©Copyright 2014

Todd Wademan Schiller

Reducing the Usability Barrier to Specification and Verification

Todd Wademan Schiller

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2014

Reading Committee:

Michael D. Ernst, Chair

Dan Grossman

James Fogarty

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Reducing the Usability Barrier to Specification and Verification

Todd Wademan Schiller

Chair of the Supervisory Committee:
Professor Michael D. Ernst
Computer Science and Engineering

A program specification is a contract between a client and a program, which describes how the program can or will behave given input from the client. In practice, formal (machine-verified) specifications take more effort and skill to write than informal natural language specifications. While some of the effort is essential to the task, much of the effort can be attributed design trade-offs and deficiencies in today's tools.

This dissertation makes three research contributions. First, it identifies tool transparency and interoperability as primary barriers limiting the use of formal specification tools; it presents empirical, observational, and controlled studies characterizing the effects of each, and provides actionable recommendations for tool designers. Second, it introduces novel verification interface features to reduce the skill barrier to verification. A study with (relatively) low-skilled freelancers found that developers writing verified specifications with an interface that incorporates the features were more productive than those using a traditional interface. Third, to address the problems of tool transparency and interoperability in the more general context of the development tools, this dissertation presents a pipeline-based approach to end-user information discovery and analysis in the IDE. In a study with undergraduate students, the students were able to quickly learn to use a tool based on the approach to answer questions that arise during development.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my adviser Michael Ernst for his constant support despite my inconstant progress. I thank Colin Gordon, Brian Burg, Sai Zhang, and the other members of the Programming Languages and Software Engineering group for their invaluable feedback. I also thank Mike Barnett, Francesco Logozzo, and Manuel Fahndrich and the other Microsoft Research members for their feedback on the Code Contract material. I thank all of the study participants who provided invaluable feedback and the anonymous reviewers for their thoughtful suggestions.

This material is based upon work supported by a National Science Foundation Graduate Research Fellowship under Grant No. DGE-0718124 and DARPA under Grant No. FA8750-11-2-0221.

DEDICATION

to my mother and father

Chapter 1

SPECIFICATION AND VERIFICATION

1.1 Introduction and Thesis

A program specification is a contract between an end-user (client) and a program describing how the program can or will behave given end-user (client) input. Specifications serve a critical role in the software development process enabling teams to answer the questions "did we try to build the right thing?" (validation) and "did we build what we tried to build?" (verification) [12].

Formal specifications, written in special mathematical or specificational languages, are more precise than natural-language specifications; machine-readable formal specifications enable automatic analysis of the specification's internal consistency and of the program's adherence to the specification.

Object-oriented and functional programming languages lend themselves to two primary kinds of formal specification: types and contracts. *Types* restrict the set of values a variable/expression can have and, thus, which operations may be performed on it. *Contracts* specify which conditions must hold when a client calls a method (preconditions) and, given that these conditions hold, specify which conditions must hold when the method returns (postconditions) [98]. Developers can use contracts to specify the full functional behavior of the software.

In practice, formal specifications are more difficult to write than informal natural language specifications. Some of the difficulty is essential (cf. [13]) — formal analyses demand complete and precise specifications. However, some is accidental, the consequence of design trade-offs or simply the result of poor tool interface design. Consider, for example, the ESC/Java2 [28] tool for verifying a Java Modeling Language [16] specification of a program. To perform verification, ESC/Java2 first computes a set of verification conditions for the program and specification, and then checks them with a theorem prover. This approach can automatically verify more kinds of properties than what previous approaches allowed. However, when the check fails, ESC/Java2 often cannot produce output that can be readily understood by a developer. Exacerbating the usability limitations of a tool's chosen approach, development resources for a tool are typically spent allowing more kinds of

properties to be verified as opposed to improving the tool interface.

Recent research confirms that poor tool interfaces are a significant barrier to developer use of static analysis tools for automatically detecting defects [74]; there is no corresponding research on tools that employ formal specifications to improve their output, though. Given that in 2002 software errors were already costing the U.S. economy an estimated $60 billion annually [131], the barriers to specification and verification have a real economic cost. Making specification and verification a viable option for a wider range of developers and projects would hold real economic benefit.

To effectively make use of specification and verification tools, developers need to be able to understand the tool output. Additionally, because each individual tool inevitably has shortcomings, developers need to be able to integrate multiple tools. In practice, however, specification and verification tools tend to be stand-alone black boxes. Such tools are both *opaque*, providing the developer little insight into their internal reasoning/state, and *monolithic*, providing limited means of integration with other tools. These two qualities translate into significant barriers to use. In particular, a developer must develop specialized knowledge of the inner-workings of a tool in order to be effective. Additionally, while various specification and (especially) verification tools are conceptually complementary, the benefit of joint use is generally not realized.

Thesis Recognizing that specification and verification tools unavoidably make design trade-offs, this dissertation argues the following:

> Making specification and verification tools transparent and interoperable — as opposed to opaque and monolithic — would facilitate their use.

Specification and verification tools that are transparent and interoperable admit user interfaces and APIs for providing input, consuming output, *and understanding the internal state of the tool*. In particular, interfaces must be provided for (1) discriminating what the tool knows and doesn't know, and (2) explaining the output of the tool. These characteristics are in stark contrast to contemporary tools for verification, such as ESC/Java2 [28] and cccheck [27], which due to their internal representations, effectively produce one of just three outputs for each clause of a specification: that the program satisfies the clause, that the program does not satisfy the clause, or that the state of the clause is unknown.

This dissertation addresses the barriers to specification and verification in the context of type- and contract-based specifications. However, it's not just specification and verification that can benefit from tool transparency and interoperability. Despite the success of Integrated Development Environments (IDEs), in practice, developers cannot readily integrate analyses and data sources within the IDE to answer questions that arise during development [83]. Inspired by the application of transparency and interoperability to specification and verification, this dissertation describes an approach to addressing the more general problem of meeting developer information needs.

Contributions This dissertation makes three primary research contributions:

- We identify transparency and interoperability as primary barriers affecting the usability of verified specifications; we present empirical, observational, and controlled studies characterizing the effects of each, and provide actionable recommendations for tool designers.
- We introduce novel verification interface features to address the skill barriers to verification. A study with (relatively) low-skilled freelancers found that developers writing verified specifications with an interface that incorporates the features were more productive than those using a traditional interface.
- Based on the insight obtained from the exploration of specification and verification, we present the design of a pipeline-based tool that enables a developer to more quickly discover and analyze information about his/her project in the IDE. A study with undergraduate students indicates that novices can use the tool to quickly answer questions that arise during development.

In support of these research contributions, this dissertation presents four engineering contributions, each of which is open-source:

- Celeriac, a tool for creating execution traces of .NET programs compatible with the Daikon invariant detection tool
- Contract Inserter, a Visual Studio add-in for discovering and inserting contract specifications into C# programs
- VeriWeb, a web-based environment for writing verified Java Modeling Language (JML) specifications of Java software
- Cupid, an information mash-up and visualization plug-in for the Eclipse IDE

This dissertation is organized into four main chapters. Chapters 2, 3, and 4 address the usability of specification and verification in the context of type-based specifications, partial contract-based specifications, and fully-functional contract-based specifications, respectively. Using these chapters as groundwork, Chapter 5 addresses the more general problem of interoperability and transparency as barriers to meeting developer information needs. Chapter 6 provides an overview of the themes in related work; each of the primary chapters discusses related work specific to the material in that chapter.

1.2 Formal Specification: From Types to Contracts

This dissertation explores three types of specifications. *Type qualifier* specifications extend a language's standard type system to further restrict the set of values a single variable/expression can have [107]. They are typically intended to be checked at compile-time. *Partial contract* specifications express and enforce aspects of the program's behavior, typically what behavior is not permitted. They are typically intended to be checked at run-time, and at compile-time where possible. *Full-functional contract* specifications express the input/output behavior of the program. Like types, they are typically intended to be checked statically at compile-time.

The remainder of this section illustrates how these approaches, taken together, subsume other approaches to specification, notably refinement types [56], effects [130], and type-state [129]. *Refinement types* allow developers to write predicates restricting the values of expressions and/or relating multiple expressions. *Effect systems* allow developers to specify which imperative effects a method may have (e.g., printing text, throwing an exception, etc.). *Type-state systems* allow the type of an object to vary over time, e.g., as the result of method calls. Showing the relation between these approaches to specification provides context as to where the results of this dissertation apply.

Each approach achieves a different trade-off between expressivity and annotation burden for the developer. In conjunction with tool support (e.g., dynamic enforcement and static verification), these factors affect the perceived and actual cost/benefit of formally specifying a piece of software. Consider, for example, the specification of a `withdraw` method for a `BankAccount` type which subtracts a *non-negative* amount from an *open* account with a *sufficient* balance.

A standard type system can specify that the amount is a number (as opposed to an arbitrary object):

```
void withdraw(double amount);
```

Type qualifiers can further specify that the amount is a non-negative number:

```
void withdraw(@NonNegative double amount);
```

With refinement types, the specification can additionally express the relationship between the withdrawal amount and the original balance:

```
void withdraw(@Refine({"v | v >= 0.0 && v <= getBalance()"}) double amount);
```

Type-state can succinctly specify the requirement that the account be open:

```
void withdraw(@Open Account this, double amount);
```

The type-state syntax appears very similar to the type qualifier syntax. Indeed, type-state can also be expressed using type qualifier (and refinement type) systems. The primary differences to the developer are (1) the quality of feedback that a tool purpose-built for verifying type-state can provide, and (2) the ability to express other properties on which the type-state might depend.

An effect system can specify that an `InsufficientFundsException` or `ClosedAccountException` might be thrown. However, effect systems typically cannot express the conditions under which the exceptions are thrown:

```
void withdraw(double amount) throws InsufficientFundsException,
                                    ClosedAccountException;
```

Finally, with contracts, in addition to all the above properties, the specification can express precisely how the method updates the balance:

```
@Requires({"amount >= 0.0" && amount <= this.getBalance()})}})
@Requires({"this.isOpen()"}})
@Ensures({"this.getBalance() = old(this.getBalance()) - amount"})
void withdraw(double amount);
```

This specification requires that the client calls the method only if the account is open and has a sufficient balance. Alternatively, the specification could specify the conditions under which an exception is thrown:

```
@Requires({"amount >= 0.0"})
@Ensures({"this.getBalance() = old(this.getBalance()) - amount"})
@ThrowEnsures({"ClosedAccountException", "!this.isOpen()"}
@ThrowEnsures({"InsufficientFundsException", "this.Balance() < amount"}
void withdraw(double amount);
```

Due to the generality of contracts, it should come as no surprise that some contract systems (e.g., Contracts for Java [30]) make no attempt to verify the contracts statically at compile-time. Instead, the contract systems dynamically enforce the specification at run-time. All the other systems, on the other hand, are designed to be statically verified.

The other salient difference between the approaches is the level of integration of the specification into the programming language itself. Language integration translates into better tool support (e.g., refactoring tools), lowering the cost of writing and maintaining the specification. Of the examples above (written in Java), only the standard type system and the effect system are fully-integrated into the language (and exceptions are the only effects supported by Java). Type qualifiers and type state make use of annotations, which are syntactically supported by programming languages, but the language itself is agnostic to the semantics of the specification. The refinement type and contract specifications utilize their own syntax for expressing properties, forfeiting tool support, but gaining the potential to support more concise syntax.

A consequence of the effects of expressivity and language integration is that specification and verification are tightly-coupled, and therefore should not be viewed in isolation. The next chapter begins the exploration of specification and verification with type qualifiers, lightweight specifications extending the guarantees provided by existing type systems.

Chapter 2

NOVICE USE OF TYPE QUALIFIERS

Static type systems are the most prevalent tool for specification and verification. For many developers, the first language they learned (e.g., Java or C++) was equipped with a static type system.

Type qualifiers extend the standard type system of a language, allowing developers to express and verify more precise properties. Because type qualifiers just extend a type system with which developers are already familiar, one might expect type qualifiers to be easy to learn and adopt. This chapter presents an observational study of undergraduate students using type qualifiers and an automated checker to eliminate `null`-dereference bugs their software. While type qualifiers and the checker were effective to eliminate errors, we observed that the students lacked a comprehensive mental model of the checking tool's operation and underutilized the nonbasic features of the toolset.

The following section provides relevant background on type qualifiers. Section 2.2 describes the Checker Framework, the tool used by the students in the study to verify their program and use of type qualifiers. Section 2.3 presents the observational study.

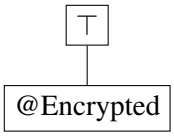
2.1 Type Qualifiers

A type describes the set of possible values an expression can take on at run-time. While types discriminate primitive values (e.g., integers from characters), in an object-oriented languages the bulk of the type system is dedicated not to directly restricting the data held by a expression, but rather what methods the expression provides. In practice this means supporting mechanisms such as interfaces, inheritance, mix-ins, etc. A type checker uses the types written by the developer (or inferred by a tool, possibly the checker itself) to check that the program is consistent with the annotations and that certain types of errors cannot occur at run-time. For example, the Java type checker prevents `MethodNotFoundException` from occurring at runtime by checking that each method call in the program refers to an existing method.

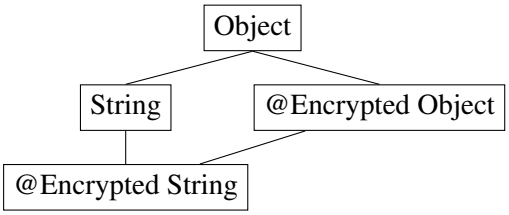
Type qualifiers further restrict (qualify) the set of values a variable/return value can have at run-time. A qualifier is an annotation that is placed on a use of a type, for example:

```
@Encrypted String data;
List<@NonNull String> strings;
myGraph = (@Immutable Graph) tmpGraph;
```

Conceptually, type qualifiers have their own inheritance hierarchy which can be distinct from the standard type hierarchy. For example, a security-conscious developer might use the following qualifier scheme to discriminate encrypted values from (possibly) unencrypted values:



where $\top$ (pronounced "top") denotes an unqualified type with no additional information (that the value may be encrypted or unencrypted, this case). This hierarchy would admit the following inheritance relation for type `Object` and its subtype `String`:



where an `@Encrypted String` can take the place of an `@Encrypted Object`, `String`, or `Object`. By annotating a method parameter with the `@Encrypted` parameter, e.g.:

```
public void sendOverNetwork(@Encrypted String message) { ... }
```

the developer gains a guarantee from the type checker that no client passes unencrypted data to the method:

```
String maybeUnencrypted = ...;
sendOverNetwork(maybeUnencrypted); // compile-time type error
```

2.2 The Checker Framework

The Checker Framework is a Java framework for defining type qualifiers and compiler plug-ins ("checkers") to detect bugs in software annotated with the custom qualifiers [107]. The type qualifiers are "pluggable" in the sense that a developer can decide if/when to use a particular type system on his/her project. Additionally, because the types gain their semantics from the checker, the developer can use different checkers or evolve a checker as his/her needs change.

Flow Sensitivity Type checkers for general purpose languages (e.g., Java and C#) are typically flow-insensitive — the type of an expression does not depend on prior expressions and statements. In the following example, the developer must still cast the variable `maybeString` despite its type being checked in the conditional:

```
Object maybeString = ...;
if (maybeString instanceof String){
   // type of maybeString is still Object, must cast the expression
   String knownString = (String) maybeString;
}
```

In contrast, the Checker Framework supports flow-sensitive type qualifier refinement.[1] This refinement is, in effect, a form of Abstract Interpretation [31] where relationships between variables, dataflow, and program annotations are used to determine the set of possible types for an expression. Refinement reduces the need for trusted (or dynamically checked) downcasts. (The cast in the example above is a dynamically-checked mechanism for downcasting the type of the expression).

Each checker defines its own rules for how an operation affects the known qualified type of an expression. For example, the Checker Framework's Nullness Checker uses a non-null check in a conditional to indicate that an expression is non-`null` in the body of the conditional:

```
if (maybeNull != null){
  maybeNull.hashCode(); // maybeNull is known to be non-null
}
```

[1] `http://types.cs.washington.edu/checker-framework/current/checkers-manual.html#type-refinement`

2.3 Observational Study

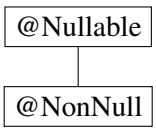
As part of a larger study on the usability of type qualifiers and the Checker Framework [42], we observed 28 undergraduate computer science students use the Checker Framework to verify the absence of null dereference bugs in their course projects. There were three primary findings:

- Students often wrote redundant qualifiers, especially early on.
- Students underutilized method annotations, likely because of insufficient instruction.
- Every student fixed at least one potential null pointer exception. The most common errors that students fixed were dereferencing the return value of `BufferedReader.readline()` without a guard, and dereferencing the formal parameter to `equals` without first checking for null. Students reported that they had not thought of these cases.

2.3.1 Nullness Type System

The Nullness Checker[2] verifies the absence of null pointer exceptions (NPEs) in a program by checking that (1) expressions with nullable type are never dereferenced and (2) variables with non-null type are never assigned a null value.

The core of the nullness type system is simple, with expressions either being `@NonNull` or `@Nullable` (may be `null`):



There is no need for a `@Null` qualifier since the purpose of the system is to prevent null dereferences and therefore the checker issues a warning for any use of a reference that that *might* be `null` at run-time. By default, the checker assumes that method parameters are `@NonNull`.

[2] `http://types.cs.washington.edu/checker-framework/current/checkers-manual.html#nullness-checker`

The nullness type system includes additional qualifiers to express common patterns. For example, the `@LazyNonNull` qualifier indicates that a field (or variable) is initialized as `null`, but becomes permanently non-`null`with the first assignment. The `@KeyFor` qualifier indicates that an expression is a valid key for a given map (and therefore `Map.get(...)` cannot return `null` due to a missing key).

Postcondition Annotations Nullness is a deceptively complex property. Because nullness properties often depend on the state of an object, the Nullness checker supports two method postcondition annotations: `@EnsuresNonNull` and `@EnsuresNonNullIf`. The former indicates that the method has initialized a particular field; the latter relates the return value of a method returning a boolean value (an indicator method) and another method. For example, consider a queue class which has two states "empty" and "non-empty" which are distinguished by the method `isEmpty()`:

```
class Queue {
    @EnsuresNonNullIf(expression="peek()", result=false)
    public boolean isEmpty();

    public @Nullable Object peek();
    ...
}
```

Because of flow-sensitivity, the developer can dereference the result of the `peek` method as long as they first check the state of the queue with the `isEmpty` method.

2.3.2 Methodology

To evaluate the Nullness Checker's ease of use for total novices, we observed 28 first-year computer science students enrolled in CSE 331 at the University of Washington. CSE 331, Software Design and Implementation, provides an introduction to design patterns, specification, and testing. The only prerequisite for the course is the introductory programming sequence, which briefly introduces the object-oriented features of Java (e.g., dynamic dispatch).

Over the course of 7 weekly problem sets, the students each developed a route-finding program over real road data. For each problem set, students submitted their solutions, received feedback, and then re-submitted their solutions. Then, as an additional problem set, the students ran the Nullness Checker to eliminate the possibility of null pointer exceptions in their code. Students received a one-hour demo, a one-hour class lecture on pluggable type systems, the Checker Framework manual, a problem set write-up, a build file, and two small documented example files. Students had to annotate both code they had written and some staff-provided code, but no testing code; additionally, calls to the Swing library and a staff-provided library were unchecked.

2.3.3 Results

Ease of Use Students reported spending only a mean of 5.4 hours reading, learning to run the tool, annotating, and bug fixing.[3] To verify their programs, each student wrote a mean of 57 annotations across 9,000 lines of code (5kLOC non-comment, non-blank, lines of code). Including all time spent, students checked and fixed their code (mean size 9 kLOC, 5 kLOC NCNB) at a mean of 1743 LOC per hour (913 LOC NCNB; σ=563 LOC, 321 LOC NCNB).

Students had no significant trouble using the basic `@NonNull` and `@Nullable` type qualifiers, which were the only ones that had been demoed. Students' usage improved with time, even over the 5 hours they spent. When annotating the first assignments, students sometimes wrote redundant type qualifiers that repeated a default or would have been inferred. Redundant type qualifiers were less common when annotating the later assignments.

Students under-used the other annotations supported by the Nullness Checker. Only 9 of the 28 students (32%) used the `@LazyNonNull` qualifier despite a member variable being lazily initialized in the staff-provided code for the first assignment. Similarly, students should have used `@KeyFor` more frequently to indicate membership in a map's key set. Finally, only 8 of the 28 students (29%) used method annotations about purity and pre- and postconditions — though they were appropriate for both staff-provided code and staff-specified student-written code. We conclude that the level of instruction that we gave to these novices was inadequate. It discussed theory but failed to focus on the pragmatic issues that they had most trouble with. And, it did not even mention the annotations for

[3]This data is from the 11 students who categorized time spent. Min: 3 hours; median: 5; max: 12. Students who worked from home had to install the Checker Framework, but the staff had pre-installed it on the lab workstations.

lazy initialization, methods, or maps, though these were covered in the Checker Framework manual.

Effectiveness All 28 students found and fixed at least one null pointer error in their code.[4] On the original submissions, 5 of the 28 students (18%) had NPEs that were detected by the staff's test suite, which consisted of system tests and some unit tests, for classes the staff had specified. After receiving feedback and resubmitting their work for automated testing, 3 students (11%) still had NPEs. After running the Nullness Checker, the staff tests detected no NPEs.

The two most common errors that students fixed were: using the formal parameter to `equals` and the return value from `BufferedReader.readLine()` without first testing for `null`. (The `readLine()` method returns null when the end of the stream has been reached). Students reported that they had not thought of these cases.

On average, each student was unable to verify the absence of a null-pointer exception at a dereference fewer than six times. These occurrences were indicated by suppressed warnings, improper workarounds, and (possibly inadvertent) use of a bug in the Nullness Checker. Students used `@SuppressWarnings` to suppress false positives warnings a mean 2.6 times. This modest number is encouraging, especially since one use per student was needed in staff-provided code. Students worked around the type checker in less than 2 methods on average (and in no more than 4 methods, except for one outlier student). These workarounds converted a NPE into a different erroneous behavior, which does not address the root cause of the problem. Examples are null-guarding a block of code and performing no action if the guard failed, or throwing a different runtime exception, such as `IllegalArgumentException`, when a null pointer was encountered. Most students made the same few workarounds, e.g., in the method containing calls to `BufferedReader.readline()`. Another error in the student annotations was that 12 students took advantage of an unsoundness in the checking of the `@KeyFor` annotation (which is now fixed) by declaring variables as keys for a map without establishing the relationship. The 20 students that used the `@KeyFor` annotation each introduced this error in less than 2 methods on average; one student made the error in 9 methods.

We asked students what would be the best use of time, if they wished to improve the quality of their code. Less than half (13 out of 28) unequivocally stated that they would use another tactic such

[4]Two students incorrectly reported that type checking had revealed no errors, but our manual examination of their code changes indicated that they had fixed a null pointer error.

as reasoning about their code, writing assertions, writing tests, or using a different pluggable type system to prevent representation exposure. We were surprised by these results. In our view, using a pluggable type checker from the beginning of a project is well worth the effort, but other activities are usually more effective than annotating all of a legacy codebase. The students concurred regarding the relative effort: the most common comment, mentioned by 9 students (32%), was that the checker would be more useful if used throughout the development process, rather than the end.

2.3.4 Discussion

The process of writing a verified specification exposed the incompleteness of the students' mental models (e.g., with respect to the behavior of the `equals` method). While all students eliminated defects, their effectiveness in eliminating defects was undermined both by their mental model of the tool's checking of annotations and the annotations themselves. The former is evidenced by the students' tendency to write unnecessary annotations. This tendency cannot be attributed to preference alone — unnecessary annotations were less common in code annotated toward the end of the assignment. The latter is evidenced by the students' taking advantage of the unsoundness of the checking tool. This pattern would not have been expected if the students were forming complete mental models of the relation between their specifications and respective projects. Rather, the pattern suggests that students instead were iteratively attempting to satisfy the checking tool.

Formal specification and verification make software specifications more transparent by eliminating the ambiguity present in natural language specifications. However, the specification and verification tools themselves can introduce complexity. Therefore, tools ought to provide transparency to help developers maintain their mental model of the software and specification. For example, the Checker Framework might be improved to output the calculated types of expressions in a program. Displaying calculated types would eliminate the need for developers to manually record their mental model with unnecessary annotations.

