and Abstract Interpretation*, pages 128–148. Springer, 2013.

[11] P. M. Cousot, R. Cousot, F. Logozzo, and M. Barnett. An abstract interpretation framework for refactoring with application to extract methods with contracts. In *OOPSLA'12, Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 213–232, New York, NY, USA, 2012.

[12] C. Csallner and Y. Smaragdakis. Dynamically discovering likely interface invariants. In *ICSE'06, Proceedings of the 28th International Conference on Software Engineering*, pages 861–864, Shanghai, China, May 24–26, 2006. Emerging results track.

[13] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *ICSE'08, Proceedings of the 30th International Conference on Software Engineering*, pages 281–290, Leipzig, Germany, May 14–16, 2008.

[14] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 233–243, Portland, ME, USA, July 18–20, 2006.

[15] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. Schiller. Building and using pluggable type-checkers. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, Waikiki, Hawaii, USA, May 25–27, 2011.

[16] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[17] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Programming*, 69(1–3):35–45, December 2007.

[18] H.-C. Estler, C. A. Furia, M. Nordio, M. Piccioni, and B. Meyer. Contracts in practice. In *Proceedings of the 19th International Symposium on Formal Methods (FM)*, Lecture Notes in Computer Science. Springer, May 2014.

[19] M. Fahndrich, M. Barnett, D. Leijen, and F. Logozzo. Integrating a set of contract checking tools into Visual Studio. In *TOPI'2012, Proceedings of the 2nd Workshop on Developing Tools as Plug-ins*, pages 43–48. IEEE, 2012.

[20] 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*, FoVeOOS'10, pages 10–30, Berlin, Heidelberg, 2011. Springer-Verlag.

[21] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 2002, Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 17–19, 2002.

[22] P. J. Guo. A scalable mixed-level approach to dynamic analysis of C and C++ programs. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 5, 2006.

[23] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 255–265, Portland, ME, USA, July 18–20, 2006.

[24] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE'13, Proceedings of the 34th International Conference on Software Engineering*, pages 672–681, San Francisco, CA, USA, May 22–24, 2013.

[25] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *ICSM 2001, Proceedings of the International Conference on Software Maintenance*, pages 736–743, Florence, Italy, November 6–10, 2001.

[26] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, March 2005.

[27] A. Leitner, I. Ciupa, M. Oriol, B. Meyer, and A. Fiva. Contract Driven Development = Test Driven Development –

writing test cases. In *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 425–434, Dubrovnik, Croatia, September 5–7, 2007.

[28] F. Logozzo and T. Ball. Modular and verified automatic program repair. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'12, pages 133–146, New York, NY, USA, 2012. ACM.

[29] F. Logozzo, M. Barnett, M. A. Fähndrich, P. Cousot, and R. Cousot. A semantic integrated development environment. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 15–16, New York, NY, USA, 2012. ACM.

[30] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, October 1992.

[31] B. Meyer. Attached types and their application to three open problems of object-oriented programming. In A. P. Black, editor, *ECOOP 2005 Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 1–32. Springer Berlin Heidelberg, 2005.

[32] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22–24, 2002.

[33] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, pages 11–20, Charleston, SC, November 20–22, 2002.

[34] R. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 2001.

[35] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 19th International Conference on Software Engineering*, pages 338–348, Boston, MA, May 1997.

[36] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 22–24, 2008.

[37] 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 22nd ACM Symposium on Operating Systems Principles*, pages 87–102, Big Sky, MT, USA, October 12–14, 2009.

[38] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *ISSTA 2009, Proceedings of the 2009 International Symposium on Software Testing and Analysis*, pages 93–104, Chicago, IL, USA, July 21–23, 2009.

[39] N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer. What good are strong specifications? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE'13, pages 262–271, Piscataway, NJ, USA, 2013. IEEE Press.

[40] M. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, Sep 2012.

[41] R. A. Rudd. An improved scalable mixed-level approach to dynamic analysis of c and c++ programs. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, January 2010.

[42] T. W. Schiller and M. D. Ernst. Reducing the barriers to writing verified specifications. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2012)*, Tucson, AZ, USA, October 23-25, 2012.

[43] T. Software. TIOBE programming community index for august 2013, September 2013.

[44] M. Staats, S. Hong, M. Kim, and G. Rothermel. Understanding user understanding: Determining correctness of generated program invariants. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA'12, pages 188–198, New York, NY, USA, 2012.

[45] N. Tillmann and J. De Halleux. Pex: White box test generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[46] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 253–262, New York, NY, USA, 2005.

[47] J. Tschannen, C. Furia, M. Nordio, and B. Meyer. Automatic verification of advanced object-oriented features: The autoproof approach. In B. Meyer and M. Nordio, editors, *Tools for Practical Software Verification*, volume 7682 of *Lecture Notes in Computer Science*, pages 133–155. Springer Berlin Heidelberg, 2012.

[48] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.

[49] E. Zoppi, V. Braberman, G. de Caso, D. Garbervetsky, and S. Uchitel. Contractor.NET: Inferring typestate properties to enrich code contracts. In *TOPI '11, Proceedings of the 1st Workshop on Developing Tools as Plug-ins*, pages 44–47, New York, NY, USA, 2011.