Chapter 3

CONTRACTS AS PARTIAL SPECIFICATIONS

3.1 Introduction

Contract specifications describe what must be true when a 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.

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 [135] and mocking frameworks has pushed testing toward specification; conversely, contracts provide a powerful semantic basis for test creation [23, 3, 88, 112]. Contracts can additionally augment or even automate refactoring [76, 33], debugging, program repair [38, 110, 92], and verification [85, 93].

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 chapter is to provide information about how developers currently use contract-style specifications, with the purpose of guiding the design of contract languages and tools. This chapter focuses on Microsoft Code Contracts (hereafter just Code Contracts) because they are officially supported by Microsoft and are designed for use with the popular C# language. The next chapter, Chapter 4, explores the use of the Java Modelling Language (JML), which is better suited for static verification, but lacks the language integration and user base of Code Contracts.

Contributions This chapter 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 integral 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 chapter presents two engineering contributions: (1) Celeriac, an open-source tool for producing Daikon-compatible traces of .NET binary executions from which to infer contracts [48], 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 chapter is organized as follows. Section 3.2 introduces four subject projects that will be referred to throughout the chapter and describes each project's use of Microsoft Code Contracts as reported by their developers. Section 3.3 analyzes the developer-written Code Contracts in 90 programs, with a focus on the four subject programs. Section 3.4 characterizes the contracts that Celeriac and Daikon can infer for the subject programs, contrasting these to the developer-written contracts. Section 3.5 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 3.6 discusses implications with respect to the design of contract languages and tools. Section 3.7 presents related work.

3.2 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.[1] 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[2] 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 3.1 provides an overview of 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

[1] http://visualstudiogallery.msdn.microsoft.com/1ec7db13-3363-46c9-851f-1ce455f66970

[2] https://www.ohloh.net/

Table 3.1: 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[a]
Mishra Reader	19K	> 27K	1-5	Fall 2011	Debugging concurrent code		✓	JetBrains R#[b]
Sando	24K	> 500	3-6	Winter 2012	Early runtime error detection		✓	NUnit[c]
Quick Graph	32K	> 75K	1	2008	Documentation & testing		✓	Pex [133], MSTest

[a] https://stylecop.codeplex.com

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

[c] http://www.nunit.org

Reader and Sando as part of developer studies in Section 3.5). 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[3] 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[4] is a Google Reader (RSS) 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[5] is a Lucene-based[6] 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

---

[3] https://labs.codeplex.com

[4] https://mishrareader.codeplex.com

[5] https://sando.codeplex.com

[6] https://lucene.apache.org/

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.

Quick Graph Quick Graph[7] 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 [133]. 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.

3.3 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 3.3.1. *What properties do developers use Code Contracts to enforce (**semantics**)?*

Research Question 3.3.2. *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 [112]). Approximately three-fourths of the contracts just check for

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

Table 3.2: 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 3.4 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%

the presence of data; an additional 3% of contracts (18% of all postconditions) repeat field assignment and return statements from the code.

3.3.1 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[8] program to categorize each contract into the finer categories of Table 3.2. We ran the program on 90 C# projects (3.5M source lines of code) that use Code Contracts. The program 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.

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.

---

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

3.3.2 Results

Table 3.2 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). The Code Contracts framework implements the semantics for these methods and features as part of 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 3.2) 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 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 3.5.4.

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 heavyweight 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 3.2.

3.3.3 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` [42].

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 benefit since they guarantee that values exist for types, and therefore support the interface guarantees provided by the type system. However, since non-`null` is the common case [19], the annotation burden is difficult to justify.

3.4 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 [48]; 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 3.4.1. *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 3.4.2. *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 3.3 (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 [112]. The results support these expectations.

3.4.1 Methodology

As a proxy for determining which contracts could be written for the subject programs, we used the Daikon invariant detector [48] 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 3.4.2), and then ran the programs using tests or example inputs. For .NET programs, each sub-component/library of the program is compiled to a library (DLL) or executable (EXE) assembly file. 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 3.5. We generated a trace for the Labs Framework by running the labs for the Rxx project[9], 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 3.4.3 addresses the shortcomings of Daikon and Celeriac as they relate to this study.

3.4.2 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[10] 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

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

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

.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.

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 [34]. 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.

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 [48]). For Celeriac, we opted to implement a conservative static comparability analysis using the CCI Code Model and AST API.[11] The primary motivation for using a static analysis was that some of the test suites for our subject programs had low branch coverage.

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

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.

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. 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 [137]. 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.

3.4.3 Results

Table 3.3 shows the inferred contracts across the projects using the categories from Section 3.3.1. Table 3.3 differs from Table 3.2 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 3.2. Therefore, the categorization would tend to make the inferred contracts appear less application-specific than the developer-written contracts.

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 3.3, 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 3.4.2).

The "Constant" row shows that many contracts specify a constant value. These contracts (and

Table 3.3: 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 3.5. The rows and columns are the same as in Table 3.2 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					Mishra Reader					Sando					Quick Graph					90 projects Written	
	Labs.dll (7.4K)					ViewModels.dll (4.4K)					Indexer.dll (2.3K)					QuickGraph.dll (21K)						
	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%*

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 developer-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).

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 [112] 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 3.3) 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.

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 [103] 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. [111] 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 [35]) 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.

3.5 Developer Case Study

Sections 3.3 and 3.4 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 3.5.1. *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 3.5.1), 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.

3.5.1 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 3.1).

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.

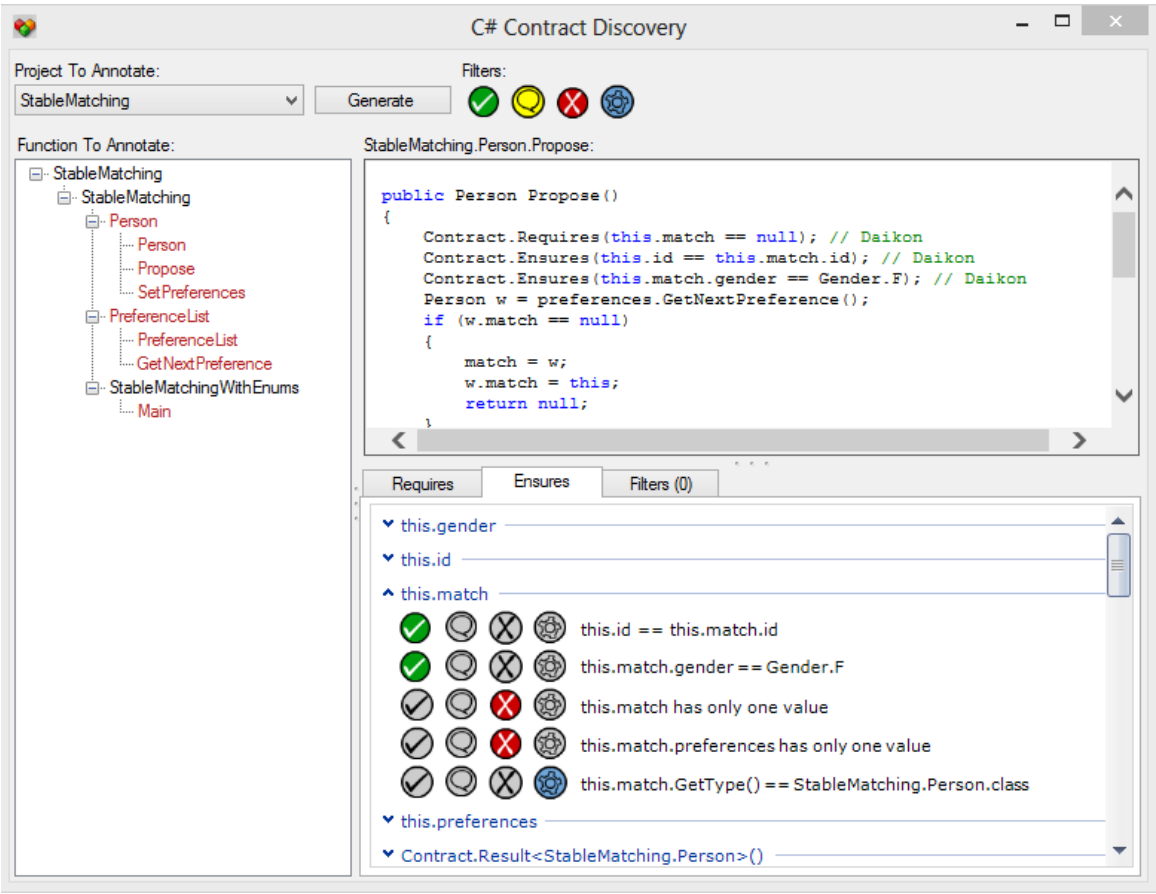


Figure 3.1: The Contract Inserter Add-In (Section 3.5.1). Left pane: the class and method tree. Top right pane: source code with preview of inserted contracts and documentation. Bottom right pane: inferred contracts, grouped by variable. The developer can insert or ignore each contract by clicking an 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).

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., specifying the greatest common denominator of two expressions), 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`.

3.5.2 Case Study Methodology

For each project, we selected an assembly (each sub-project of a .NET project corresponds to a separate assembly) 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 3.4.1). 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.

3.5.3 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.[12] 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 3.4, 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.

The decision to only specify module boundaries with contracts is not uncommon (e.g., in the language Racket, contracts by default are only enforced on module boundaries [114]). However, this decision appears 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).

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

3.5.4 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-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 3.2).

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 [32]. 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.

3.5.5 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.

3.6 Recommendations

This section discusses the 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, the reader can 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 3.4.3, 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 [42], 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 [19, 107]. Eiffel's type system supports types that are non-null ("attached") by default [99].

Contract Suggestions Contract suggestions, like those provided by Daikon and the Contract Inserter, serve to both reduce annotation burden and to educate developers. 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 developers due to annoyance or distrust [75]. 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 [147], 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.

3.7 Related Work

Specifications Each contract framework chooses a different trade-off between expressivity, verbosity, and tooling. The Java Modeling Language (JML), explored in the next chapter, aims at expressing full behavioral specifications for both runtime checking and static verification [16, 85]. Statically verifiable specifications make heavy use of JML's modeling functionality [55], however support for dynamically checking these specifications is nascent [21]. JML's lack of language integration has translated into a lack of modern tool support [16]. 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 [51] enables dynamic checking, even for features such as prestate values. Microsoft also provides a static verifier `cccheck` [52]. 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 [98, 112], but Eiffel is not a mainstream language [125]. Eiffel's contracts are designed for dynamic checking — the only static checkers are research tools (e.g., [136]). 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# [5] 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 [112].

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 [18]. 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 [50]. 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 [20].) 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 [47, 48] 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 [115], though it oddly excludes tools such as Daikon. Tools ("front-ends") for generating Daikon compatible traces exist for other programming languages including Java [48], C/C++ [65, 116], and Eiffel [111]. An important part of a Daikon front end is a comparability analysis [64] 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 [105] does a similar task more precisely but less scalably; like our analysis, it is inspired by static type inference of abstract types [2, 106]. 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; developers 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 [111], 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 developers significantly more effective at writing verified specifications [104]. Polikarpova et al. [111] 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 developers writing verified specifications in our previous work [121] reported that the included inferred contracts were distracting or even harmful. Directly studying contract comprehension is a difficult problem. Staats et al. suggest that developers face difficulty determining whether or not postconditions inferred by Daikon are true [127]. 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.

Chapter 4

VERIFIED SPECIFICATIONS

4.1 Introduction

Chapter 3 investigated how developers use contracts as partial specifications, typically to catch degenerate behavior (e.g., null values). However, contracts can also be used to specify and verify the functional behavior of a program — how the program *should* behave, as opposed to how it *should not* behave.

Writing a verifiable specification using contracts is a Herculean task. Much of the challenge derives from the fact that even when programs are well modularized, changes to one method contract may require changes to others as a result of interdependencies between program elements — a method's contract may depend on the contracts for other methods it calls. Attempting to decompose the task into subtasks results in information (context) loss, the severity of which depends on the quality of the program's design and documentation; incomplete information leads to confusion, mistakes, and wasted or duplicated effort. For writing and verifying formal specifications, the difficulty is exacerbated by the complexity and inaccessibility of current verification interfaces.

This chapter investigates the bounds of usability in writing verified specifications — exploring what features would be needed to crowdsource the writing of verified specifications to relatively low-skilled workers, as opposed to depending on expert developers. Given the high cost of development labor (the average developer in the United States earns $90,170 per year [138]), the results provide insight into the factors that will make verified specifications economically attractive for general-purpose use. To the extent that tools can also be designed to decompose the problem into sub-problems, the tools would further improve the economics of verification by enabling parallel and distributed workflows (e.g., crowdsourcing to workers in a global marketplace).

VeriWeb Verification Interface This chapter presents VeriWeb, a tool for creating Java Modeling Language (JML) [16, 84] specifications for existing programs that are verified with ESC/Java2 [28].

Figure 4.1: An example partial Java Modeling Language (JML) contract for the enqueue method of a queue. “Requires” clauses state properties that must hold when the method is called. “Ensures” clauses state properties that must hold when the method exits normally. “Exsures” clauses state properties that must hold when the method throws the declared exception. “Invariants” state properties that must hold whenever an object is visible, e.g., after a constructor or a public method call. A tool such as ESC/Java2 [28] can verify that the program meets the specification, and that no unexpected exceptions (e.g., NullPointerException) will be thrown.

```
class Queue {
  int /*@spec_public*/ currentSize;
  /*@invariant currentSize >= 0; */
  ...

  /*@ requires x != null;
    @ ensures currentSize == \old(currentSize) + 1;
    @ exsures (RuntimeException) ... */
  public void enqueue(Object x)
    throws RuntimeException {
    ...
  }
}
```

VeriWeb works by leveraging the “crowd,” a (potentially distributed) set of workers solving problems via web client.

VeriWeb decomposes the larger task of writing a verifiable program specification into smaller subtasks: creating one method’s preconditions and postconditions. To compensate for the effects of decomposition and to lower the tool’s skill requirement, VeriWeb includes several novel interface features: drag and drop contract construction, concrete counterexamples, contract inlining, and context clues. VeriWeb additionally includes contract suggestions inferred from dynamic traces. VeriWeb can be viewed as an IDE for verification, where the combined effect of the features is greater than the sum of the individual parts.

By enabling the use of less-skilled labor for verifying an existing program, VeriWeb admits the following workflow for skilled feature developers:

1. The skilled developer writes a program or feature.

2. The skilled developer (optionally) writes a partial JML specification of the program or feature.

3. The skilled developer submits the program and partial specification to VeriWeb, which utilizes the "crowd" to complete a verifiable JML specification.

Extended Static Checking Internally, VeriWeb uses ESC/Java2 [28] to perform Extended Static Checking [40, 87, 41, 55, 24, 5] of the program against the generated specification. Extended Static Checking is a verification approach that can be viewed as a front-end or user interface to an automated theorem prover [101, 39, 37]. The developer writes a specification consisting of method preconditions, method postconditions, and object invariants. (Figure 4.1 shows a partial example of a method contract, expressed in the Java Modeling Language (JML) [16, 84].) The extended static checker converts both the specification and the program code into logical formulas, passes the combined formulas to the theorem prover, and reports to the developer whether or not the program satisfies the specification. If not (that is, the verification attempt failed), the developer revises the specifications or the code using the feedback from the checker, and then tries again.

One example verification task is to prove that a program will never throw an unexpected null pointer exception, regardless of input. If a public method unconditionally dereferences a parameter, then the property is true only if the method is never called with `null` as an argument. The developer can express this requirement by writing a method precondition that the parameter cannot be null. ESC/Java2 would then be able to prove that the method cannot throw an unexpected null pointer exception. However, the addition of the precondition introduces the requirement that the corresponding argument at each call site is non-`null`. In the end, ESC/Java2 verifies that all the developer-written contracts are mutually consistent, that the developer-written contracts are consistent with the code, and that there are no null pointer exceptions. Note that, while this is partial verification, and not verification of full functional correctness, any requisite functional properties (e.g., data value properties) are verified *en passant*.

VeriWeb Use Case VeriWeb is targeted at the verification of domain-independent and expert-specified properties in existing client code. VeriWeb is not targeted at the discovery of new application-specific properties, verification of full functional correctness, the verification of library code, or the development of new programs.

VeriWeb focuses on client code rather than library code. We have observed that library code use is cross-cutting, whereas client code is more lightly coupled. Furthermore, the specifications of library code are complicated, in order to support general use — consider the C++ Standard Template Library. By contrast, the specifications for client code are relatively simpler, and the specifications required to show the absence of unexpected exceptions in client code are even simpler. Therefore, there is ample opportunity to reduce the cost of partial formal verification by focusing on client code.

Since VeriWeb operates by having developers solve method subtasks, the tool is unlikely to uncover complex object invariants and global application properties. VeriWeb is better-suited to verifying language properties (e.g., that the program will never deference a null pointer) and localized application properties. This use case is aligned with the underlying tool ESC/Java2 [28], as well as similar techniques such as Microsoft's Code Contracts [27].

Evaluation To measure the time and monetary cost of verification, we performed a comparative study in which workers from Exhedra Solutions's vWorker [141] labor marketplace performed verification with ESC/Java2, either through its Eclipse interface or through VeriWeb. To understand the potential for using ad-hoc crowdsourced labor for verification, we recruited workers from Amazon's Mechanical Turk to use VeriWeb. To understand the overhead incurred when performing verification collaboratively with VeriWeb, we observed undergraduate students using the tool to collaboratively verify a small program.

Contributions This chapter makes 3 primary contributions:

1. We present novel verification interface features to address the challenges of decomposition and a distributed workforce. VeriWeb, a browser-based tool for program verification, incorporates these features.

2. We quantitatively characterize the time and monetary costs of verification.

3. We identify and describe challenges and threats to validity that are unique to studying verification in a global and collaborative setting.

The chapter is organized as follows. Section 4.2 describes VeriWeb's design principles and implementation. Section 4.3 poses three primary research questions. Section 4.4 quantifies the monetary cost of verification when contracting semi-skilled labor on an hourly basis. Section 4.5 explores the use of ad-hoc labor from Amazon's Mechanical Turk for performing verification. Section 4.6 characterizes the overhead incurred when performing collaborative verification with VeriWeb. Section 4.7 discusses the results. Finally, Section 4.8 presents related work.

4.2 VeriWeb

We created VeriWeb, a tool for verifying Java programs. Internally, VeriWeb uses the ESC/Java2 verification tool.[1] VeriWeb decomposes the task of verification into subproblems that users solve in a web interface (Figure 4.2). A live demo is available at the VeriWeb project page `http://www.cs.washington.edu/homes/tws/veriweb/`. We designed VeriWeb around two major principles: active guidance (Section 4.2.1) and explanations in context (Section 4.2.2).

4.2.1 Active Guidance

Active guidance — encouraging workers to reason in a certain way, or restricting their set of actions — is aimed at aiding reasoning and preventing time-wasting mistakes. VeriWeb guides workers *between* subproblems by choosing the next subproblem for workers (Section 4.2.1) and *within* a subproblem by making suggestions (Section 4.2.1) and preventing syntax errors (Section 4.2.1).

Guided Decomposition

VeriWeb guides the worker through the verification task by asking the worker to solve 4 types of subproblems:

1. Select method preconditions from a list

[1] Other verification tools exist, such as Microsoft's Code Contracts [27]. We originally wanted to use Code Contracts for this research, but ESC/Java2 can statically reason about array properties that Code Contracts cannot. ESC/Java2 has its own disadvantages: it only fully supports the Java 1.4 specification (Java 5 was released in 2004), and it can be unsound.

2. Write method preconditions

3. Write method postconditions (that are true when the method exits normally)

4. Write method exceptional postconditions

VeriWeb offers the selection and writing of preconditions as separate problems not just because this focuses the worker on the most relevant task at any moment, but also because they require different modes of reasoning. Selecting preconditions from a list of possibilities suggested by VeriWeb requires forward reasoning, to determine whether an error will occur. Writing preconditions is a more difficult task that requires backward reasoning from an error to determine the weakest precondition that will prevent it.

Active Subproblems A subproblem is *active* if it requires work from a worker. There are three reasons that a subproblem may be active:

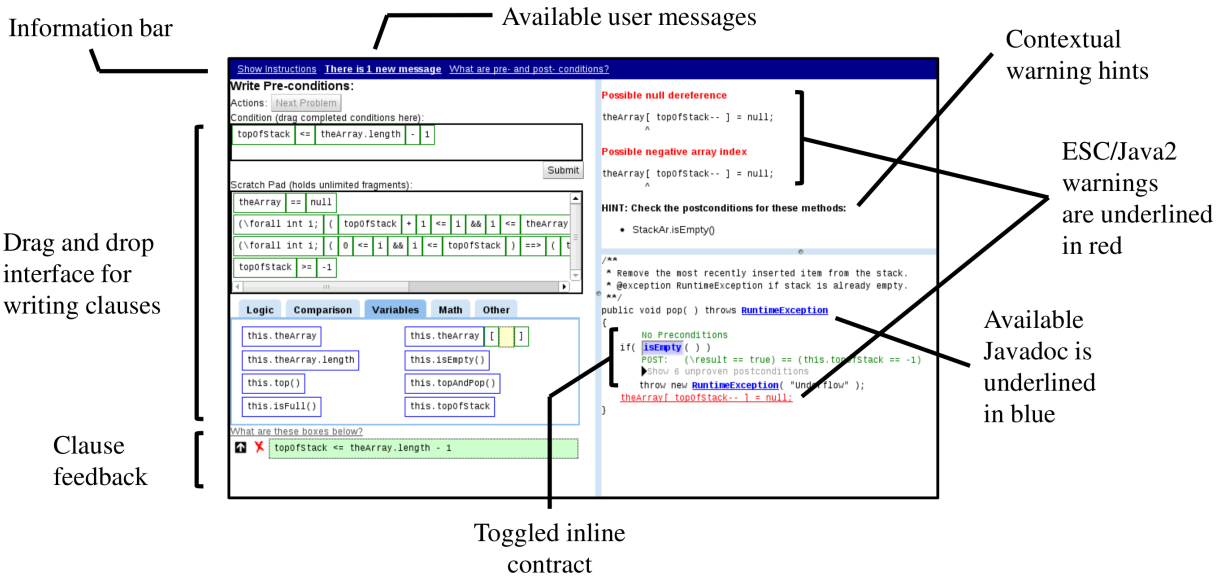
1. The subproblem is the cause of an ESC/Java2 warning. If there is a warning within a method body, the precondition set is considered to be the cause.

2. A worker has identified the subproblem as being the cause of an error elsewhere. For example, an unverifiable postcondition might be caused by a too-weak precondition or by a too-weak postcondition on a callee method. Likewise for ESC/Java2 warnings occurring within the method body.

3. Every postcondition problem is initially marked as active. Workers can often quickly write a few obvious postconditions (e.g., about the return value for a boolean method), and we found that doing so can substantially speed the verification process.

As long as any subproblems are active, the worker is presented with an active subproblem to solve. When no subproblem is active, the program has been verified.

Figure 4.2: VeriWeb client interface showing a "write preconditions" subproblem. Top: information bar linking to instructions, FAQ, and messages about the problem (Section 4.2.2). Left: the drag and drop interface for writing conditions (Section 4.2.1). Right Top: ESC/Java2 warning locations are underlined in red in the source code view, and the warnings are shown at the top right. Lower right: source code view. Javadoc is available for code highlighted in blue. To view the documentation and warnings, the worker simply hovers their mouse over the underlined code. The worker has toggled the inline specifications (Section 4.2.2) for method isEmpty.



Subproblem dependencies VeriWeb computes dependencies among subproblems and stores these as a directed dependence graph that exhibits the same high-level structure as the call graph. Subproblem P_1 depends on subproblem P_2 if a change in the solution to P_2 may invalidate the solution to P_1. This property is independent of whether subproblem P_1 has actually been solved yet.

Whenever a worker changes the solution to a subproblem, VeriWeb activates some or all of the subproblems that depend on it (Figure 4.3):

- If a precondition is strengthened, VeriWeb activates the precondition subproblems for the method's callers.
- If a precondition is weakened, VeriWeb activates the postcondition subproblems for the method.
- If a postcondition is weakened, VeriWeb naively activates all problems for the method's callers.

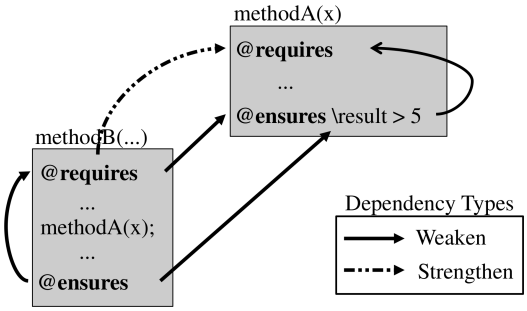


Figure 4.3: Intra- and intermethod depends-on graph for a method B that calls a method A. When a contract clause is weakened, subproblems with "weaken" edges leading to the contract are activated. For example, if the ensures clause for method A was weakened to `\result` > 4, both the precondition and postcondition problems for method B would be activated. When a contract clause is strengthened, subproblems with a "strengthen" edge leading to the contract are activated. For example, if a new precondition $x < 3$ is added to method A, the precondition subproblems for method B would be activated to ensure the call to method A uses a valid x value.

In each of these situations, VeriWeb re-checks each subproblem and only activates subproblems for which an error occurs. If no error occurs, then no additional work is required and VeriWeb does not require the worker to re-visit the subproblem.

Note that these basic rules capture more complex relationships between properties via composition. For example, according to the second activation rule, if a method's preconditions are weakened, VeriWeb will activate the postcondition problem for that method. If the change causes the postcondition set to be weakened, then VeriWeb will activate all the problems for the method's callers according to the third activation rule.

The leaves of the active subproblem graph — those nodes that are active and have no active children — are the set of subproblems that can be productively assigned to workers. In fact, VeriWeb can assign these subproblems to be solved in parallel by different workers. When assigning a subproblem, VeriWeb gives preference to workers who have already worked on a subproblem, and to workers who have marked that a specification was incorrect.

Collaborative Use To support multiple workers simultaneously, VeriWeb currently just naively performs synchronization on an entire project. Each worker keeps a local view of the master project specification; the local view is updated whenever the worker requests a new subproblem.

When a worker submits a solution to a subproblem P_1, VeriWeb checks whether the solution invalidates the assumptions of any other subproblem P_2 currently being worked on. VeriWeb lets the other worker continue working on P_2. However, when the worker submits P_2, VeriWeb records the clauses in the solution for future use, but does not modify the master specification.

Recursion The current implementation of VeriWeb only supports single-method recursion. It is straightforward to extend VeriWeb to mutually recursive methods: make a tree of the (possibly degenerate) strongly connected components (SCCs), and arbitrarily choose one method from each SCC that is a leaf. Once the worker solves this subproblem, the SCC is either broken or is smaller.

Object Invariants Object invariants require non-modular reasoning about all public methods in a class, which is at odds with VeriWeb's decomposition into subproblems. VeriWeb does not display object invariants nor give the worker the opportunity to write object invariants directly.

VeriWeb does, however, compute and utilize object invariants by lifting conditions that appear as preconditions and postconditions for all the public methods for the type. The tool expedites object invariant discovery in the following ways:

- Any precondition that refers to object state is automatically checked as a (potential) postcondition for the method. (Section 4.2.1 discusses how VeriWeb suggests and checks potential contracts.)

- Any method invariant — a clause in both a method's pre- and postconditions — is automatically suggested in precondition selection problems for other public methods in the class, and is automatically checked as a (potential) postcondition for other public methods.

- When a clause has been established as a method invariant for at least half of the non-pure public methods in the class (those methods annotated as not mutating state), the interface prompts the worker to indicate whether or not the clause is an object invariant; if the worker agrees, the condition is added as a precondition to the other methods and subproblems are activated as previously described.

This scheme for handling object invariants is far from perfect; non-trivial object invariants should by written directly by the feature developer.

Contract Suggestions

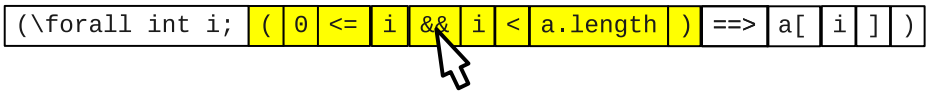
Writing contracts from scratch is difficult, especially for workers unfamiliar with the specification language. VeriWeb generates, and presents to workers, a suggested set of possible clauses. Workers can often complete their task just by selecting clauses; in other cases, workers can select some clauses and write others.

Contract suggestions convey two kinds of benefits. First, it is much faster, and requires less creativity, to select clauses rather than writing them from scratch. Second, the inferred clauses serve as a model when writing new contracts: they can both illustrate the syntax in which clauses are written and, even when slightly incorrect, can inspire workers to write similar clauses. We have observed all of these benefits.

VeriWeb makes suggestions for both pre- and postconditions; however, the list of subproblem types (Section 4.2.1) includes "select preconditions" but not "select postconditions" because VeriWeb automatically performs all postcondition "selection." When postcondition problems are presented to the worker, VeriWeb queries ESC/Java2 regarding each suggested postcondition, displays them all, and indicates which ones are verifiable and which ones are not verifiable; the worker may hide the unverifiable ones.

There are multiple ways to generate suggested clauses. The current VeriWeb implementation uses Daikon [47, 48] to dynamically infer likely clauses from execution traces. Because the suggested clauses generalize observed executions, they may not be true; even if true, they may be beyond ESC/Java2's ability to verify them. However, Nimmer and Ernst [104] found that (1) developers annotating programs are not encumbered by false inferred clauses, and (2) for two of their three subject programs, including inferred clauses had a statistically significant positive correlation with successful verification.

Figure 4.4: Subfragment highlighting, shown here in the drag and drop interface, helps workers read and understand clauses.



Drag and Drop Contracts

We observed that many workers wasted time attempting to write invalid clauses — clauses that were either syntactically or semantically incorrect. For example, some workers attempted to write facts about local variables in a method's preconditions. To guide workers in writing expressions, we developed a drag and drop interface to allow workers to construct clauses from a pool of starter "fragments." Invalid fragments, such as the `\result` and `\old` fragments when writing preconditions, are not available. Fragments have holes where other fragments can be inserted or removed. Contracts displayed in documentation can be added to the drag and drop interface by clicking on a button next to the contract.

Large contracts and fragments, especially those involving complex constructs such as universal quantification, are difficult for humans to read. VeriWeb's subfragment highlighting (shown in Figure 4.4) enables workers to better understand the subexpression groups. Additionally, the highlighting indicates the subfragment that will be removed when the worker clicks and drags. Subexpression highlighting is also enabled for contracts displayed in other parts of the interface.

4.2.2 Explanations in Context

Program verification tools tend to be inscrutable. After a verification failure, it can be difficult for workers to understand the tool's internal state or reasoning steps, and to know what changes would permit a specification to be verified. Workers are often frustrated as they try to form and maintain their own mental model of what the tool knows and/or can prove. VeriWeb offers clues to make this work easier or to eliminate it entirely. VeriWeb explains the relationship of elements *within* a subproblem with concrete counterexamples (Section 4.2.2) and tool tips (Section 4.2.2), and *between* subproblems with contract inlining (Section 4.2.2) and intermethod dependency information (Section 4.2.2).

Figure 4.5: VeriWeb clauses are "executed" over a dynamic trace. If the clause is falsified by the trace, the parameter and field values before and after the call are displayed in an expandable tree grid.

Name	Before Call	After Call
this.theArray	ref@14247437	ref@14247437
this.theArray[..]	length 2	length 2
this.theArray[0]	ref@6588476	ref@6588476
this.theArray[1]	ref@2891371	null
this.topOfStack	1	0

Concrete Counterexamples

A typical program verification tool indicates that a given clause is either provable or unprovable. Given an unprovable clause, a worker does not know whether the contract is false, the clause is true but beyond the reasoning abilities of the verification tool, or the clause would be provable if other parts of the specification were improved. We observed workers spending significant amounts of time fruitlessly attempting to prove false clauses. To eliminate this wasted effort, VeriWeb presents the worker with concrete counterexamples for clauses that are demonstrably false with respect to a set of real executions (e.g., from running the test-suite). VeriWeb does not currently generate concrete executions or tests; these are provided by the feature developer.

VeriWeb displays the counterexample information in two ways. First, when the worker hovers the mouse pointer over a subexpression of the clause, a tooltip shows the value of that subexpression. Second, the worker can explore all the parameter and return values via an expandable tree grid (see Figure 4.5). Expanding the grid shows the fields of each object.

Our run-time checking currently has two limitations: First, no testing is done of conditions in exceptional postconditions (because our trace generator does not handle exceptional exits). Second, expressions involving universal quantification can be tested only if the quantified-over variable has explicit lower and upper bounds.

Task-specific Tooltips

Contextual help messages are weaved into the interface. For example, in the drag and drop interface, hovering the mouse over a hole in a `\forall` expression shows what type of expression should be placed there (e.g., a predicate to bound the quantified variable).

Other examples of contextual help messages include tips shown between problems (while the next problem is loading), FAQ links displayed above and below interface elements, and additional tooltips.

Contract Inlining

When verifying a program, information about the absence of knowledge (e.g., "Why doesn't the tool have enough information to prove this postcondition?") can be as important as information about what the tool does know. To help workers locate "information gaps" between method calls, VeriWeb offers contract inlining in the source view. Contract inlining displays a method's contracts around a method call in the source code: the preconditions are above, the postconditions are below, and everything is horizontally aligned with the method call. An example is shown in Figure 4.6. The inlined contracts currently provide two pieces of additional information: (1) the set of preconditions that are not met at the call site and (2) a worker-toggleable list of unproven (potential) postconditions for the callee method.

Initially, no contracts are inlined; the worker can display as many or as few as desired. Contract inlining even works for source lines with multiple method calls: the inlined contracts are displayed from outside to inside in the order that the calls appear on the line (i.e., horizontally aligned with the corresponding call). Contracts can be verbose and might clutter the display, but we hypothesize that workers only need to inline contracts for 2–3 methods at a time (to visualize the information gaps). Therefore, our approach should scale even to methods that make many method calls.

Intermethod Dependency Information

Postcondition Dependencies We observed that it is difficult for new workers to form and maintain a mental model about what the verifier knows after a method call. One ramification of this is that the workers do not realize that warnings in a method may be caused by deficient postconditions for

Figure 4.6: Contract inlining: contracts for a method are aligned with the method in the source code. Workers can toggle the display of unproven postconditions. Inlining the specs for multiple methods can help to identify "information gaps."

```
public Object top( )
{
        No Preconditions
    if( isEmpty( ) )
        POST:(\result == true) == (this.topOfStack == -1)
        ▼Hide unproven postconditions
        POST:this.theArray != null
        POST:(\result == false) == (this.topOfStack >= 0)
        POST:this.topOfStack <= this.theArray.length-1
        POST:this.topOfStack >= -1
        POST:(\forall int i; (this.topOfStack+1 <= i && i <= this.theArray.length-1)
             ==> (this.theArray[i] == null))
        POST:(\forall int i; (0 <= i && i <= this.topOfStack) ==> (this.theArray[i] !=
             null))
       return null;
    return theArray[ topOfStack ];
}
```

the method's callees. Contract inlining (Section 4.2.2) addresses this problem. To further address it, when VeriWeb displays verifier warnings, it also lists the methods that are called before the warning and indicates that the worker may need to refine the postconditions of those methods. While data flow analyses could be utilized to make this information more precise, pilot tests showed that workers still found these messages helpful.

Information Transfer Because methods depend on one another, some information must transfer between subproblems. In VeriWeb, when a worker assigns blame (e.g., marking that a callee's postconditions are too weak), the worker is asked to explain the problem either in English or pseudo-code.

Relevant messages from other subproblems are displayed with the current subproblem. The worker completing the newly created task can mark whether or not the comment was helpful. If the worker indicates that the comment was not helpful, the worker is prompted to explain why they did not find the comment helpful. In the future, we plan to use this feature to automate the handling of payments when deploying VeriWeb on an ad-hoc marketplace such as Mechanical Turk [1].

4.3 Research Questions

To validate the VeriWeb tool, as well as the general potential for crowdsourced program verification, we posed three research questions:

Research Question 4.3.1. *What is the cost (time and money) of program verification?*

To answer this question, we performed a comparative study of VeriWeb and the ESC/Java2 plugin for Eclipse (Section 4.4). The subjects were 14 programmers recruited from Exhedra's vWorker labor marketplace. We found that the VeriWeb workers took both less time and money on average than the Eclipse workers for the subject program. Additionally, we found that worker progress with VeriWeb was more consistent than with Eclipse, which was characterized by work toward incorrect solutions and oscillations around local optima.

Research Question 4.3.2. *Can ad hoc labor be used to crowdsource program verification?*

To explore this question, we recruited workers from Amazon's Mechanical Turk at varying pay levels to complete VeriWeb subproblems (Section 4.5). We found the labor pool to be very shallow; additionally, the expected pay was not competitive with that of hourly contracted workers (cf. Section 4.4). The preliminary results suggest that current ad hoc labor marketplaces are not well-suited for verification.

Research Question 4.3.3. *How does decomposition and communication overhead affect the performance of collaborative verification?*

To characterize the overhead, we observed two pairs of computer science undergraduates each using VeriWeb to collaboratively verify a small program (Section 4.6). We found that, for each pair, VeriWeb generally isolated one participant from the other — for one pair, neither participant observed any communication from the other. Additionally, the observations highlighted that while collaboration can speed verification by enabling workers to address a root cause in parallel, a single poor-performing worker can significantly derail verification.

Pilot Studies During the development of VeriWeb, we performed pilot studies similar to the first study above. These studies were performed with both hourly contracted workers and more than 40 computer science undergraduate students.

4.4 The Cost of Program Verification

Creating a cost- and time-effective tool requires solving an optimization problem [120]. Though the problem involves many complicated factors, an approximation consists of only two complementary questions:

Research Question 4.4.1. *Given a fixed amount of money, how "much" verification can you buy?*

Research Question 4.4.2. *Given a fixed amount of time, how "much" verification can you buy?*

In this section, we begin to quantitatively answer both of these questions by contracting workers on Exhedra Solutions's vWorker [141] marketplace to verify a small project `StackAr`, which we consider to be of moderate difficulty based on prior work [104].

vWorker Labor Marketplace The vWorker [141] marketplace boasts a global workforce of over 320,000 registered workers, over 150,000 registered employers, and 1,500 – 2,500 projects in open bidding at any given time. Employers typically post small to medium business projects ($50 – $1000), including design and programming jobs, for workers to bid on in an open auction (invite-only auctions are also possible). The employer then selects a worker(s) to work on the project. For its role as a matchmaker and arbiter, vWorker takes a 7.5% – 15% cut from worker pay. Several sites offer services similar to vWorker. We chose vWorker due to my positive experience with the service in the past.

4.4.1 Experimental Design

We posted a project with the headline "Write Java program specifications" on vWorker. The project posting had workers place hourly bids for up to 6 hours of work and stated that we would accept multiple bids. We placed no restrictions on the workers. (vWorker lets you accept bids only from workers in developed countries, or mandate that the worker use a webcam so that you can monitor his/her work.)

Subject Program We used the `StackAr` program [143], an implementation of an array-based stack, from a previous study of program verification [104]. The program consists of a data type (library

class) paired with a test program that throws run-time exceptions if certain correctness properties do not hold. For example, the client checks that a call to `isEmpty()` returns `true` for a new stack.

The `StackAr` class has 8 methods, consisting of 49 NCNB lines of code. The client class consists of 79 NCNB lines of code. Using ESC/Java2, 23 JML annotations are necessary to show that the data type and client suffer no unexpected run-time exceptions.

We added clauses and annotations asserting the types of the objects and arrays so that we did not have to teach the participants the type syntax of JML (these clauses were trivial, and were automatically inferred by Daikon).

The Daikon-inferred clauses, which were offered to both VeriWeb and Eclipse workers (see below), came from Nimmer's client code, which used the library in realistic ways [104]. By contrast, VeriWeb's counterexamples came from the included ADT clients, which are (rather impoverished) example uses. As a result, the VeriWeb counterexamples did not add any additional information beyond what the Eclipse workers could learn by inspecting the included ADT clients.

Treatments We used the ESC/Java2 plugin for Eclipse 3.5[2] as the "standard" user interface. VeriWeb suggests possible clauses inferred by Daikon, so to level the playing field, we inserted those same inferred specifications in the subject programs used by Eclipse. Both interfaces were hosted on a group of Rackspace Cloud server instances (Eclipse was served via Windows terminal services). Progress in Eclipse was logged by recording each text edit and recording when the worker invoked ESC/Java2 to check the project.

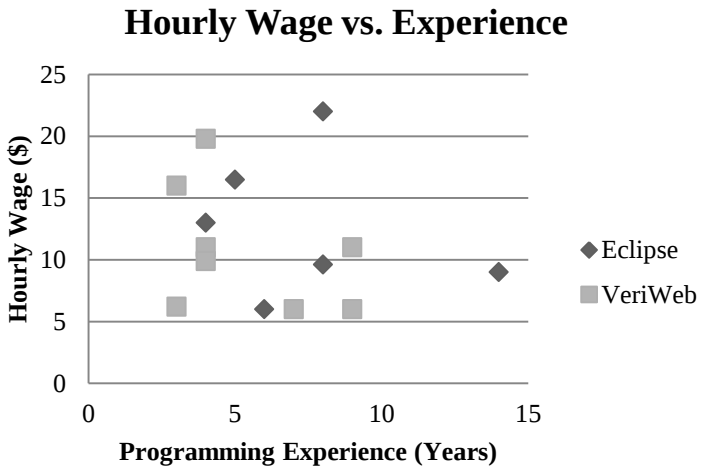
Participants We received 26 official bids ranging from $6/hour to $110/hour (Mean: $21.8/hour; Median: $14.5). For the experiment, we accepted the 18 bids in the $6/hour – $22/hour range and split the participants into treatment groups to produce roughly equivalent rate distributions. Complete bid data, including worker country, can be found on the project website.[3]

Worker Skill We expected that assigning the groups based on requested pay would result in groups with similar programming skill distributions. We asked each worker both his/her programming and

[2] `http://secure.ucd.ie/products/opensource/ESCJava2/`

[3] `http://www.cs.washington.edu/homes/tws/veriweb/`

Figure 4.7: Requested hourly pay (bid) versus programming experience reported upon completion of the project. Workers' bids were not an accurate proxy for programming experience.



Java experience.[4] Figure 4.7 shows that workers' bids do not correlate with experience, placing into doubt our initial assumption that requested pay is a viable proxy for skill.

Instructions Each worker was given a web link to a description of JML [16, 84] contract syntax and instructions for how to run his/her respective tool. Workers were instructed to spend approximately 1 hour on a warm-up tutorial. Upon completing the tutorial, they were given a quiz about JML specifications and their tool to ensure comprehension. For each worker's initial quiz attempt, we provided a list of questions answered incorrectly, references to the sections containing the answers to the questions, and had the workers correct their answers before continuing with the main task. Aside from one worker, all of the workers answered at least one question incorrectly on their initial quiz attempt. 9 out of the 14 workers finishing the project reported spending longer than an hour on the warmup. A deadline of one week was given for the project; though this deadline was not enforced, it may have contributed to the 4 workers who did not complete the project.

[4]We asked for this information after the experiment. Asking before might have given workers a motivation to lie, in order to obtain a higher wage.

Edit Distance Performance Metric The number of warnings reported by ESC/Java2 is not an accurate measure of progress, as one incorrect annotation can mask other warnings. Therefore, we instead use *edit distance*, the minimal number of additions and removals of top-level JML annotations that yields a known verifiable solution. This is a lower bound on the amount of work required to complete the specification task. (Actually, we use a modified version of the metric that accounts for the differences between the tools. See the Appendix.)

A verification problem has many possible (legal) solutions. The target solution set included solutions from [104], our pilot studies, and the solutions discovered during the study themselves. Whenever a solution included an object invariant, we added another solution in which the object invariant was explicitly listed as method conditions. The edit distance is the distance from a worker's current version of the program, to the nearest of any of these legal solutions.

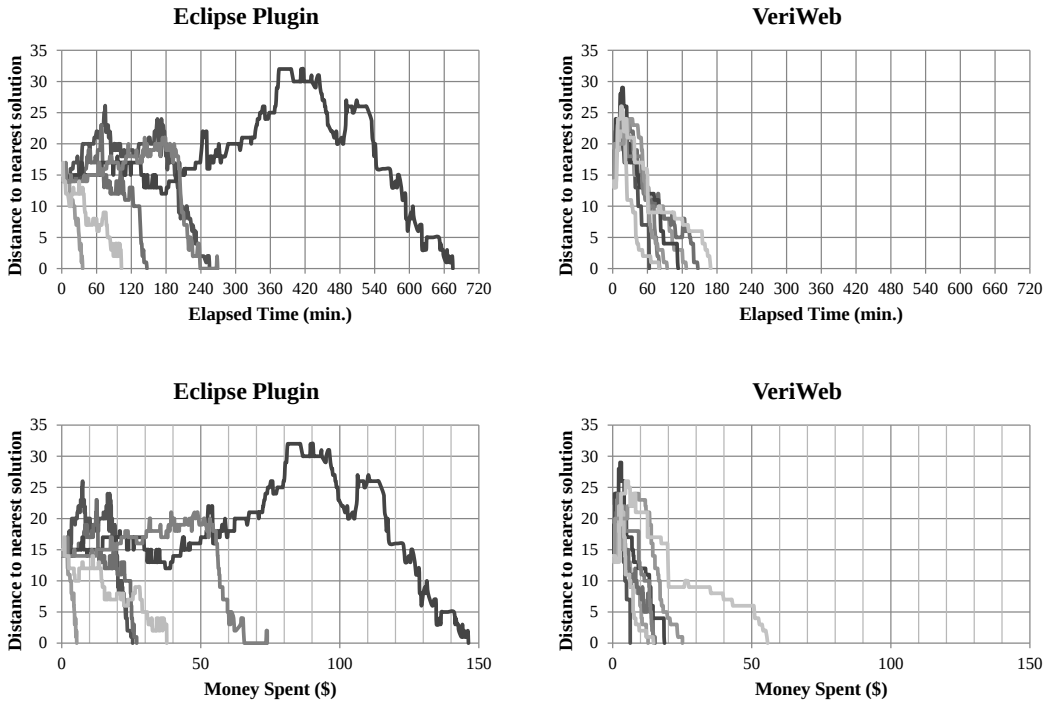
To efficiently calculate distance to verifiable specifications, we wrote a tool to normalize the specifications and perform a textual comparison. The normalization rewrites constant subexpressions and inequalities, uses De Morgan's law to split conjunctions (e.g., `P ==> Q && R` is split into `P ==> Q` and `P ==> R`), etc.

4.4.2 Results

8 VeriWeb workers and 6 Eclipse workers completed the project; the other workers (3 Eclipse workers, and 1 VeriWeb worker) would not comment on why they dropped out of the project. Figure 4.8 shows the distance to the nearest verifiable solution for the workers as both a function of time and money spent. The graphs illustrate three salient features.

First, on average, the VeriWeb workers completed the project faster and for less money than the workers using Eclipse. However, the relatively small number of workers in the study prevented the rejection of both the null-hypothesis that the completion time distributions are the same and the null-hypothesis that the money spent distributions are the same (the one-tailed Mann-Whitney U-test p-values are 0.0985 and 0.0694, respectively). If the Eclipse worker with 14 years of programming experience — an American who only charged $9/hr — is excluded from the sample, the differences are both significant (one-tailed p-values of 0.0202 and 0.0116, respectively). The variance in completion time is smaller for VeriWeb. This is promising because one main goal of VeriWeb

Figure 4.8: Top: progress as a function of time. Bottom: progress as a function of money spent. Data series (line) coloring is consistent between the graphs, e.g., the darkest lines in both Eclipse graphs both correspond to the same worker.

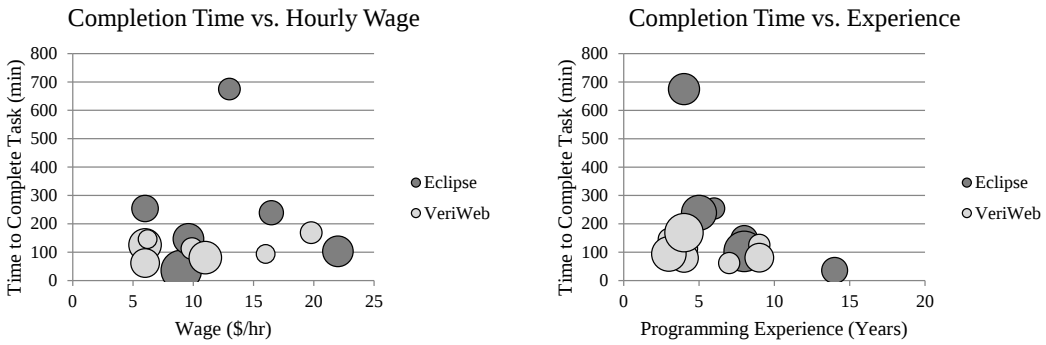


is to commoditize verification, and low variance (predictability) and commoditization go hand in hand. However, the null-hypothesis that the variances are equal cannot be rejected when using the non-parametric Brown-Forsythe Levene-type test (p-value is 0.0725).

Second, the VeriWeb workflow is characterized by continual progress with intermittent backtracking. By contrast, the Eclipse workflow is characterized both by work toward incorrect solutions and by oscillations around local minima. As with the reduced variation in completion time, this is encouraging with respect to commoditization.

Third, while the total cost of verification is linearly correlated with total time for Eclipse ($r = 0.946$), the correlation is lower for VeriWeb ($r = 0.694$) indicating that different workers should be chosen whether you are optimizing for time or money.

Figure 4.9: Left: time to complete the task as a function of hourly wage. Bubble size indicates the worker's relative programming experience. Right: time to complete the task as a function of the worker's self-reported programming experience. Bubble size indicates the worker's relative hourly wage. There is no relationship between wage and productivity for either tool — this is likely explained by the absence of a relationship between requested hourly wage and programmer experience (see Figure 4.7).



Effects of Skill

Figure 4.9 shows the time taken to complete the task given the workers' hourly rates and reported programming experience. As previously noted, the rate is likely not a good proxy for skill, as it was not correlated with worker experience. Multiple factors may confound the relationship between bids and skill.

- Variations in the cost of living and exchange rates between countries, and within countries, may cause the assumption to be violated if the geography of the workforce is diverse. We adjusted the bids based on an OECD report on purchasing power parity [113], but this still did not account for the differences (due to *intra*national bid variation).

- vWorker employs a worker rating system. Workers with higher ratings may demand a premium; skilled workers with a short work history may proffer a low bid in order to establish a good rating.

Feature Usage

The 14 successful workers took a short survey about their tool's specific features.

Suggested Conditions Contrary to what we found in pilot studies, the VeriWeb workers only found the suggested conditions moderately helpful (mean of 3 5/8 on a 5-point Likert item), due to the inclusion of incorrect or irrelevant conditions. One worker also felt that some of the conditions took too much effort to understand. The Eclipse workers found the suggested conditions more helpful (mean of 4 1/3 on a 5-point Likert item).

Drag and Drop Interface Three of the workers preferred the drag and drop interface, four workers preferred the text interface, and one worker used both depending on the complexity of the condition (preferring the drag and drop interface for complex conditions). As expected, the drag and drop interface was preferred for not having to worry about incorrect syntax; text entry was preferred for speed.

Contract Inlining All of the VeriWeb workers reported using the toggle-able contract inlining to solve the task (however two workers' responses indicate that they misunderstood the feature to which the question was referring). Workers reported using the feature to (1) remember conditions for methods that had already been visited, (2) identify missing postconditions, (3) view multiple method specifications simultaneously, and (4) understand unexpected exceptions.

Counterexamples VeriWeb generated concrete counterexamples for 4 of the 24 inferred clauses provided to the workers. While VeriWeb also prevented workers from writing additional clauses that violated the trace, Eclipse did not — the successful Eclipse workers introduced an additional 26, 13, 9, 9, 3, and 1 falsifiable clauses when working on `StackAr`. The distribution of these clauses' lifetimes were highly skewed with a mean of 34 minutes, and a median of just 2 minutes. Unsurprisingly, the Eclipse worker who introduced 26 falsifiable clauses took the longest to complete the task. For 2 of the workers, the number of falsifiable preconditions met or exceeded the number of falsifiable postconditions introduced. Falsifiable preconditions are especially harmful because workers "propagate" the false preconditions to the method's callers.

VeriWeb provided counterexamples to each of the workers, however only four of the eight workers reported being aided by counterexamples. One worker reported using the feedback to identify an off-by-one error. Another worker reported using the feedback to identify a missing condition for a

callee.

4.4.3 Discussion

Feature evaluation Directly measuring the effect of each VeriWeb feature on performance would have required an intractably large number of subjects. The survey responses and quantitative results indicate that each of the features contributed to the usability of VeriWeb. Feature usage metrics suggested that certain features, especially concrete counterexamples, are good first candidates for individual evaluation.

Threats to Validity Due to the small study size and single subject program, the performance characteristics observed may not generalize. In particular, the following caveats apply:

- VeriWeb's speed is bounded by ESC/Java2's speed, and ESC/Java2 does not scale.
- The subject program is array-based, whereas much Java code is collection-based. We opted to not use a subject program with collections because the ESC/Java2 specifications for collections are fragile (often resulting in unsoundness) and significantly slow the checker.

More general threats to validity are discussed in Section 4.7.3.

4.5 Verification with Ad hoc Labor

In this section, we explore the (lack of) ad hoc verification labor supply on Amazon's Mechanical Turk marketplace [1] for performing VeriWeb subproblems.

4.5.1 Mechanical Turk Labor Marketplace

Amazon's Mechanical Turk [1] is an online marketplace where employers can post human intelligence tasks (HITs) to be solved by a global ad hoc workforce. HITs are typically small tasks that are easy for a human to perform but difficult to automate. Common tasks include image labeling, preference surveys, and audio transcription.

4.5.2 Experimental Design

For our experiment, we created a batch of 50 HITs with the title "Answer questions about Java methods" and description "Describe what must be true when a method is called and after it runs." Each HIT required the worker to complete at least three VeriWeb subproblems (guaranteeing that each worker solved at least one precondition problem); workers could complete additional subproblems within the HIT for additional pay.

Subject Program The subject program was the array-based stack, `StackAr`, previously described in Section 4.4. To provide a consistent VeriWeb experience for each worker, each worker worked on his/own own copy of the project. Workers did not work collaboratively with other workers. The subproblems presented to a worker followed the methodology described in Section 4.2.1.

Variable Pay Mechanical Turk permits requesters to embed external websites within a HIT, informing the website whether the HIT is in preview mode, or has been accepted by the worker. We used this information to allow the workers to try using VeriWeb before accepting the HIT.

Typical HITs pay a fixed rate, however, the service offers the ability to pay a bonus for good work. Combined with the preview feature, we use this functionality to randomly pay a different rate to each worker (between $0.15 and $0.35 a subproblem). We set the static pay rate of the HIT to $0.00 and appended the phrase "BONUS PER QUESTION" to the HIT title and description. The preview screen for the HIT informed each worker the amount that they would be paid for solving each subproblem. Browser sessions were used to ensure that each worker only saw a single price, even when previewing the HIT multiple times. The VeriWeb interface was modified to display the amount of money the worker had earned so far in the top information bar.

Learning Curve VeriWeb's steep learning curve relative to other tasks on Mechanical Turk is a major obstacle. We opted to not provide a formal tutorial, instead relying on VeriWeb's contextual help (see Section 4.2.2); additionally, the first subproblem was chosen to be trivially easy — a "select requires" problem consisting of a single choice sufficient to eliminate all warnings — to encourage acceptance.

4.5.3 Results and Discussion

Ideally, the results of the experiment would enable the creation of a labor supply curve: the number of subproblems solved vs. pay (per subproblem). However, worker response was unenthusiastic — fewer than 10 workers accepted the HIT over the course of 3 days. Workers required at least $0.25 per subproblem to complete any subproblems. No Mechanical Turk worker completed more than 5 subproblems, indicating that the rate would have to be further increased to account for problem difficulty. We elected not to perform the study again with higher rates, as the effective hourly rate would not have been competitive with the hourly rate for workers contracted via vWorker (see Section 4.4).

Threats to Validity In addition to a possible lack of generalizability due to the use of the single subject program `StackAr`, the following three factors likely affected the observed result:

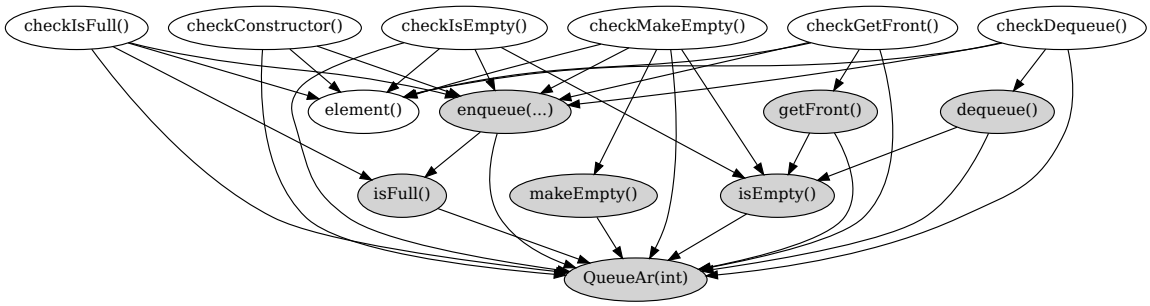
- The absence of a fixed price for the HIT reduces views of the HIT by workers who search for HITs by payment range.
- The HIT is unique and unusual. Workers are unlikely to "invest" in learning to perform a HIT if they perceive the opportunity to perform similar tasks in the future is low. Conversely, the novelty of the HIT may attract some workers that would not normally work on the HIT.
- Labor supply may vary throughout the week. However, allowing a HIT to run for a whole week would be sub-optimal — as the HIT ages, its position in the "latest HIT" list lowers, decreasing the chance that it will be seen by a worker.

Despite these threats to validity, it is clear that VeriWeb (or the presentation of VeriWeb) is insufficient to address the ease-of-use and profitability demands of the Amazon Mechanical Turk labor pool.

4.6 Overhead of Collaborative Verification

Since VeriWeb decomposes the task of writing a verifiable specification into subtasks, multiple workers can work on the task simultaneously. For programs with limited coupling (modulo library code), workers can largely work in parallel. To characterize the overhead incurred when workers

Figure 4.10: Method dependency graph for the subject program in the collaborative study (Section 4.6). The queue data type classes are shown in gray, the client classes are shown in white. The program has many methods for which the subproblems can be performed in parallel. For example, once the QueueAr(int) subproblems are solved, workers can simultaneously work on the subproblems for isFull(), makeEmpty(), and isEmpty().



work on interdependent subtasks, we observed five computer science undergraduate students using VeriWeb to individually and collaboratively verify a queue data type and client class.

4.6.1 Experimental Design

Participants The study participants were computer science undergraduate students at the University of Washington with 2 ½–5 years programming experience. Use of undergraduates was intended to emulate the use of semi-skilled workers.

Treatments We observed three treatments. For each treatment, the participants first (individually) performed a tutorial, and then verified the subject program.

- Treatment 1: a single participant performed the tutorial, took and received feedback on a comprehension quiz, and then verified the subject program.

- Treatment 2: two participants each individually performed the tutorial, and then collaboratively verified the subject program without the assistance of dynamic counterexamples.

- Treatment 3: two participants each individually performed the tutorial, individually took and received feedback on a comprehension quiz, and then collaboratively verified the subject program.

Treatment 2 was performed before the other treatments. The lack of dynamic counterexamples for Treatment 2 was due to a tool setup error. The quiz from the study in Section 4.4 was added in light of the Treatment 2 results (see Section 4.6.2) — we wanted to separate the effects of learning and tool comprehension from the effects of collaboration.

Subject Program We used the `QueueAr` program [143], an implementation of an array-based queue, which was also used in a previous study of program verification [104]. Like `StackAr` in the comparative study (Section 4.4), the program consists of a data type (library class) paired with a test program that throws a run-time exception if certain correctness properties do not hold. Figure 4.10 shows the dependency graph for the subject program; we selected this subject program because it exhibits a small amount of parallelism suitable for the number of workers in the study.

4.6.2 Results

Completion Time The individual participant (Treatment 1) with 2 ½years of programming experience completed the task in 52 minutes. For reference, Nimmer and Ernst had previously reported that none of their 41 participants could complete the task within a 60 minute period (participants were stopped after an hour) [104].

In both Treatment 2 and Treatment 3, the participants worked in parallel on subproblems containing warnings caused by a single root cause — a missing exceptional postcondition for the `enqueue` method. For both treatments, these subproblems were the most difficult for the participants to solve.

The pair in Treatment 2 completed the task in 54 minutes (wall clock time). 20 of the 54 minutes was spent assessing the missing exceptional postcondition: one participant spent 20 minutes studying the `checkConstructor` method; the other spent 10 minutes on the `checkIsEmpty` method. As both participants had stalled, we asked the participants to explain their reasoning out loud. Based on this information, we reminded the participants how to mark a callee's exceptional postconditions as insufficient. Both participants were then able to continue.

The pair in Treatment 3 had not completed the task after 55 minutes (wall clock time), but could not continue as an input caused the tool to crash. One participant correctly added an exceptional postcondition to `enqueue`, but this was insufficient as the `isFull` method was missing a postcondition about its return value. The other participant then began needlessly strengthening method preconditions

in an attempt to avoid the exception, ultimately triggering a combination that caused the tool to crash.

Messages During the course of the task, neither participant in Treatment 2 worked on a subproblem that had associated messages from the other participant. In Treatment 3, one of the participants worked on a problem with a message from the other participant that consisted of two syntactically correct JML clause suggestions. Since the tool does not attempt to parse messages, the participant saw the message as just an extra step to adding the missing conditions.

The lack of direct communication was a result of VeriWeb's rules for assigning subproblems, which prefers workers with knowledge of the subproblem (see Section 4.2.1). Additionally, for Treatment 2, the participants did not introduce incorrect specifications that required multiple levels of backtracking.

4.6.3 Discussion

The results highlight a major benefit of collaborative verification with VeriWeb: workers can simultaneously work on the same underlying issue. If one worker gets stuck, another worker might be able to identify and solve the issue by approaching it from a different perspective. The Treatment 3 results also expose a major drawback of collaborative verification with VeriWeb: a single worker can derail the verification process by solving subproblems incorrectly.

For the subject program, VeriWeb's problem preference system performed well for presenting a consistent workflow to the individual participants. Therefore, future research on the efficacy of the message system must still be performed, as the opportunity for interaction increases as the number of workers and size of the call graph increases.

Experimental Design Having the participants work side by side conveyed the benefit of a single observer being able to observe the workers simultaneously. Unfortunately, the setup precludes having the participants narrate their thought process. Additional studies should have multiple observers observing the participants in isolation to enable narration.

Threats to Validity In addition to the threats described in Section 4.4.3, the following threats to validity exist:

- The results may not scale to a larger number of workers working simultaneously due to increased interaction.

- All of the participants were native English speakers, and the documentation and variable names were in English. Workers with limited ability to read / write a common language are likely to incur greater overhead when communicating with other workers.

- The participants had all completed the same core computer science curriculum. In practice, workers may not have shared computer science training, or formal training at all.

Overall, we believe the benefit of being able to simultaneously observe the participants in person outweighed the threats to validity caused by our participant selection.

4.7 Discussion

4.7.1 Crowdsourced Verification in Practice

Our primary study (Section 4.4) explored the hourly pay model for verification in a global marketplace. Such a model lies in the middle of the spectrum between ad-hoc labor marketplaces like Mechanical Turk (Section 4.5) where work engagements are fleeting, and contract labor that is typical of more traditional approaches such as outsourcing. It remains to be seen under what conditions the different positions on the spectrum are optimal for program verification and other software engineering tasks.

In any case, verification and specification experience is scarce. Therefore, for the time being, employers that opt to use crowdsourcing must decide whether to pay a premium for workers with formal methods experience or to "train" new developers to use the technology, as we did in the vWorker study. The most cost-effective approach is most likely to meet the workers half-way: building tool support to decrease the skill demands of the task, and, at the same time, offering targeted training to set the workers up for success.

4.7.2 Validity of Specifications

VeriWeb generates a specification that is sufficient to prove the lack of runtime exceptions; the "correct" or intended specification is unknowable without input and validation from the feature

designers. However, there are three ways that VeriWeb can be led to the intended specification:

1. Informal documentation (Javadoc)

2. Checks in the implementation (e.g., checking of preconditions and throwing an IllegalArgumentException)

3. The feature designer can write JML specifications for high-level properties

Approaches #1 and #2 can be used by VeriWeb with no extra effort from the feature designer. Approaches #2 and #3 induce verification criteria that lead VeriWeb to find the "correct" specification, while Approach #1 depends on the workers taking cues from the documentation.

4.7.3 Threats to Experimental Validity

In addition to the experiment-specific threats enumerated in Sections 4.4, 4.5, and 4.6, the following threats to experimental validity apply.

Learning Effect Workers may perform better because they acquire more experience (a "learning effect"). For the studies in Section 4.4 and Section 4.6, we employed a mandatory warmup and quiz to mitigate the learning effect.

Program Correctness All of our subject programs are "correct" in the sense that ESC/Java2 can verify lack of run-time exceptions without modifying the source code. The performance and mindset of workers likely depends on (1) whether the program is correct, and (2) whether they believe that the program is correct. We did not tell the vWorker and Mechanical Turk workers that the programs were correct or incorrect. The vWorker workers might have inferred this from the task description. It is possible that (accurate) belief that a program is correct might qualitatively change the way that the workers would work.

Future research should investigate workflows for verification when software modifications are necessary, such as for fixing bugs or improving design. An intuitive workflow for dealing with software bugs would be to escalate areas that cannot be verified to software developers (possibly escalating to more-skilled program verifiers first).

Program Complexity Real programs often contain poor design: complex control flow, long methods, undesirable dependencies/coupling, etc. By contrast, the `StackAr` and `QueueAr` programs used in the experiments are very simple, and lack the use of common features, such as collections. Complexity and poor design make all software engineering tasks, including writing contracts, more difficult. Well-written, simpler code will cost less to work with.

Specification Complexity General-purpose libraries, such as the Java JDK, have extremely complicated specifications. In our experience, the JML contracts for client code are significantly less complicated. Just as it is not desirable to have low-skilled workers design libraries, it is not desirable to have low-skilled workers write specifications for libraries.

Additionally, in practice, JML specification writers simplify contracts by splitting them into cases. VeriWeb currently does not support cases, so developers must use implication, which becomes unwieldy. One way to support cases in VeriWeb would be introduce a new subproblem type that asks the developer to write (or select) the cases for a method contract.

4.8 Related Work

In this section, we survey related work in program verification and crowdsourcing.

Interfaces for Traditional Verification Tools The traditional program verification literature does not focus on user interface design, but sometimes contains it as a component. For example, Houdini, a static contract inference tool for ESC/Java, produces static HTML reports that contain hyperlinks to locations in the source code [53]. Flanagan and Leino reported that when working with Houdini, they repeatedly asked questions such as "Why didn't Houdini infer this precondition?" They note that anecdotal evidence shows that the presentation of the refuted annotations and the corresponding causes are the most important aspect of the user interface. Unfortunately, the authors did not enumerate any of their other questions.

Other work has focused on explicit deficiencies found in tools. For example, Kiniry [78] augmented the output of ESC/Java with warnings when the analysis may be either unsound or incomplete.

Pex for Fun (`http://www.pexforfun.com/`) is a web-based game designed to help students

practice programming [134]. Levels come in two forms: puzzles and coding duels. Puzzles ask the player a question about a method, or group of methods, e.g., for what input values does the program throw an exception? In a coding duel, the player must write a method that behaves the same as a secret implementation. The Pex tool is used to generate relevant input values; problems can also include Code Contracts to guide Pex [3].

Usable Program Verification Microsoft Code Contracts [27] provides language support for integrating contracts into programs in the .NET languages, as opposed to using a separate specification language such as JML. The productized version of the tool can statically verify some properties; other specifications are checked at run time. Leino's Dafny language and verifier also aims to provide guarantees for imperative languages beyond that of extended static checking [86]. It provides support for a richer set of properties, translating Dafny code to the Boogie verification language [4] to produce the necessary conditions for a SAT solver.

There is a growing body of work on inferring contracts, using static analysis, dynamic analysis, and a combination of both. Independent of this work, Yi Wei et al. utilized traces to invalidate quantified expressions generated by generalizing contracts [142].

There is also a push to depart from the de facto delineation of program, contracts, and verifier — these new approaches often adopt a "pay-as-you-go" mentality. For example, pluggable type checking techniques [107, 42] allow developers to incrementally extend the standard type system to check richer properties such as nullness, interning, and information tainting. In [123], Sheard et al. outline how the same philosophy can be applied in dependently typed languages that provide language-based verification. They argue that cognitive burden is reduced because properties fit naturally into the language.

Crowdsourcing Software Engineering Despite the growing base of crowdsourcing literature, there has been little academic exploration of crowdsourcing in software engineering. The only formal crowdsourcing research of which we are aware focused on end-user programming [128]. The commercial sector appears to have taken greater interest. For example, uTest is a start-up that uses crowdsourcing to provide on-demand software testing services [139].

Ad hoc Labor Markets The creation of Mechanical Turk in 2005 spurred crowdsourcing research by providing easy access to a global workforce for ad hoc tasks. As the field matures, research is transitioning from one-off proof of concept projects to crowdsourcing frameworks [91] and formal modeling of workflows [36].

As the understanding of dynamics of the labor market develops, tools that combine ad hoc labor to achieve a larger goal are possible. For example, Soylent, a word processor backed by MTurk, introduces a Find-Fix-Verify pattern for managing worker variance in tasks [7]. In the Find phase, workers identify parts of the document that need more work. In the Fix phase, workers propose revisions to the parts identified during the Find phase. Finally, in the Verify phase, workers determine which suggestions are the best.

Economics of Ad Hoc Crowdsourcing Horton and Chilton [70] construct a formal model of labor supply based on reservation wage — the wage that causes the benefit of performing a task to exceed the benefits of performing other tasks. They characterize this wage with two experiments: (1) increasing task difficulty while keeping a workers wage constant and (2) decreasing the wage while keeping the difficulty constant. The task they used, clicking between two vertical rectangles, requires little skill. The lack of skill requirement is commonplace among the economic crowdsourcing literature.

Mason and Watts [97] explore the rationality of the Mechanical Turk workplace, that is to what extent the workspace satisfies traditional economic models in which pay determine economic quality and quantity. To vary the price paid to workers, they set a base rate for the HIT and use Mechanical Turk's preview functionality to inform workers of an additional bonus. They find that pay has a positive relationship with the quantity of work performed on a task, but does not have a measurable effect on the quality. Furthermore, they provide evidence that enjoyment, intrinsic motivation, and other factors play a significant role in the marketplace.

Chapter 5

MEETING DEVELOPER INFORMATION NEEDS

5.1 Introduction

The previous chapters characterized how writing a verified specification requires a developer to gather, synthesize, and act on information. These requirements are not unique to specification and verification, however. Successful IDEs such as Eclipse and Visual Studio provide a comprehensive environment for these activities, and they can be extended with third-party plug-ins (add-ons) that provide new analyses, views, and actions. Despite this, many end-user developer information needs are not met by today's IDEs. In particular, end-user developers cannot readily combine information from multiple sources because the standard static extension point model demands that plug-in developers accurately anticipate future combinations [122]. In other words, while IDEs provide an environment for running multiple components, each of these components is in effect a black box, neither transparent nor interoperable.

This chapter introduces Cupid, an analysis framework for Eclipse that enables end-user developers to seamlessly discover, synthesize, and visualize information about their project and team. Computation in Cupid is based on pipelines. The pipeline model, made popular by Unix, provides greater flexibility than the static extension point model by allowing the developer to dynamically modify the connection between components. However, working within the IDE poses a unique challenge because, despite being part of an "integrated" environment, plug-ins generally don't provide open-access to information via published interfaces [83]. Cupid addresses this challenge using two key insights. First, the Standard Widget Toolkit (SWT) used by plug-ins provides a de facto API for information publishing. Second, type information provides context for how data may be combined and presented.

Cupid differs from previous work, most notably Mariani and Pastore's recent MASH tool for building and executing task-based workflows [96]. While MASH is targeted at performing developer-invoked sequences of actions, Cupid is targeted at providing contextual information in real-time.

To this end, composition in Cupid is based on lightweight components which are combined and visualized semi-automatically within the developer's existing workflow.

Cupid is not meant to be a total replacement for plug-in development. Instead, combining aspects of scripting and mash-ups, Cupid aims to bridge the gap between end-user analysis and plug-in development. Therefore Cupid attempts to (1) provide information automatically when possible, and (2) when developer action is required, minimize the knowledge the developer must have of plug-in APIs.

Contributions This work makes three contributions:

- The design of a pipeline-based information ecosystem for IDEs that enables developers to quickly combine, transform, and visualize information from multiple sources. Cupid, a prototype implementation of the ecosystem for Eclipse, is available open-source at `http://cupidplugin.com`.

- A "functional" model of developer information needs, the information required to answer questions that arise during development. The functional model is grounded in the availability of information as opposed to the often vague questions that developers ask. We demonstrate that this model explains the relationship between prior work and also show how Cupid's computation model subsumes that of prior work.

- A study with undergraduate students that indicates that novices can use Cupid's pipeline interface to quickly answer important questions taken from the software engineering literature.

We postulate there are three primary reasons why a typed pipeline architecture has not been realized in an IDE before. First, the lack of official common data interfaces between plug-ins meant a lack of inputs for pipelines. Second, the type representation needed to properly handle generic types was only recently introduced (Type Tokens, introduced to Google Guava [59] in 2012). Third, researchers have placed an unwarranted emphasis on iteration (which Cupid handles by supporting scripting, just as in Unix-based systems).

This chapter is organized as follows. Section 5.2 provides an example user story of a developer using Cupid. Section 5.3 describes the Cupid tool and implementation. Section 5.4 presents a

model of developer information that explains the relationship between both prior work and Cupid. Section 5.5 presents a controlled study of developers using Cupid to answer development questions. Section 5.6 describes related work.

5.2 Usage Example

This section presents a narrative of a developer, John, using Cupid to gain insight into his project and team. John has just joined a new team and has been assigned to work on a new feature. He believes he has found the component to modify, but is unsure how the code operates.

5.2.1 Whom should I contact about this file?

John right-clicks a file to open the Git log view and looks through the list of authors to see who has edited the file most frequently. This isn't bad for the first few files, but it quickly becomes tedious.

Therefore, John would like the most frequent author for a file to be automatically displayed whenever he selects the file in the Package Explorer or anywhere else in the workbench. John could write a custom Eclipse plug-in with this functionality, but it is easier to do in Cupid.

John plans to customize Cupid's Selection Inspector view, which displays information about the currently selected workbench item. Figure 5.1 shows John's goal: the Selection Inspector displays a new "Most Frequent Author" for a file along with information such as the file's "Last Modified Date." Conceptually, John wants to take the commit history for the file, select the author of each commit, and then compute the most common author. As shown in Figure 5.2a, these steps can be thought of as a pipeline of operations taking a file as input.

To make the pipeline in Cupid, John opens the Pipeline Creator. To produce the author of each commit, John first expands the "Git History" capability, which returns the commit history for a resource (e.g., file), and then John double-clicks the `[getAuthor]` entry to add it to the new pipeline. The dialog now indicates that the current pipeline has an input type `IResource` (a file, package, etc.) and output type `List<String>`. Figure 5.3 shows the Pipeline Creator after John has added the component that gets a list of commit authors for a file. To produce the most frequent commit author, John adds the "Most Frequent Element" capability to the pipeline by double-clicking the capability in the capability tree. (Cupid hides the incompatible capabilities. Because the "Most Frequent Element"

Figure 5.1: The Selection Inspector. The selection inspector shows information about the currently selected workbench item (a file in the Package Explorer, in this case). The developer has customized it to display the "Most Frequent Author."

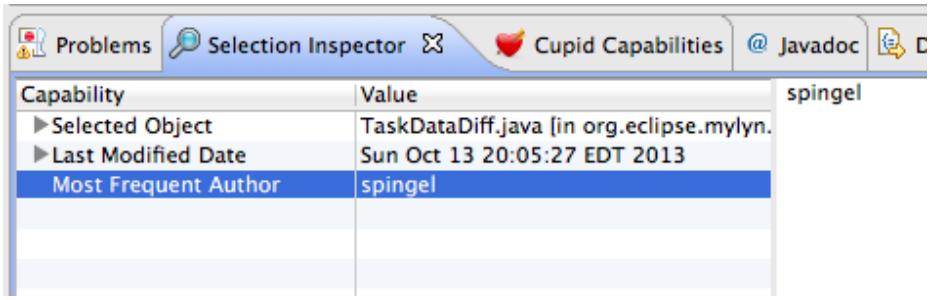
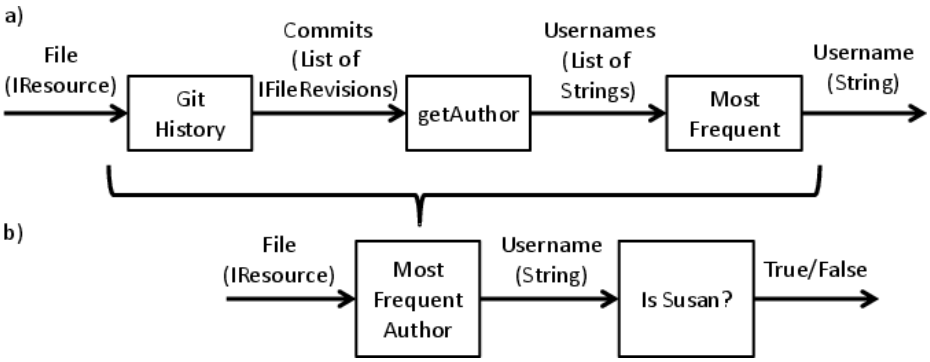


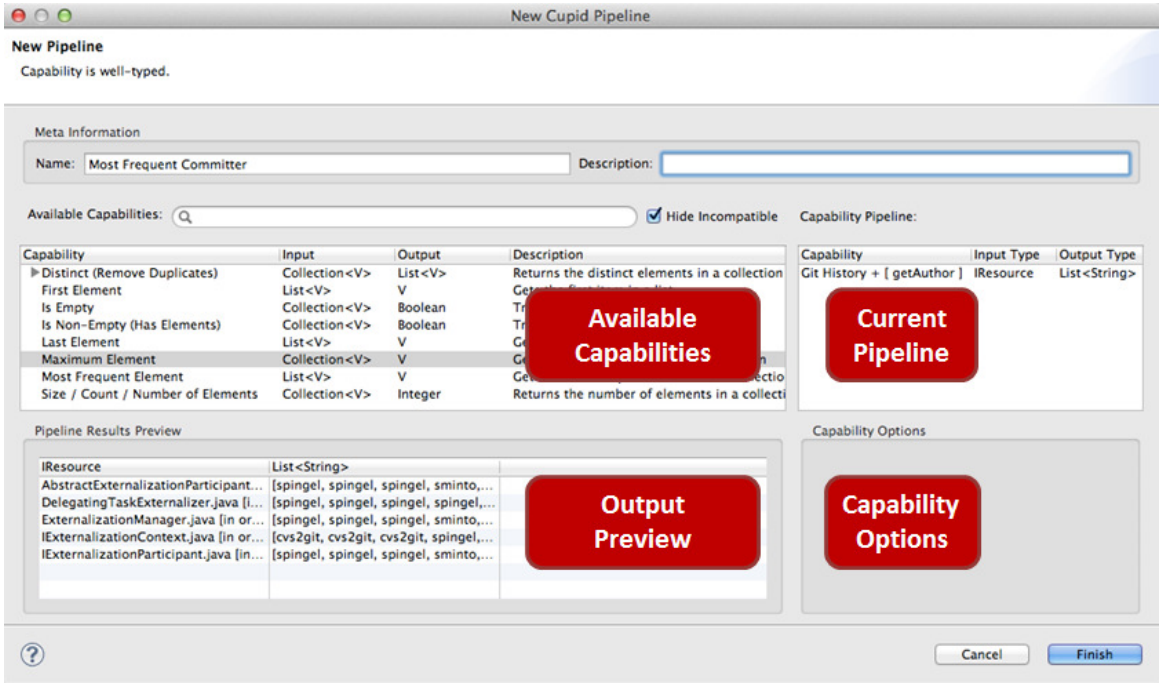
Figure 5.2: Top (a): a pipeline for computing the most frequent author for a file. Each box represents a capability; the arrows represent data flow. The data flow between capabilities is typed. In a pipeline, the output type of a capability must be compatible with the input type of the capability consuming the output. Bottom (b): a pipeline for determining whether Susan is the most frequent committer to a file. The bottom pipeline reuses the top pipeline as a capability.



capability is displayed, it is valid to add it to the end of the pipeline). Cupid shows that the pipeline has the input type `IResource` and output type `String`. John names the pipeline and clicks OK. Based on the input type `IResource`, Cupid knows that the most frequent commit author information is relevant for files (and other resources). For example, if john has the Selection Inspector view open, whenever John selects a file *anywhere* in the workbench, Cupid will automatically compute and display the most frequent committer for the selected file (Figure 5.1).

John sees that Susan is the most frequent committer to the files that he needs to modify. Before contacting Susan, though, John wants to see what other parts of the project Susan has worked on to

Figure 5.3: The Pipeline Creator. The pipeline creator lets the developer combine and transform information from multiple sources. The capability tree (middle left) shows the capabilities that can be attached to the end of the pipeline. That is, their input type is compatible with the current output type of the pipeline. For each capability, the tree also shows accessor capabilities (not shown) which call a getter on the result. Adding the accessor capability adds both the main capability and the accessor to the pipeline (shown as a single capability in current pipeline because it conceptually acts as a single capability). The middle right pane shows the current pipeline, currently "Git History + [getAuthor]" which takes a resource (e.g., file) and returns a list of strings. The bottom left pane shows a preview of the output for the current pipeline for the files the developer had selected. The options pane (lower right) shows options, if any, for the currently selected capability.



better understand her role on the team.

5.2.2 *Which files are owned by a particular developer?*

John would like to highlight all files in the Package Explorer for which Susan is the most frequent committer (Figure 5.4). Conceptually, John wants to take the data produced by the pipeline he just created, filter (apply a predicate to) the data, and then apply formatting to all the items that passed the filter (Figure 5.2b).

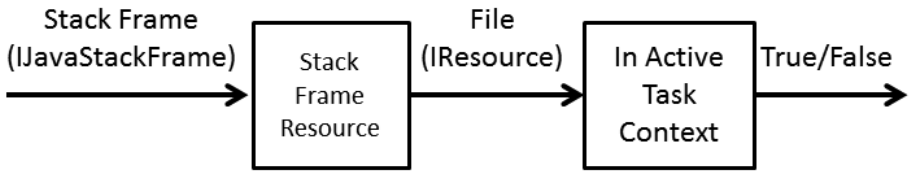
John can accomplish this using Cupid's conditional formatting rule feature. Conditional format-

ting allows John to specify an input, a condition (predicate), and a format (e.g., background color) to apply when the condition is met. John right-clicks on a file and selects the "Create Formatting Rule" option from the context menu. In the menu, John selects the capability he just created for computing a file's most frequent committer. John then writes a snippet of code to act as a predicate (`val.Equals("Susan")`). Figure 5.5 shows the wizard giving John auto-completion support when writing the condition. After writing the snippet, John clicks to go to the next screen in the wizard where he can select a formatting to apply when the condition is met. John indicates that elements matching the rule be given a yellow background. When John clicks "Finish", all of the files owned by Susan are highlighted in yellow (Figure 5.4).

With Susan's help, John begins implementing the feature. While debugging, he hypothesizes that an error in his new code is producing bad data that is propagated via method calls eventually causing an exception. Therefore, John wants to see which parts of the execution correspond to his modifications.

5.2.3 Which stack frames correspond to the active task?

John decides to create another formatting rule to bold the stack frames that correspond to files in the active task's context. To do so, John first creates a pipeline that gets the resource associated with a stack frame and then returns true if the resource is in the active task context.



John opens the Pipeline Creator wizard and selects the `Resource` output for the "Stack Frame Resource" capability. He then attaches the "Mylyn in Active Task Context" capability. John then right-clicks on a stack frame and selects "Create Formatting Rule" from the Cupid context menu. John selects the pipeline he just created and indicates that the font should be bolded for stack frames that match the rule. When John clicks "Finish," all of the stack frames corresponding to the active Mylyn task are bolded.

5.3 Tool

This section provides an overview of the design and implementation of the Cupid ecosystem. Cupid is available as an open-source project at `http://cupidplugin.com`. We encourage the reader to view the demo videos on the website as a supplement to reading this section.

Plug-in Terminology Each component of Eclipse is a plug-in. Even the core Eclipse resources component and Java Development Toolkit (JDT) are structured as plug-ins. Therefore, this paper uses the term "plug-in" to refer to both 3rd-party plug-ins and the Eclipse distribution itself.

5.3.1 Design Goals

We designed Cupid to achieve a single primary goal:

Design Goal 1. *Make information discovery and visualization simple; make complex information synthesis possible.*

Cupid is not meant to be a total replacement for plug-in development. However, Cupid aims to bridge the gap between end-user analysis and plug-in development. Cupid is most appropriate for providing rapid access to information in a digestible form, both with respect to visualization and to location within the workbench. Cupid is not appropriate for UI automation, long tasks blocking workspace resources, tasks that require developer feedback, or tasks that modify the behavior of the UI. To this end, the design of Cupid is also informed by the following two corollary goals.

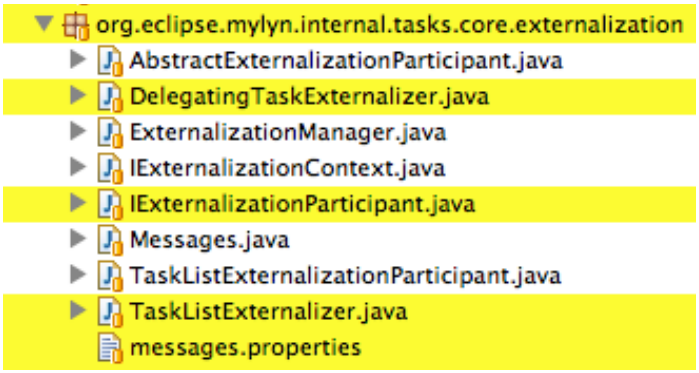
Corollary Design Goal 1. *Provide information automatically when possible.*

Cupid should display all relevant information automatically. If the developer is not interested in the information, she should be able to hide the information.

Corollary Design Goal 2. *When developer action is required, minimize the knowledge the developer must have of plug-in APIs.*

The developer should not have to interrupt her flow to learn a plug-in's API. Seamless data discovery and analysis is critical for enabling developers to pose and answer new questions during development.

Figure 5.4: The Package Explorer highlighting files owned by a particular developer. This was achieved via a conditional formatting rule.



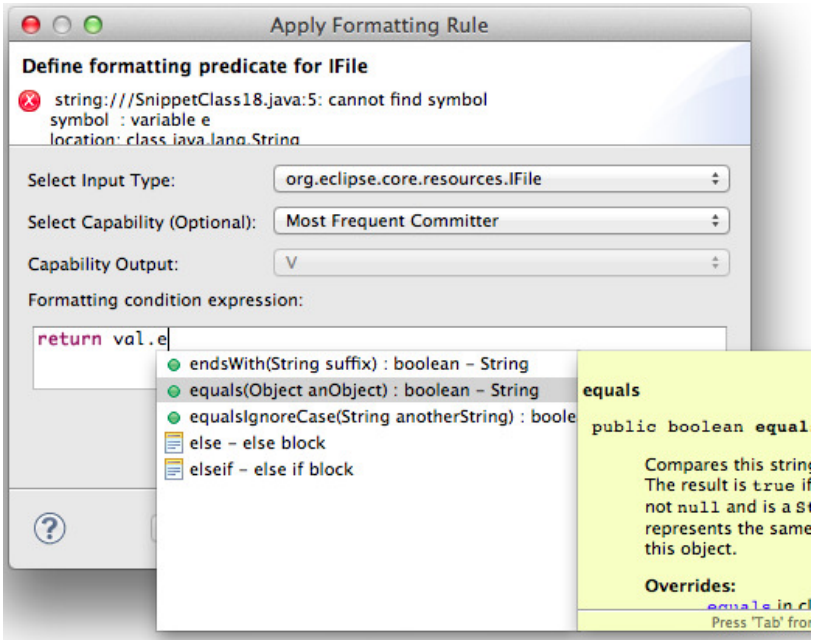
The following sections describe the computation model and features of Cupid, and how they support these goals.

5.3.2 Capabilities

Cupid's computation model is based on *capabilities*: typed services that, given an input, produce a future (i.e., promise) that calculates an output. A capability can have multiple inputs and outputs. For example, the "Mercurial Diff" capability takes as input a text buffer and a revision number, and outputs the line ranges that were added, modified, and deleted. The "Stack Frame Resource" capability takes as input a Java stack frame and outputs both the corresponding resource (`IResource`) and the line number.

The capability-based model provides two primary benefits to the developers and the execution engine. First, the developers are already familiar with typed interfaces and chaining calls to typed interfaces (as sub-routine calls, or pipelines similarly to Unix and Windows PowerShell). Second, since units of computation are explicitly represented, the computation engine can optimize and concurrently execute the computation (see Section 5.3.2). Conceptually, Cupid's computation model is similar to dataflow (e.g., Simulink [124]) and task/workflow (e.g., [96]) computation models. The salient difference is that a capability produces a future for its input rather than executing directly — a distinction that is important because capabilities are often executed concurrently (e.g., for each item in a table when performing conditional formatting). Additionally, capabilities are intended to be

Figure 5.5: The Conditional Formatting Rule Editor. The developer writes a predicate for the rule with the assistance of autocomplete. The developer can also additionally select an existing capability to first apply to the input; in this case, the developer is writing a rule that highlights files in the Eclipse UI based on their most frequent committer (Section 5.2).



more lightweight than tasks. For example, capabilities which just call an accessor (getter method) on data are frequently used when creating a pipeline of capabilities (Section 5.3.3).

End-users create new capabilities using wizards and scripting (Section 5.3.3). Plug-ins can expose capabilities to Cupid (and the end-users) by implementing a publication interface, `ICapabilityPublisher`, that clients use to request the set of capabilities provided by that plug-in and listen for changes to that set. The Cupid Platform itself serves as a centralized aggregator and publisher of all capabilities. A plug-in can register capabilities with the centralized aggregator, or implement its own aggregator (e.g., that enforces some policies, such as origin or code signing), which can in turn (optionally) re-publish to the centralized aggregator.

Capability Typing

Cupid capabilities are typed, and Cupid ensures that when a pipeline is constructed, each capability's input is compatible with the previous capability's output.

Cupid automatically uses type adapters to enable capabilities to work together when they represent the same information using different types. For example, the JDT plug-in includes Compilation Units and Java Projects, which correspond to Eclipse Files and Projects, respectively. Developers can register type adapters by writing a plug-in that extends the type adapter Eclipse extension point.

Computation Engine

Cupid's computation engine manages the concurrent execution, caching, and coalescing of futures produced by capabilities. Cupid concurrently executes futures using the Eclipse Jobs API [45]. The Jobs API handles the scheduling and progress monitoring based on resource utilization, job priorities, and scheduling policies.

Capability Composition Each capability controls which capabilities it calls — Cupid's execution engine operates on capabilities, not on an execution graph of capabilities. A pipeline of capabilities is structured as a coordinator capability that calls each of the component capabilities in turn. In a graph-based execution model, the pipeline would instead be represented as a DAG of capabilities. The benefit of having coordinator capabilities is that encoding value-dependent iteration is straightforward. In a graph-based model, this might be encoded as a cycle with metadata, or by dynamically generating additional sub-graphs.

To compose multiple capabilities (either via a wizard or code), a developer creates a new capability which calls the existing capabilities. These existing capabilities may themselves be combinations of capabilities. The calls are, in effect, sub-routine calls with two primary characteristics: (1) Cupid may perform caching and/or coalescing on the sub-routine call (see below), and (2) the calls are by name, so the referenced capability may be re-bound (e.g., via the scripting feature described in Section 5.3.3).

Caching and Coalescing Cupid optimizes computation in two ways. (1) Cupid caches the result of capabilities, invalidating entries whenever the input changes. (2) Cupid coalesces multiple simultaneous job requests for a capability and input. That is, if a new request is issued for a capability that is already running, the new request is associated with the currently running capability instead of re-running the capability. These optimizations are analogous to removing redundant sub-graphs from

an explicit graph-based representation of the computation.

5.3.3 Information Discovery and Synthesis

To satisfy Corollary Design Goal #2 (minimize required knowledge of APIs), Cupid leverages existing plug-in UI (i.e., tables and trees), which the developer already interacts with, as the starting point for the information discovery and synthesis process. Once the developer has established a reference to plug-in information, Cupid guides the developer in transforming and visualizing the data.

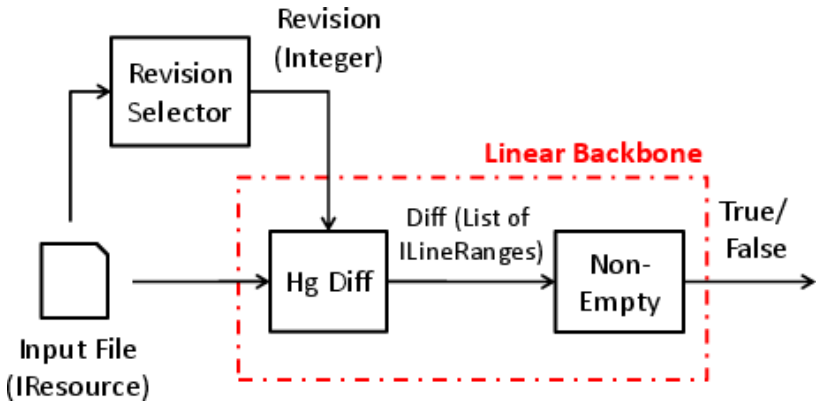
Information Extraction

Information extraction is the process of retrieving (e.g., via accessor methods) information from a plug-in so that it can be transformed and visualized. This information may or may not be displayed in the UI, and may only be accessible by opening a particular dialog or view. Extracted information is used as input to the pipelines, scripts, and visualization described later in this section.

To extract information with Cupid, a developer selects one or more elements in a table or tree in the Eclipse workbench and selects the "Extract Capability" context menu option. Cupid shows the developer a listing of all available accessors for the selected object (or the common supertype, if multiple objects were selected). When the developer selects an accessor, a new capability is created which calls the selected accessor on its input.

UI-based end-user information extraction is effective when a plug-in computes information but does not display it. However, some information must be generated by an API call. To enable UI-based pipelining, plug-ins need to expose API methods as capabilities. To do so, a plug-in implements the `ICapabilityPublisher` interface and contributes to the corresponding publisher extension point. Creating a capability generally just involves wrapping one or more pre-existing API calls and declaring the inputs and outputs. Plug-ins exposing capabilities for other plug-ins can be written and distributed by a 3rd party. Cupid come pre-packaged with capabilities for the popular the popular Mercurial Eclipse, EGit, and Mylyn plug-ins, as well as the Eclipse JDT and Resources components.

Figure 5.6: A pipeline for determining if a file differs from a previous revision. The "Hg Diff" and "Non-Empty" capabilities each have a single required input and form the linear backbone of the pipeline (the "Hg Diff" capability uses the head revision by default). The "Revision Selector" capability provides the input for the optional revision input of the "Hg Diff" capability. While the "Revision Selector" capability is shown as a capability, it could itself be a pipeline consisting of multiple capabilities (since a pipeline is a capability).



Pipeline Wizards

Cupid provides wizards for creating pipelines of capabilities and cross-referencing the outputs of two capabilities to produce a map. These provide UI support for common relational operations/transformations, e.g., "select" and "group by".

Pipeline Creator The pipeline creator (Figure 5.3) enables developers to create a linear combination of capabilities. The interface is designed to support a Select-Transform-Synthesize pipeline structure. To simplify the interface, the pipeline creator limits developers to building pipelines with a linear backbone — each component can only have one required input. However, optional inputs are allowed, and the developer can specify a capability for an optional input as long as that capability is compatible with the input type for the pipeline (see Figure 5.6).

As when performing information extraction, the wizard lists the accessors for the outputs of each of the capabilities. Under the hood, the capability produced for the pipeline will call an accessor capability (which takes an object and an accessor name as input). When a developer has added a capability to the pipeline, Cupid indicates which capabilities can legally be attached to the end of the pipeline (i.e., the input type of the capability is compatible with the output type of the current

pipeline). If the developer is creating a capability for information they've selected in the Eclipse workbench, Cupid shows a preview of the output for the current pipeline. These three features assist the developer in using capabilities for unfamiliar plug-ins.

Additionally, to ease use, Cupid comes pre-packaged with a library of utility capabilities for working with collections (e.g., producing the most frequent element, maximum element, distinct elements, etc.).

Mapping Wizard The mapping wizard is an interface for cross-referencing the outputs of two capabilities (i.e., performing a "group by" operation). The wizard was originally designed to be used with the Map View (Section 5.3.4). Section 5.4.3 presents a case study of implementing a prototype of the hierarchical view described in [57].

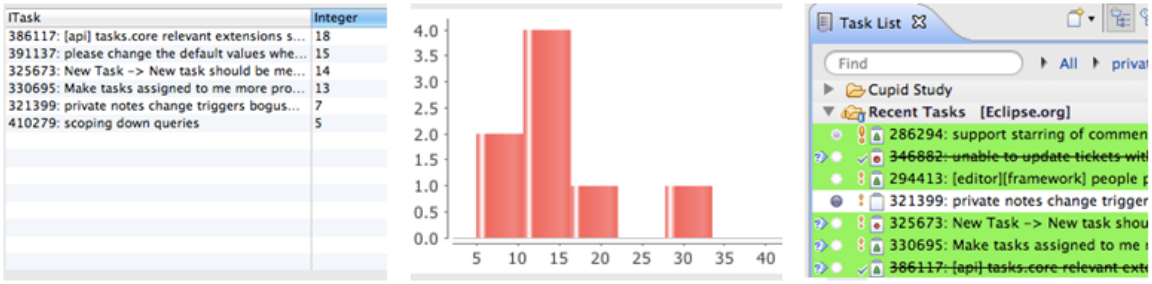
To use the mapping wizard, the developer selects a key generation capability (or an object type), a value generation capability, and a grouping predicate (currently only equality is supported). For example, to create a mapping from each author to their commits (when using Mercurial), the developer would select the "getAuthor" accessor of the "Hg Log" capability for the key generator, the "Hg Log" capability for the value generator, and link the results using the "getAuthor" accessor.

Capability Scripting

To allow developers to perform complex analyses, Cupid enables developers to script new capabilities with Java without leaving their workspace (and without running multiple Eclipse instances). To do so, Cupid maintains a Java scripting project within the developer's workspace. Cupid watches for changes to the project's class files and dynamically reloads the capabilities. Therefore, changes to capabilities are immediately reflected. Since this approach works with any JVM language, Cupid could support Scala, Groovy, or JavaScript (via Nashorn) as a more concise scripting language.

Iteration While the wizards currently only support linear combinations of capabilities, scripts can support arbitrary combinations of capabilities, including iteration. Each capability controls which capabilities it calls — Cupid does not maintain a computation graph. Optimization is performed via caching and coalescing (see Section 5.3.2).

Figure 5.7: The Report View (left), Histogram View (middle), and Conditional Formatting (right) displaying information about the number of comments for Mylyn tasks (i.e., issues). The Report and Histogram Views are packaged with Cupid. The Mylyn Task list is provided by the Mylyn plug-in. The developer has applied a rule to highlight tasks which have more than a specified number of comments. Custom formatting is possible without modifying the Mylyn source because Mylyn uses the Standard Widget Toolkit (SWT) for displaying data.



Build Path The scripting environment uses the Eclipse bundle information to determine the location of external plug-in classes, and provides quick fixes to the developer to reference the classes and import the correct packages. Cupid automatically updates the scripting project's references each time Eclipse is loaded in order to account for plug-ins updated by the developer. The scripting project additionally serves as the context for Java auto-completion support (e.g., for the conditional formatting rule snippet editor in Section 5.3.4).

5.3.4 Visualizations

Capabilities specify how to generate information. Visualizations allow developers to consume this information *where* and *when* they need it. Cupid's visualizations can be broken into two groups: (1) visualizations that generalize idioms that have proven to be effective in IDEs for specific purposes, and (2) visualizations that are commonly used for data analysis, but are not traditionally found in an IDE. For example, the Eclipse debugger excels at helping developers explore executions of code for which they are unfamiliar; Cupid's inspector view and conditional formatting visualizations borrows from these interfaces to help developers explore project information for which they are unfamiliar.

Cupid is currently packaged with seven mechanisms for visualizing information:

- The *inspector view* (Figure 5.1) shows the output of capabilities that apply to the item currently selected in the workbench (e.g., a compilation unit).

- The *report view* (Figure 5.7) shows input/output pairs for multiple items selected in the workbench; the developer can sort the rows by clicking the column header to sort by.

- *Conditional formatting* (Figure 5.7) support for common SWT widgets (currently tables and lists) based on capabilities. To the extent that plugins use these common components, the developer can apply conditional formatting rules based on capabilities anywhere in the workbench.

- The *editor ruler* augments Eclipse text editors with a view of capabilities that produce line ranges. For example the Mercurial "Hg Diff" capability returns ranges that have been added or modified between two revisions.

- The *map view* is a graphical view that shows the output of capabilities that produce a mapping between two collections of data.

- The *chart view* allows the developer to chart the output of a capability. Like the inspector view, the chart view supports outputting the result for the current workbench selection. Currently Cupid supports histograms (Figure 5.7) and pie charts via JFreeChart [72].

- The *marker* system shows Eclipse markers for capabilities that produce markers.

Selection Inspector View The selection inspector is the de facto view for consuming information in Cupid. Its design is inspired by the Eclipse debugger's Expressions View. The selection inspector displays the currently selected workbench item and the output for all compatible capabilities. The developer can expand values in the inspector to view their accessors (which can be expanded, in turn). Since the developer can always have the inspector view open, it provides a way to instantly view information that an existing plug-in interface does not expose directly. Providing both immediate data access and exploration, the selection inspector is one of the key features in making Cupid effective for end-user exploratory analysis.

Conditional Formatting Conditional formatting (Figure 5.4) is inspired by conditional breakpoints. To create a conditional formatting rule, the developer selects one or more elements in a tree or table and then selects "Create Formatting Rule" from the context menu. The developer can optionally choose a capability rule to transform the input. The developer then writes a snippet of code that specifies when the formatting rule should be applied (Figure 5.5) and selects the formatting to apply. Cupid provides auto-completion to the developer when writing the snippet, to reduce the need for the developer to be familiar with the plug-in API.

While Eclipse provides a decoration API for tables and trees, not all plug-ins expose these extension points. Therefore, Cupid implements conditional formatting by walking the SWT tree for all views to locate the SWT tables and trees.

5.3.5 Discussion

While Cupid provides new views and adds additional sinks to existing views, Cupid does not enable the composition of visual contributions by plug-ins. Similarly, it does not enable the composition of actions. Since composition of UI elements and actions are not dataflow-based, we believe that the pipeline model is not well-suited for these purposes. Work by Lerner et al. on composing browser plugins that modify page rendering suggests that an aspect-oriented approach may be promising [90]; Cupid's pipelines could be a good mechanism for producing advice for join-points.

5.4 Expressivity and Validation

This section validates Cupid's design and implementation for meeting developer information needs. Where applicable, it compares and contrasts Cupid to related work. Section 5.4.3 presents two case studies of how Cupid allows plug-in developers and researchers to rapidly prototype new interfaces and analyses that better meet developer information needs.

5.4.1 Developer Information Needs

In [57], Fritz and Murphy present 78 questions that they obtained by interviewing professional developers. For each question, they identify the information "domains" required to answer the question: source code, change sets, teams, work items, comments, web/wiki, stack trace, and test

Name	Previously Implemented?	Pre-computed?	Number	Description	Example
Pre-computed	Yes	Yes	13 (17%)	Questions that can be answered using information pre-computed by an existing tool	"Which code reviews have been assigned to which person?" can be answered by looking the code review tool's dashboard
On-demand	Yes	No	10 (13%)	Questions that can be answered by invoking an existing analysis; these can include one-off tools/analyses such as [15]	"Who changed this code?" can be answered by running the version control system's `blame` command
Relational	No	Yes	32 (41%)	Questions that can be answered by combining information pre-computed by existing tools	"Who is working on what?" can by answered by grouping the open tasks by assigned team member
Functional	No	No	8 (10%)	Questions that can be answered by combining pre-computed and on-demand information	"Who is responsible for a failing test case? (stack trace)" can be answered by computing the `blame` for each stack frame and grouping frames by committer
Ill-defined	?	?	9 (12%)	Questions that are too vague to categorize as relational or functional	"How do test cases relate to work items?" might simply involve cross-referencing work item text or could involve associating code covered by a test with task contexts
Other	?	?	6 (8%)	Tool-dependent questions and questions using web-resources	"Which API has changed (check on web site)?"

Table 5.1: Developer information needs categorized by information availability for the 78 developer questions identified in [57]. The "Previously Implemented?" column indicates whether the information is, or can be, generated by an existing plug-in. The "Pre-computed?" column indicates whether the underlying information is pre-computed or must be computed on-demand. Relational and functional models are a good match for answering developer questions.

cases. The labeling highlights that questions with similar wording may differ significantly due to their information domain. For an IDE such as Eclipse, information for each domain is produced by a plug-in. For example, source code information is provided by the Java Developer Toolkit (JDT) plug-in, while change set information is provided by a plug-in such as EGit or Mercurial Eclipse. The correspondence between tool and domain suggests that the list of developer questions is not exhaustive, as other domains each have their own questions, such as questions about code coverage, static analyses, etc. Indeed, varying interpretations of the questions could demand information from additional domains/plug-ins. For example, to obtain a precise answer to the question "How do test cases relate to work items?", one might use trace and/or slicing information. Information domain(s) is just one facet of developer information needs.

Information Availability Model A more important — from a practical perspective — facet of developer information needs is the *availability* of information. Table 5.1 provides a break-down of the questions in [57] by (1) the location of the information: whether the information can be generated by an existing plug-in, and (2) whether the information can reasonably be pre-computed, or must be computed on-demand. 23 of the questions (30%) can be answered using an existing plug-in (either pre-computed or by invoking an existing analysis). 32 of the questions (41%) can be answered by combining information pre-computed by existing tools. We call this category "Relational" since standard relational operations a la SQL (e.g., joins, grouping, sorting, etc.) are sufficient to compose the information. 8 of the questions (10%) require combining pre-computed information and on-demand information. We call this category "Functional" since they can be encoded using functional patterns — relational queries plus methods to create data on-demand. Of course, this categorization is dependent on one's interpretation of the question; demanding a more precise answer to a question may require dynamically generated information (e.g., running an analysis on a sub-query). The "Ill-defined" category consists of 9 questions (12%) that require either relational or functional composition, but that we could not categorize because the categorization depends on the interpretation of the question.

The analysis suggests developer information needs are well-handled by known patterns (relational and functional). Therefore, the primary barriers to meeting developer information needs are (1) making data and APIs readily accessible, and (2) providing interfaces for applying these patterns.

These are exactly the demands that the Cupid interface and computation engine are designed to meet.

Questions Developers Don't Ask In distinguishing between data and analysis, our availability-based model provides insight into how/where the developer-question based approach doesn't exhaustively enumerate information needs. Indeed, Fritz and Murphy ended up using data domain to distinguish between developer questions that sound similar, but had different intent [57]. For example, they identified that the question "Who is working on what?" refers to both source code and changeset information in addition to the team and work item information required to answer the nearly identical question "What have people been working on?" When a developer states a question, they may in fact be asking about a particular piece of data (and corresponding analysis). Therefore, it can be informative to start at the data (and analysis) and "back-out" the question.

For example, suppose a developer has test coverage information for each method in a project. She could aggregate the information by class and then find the element with the minimum coverage. Performing these two operations in sequence would correspond to the question "Which class has the least test coverage?" Note that this data-centric question does not include an intent or purpose. The higher-level question is "Which class is most in need of testing?" Test coverage was introduced as a metric for answering this question, not vice versa — developers did not wait for coverage to be introduced to start asking which classes needed more testing. The set of available tools and data do matter, though: a developer with access to production data might instead ask the question "Which class has had the highest defect rate the past month?" to determine which class is most in need of testing.

Indeed, exploring the space of data and analysis combinations can be informative to demonstrate alternative ways to answer (or better answer) existing questions. As Cupid's interface demonstrates, types provide a helpful mechanism for restricting the exploration to meaningful combinations.

5.4.2 Expressivity

Cupid's computation model, which is backed by capabilities and scripts, is as expressive as general programming. This chapter focuses on a pipeline interface (Section 5.3.3), which lends itself to rapid creation of analyses that follow a Select-Transform-Synthesize pattern that many developer questions require. However, it would be entirely possible to also implement relational [57], algebraic [109],

and graphical [77] interfaces/models on top of Cupid which may be more favorable for answering certain questions. In practice, we find that many questions don't involve crunching computationally intractable amounts of data, only too much data for a developer to calculate manually. Therefore, we speculate that Cupid's basic caching would allow the aforementioned models to be performant when implemented on top of Cupid.

Composition vs. View Prior work [57] distinguishes between composition and view. There is value in the distinction since composed data can be used in multiple views and views can often be effectively handled with existing UI patterns. However, it is not possible (as is done for information fragments in [57]) to claim that enforcing the distinction eliminates burden for the developer. Pushing relational operators (e.g., the count operator) into the view just results in a UI for forming relational queries. The developer must still understand which operations to apply.

Additionally, a full separation between composition and view implicitly assumes a standard mapping across data sources such that no data must be transformed. In practice, there is no such common mapping. Therefore, it is important for Cupid to support the ability to transform data — providing an intermediate view — in the composition interface.

5.4.3 Validation

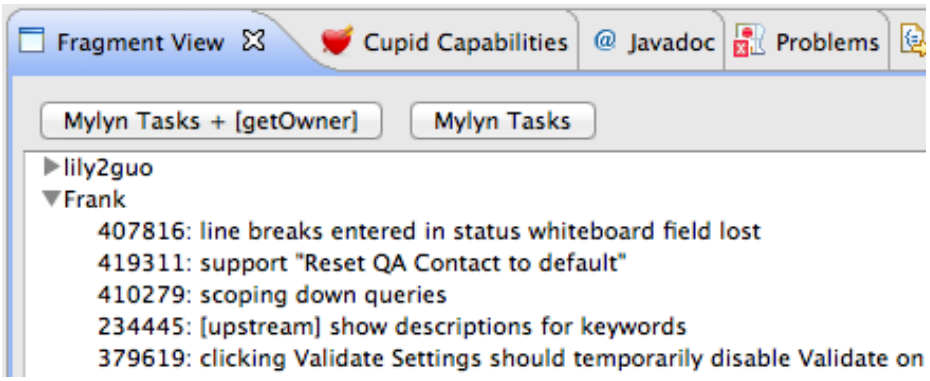
This section presents two case studies demonstrating Cupid's ability to enable rapid prototyping of novel user interfaces and analyses.

Hierarchical Grouping View

In [57], Fritz and Murphy present a hierarchical view to present composed (grouped) information to developers. A developer can switch the order of the hierarchical grouping by rearranging buttons in a toolbar corresponding to each information source.

Prototype Implementation We wrote a prototype implementing a hierarchical view for information the developer creates with Cupid's Mapping Wizard (Figure 5.8). The developer can switch between the two grouping orders (i.e., reversing the mapping) by rearranging the toolbar buttons.

Figure 5.8: Prototype of the hierarchical view from [57] displaying tasks that each team member is working on. The developer can view the team members working on each task by switching the order of the grouping buttons (via drag-and-drop).



Like the other Cupid views, the view is selection-driven, displaying the output of the mapping for any workbench item the developer selects.

Only 303 NCSL (non-comment, source lines) of Java were required to implement the hierarchy view prototype, much of which is view boilerplate and drag-and-drop code. In fact, implementing drag-and-drop to allow the developer to re-order the grouping was the most challenging part of building the prototype.

Discussion The prototype could have alternatively allowed the developer to select the component capabilities (information sources) as in [57], eliminating the extra step of explicitly defining the mapping with the Mapping Wizard. However since Cupid does not assume or pre-define any links between information, the interface would then require that the developer explicitly specify these links (as is done in the Mapping Wizard).

The prototype is limited in functionality in that it only supports one level of grouping. Supporting additional levels would involve chaining mappings the developer has created with the Mapping Wizard. Cupid's computational model would enable the output of the sub-mappings to be computed dynamically when the developer requests the information (e.g., when the developer expands the node for a grouping).

Continuous Testing

Continuous Testing, now a central part of Test-Driven Development (TDD), runs tests continuously in the background while a developer works [118]. Eclipse implementations generally mark failed tests as problem markers. Since tests are long running, implementations often include logic to prioritize the tests, such as "Most Recently Failed First". Implementations for Eclipse include InfiniTest [71] and CT-Eclipse [29]. Previous work found Continuous Testing to significantly benefit programmer performance [119, 117]; however, the initial continuous testing implementation required a significant implementation effort [118].

Prototype Implementation We wrote a prototype implementing a basic Continuous Testing feature for Eclipse. The prototype implements the core feature of continuous testing: running tests continuously and marking failed tests using warning markers. We did not attempt to replicate features that are analysis-focused, such as test prioritization. To implement continuous testing, we wrote three capabilities:

1. A "testing capability" that runs a test launch configuration and returns the corresponding JUnit test session.

2. A "failed test capability" that returns the set of failed tests from a JUnit test session.

3. A "test marker capability" that builds a marker from a JUnit test run.

We then wrote an additional capability that executes these three capabilities in a pipeline. Note that the constituent capabilities are general, and could be reused in other pipelines to integrate test information. Additionally, we created a standard Eclipse preference page for selecting which launch configurations to use for continuous testing.

Despite optimizing for readability, not size, only 502 NCSL (non-comment, source lines) of Java were required to implement continuous testing. The capabilities for running and logging test configurations were 186 NCSL of Java; the capability for creating markers from test configurations was 76 NCSL; the preference page required an additional 112 NSCL of Java.

Discussion The size of the prototype belies the effort required to build the prototype; I spent spent 7–10 hours discovering the correct JUnit APIs to call by reading the Eclipse source code. Implementing advanced continuous testing features, such as test prioritization, would require even deeper understanding of the JUnit API.

5.5 Learnability Study

This section describes a study exploring the question:

Research Question 5.5.1. *How difficult is Cupid's pipeline and view interface to learn?*

We ran a study in which 8 undergraduate students performed a series of time-constrained tasks using Cupid. Each task had the student answer a question about the subject project (Table 5.2). The results indicate that participants became statistically-significantly more successful at completing tasks with Cupid after just 4 tasks. As expected, there was variability across students; the instructions and feedback given during the study were insufficient to make some of the novice users successful.

5.5.1 Methodology

This section describes the experimental design; all study materials are available at `https://homes.cs.washington.edu/~tws/cupid`.

We had 8 undergraduate computer science students complete a Cupid tutorial and then measured their success rate answering the 8 developer questions described in Table 5.2. We fit a generalized linear mixed model to compare student performance on the first four tasks to the second four tasks. The model accounts for inter-subject variability.

Tutorial Each subject completed a tutorial covering the Cupid features necessary to perform the tasks. The tutorial included examples for the subjects to work through. The tutorial took participants 20–45 minutes to complete.

Tasks After completing a tutorial covering the main features in Cupid, each subject performed the 8 tasks described in Table 5.2. The subjects performed the tasks in a workspace containing 8

Table 5.2: Questions and task descriptions for the Learnability Study (Section 5.5). Questions #1–4 and #8 were taken from the developer questions identified in [57]. Task descriptions were provided in addition to the questions to eliminate variability due to the participants' interpretation of the questions. The time limit for each task was $5\frac{1}{2}$ minutes.

Task	**Question**	**Task Description**
1	Who should you talk to if you have to work with the specified package?	List the three most frequent committers (i.e., authors) to the package
2	Who is working on the same classes as I am and for which work item?	List the other tasks that share context with the work item (i.e., task).
3	Who owns this piece of code?	List the most frequent committer (i.e., author) to the file.
4	What's the most popular class? [Which class has been changed most?]	List the class in the package that has been modified most frequently.
5	Which files in the package were last modified by the specified developer?	List the files in the package last modified by the developer. That is, files where the most recent committer/author is the developer.
6	Which project(s) have the most open work items?	List the projects in the workspace that have the most open work items (i.e., non-completed tasks).
7	Which project(s) were modified recently?	List the projects for which a commit has been made within the last 30 days.
8	Which work item(s) are the most active?	List the task(s) with the most number of comments.

subprojects of the Mylyn project;[1] we selected Mylyn because it is open source with many tasks having associated context information. While completing the tasks, subjects were allowed to refer back to the printed tutorial and to the example web page. This design was chosen to mimic how developers learn and use tools in practice — searching for similar examples and then modifying them to accomplish their specific task. In practice, the task time limit (discussed below) precluded participants from using the examples. After each task we gave the subject a valid written solution for the task (even if they had succeeded, for consistency). We instructed the subject to follow the solution if they had failed or had completed the task using a different approach.

Time Limit For each task, the subject read the description of the task (asking for clarification as necessary) and then started a timer to record the time needed to set up Cupid to complete the task (e.g., creating a pipeline, opening a report, etc.). The maximum time was capped at $5\frac{1}{2}$ minutes, after which the subject timed-out for the task. We chose the time limit based on pilot studies. Our goal was for the overall success rate to be 50%. If the participants instead succeeded at nearly all the tasks (or failed at nearly all the tasks), the analysis may not have been able to detect differences in performance with just 8 subjects.

Task Order The task order for the subjects followed a Latin Squares assignment such that task position was balanced across subjects — each task was performed in each position once (e.g., Task 1 was the first task performed for a single subject, the second task performed for a single subject, etc.). Table 5.3 lists the task assignments.

Questionnaires Before beginning the study, each subject completed a pre-questionnaire about their Eclipse and plug-in usage. After completing the tasks, each subject completed a post-questionnaire about their experience using Cupid. The results of this questionnaire are provided in Section 5.5.2.

---

[1] http://www.eclipse.org/mylyn/

5.5.2 Results

Task Performance

Table 5.3 shows the results of the user study. To test whether participants performed significantly better after the first four tasks, we fit a generalized linear mixed model using maximum likelihood. The generalized linear mixed model accounts for inter-subject variability. At the $p < .01$ level, we are able to reject the null hypothesis that participants had the same success rate for the first and second set of 4 tasks they performed.

Familiarity with Eclipse ended up being more of a factor than we expected. For example, Subject 8, who only successfully completed a single task, reported not having "spent too much time coding in Eclipse". Similarly, despite being covered in the tutorial, some participants confused tasks (in the Mylyn task view) with resources (e.g., files, projects).

User Experience

On average, the participants were neutral in their self-assessment of Cupid's learnability: the mean response to "I found it easy to figure out how to use Cupid to perform tasks" was 2.9 on a 5-point Likert scale. The structure of the study — that time limits were chosen such that participants would time out on about half of the tasks — likely played a role in the subjects' perception. Subject 6, for instance, reported that the "the pipeline wizard will take some time to get used to, but it will give you a lot of freedom and power over the information you want to see. [...] without the time pressure, I would've thought it was easy / straightforward."

Participants were slightly more positive about the overall user experience provided by Cupid. The mean response to the question "I liked the user experience provided by Cupid" was 3.7 on on a 5-point Likert scale; the mean response to the question "I would use Cupid in my own development" was 3.6. Due to the subject population (undergraduate students), the response to the latter question seems reasonable, as students working on class projects do not rely heavily on version control and issue tracking.

API Exploration Surprisingly, just two subjects mentioned the absence of documentation for capability outputs (which are derived via reflection) as an area of improvement. We observed a

Table 5.3: Results for the tasks described in Table 5.2. Subjects were statistically significantly more successful at the second four tasks that they performed than the first four tasks (see Section 5.5.2).

	*N*th Task Performed								
	1	2	3	4	5	6	7	8	Success (%)
Subject 1	Task 2 ✓	Task 5 ✗	Task 3 ✓	Task 8 ✓	Task 4 ✗	Task 1 ✗	Task 7 ✓	Task 6 ✓	63%
Subject 2	Task 5 ✗	Task 8 ✓	Task 6 ✗	Task 3 ✓	Task 7 ✓	Task 4 ✓	Task 2 ✓	Task 1 ✓	75%
Subject 3	Task 1 ✗	Task 4 ✗	Task 2 ✗	Task 7 ✗	Task 3 ✓	Task 8 ✓	Task 6 ✓	Task 5 ✓	50%
Subject 4	Task 7 ✗	Task 2 ✗	Task 8 ✓	Task 5 ✗	Task 1 ✓	Task 6 ✗	Task 4 ✓	Task 3 ✓	50%
Subject 5	Task 8 ✗	Task 3 ✗	Task 1 ✓	Task 6 ✗	Task 2 ✗	Task 7 ✗	Task 5 ✓	Task 4 ✓	38%
Subject 6	Task 6 ✗	Task 1 ✓	Task 7 ✗	Task 4 ✓	Task 8 ✓	Task 5 ✗	Task 3 ✓	Task 2 ✗	50%
Subject 7	Task 3 ✗	Task 6 ✓	Task 4 ✗	Task 1 ✗	Task 5 ✓	Task 2 ✗	Task 8 ✓	Task 7 ✓	50%
Subject 8	Task 4 ✗	Task 7 ✗	Task 5 ✗	Task 2 ✗	Task 6 ✓	Task 3 ✗	Task 1 ✗	Task 8 ✗	13%
Success (%)	13%	38%	38%	38%	63%	38%	88%	75%	48%

few cases where a misleading field name stymied subjects. For example, one subject mistook the `getNumber` method of the `ITaskComment` type to be the number of comments. While Cupid's type filtering prevented participants from creating non-sensical capability combinations, there is still potential for documentation and other API information (e.g., snippets) to improve the developer's ability to explore unfamiliar data.

Context Cupid's selection-driven interface caused confusion for some participants. Subject 3 suggested indicating the current selection in the Cupid visualizations. (Multiple items can appear to be highlighted in the Eclipse workbench, though only the most recently selected item(s) comprise the current selection.) Subject 4 suggested clarifying that opening the Pipeline Creator from an item selected in the Eclipse Workbench would cause the dialog to only show capabilities compatible with the selected input. Cupid, as a tool that provides information based on context, ought to clearly communicate that context to developers, especially those new to Cupid.

Visualizations The study confirmed that visualizations are important to developers. Subject 6 reported "the visualization tools were especially helpful. I'm a more visual worker, so those really helped show the big picture." However, the subjects identified areas of improvement. Three subjects suggested including numbers in the Pie Chart display. Including numbers would give developers immediate access to the underlying data while also conveying the high-level information (the relative frequencies). Multiple participants wanted to be able to create a report or chart directly from the Selection Inspector (at the time, the participants had to create reports and charts from the Capability Bulletin Board). This desire helps validate Cupid's design goal of making information discovery and validation both simple and immediate.

5.5.3 Discussion

The success rate we observed was lower than the success rate Fritz and Murphy observed for their Information Fragments interface [57]. We expect that this can be attributed to three factors. First, we selected the task time-limit based on pilot testing to target a 50% success rate. Second, all of the participants in their study were professional developers, whereas the subjects in our study were computer science undergraduates, some of whom had limited experience with Eclipse, Mylyn,

and version control. Third, the Cupid interface is less constrained, making naive experimentation unsuccessful when time-constrained. If the Information Fragments interface were expanded to support additional information types and composition, we expect naive experimentation to become more costly for that interface as well.

Threats to Validity

Experimental Design We designed the experiment to mimic how developers learn to use tools in the wild. However, there are two primary ways in which the study departs from reality. First, the subjects performed a formal tutorial. When using a tool in the wild, some developers may not perform a formal tutorial, instead jumping straight into performing a task by referencing examples. Second, showing subjects a solution to the task immediately after task failure is not realistic — in reality the subject would either (1) continue working on the task until they found a solution, (2) abandon the task, or (3) ask for help (e.g., from a colleague, on Stack Overflow, or the mailing list) and then be given the solution after some delay. If Cupid gains adoption, however, the addition of public information and tutorials from a variety of sources would make the study design more realistic. Additionally, we expect developers would be less likely to abandon the tasks after a short period of time, due to social-proof (i.e., a developer would put more effort in learning to use Cupid as they've seen evidence that other developers have found Cupid useful).

Task Selection The results are, of course, sensitive to our selection of tasks. We chose the eight tasks to (1) include questions identified in [57] and (2) have some re-use of concepts. Our study omits some features of Cupid, such as scripting and conditional formatting.

5.6 Related Work

Surveying development information needs has been a popular topic [80, 57]; the research has also begun to hone in on the difference in information needs between various roles such as developers and managers [17]. Much less work has been done in creating general frameworks for addressing these information needs.

Mariani and Pastore frame the problem as end-user programming [96, 79]. However, the problem that Cupid is addressing is different in that the target "end-user" is a programmer (software engineer),

which means that the end-user is (1) is familiar with programming concepts such as types, and (2) can write code. The important similarity is that the developer wants to encode their domain-knowledge without the overhead of learning tool APIs.

Mash-Up Tools The work most similar to that presented in this paper is Mariani and Pastore's MASH tool, which supports a workflow model for combining tasks in Eclipse [96]. Special task-based plug-ins expose "tasks" which the developer can combine into a workflow. Additionally developers can record their actions within Eclipse to create new tasks. There are three fundamental differences between Cupid and MASH. First, Cupid can use data displayed in plug-in's UI without the plug-in developer having to expose tasks (while a developer recording in MASH could open a UI view, the data shown would not be directly available to subsequent tasks). Second, Cupid uses type information to automatically output contextually relevant information; a developer using Cupid can create a new capability in a few clicks, as opposed to having to create a new workflow file and invoke it. Third, Cupid visualizations integrate into and augment existing plug-in UI.

Mashup tools (both academic and commercial) have largely been applied to combining information on the web [60]. Grammel et al. recently proposed a web-based environment for creating mashups of project and team information [61]. Such a tool would be supported by approaches such as Ghezzi and Gall's recent *ontology-based approach* for exposing software analysis tools as remote services [58]. We don't know of any efforts to reconcile the push toward "analyses as a service" with "integrated development environments." Cupid, however, could be used as a basis for such an effort by providing bindings to web services (or automatically binding based on published interfaces).

Like the typed interfaces in our work, the ontologies that Ghezzi and Gall describe guide the use of services and inform which services can be composed. However, we believe that type systems, coupled with developer understanding, serve as a better (and often more precise) oracle for what information can by synthesized when working in the context of the IDE. Nevertheless, we plan to explore ontologies as a way of *recommending* capability combinations in future work.

Scripting In September 2013, members of the e4 Eclipse community incubation project revived an effort to build a unified scripting engine (SE) that supports "easy customization, and automation of common workflows" [44]. Previous Eclipse scripting projects include EScripts [49], EclipseMonkey

(part of the Dash Project) [46], and Groovy Monkey [62]. All of these projects are no longer actively developed.

The functional requirements gathered for the new project highlight three ways in which the goals and design choices of the Cupid and the scripting tools differ. First, the scripting tools are focused on calling Eclipse APIs to perform actions, not integrating data from multiple plug-ins. Second, the projects eschew Java, instead focusing instead on "true" scripting languages such as JavaScript, Python, and Groovy. Third, the scripting projects make no attempt to aid visualization aside from providing an interactive console.

Other Approaches to Information Integration Fritz and Murphy propose and evaluate using composable *information fragments* from different information domains to answer developer questions [57]. Our work is complementary, as Cupid provides a natural mechanism for creating and exposing new information fragments, and the composition operators can be created using pipelines.

Buse and Zimmerman frame developer information needs in terms of development metrics [17], and identify an *analytics matrix* of exploration, analysis, and experimentation, with respect to past, present, and future versions of the software. As described in this paper, Cupid provides an easy way to add these analytics to the IDE and developer's workflow for the current version. As support for past and future artifacts (à la [100]) is added, Cupid will provide a seamless framework for exploration and experimentation.

Chapter 6

RELATED WORK: THEMES IN TOOL DESIGN AND RESEARCH

This section describes related work in specification and verification following the general themes of tool feedback, developer understanding, tool interoperability, and IDE integration.

6.1 Tool Feedback and Developer Understanding

Tool understanding research takes place in the context of either educational or developer use. Educational research almost exclusively deals with novice programmers and therefore targets the compiler and other "basic" tools (e.g., the debugger). Evaluations include observational classroom studies and controlled experiments with students, and the community often builds special tools and then evaluates them. Developer research, on the other hand, isn't as formal — tools are built, but their effects on development are not directly studied. However, acceptance and feedback from the developer community can serve as a proxy.

There are three approaches taken to enhancing developer tool understanding: increasing awareness, improving explanations, or providing sensitivity analysis. This section presents work in these areas that targets developer use, though related educational work is addressed where appropriate.

Awareness Information

Tools support developer awareness by reporting the decisions they make and the actions they take. One approach to reporting decisions is to add logging. St-Amour et al. add logging to the Typed Racket optimizer and the Racket inliner to notify developers where optimizations were performed, or were nearly performed ("near-misses") [126]. The tool uses optimization-specific proximity metrics to determine when to show information. However, no mechanism is provided for standardizing the proximity metrics *across* optimizations. Therefore, it is not clear whether the developer will be made aware of the most pertinent information.

Alternatively, an external analysis can be performed to provide information about a tool. For

example, Kiniry et al. augment ESC/Java2's output with soundness and completeness warnings by performing an AST analysis during pre-processing [78]. While the approach avoids dependence on the underlying checking implementation and avoids adding complexity to the checker, care must be taken to keep external analyses in sync with the operation of the tool.

Explanation Information

Tools can (and should) additionally provide information about why a decision was made, or how an action was performed. However, making this information useful is difficult, even for well understood analyses such as type-checking. Much of the problem arises from the diversity of end users: lay-developers benefit from simple messages, while seasoned developers benefit from precise messages.

In practice, the demand for simplicity must win. This is especially true when developers are first deciding whether or not to use a tool or analysis [8], but also remains true at later stages of use. For example, Coverity, the preeminent commercial static checking tool, does not ship certain static checkers since the company has found that their output is too difficult to explain to lay-developers [8]. Additionally, developers reasonably expect results to be stable across runs. This precludes the use of non-determinism (randomness) to make algorithms more powerful, and provides a practical limit on how much analyses can change across versions, even if they are being improved [8].

The previously described optimization notification tool (Section 6.1) faces a similar explanation barrier due to the fact that compilers perform optimizations in a pipeline — later optimizations may be performed on code that the developer did not write. The tool's authors opt not to present information that can't be linked to the source code [126]. As a result, developers may miss valuable optimization opportunities; the authors don't characterize the effect of this design decision. Additionally, given that the major contribution of the work is a methodology for extracting pertinent information from optimizer logs, it's unfortunate that the work lacks a proper usability study to evaluate whether or not the information is indeed both apt and actionable.

From a learning perspective, Marceau et al. identified three major roadblocks to novice comprehension of compiler error messages though classroom observation and quizzes: (1) insufficient vocabulary, (2) misleading error location information, and (3) misleading and unsound sugges-

tions [95]. While the latter two concerns are clearly legitimate, the work does not address whether vocabulary is a significant issue when developers have instant access to searchable online resources. Without this, it's not clear whether their prescribed "simplified vocabulary" for educational tools would be more effective, especially since the students must eventually learn to use real tools.

An alternative approach to changing vocabulary and output format of the tool is modify the output of the tool post-hoc. STLFilt is one such tool that performs a text-to-text simplification of the standard type errors encountered when using the C++ Standard Template Library (STL) [146]. The tool is integrated into multiple development environments, enjoys a sizable developer base and has been publicized in books and online resources. The tool has two primary advantages: first, translation is performed on demand, so precision is not lost; second the translation performed is transparent and developer-customizable. As a text-to-text translation, however, the tool cannot take advantage of structural or semantic information calculated by the compiler.

For some developers, text explanations may not be sufficient, especially if the developers are unfamiliar with an analysis or technique. Binkley et al. introduced the "Feedback Compiler" which constructs and shows animations of a variety of compiler optimizations by adding passes over internal compiler data structures [11]. While the motivation as a teaching aide makes sense, it is not clear why developers would benefit from seeing how an optimization is performed, or would primarily benefit from knowing the optimization occurred. Given that their controlled experiment found that animations provided no significant benefit in helping undergraduates manually perform common sub-expression elimination grants credence to this doubt.

Sensitivity Information

Sensitivity analyses indicate what changes would affect a decision or action by either identifying likely mistakes or by mimicking analyses that a developer (or expert) would perform manually.

A natural method for discovering likely mistakes is finding nearby solutions that are correct. Lerner et al. use a search procedure to improve type-error messages for the Caml language by offering nearby expressions with compatible types [89]. Their evaluation finds that the generated messages perform better under a "good location" metric. However, as implied by Marceau et al.'s result previously described [95], the efficacy of location information is developer-dependent. Therefore,

having the authors (experienced Caml developers) determine location quality when evaluating the tool likely overstated the benefit of the approach, especially since the tool is targeted at newer developers who don't fully understand the type system. Another problem with type-based search tools is their efficacy of the work depends on types being relatively descriptive and precise. Ideally, there'd only be a single value of each type in a context, or parameter list, however in practice this isn't the case. Ordering heuristics (e.g., length of expression) therefore must be empirically evaluated.

Analysis need not be completely automatic. Ko and Myers introduce Whyline, a debugging tool which answers questions about "why" and "why not" questions about program behavior [81]. The tool is a question-based interface to static and dynamic program slicing, saving the developer the effort of manually reasoning about control and data flow. Evaluations suggest that the tool is effective for debugging programs with graphical user interfaces, it is less clear whether or not the tool is effective for debugging computation-centric software; additionally, the heavy-weight instrumentation may not scale to real software.

An interesting related subfield that relies on sensitivity is mutation testing. Mutation testing evaluates the quality of a test suite by its ability to detect small changes (mutations) to the program, providing a basis for augmenting existing tests [73]. In some ways, the ability to detect mutants is a proxy for the completeness of specification the test suite provides, however it is not known whether the ability to detect mutants translates into an ability to detect real program errors, or if its benefit solely comes from increased coverage.

6.2 Tool Synergies

All approaches to specification and verification have relative strengths and weaknesses; by combining approaches with complementary strengths, the weaknesses of the individual approaches can be overcome. Approaches combining over-approximations (for generality) and under-approximations (for precision) have been particularly successful. In practice, this combination is realized by coupling static and dynamic analyses[1], or similarly predicate (i.e., abstract) and explicit analyses.

At a high level, there are three ways to combine two analyses: one analysis can use another as subroutine, two analyses can be run in parallel, or two analyses can be chained together. Chains can

[1]Many runtime and debugging tools use static analyses to improve performance, e.g., by determining where information is not required. These runtime tools are beyond the scope of this work.

either be *naive*, simply passing the output of one independent tool to the next, or *explicit*, additionally passing meta-data to make the subsequent analyses more effective.

Analyses as Subroutines

The subroutine pattern is typically used to avoid making inference analyses sound: suggestions are created heuristically and then invalid suggestions are eliminated by repeatedly running a checker as a black box. The checking can be on a per-suggestion basis, as in Lerner et al.'s work which invokes Caml's type checker on expressions "near" another expression [89]. Similarly, VeriWeb (see Section 4) checks each newly suggested specification clause by invoking the ESC/Java2 checker on the baseline specification augmented with the new clause. Alternatively, checking can be used iteratively to prune a whole set of interrelated suggestions. Flanagan and Leino's Houdini tool heuristically generates candidate Java Modeling Language (JML) clauses from program text and iteratively runs ESC/Java2 to eliminate false clauses, resulting in a provably maximal verifiable subset [54]. The approach scales well enough that they were able to run the tool on thousands of lines of code; they found real bugs by manually inspecting sites where ESC/Java2 could not prove that the maximal verifiable subset was sufficient.

There are two limiting factors in subroutine based approaches: the recall of the initial suggestion sets, and the run-time of the black box analysis. The relative impact of these factors is domain dependent. For example, while Houdini is unable to generate disjunctions, this was not a dealbreaker for proving the absence of many possible runtime exceptions. Similarly, there are no major runtime requirements. For VeriWeb, an interactive tool, ESC/Java2's runtime affects the developer experience. The tool runs checks in parallel when possible, i.e., when checking post-conditions.

Parallel Composition

Parallel combinations of tools can be either heterogeneous or homogeneous. Heterogeneous combinations rely on structural differences for diversity; homogeneous combinations are made diverse by varying input parameters (e.g., the random seed).

Heterogeneous combinations typically also include communication between the analyses; parallel analyses need not be independent. Beyer et al. propose dynamically switching between parallel

analyses, in particular, explicit and predicate states [9]. However, their system only works one direction — tracking explicit values for an expression until it becomes too expensive, then tracking predicates for the expression. In some ways, the approach is the converse of CEGAR, which iteratively refines an abstract model based on concrete counter-examples [26], as it approaches explicit values for the degenerative case. Similarly, Gulavani et al.'s SYNERGY algorithm combines execution and theorem proving to simultaneously hunt for bugs and proofs of correctness [63]. Test information is used to guide predicate abstraction, and verification information (i.e. abstract error traces) is used to guide testing. The authors note that the approach performs better than running a model checker and directed test generator independently, primarily because it avoids the path explosion caused by *if-then-else* constructs that stifles directed testing.

The benefit of homogeneous combinations is that they are straight forward. Based on the observation that the run-time of explicit state model checkers is very sensitive to their starting state, Dweyer et al. run multiple instances of the same model-checker with each instance randomly starting at a different location [43]. The major strength of this approach is its generality, it can be applied to any search-based technique that uses explicit-state (e.g., explicit-state model checkers). Unfortunately, the vast majority of static analyses use abstract states; it would be interesting to combine this approach with Bayer et al.'s [9] — using explicit state to randomize the starting state, then proceeding with standard abstract state model checking.

Naive Chaining

Intuitively, the second tool in a naive chain can either refine the output of the first tool, or expand the first tool's output. The former results in analyses very similar to the subroutine pattern, in which a checking analysis vets suggestions from an inference analysis.

A canonical example of refinement is Nimmer and Ernst's work which infers contracts from dynamic traces using Daikon, and then uses ESC/Java to eliminate the spurious contracts [102]. For the toy subject programs studied, they found that ESC/Java was able to verify the majority of the inferred contracts. However, this result is not meaningful because they apparently did not account for the fact that ESC/Java's modular checking means that warnings can mask other warnings (i.e., a property can be implied by an unproven condition). The way to avoid the problem would have

been to iteratively run ESC/Java, as in the aforementioned Houdini work [54]. Additionally, the work does not address the effect that the quality of the inferred preconditions has on the specification as whole — dynamic analyses notoriously infer strong (non-general) preconditions, which would imply artificially strong postconditions.

Yorsh et al. take a similar approach, inferring predicates from concrete executions and checking them with a theorem prover, but then additionally continue the process by generating concrete states for failed properties to use as a starting point for additional execution [145]. The additional iteration is closely related to concolic testing tools which switch between abstract execution (interpretation) and concrete execution.

As a followup ot their previous work, Nimmer and Ernst evaluated providing inferred contracts from both Houdini and Daikon as an aide for developers writing verified specifications with ESC/-Java2 [104]. For the same subject programs as in [102], both tools had significant positive impact on the success rate (participants were only given one hour per program), however, the quantitative effects were indistinguishable. The problem with this type of work is that it approaches verification from the point of view of the contract inference tool: individual clauses. In reality, specifications are organized by cases, which are made up of clauses; it's not clear if existing tools would be an effective aid for writing this kind of structured specification. The same criticism applies to the verified specification work using ESC/Java2 described in Chapter 4.

Expansion or augmentation is common in test case generation, where analyses produce additional test to gain more coverage (with respect to lines of code, control flow, etc.). Tools in the automated test generation literature typically uses trivial oracles such as the absence of uncaught exceptions, as they do not require human input. The Pex test generation tool can use Code Contracts to guide unit test generation [3]. A benefit of this approach is that it can make use of partial formal specifications. However, it is unclear how to determine the effect of inconsistent conditions; there appears to be no indication of which contracts are being used.

Explicit Chaining

Explicit chaining improves on naive chaining by including metadata to support downstream analyses. Beyer et al. introduce conditional model checking, in which a non-terminating / non-conclusive

model checker reports the checked state predicate (or another condition) for consumption by a down-stream analysis, such as another model checker [10]. Preliminary experimental results indicate that sequential combinations of checkers can verify additional properties. While the work is a step in the right direction, there are three problems with approach: first, it requires that all of the model checkers use an abstract reachability tree (ART) model of state. Second, the choice of message format can disrupt optimizations by the individual checkers. Finally, their work doesn't directly address programs with interacting domains — their evaluation uses synthetic "hard" programs produced by concatenating benchmarks.

Independently, Christakas et al. propose a similar approach of explicitly tracking verifier assumptions that make the analysis unsound and then automatically generating complementary tests using concolic testing [25]. The major benefit of their approach over Beyer et al.'s work is that the assumptions are in terms of the code, not the internal representation of the static analysis. This enables broader combinations of tools, but may limits the expressivity of the communication between tools (e.g., it may not be possible to tractably serialize the state of a model checker explicity in terms of the code). A side benefit is that some of the verifier output may be human readable.

6.3 Effective IDE Integration

The work that most directly addresses intergrating specification and verification support into the IDE is Logozzo et al.'s Semantic Integrated Development Environment [94]. The IDE takes advantage of Code Contracts and abstract interpretation to augment refactoring, and additionally identifies and suggests fixes for semantic errors.

In general, there are three interelated themes in IDE integration research: problem awareness, information retrieval, and information composition (the focus of Chapter 5). Awareness research aims to alert the developer to (potential) problems; information retrieval research aims to effectively expose apt information either automatically or on demand; information composition research research aims to enable developers to synthesize and understand information from multiple sources. A shared goal of all three themes is reducing or eliminating developer context switching, which impairs productivity.

Problem Awareness

Real-time feedback about compilation warnings and errors is now standard. However, it is not always clear what additional analyses should be computed, and when and how their warnings should be displayed.

Inexpensive analyses can be performed in real-time. The Check Style plug-in for Eclipse, for example, flags a variety of problematic style violations such as empty blocks [22]. Other analyses, such as continuous testing [118], can be long-running and demand significant resources. Long running analyses can be optimized to provide valuable information faster (e.g., test prioritization as in [118]), however, there is no work on frameworks for allocating resources across multiple analyses.

Even when analyses can be continuously computed, the IDE may not be the best place for an alert. For instance, Brun et al.'s Crystal runs from the system tray and proactively alerts developers of merge conflicts with other developer's local repositories [14]. Running from the system tray provides the benefit of being developer tool and project agnostic; the developer can be alerted when performing another task such as email. However, lack of IDE integration means that the developer does not have a natural environment for examining and understanding the consequence of the alerts that Crystal provides. Crystal, and similar analyses, could benefit being exposed both as a stand-alone tool, and as an IDE plug-in.

Reporting information on *potential* problems raises even more questions, but can be a valuable way to prevent wasted effort. Holmes and Walker propose a general approach to providing developer-specific alerts called YooHoo [69]. Their approach is not specifically targetted at problem awareness, but is also targetted at change awareness on large teams. Providing apt information about changes is more difficult than problems because, unlike problems, the developer may not consider a change to be relevant. They propose using domain-knowledge and heuristics to determine relevance and severity (e.g., did only white space change?), and passive notifications to minimize the adverse effects of providing too much information.

Information Retrieval

Čubranić and Murphy built Hipikat, an Eclipse plug-in which presents developers with relevant artifacts when performing a task [140]. Interestingly, tasks are defined in terms of Bugzilla tasks,

rather than developer activities (debugging, etc.). The benefit of this is that the tool doesn't have to infer developer intent, however it makes the pertinence of results heavily dependent on the quality of Bugzilla tasks. In their study, they found that while developers can filter non-pertinent information, retrieving too much information hurt productivity. There are countless other recommender systems, I omit their discussion here since they all share the same basic approach and problems. An important finding from recommender system research to highlight, however, is that providing explanations with recommendations increases developer acceptance [66].

The core resource recommender system literature ignores developers and managers as sources of information. Ye et al. propose a socio-technical approach based on building an information resource graph that includes both software development artifacts and developers [144]. When performing a task, the developer traverses the graph as necessary to find information. If development artifacts are insufficient, the system allows the developer to ask an anonymized group of relevant contacts. Since the group members are anonymous, each individual developer can opt to not respond without stigma; the developer that responds is identified, allowing them to receive credit.

While Ye et al.'s basic approach is reasonable, the tool goes further by also trying to optimize social interaction by keeping score of what each developer "owes" to each other programmer and the group as a whole. A less formal approach is taken by Bagel et al.'s Hoozizat tool that aggregates social, organizational, and artifact information to help a developer decide who to contact with a question [6]. The information is not anonymous — in fact pictures are included – enabling developers to easily identify information sources they may know personally. This more organic approach is likely more effective when teams are located near each other and personal networks already exist.

Chapter 7

FUTURE WORK AND CONCLUSION

This dissertation characterized how a lack of tool transparency and interoperability undermine the usability of formal specification tools. While empirical studies played a role in the research, the research is predominately tool- and feature-centric. Future work, especially in the near-term, should aim to create a developer-centric perspective on formal specification.

What kinds of properties do developers (want to) specify? The most widely used approaches to specification are documentation, tests, and types. Based on their wide-spread use, future work should characterize the kinds of properties that developers are using each of each of these approaches to specify. Future work should additionally characterize the extent to which contemporary formal specification tools support developers in specifying and verifying these properties. Investigating these questions will provide insight into which formal specification approaches inherently hold the most promise, and how existing tools can be improved.

To begin to explore these questions, I designed and implemented a tool for tagging specifications appearing in documentation (e.g., Javadoc). The goal was to use the tool to construct a large corpus of tagged documentation which could provide insight into written specifications. The tool allows the annotator to tag phrases in documentation that act as specifications (specification phrases). For each phrase, the annotator can tag the phrase with the kind of property expressed (e.g., type state), the kind of location in the documentation (e.g., parameter documentation), and the phrase's relation to other specificational phrases (e.g., implies, mutually exclusive, etc.).

The model of documentation as a collection of specification phrases proved to be inadequate. The documentation I tried tagging (e.g., the Java standard library) did not match typical formal specifications in two primary ways. First, many of the properties were expressed in application-specific terms precluding the use of a generic set of tags. Second, much of the documentation was based on examples rather than descriptions. These two characteristics likely improve readability, but

make identifying certain aspects of the specification difficult.

Future work should take a different approach to address the research question. One potential approach is to translate the natural-language documentation using formal specifications and test cases, identifying where translation is not possible or difficult. Another approach is to write contracts based on the oracles (assertions) in a test suite, identifying where there is no corresponding contract.

How do developers view specification and verification tools? Similar to Johnson et al.'s work on static analysis tools [74], future work should survey and interview professional developers to clarify developer view points toward formal specifications and tools. Clarifying developer view points will help distinguish perceived problems from real problems.

The best opportunities to survey and interview developers using contract-based specifications lie with Eiffel and C#. Eiffel has a culture of design and specification by contract. Eiffel developers would likely show an inclination toward contracts, however it would be informative to characterize how each developer's view of contracts has changed with continued use. C#, on the other hand, has no corresponding culture. Therefore Code Contracts (the C# implementation of contracts) has to compete against static analysis and testing tools on merit. In addition to identifying new areas of improvement, surveying and interviewing developers would help in prioritizing known problems (e.g., reducing build time vs. improving the static checker).

The introduction of type annotations in Java 8 provides an opportunity to observe how developers adopt formal specification tools. While the Java Modeling Language suffered poor tool support because it extended the Java syntax, type-qualifier based approaches (cf. Chapter 2) now have syntactic support. Future work should characterize the adoption of type qualifiers (similar to Parnin et al.'s study of Java generics in [108]). Additionally, as type qualifiers enable developers to implement their own application-specific analyses, future work should characterize how teams decide when to introduce custom type qualifiers and verification rules.

How should desktop-based IDEs integrate with the web? To address the more general problems of tool transparency and interoperability, future work should aim to reconcile the desktop-based IDE and the web. The response of some projects, such as Light Table [82] is to move the IDE to the web. Other projects, such as Eclipse Flux [132], instead aim to re-architect desktop-based IDEs to better

integrate with web services. The approach described in Chapter 5 could also serve an an integration point with web APIs.

Integration of the IDE and the web provides new opportunities to augment the development lifecycle. For example, access to elastic computing resources makes some analyses (e.g., static verification) more practical to run frequently, even continuously. However, increased access to data also brings new challenges. First, services must be organized in order to be discoverable (e.g., using ontologies as in [58]). Second, services must provide transparency into data provenance and quality so that developers can make decisions with confidence. Future work should explore how to best balance the need for both structure and flexibility.

Conclusion In practice, machine-verifiable specifications are more difficult to write than natural-language specifications. Some of the difficulty is essential and is therefore inevitable. However, as this dissertation demonstrates, much of the difficulty comes from a lack of tool transparency and interoperability. Therefore, there is still ample opportunity to make specification and verification tools easier to use.

My vision is a world in which software verification is more feasible, and therefore is performed more often. The path toward this is making formal specification accessible to a broad range of developers, even those with relatively little specialized training. This dissertation is one small step along this path.

BIBLIOGRAPHY

[1] Amazon Mechanical Turk, May 2012. https://www.mturk.com/.

[2] Henry Baker. Unify and conquer (garbage, updating, aliasing, ...). In *LFP*, pages 218–226, June 1990.

[3] 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.

[4] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs 002, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[5] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS*, pages 49–69, Mar. 2004.

[6] Andrew Begel, Khoo Yit Phang, and Thomas Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *Proceedings of the 32th International Conference on Software Engineering*, May 2010.

[7] Michael S. Bernstein, Greg Little, Robert C. Miller, Björn Hartmann, Mark S. Ackerman, David R. Karger, David Crowell, and Katrina Panovich. Soylent: a word processor with a crowd inside. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 313–322, New York, NY, USA, 2010. ACM.

[8] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, Feb. 2010.

[9] D. Beyer, T. A. Henzinger, and G. Theoduloz. Program analysis with dynamic precision adjustment. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 29–38, Washington, DC, USA, 2008. IEEE Computer Society.

[10] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional model checking: A technique to pass information between verifiers. In *Foundations of Software Engineering*, 2012.

[11] David Binkley, Bruce Duncan, Brennan Jubb, and April Wielgosz. The feedback compiler. In *In IEEE Sixth International Workshop on Program Comprehension*, 1998.

[12] Barry Boehm. Software risk management. In C. Ghezzi and J.A. McDermid, editors, *ESEC '89*, volume 387 of *Lecture Notes in Computer Science*, pages 1–19. Springer Berlin Heidelberg, 1989.

[13] Frederick P. Brooks, Jr. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[14] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 168–178, New York, NY, USA, 2011. ACM.

[15] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Early detection of collaboration conflicts and risks. *IEEE TSE*, 39(10):1358–1375, Oct. 2013.

[16] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, June 2005.

[17] Raymond P. L. Buse and Thomas Zimmermann. Information needs for software development analytics. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 987–996, Piscataway, NJ, USA, 2012. IEEE Press.

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

[19] Patrice Chalin and Perry R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP*, pages 227–247, Aug. 2007.

[20] Patrice Chalin and Frédéric Rioux. Non-null references by default in the Java Modeling Language. In *SAVCBS*, Sep. 2005.

[21] Patrice Chalin and Frdric 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.

[22] The Checkstyle plug-in for Eclipse, may 2014. `http://eclipse-cs.sourceforge.net/`.

[23] Yoonsik Cheon and Gary 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.

[24] Brian V. Chess. Improving computer security using Extended Static Checking. In *IEEE Symposium on Security and Privacy*, pages 160–173, Berkeley, California, May 12–15, 2002.

[25] Maria Christakis, Peter Müller, and Valentin Wüstholz. Collaborative verification and testing with explicit assumptions. In *Proceedings of the 18th International Symposium on Formal Methods (FM)*, 2012.

[26] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 154–169, London, UK, UK, 2000. Springer-Verlag.

[27] Code Contracts user manual. `http://download.microsoft.com/download/C/2/7/C2715F76-F56C-4D37-9231-EF8076B7EC13/userdoc.pdf`, Sep. 2010.

[28] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, CASSIS 2004, Revised Selected Papers*, volume 3362 of *LNCS*, pages 108–128, Marseille, France, Mar. 10–13, 2004.

[29] CT-Eclipse, 2012. `http://ct-eclipse.tigris.org/`.

[30] Contracts for Java, May 2014. `https://code.google.com/p/cofoja/`.

[31] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[32] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *Verification, Model Checking, and Abstract Interpretation*, pages 128–148. Springer, 2013.

[33] Patrick M. Cousot, Radhia Cousot, Francesco Logozzo, and Michael 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.

[34] Christoph Csallner and Yannis Smaragdakis. Dynamically discovering likely interface invariants. In *ICSE*, pages 861–864, May 2006. Emerging results track.

[35] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *ICSE*, pages 281–290, May 2008.

[36] Peng Dai, Mausam, and Daniel S. Weld. Decision-theoretic control of crowd-sourced workflows. In *AAAI Conference on Artificial Intelligence*, pages 1168–1174, 2010.

[37] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, Apr. 2008.

[38] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA*, pages 233–243, July 2006.

[39] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, Palo Alto, CA, July 23, 2003.

[40] David L. Detlefs. An overview of the Extended Static Checking system. In *Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, Jan. 1996.

[41] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, Dec. 18, 1998.

[42] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. Building and using pluggable type-checkers. In *ICSE*, pages 681–690, May 2011.

[43] Matthew B. Dwyer, Sebastian Elbaum, Suzette Person, and Rahul Purandare. Parallel randomized state-space search. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[44] E4/scripting, September 2013. `http://wiki.eclipse.org/E4/Scripting`.

[45] Eclipse jobs api documentation, 2012. `http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/core/runtime/jobs/Job.html`.

[46] Eclipse monkey, September 2013. `http://wiki.eclipse.org/Eclipse_Monkey/Overview`.

[47] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, Feb. 2001.

[48] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.

[49] Escripts, September 2013. `http://marketplace.eclipse.org/content/escripts`.

[50] H.-Christian Estler, Carlo A. Furia, Martin Nordio, Marco Piccioni, and Bertrand Meyer. Contracts in practice. In *Proceedings of the 19th International Symposium on Formal Methods (FM)*, Lecture Notes in Computer Science. Springer, May 2014.

[51] Manuel Fahndrich, Michael Barnett, Daan Leijen, and Francesco 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.

[52] Manuel Fähndrich and Francesco 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.

[53] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods Europe*, pages 500–517, Mar. 2001.

[54] Cormac Flanagan and K. Rustan 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*, FME '01, pages 500–517, London, UK, UK, 2001. Springer-Verlag.

[55] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, pages 234–245, June 2002.

[56] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM.

[57] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 175–184, 2010.

[58] Giacomo Ghezzi and Harald C. Gall. SOFAS: A lightweight architecture for software analysis as a service. In *Proceedings of the 2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, WICSA '11, pages 93–102, 2011.

[59] Guava: Google core libraries for Java 1.6+, 2012. `https://code.google.com/p/guava-libraries/`.

[60] Lars Grammel and Margaret-Anne Storey. An end user perspective on mashup makers. *University of Victoria Technical Report DCS-324-IR*, 2008.

[61] Lars Grammel, Christoph Treude, and Margaret-Anne Storey. Mashup environments in software engineering. In *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*, Web2SE '10, pages 24–25, 2010.

[62] Groovy monkey, Sep 2013. http://groovy.codehaus.org/Groovy+Monkey.

[63] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. Synergy: a new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 117–127, New York, NY, USA, 2006. ACM.

[64] Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic inference of abstract types. In *ISSTA*, pages 255–265, July 2006.

[65] Philip Jia Guo. A scalable mixed-level approach to dynamic analysis of C and C++ programs. Master's thesis, MIT Dept. of EECS, May 5, 2006.

[66] Jonathan L. Herlocker, Joseph A. Konstan, and John Riedl. Explaining collaborative filtering recommendations. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, CSCW '00, pages 241–250, New York, NY, USA, 2000. ACM.

[67] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Corrigenda: "Laws of programming". *Communications of the ACM*, 30(9):771, Sep. 1987. See [68].

[68] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, Aug. 1987. See corrigendum [67].

[69] Reid Holmes and Robert J. Walker. Promoting developer-specific awareness. In *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*, CHASE '08, pages 61–64, New York, NY, USA, 2008. ACM.

[70] John Joseph Horton and Lydia B. Chilton. The labor economics of paid crowdsourcing. In *Proceedings of the 11th ACM Conference on Electronic Commerce*, EC '10, pages 209–218, 2010.

[71] InfiniTest, 2012. http://infinitest.github.com/.

[72] JFreeChart, 2012. http://www.jfree.org/jfreechart/.

[73] Yue Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649 –678, sept.-oct. 2011.

[74] B. Johnson, Yoonki Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681, 2013.

[75] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE*, pages 672–681, May 2013.

[76] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *ICSM*, pages 736–743, Nov. 2001.

[77] Christoph Kiefer, Abraham Bernstein, and Jonas Tappolet. Mining software repositories with isparql and a software evolution ontology. In *Proceedings of the 29th International Conference on Software Engineering Workshops*, ICSEW '07, pages 10–, Washington, DC, USA, 2007. IEEE Computer Society.

[78] Joseph R. Kiniry, Alan E. Morkan, and Barry Denby. Soundness and completeness warnings in ESC/Java2. In *Proceedings of the Fifth International Workshop on Specification and Verification of Component Based Systems (SAVCBS)*, 2006.

[79] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. The state of the art in end-user software engineering. *ACM Comput. Surv.*, 43(3):21:1–21:44, Apr. 2011.

[80] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.

[81] Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

[82] Inc. Kodowa. Light table: the next generation code editor, May 2014. `http://www.lighttable.com/`.

[83] A. Kuhn. Ides need become open data platforms (as need languages and vms). In *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, pages 31 –36, June 2012.

[84] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM Softw. Eng. Notes*, 31(3), Mar. 2006.

[85] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, Mar. 2005.

[86] K. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer Berlin / Heidelberg, 2010.

[87] K. Rustan M. Leino and Greg Nelson. An extended static checker for Modula-3. In *Compiler Construction '98*, pages 302–305, Apr. 1998.

[88] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. Contract Driven Development = Test Driven Development − writing test cases. In *ESEC/FSE*, pages 425–434, Sep. 2007.

[89] Benjamin Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2007. ACM Press.

[90] Benjamin S. Lerner, Herman Venter, and Dan Grossman. Supporting dynamic, third-party code customizations in JavaScript using aspects. *SIGPLAN Not.*, 45:361–376, October 2010.

[91] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. Turkit: tools for iterative tasks on mechanical turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '09. ACM, 2009.

[92] Francesco Logozzo and Thomas 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.

[93] Francesco Logozzo, Michael Barnett, Manuel A. Fähndrich, Patrick Cousot, and Radhia 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.

[94] Francesco Logozzo, Patrick Cousot, Radhia Cousot, Manuel Fahndrich, , and Mike Barnett. A semantic integrated development environment. In *Proceedings of the to the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2012, 2012.

[95] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Mind your language: on novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, ONWARD '11, pages 3–18, New York, NY, USA, 2011. ACM.

[96] Leonardo Mariani and Fabrizio Pastore. Mash: a tool for end-user plug-in composition. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1387–1390, Piscataway, NJ, USA, 2012. IEEE Press.

[97] Winter Mason and Duncan J. Watts. Financial incentives and the "performance of crowds". In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '09, pages 77–85, New York, NY, USA, 2009. ACM.

[98] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, Oct. 1992.

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

[100] Kıvanç Muşlu, Yuriy Brun, Michael D. Ernst, and David Notkin. Making offline analyses continuous. In *ESEC/FSE*, pages 323–333, Aug. 2013.

[101] Greg Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, Palo Alto, CA, 1980. Also published as Xerox Palo Alto Research Center Research Report CSL-81-10.

[102] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *RV*, Paris, France, July 23, 2001.

[103] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA*, pages 232–242, July 2002.

[104] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *FSE*, pages 11–20, Nov. 2002.

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

[106] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *ICSE*, pages 338–348, May 1997.

[107] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, July 2008.

[108] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Java generics adoption: How new features are introduced, championed, or ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 3–12, New York, NY, USA, 2011. ACM.

[109] Santanu Paul and Ataul Prakash. A query algebra for program databases. *IEEE Trans. Softw. Eng.*, 22(3):202–217, Mar. 1996.

[110] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *SOSP*, pages 87–102, Oct. 2009.

[111] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *ISSTA*, pages 93–104, July 2009.

[112] Nadia Polikarpova, Carlo A. Furia, Yu Pei, Yi Wei, and Bertrand 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.

[113] Purchasing power parities for GDP and related indicators. `http://stats.oecd.org/Index.aspx?DataSetCode=PPPGDP`, Apr. 2012.

[114] Racket contracts and boundaries, May 2014. `http://docs.racket-lang.org/guide/contract-boundaries.html`.

[115] 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.

[116] Robert Andrew Rudd. An improved scalable mixed-level approach to dynamic analysis of c and c++ programs. Master's thesis, MIT Dept. of EECS, Jan. 2010.

[117] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *ISSRE*, pages 281–292, Nov. 2003.

[118] David Saff and Michael D. Ernst. Continuous testing in Eclipse. In *2nd Eclipse Technology Exchange Workshop (eTX)*, Barcelona, Spain, Mar. 2004.

[119] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA*, pages 76–85, July 2004.

[120] Todd W. Schiller and Michael D. Ernst. Rethinking the economics of software engineering. In *FoSER*, pages 325–330, Nov. 2010.

[121] Todd W. Schiller and Michael D. Ernst. Reducing the barriers to writing verified specifications. In *OOPSLA*, pages 9–112, Oct. 2012.

[122] Todd W. Schiller and Brandon Lucia. Playing Cupid: the IDE as a matchmaker for plug-ins. In *Tools as Plug-Ins*, 2012.

[123] Tim Sheard, Aaron Stump, and Stephanie Weirich. Language-based verification will change the world. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 343–348, New York, NY, USA, 2010. ACM.

[124] Simulink, March 2014. `http://www.mathworks.com/products/simulink/`.

[125] TIOBE Software. TIOBE programming community index for August 2013, Sep. 2013. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[126] Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching. In *OOPSLA 2012*, 2012.

[127] Matt Staats, Shin Hong, Moonzoo Kim, and Gregg 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.

[128] Kathryn T. Stolee and Sebastian Elbaum. Exploring the use of crowdsourcing to support empirical studies in software engineering. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 35:1–35:4, New York, NY, USA, 2010. ACM.

[129] R.E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *Software Engineering, IEEE Transactions on*, SE-12(1):157–171, Jan 1986.

[130] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, June 1994.

[131] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. May 2002.

[132] Eclipse Flux Team. Flux, May 2014. https://projects.eclipse.org/proposals/flight.

[133] Nikolai Tillmann and Jonathan 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.

[134] Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Pex for fun: Engineering an automated testing tool for serious games in computer science. Technical Report MSR-TR-2011-41, Microsoft Research, Redmond, WA, March 2011.

[135] Nikolai Tillmann and Wolfram 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.

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

[137] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, Oct. 2005.

[138] U.S. Bureau of Labor Statistics. Computer software engineers, applications, May 2010. `http://www.bls.gov/oes/current/oes151031.htm`.

[139] uTest: Software testing, Apr. 2012. `http://www.utest.com`.

[140] Davor Čubranić and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society.

[141] vWorker.com: More capable, accountable and affordable. guaranteed., Apr. 2012. `http://www.vworker.com`.

[142] Yi Wei, Carlo A Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 191–200. ACM, 2011.

[143] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison Wesley Longman, 1999.

[144] Yunwen Ye, Yasuhiro Yamamoto, and Kumiyo Nakakoji. A socio-technical framework for supporting programmers. In *ESEC/FSE*, pages 351–360, Sep. 2007.

[145] Greta Yorsh, Thomas Ball, and Mooly Sagiv. Testing, abstraction, theorem proving: better together! In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 145–156, New York, NY, USA, 2006. ACM.

[146] Leor Zolman. STLFilt: An STL error message decryptor for C++. `http://www.bdsoft.com/tools/stlfilt.html`.

[147] Edgardo Zoppi, Víctor Braberman, Guido de Caso, Diego Garbervetsky, and Sebastián 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.

Appendix: Specification Edit Distance Adjustment

This Appendix describes the distance metric used in Section 4.4. The edit distance metric assumes that each contract is either present or not present. However, VeriWeb introduces a third "state" for a contract, because VeriWeb remembers contracts that are not currently in use but may be tried or presented to the developer in the future. This state is not relevant to the Eclipse computation.

We adapt the edit distance metric as shown in Table 1. Any developer-written clause that is not in the target set counts toward the edit distance, as does any clause that is in the target set but not in the candidate set either as a developer-written or VeriWeb-suggested clause. In general, a VeriWeb-suggested clause is treated like a developer-written one. The exception is that a correct postcondition that has not yet been proved by VeriWeb is not counted against the developer. Such a clause typically exists only because the developer has not proceeded to the appropriate postcondition subproblem; when the developer does, the postcondition will be automatically added by VeriWeb without any human intervention.

A clause can be proven but "wrong" either because the proof depends on other wrong clauses, or because the target specification does not include that clause (another verifiable specification, however, might include the clause).

Sensitivity to Inferred Invariants The distance metric is sensitive to the inferred set of invariants. Let I be the inferred specification with preconditions I_R and postconditions I_E. Let X be the nearest verifiable specification with preconditions X_R and postconditions X_E. The distance for the developer using Eclipse Eclipse is

$$|X \setminus I| + |I \setminus X|$$

, the number of conditions that were not inferred by Daikon plus the number of conditions that were incorrectly inferred by Daikon. For a possibly different nearest verifiable solution X, the distance for the developer using VeriWeb is

$$|X_R| + |X_E \setminus I_E|$$

, the number of preconditions in the solution plus the number of postconditions that inference failed to detect. The difference in distance is given by:

$$
\begin{aligned}
&(|X_R \setminus I_R| + |X_E \setminus I_E| + |I \setminus X|) - (|X_R| + |X_E \setminus I_E|) \\
&= (|X_R \setminus I_R| + |I_R \setminus X_R| + |I_E \setminus X_E|) - |X_R| \\
&= (|I \setminus X| + |I_R \setminus X_R|) - (|I_R \setminus X_R| + |I_R \cap X_R|) \\
&= |I \setminus X| - |I_R \cap X_R|
\end{aligned}
$$

, the number of incorrectly inferred conditions less the number of correctly inferred preconditions.

Table 1: Which clauses count toward the edit distance metric, when comparing a candidate specification (set of clauses) with a target specification. When considering a contract clause in the candidate set, "WRONG" means to count any clause that is not in the goal set, and "RIGHT" means to count any clause that is in the goal set.
The Eclipse column indicates that the edit distance for a precondition is the sum of the number of wrong developer-written clauses, plus the number of missing (but necessary, or "RIGHT") clauses. The VeriWeb column indicates that the edit distance is the sum of the number of developer-written and developer-selected wrong clauses, plus the number of non-developer-selected (but necessary) clauses, plus the number of missing (but necessary) clauses.

Invariant source	Eclipse	VeriWeb
Object invariants		
Developer-written	WRONG	n/a
Generalized	n/a	WRONG
Missing	RIGHT	RIGHT
Preconditions		
Developer-written	WRONG	WRONG
Developer-selected	n/a	WRONG
Not developer-selected	n/a	RIGHT
Missing	RIGHT	RIGHT
Postconditions		
Developer-written	WRONG	WRONG
Proven	n/a	WRONG
Unproven	n/a	(none)
Missing	RIGHT	RIGHT