A comparative study of programmer-written and automatically inferred contracts

Nadia Polikarpova, Ilinca Ciupa, Bertrand Meyer
Chair of Software Engineering, ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

ABSTRACT

Where do contracts — specification elements embedded in executable code — come from? To produce them, should we rely on the programmers, on automatic tools, or some combination?

Recent work, in particular the Daikon system, has shown that it is possible to infer some contracts automatically from program executions. The main incentive has been an assumption that most programmers are reluctant to invent the contracts themselves. The experience of contract-supporting languages, notably Eiffel, disproves that assumption: programmers will include contracts if given the right tools. That experience also shows, however, that the resulting contracts are generally partial and occasionally incorrect.

Contract inference tools provide the opportunity for studying objectively the quality of programmer-written contracts, and for assessing the respective roles of humans and tools. Working on 25 classes taken from different sources such as widely-used standard libraries and code written by students, we applied Daikon to infer contracts and compared the results (totaling more than 19500 inferred assertion clauses) with the already present contracts.

We found that a contract inference tool can be used to strengthen programmer-written contracts, but cannot infer all contracts that humans write. The tool generates around five times as many relevant contract elements (assertion clauses) as written by programmers; but it only finds around 60% of those originally written by programmers. Around a third of the generated assertions clauses are either incorrect or irrelevant. The study also uncovered interesting correlations between the quality of inferred contracts and some code metrics.

1. INTRODUCTION

Embedding contracts (executable specification elements) in software texts yields a number of benefits [17]: contracts provide a basis for program verification techniques; they are essential for automated testing strategies by helping to filter out invalid inputs and acting as automated oracles; they support debugging by providing information about the locations of faults; they serve as documentation aid; they enhance the analysis and design process. These diverse applications make contracts an invaluable tool in support of software quality.

In spite of wide recognition of contracts' benefits, only a very small part of existing code is contracted. Part of the reason is notational: as the vast majority of programming languages offers no built-in support for contracts, programmers have to resort to mechanisms such as `asserts` to include conditions for run-time checking. This is sufficiently awkward and partial (with, for example, the difficulty of supporting the notion of class invariant, and the inheritance of contracts) to cause reluctance on the programmers' part.

The situation is different in languages with support for Design by Contract (DbC), such as Eiffel [18], JML [16] and Spec# [4]. These languages and their associated environments (IDE) provide a variety of supporting mechanisms: in the language, the ability to equip routines (methods) with preconditions and postconditions and classes with invariants, with associated semantics and well-defined inheritance rules; in the compiler, options for enabling and disabling the checking of contracts or their individual elements at run time; in the IDE, documentation views of a class interface including its contracts.

Such support (possibly with other factors) makes a big difference in developers' willingness to write contracts, as indicated by an extensive study [6], which shows that Eiffel classes contain a higher proportion of assertion clauses[1] than classes written in languages not supporting DbC and that 97% of assertion clauses present in Eiffel code are located in contracts, rather than in inline checks. This suffices to disprove the commonly held view that "programmers won't make the effort to write contracts", suggesting instead that language and environment issues are what restrains programmers.

This is not, however, the full story regarding programmer-written contracts, since closer examination (for languages with direct support) shows that such contracts are often incomplete, and sometimes incorrect in the sense of contradicting the implementation or the informal intent [7]. Contract quality is clearly a prime concern for all the applications of contracts mentioned above.

To address the difficulty of getting contract-equipped programs in languages not supporting DbC, researchers have investigated ways of automatically generating assertions. A notable outcome of this research is the notion of contract detector, as illustrated in particular by the Daikon tool [10], which produces likely assertions by observing properties that hold during executions of the system. The approach faces some objections of principle: the results depend on the particular input values exercised during execution and

[1]A contract element (also called *assertion*) such as a precondition, postcondition, class invariant or loop invariant consists of a number of clauses, combined using logical conjunction; the term *assertion clause* denotes such a clause.

there is a risk of documenting behavior of the software as it is, bugs included, rather than intent, which can only come from an explicit specification. In practice, however, Daikon has proved effective at inferring interesting contracts [10, 19, 20].

As a result of this effort the community now has at its disposal two bodies of meaningful contracts: those which programmers have written in two decades of Eiffel programming and several years of usage of such languages as JML and Spec#; and those which Daikon and related tools can infer. Intuitively, we may guess they have different properties, but no study so far, to our knowledge, has performed a systematic comparison. The benefits of such a comparison, as performed in the work reported here, include a better understanding of the possible role of automatically inferred contracts, the limitations of programmer-written contracts, and how to improve both kinds.

Contribution.

We performed an experiment to compare contracts written by programmers in existing, production-grade and student-written Eiffel code (freely available, so that others can repeat and continue our experiment), and assertions inferred for the same code by the Daikon tool equipped with a front-end for Eiffel that we developed. We were in particular investigating answers to the following questions:

- What proportion of the programmer-written assertion clauses are implied by the inferred assertions and vice-versa?
- What proportion of the inferred assertions clauses are relevant? (to the extent that we can assess this partly informal property)
- How can contract inference be used to assist programmers or to improve the programmer-written assertions?
- What factors influence the quality of the inferred contracts? More precisely, can we find correlations between any code metrics and the quality of the contracts inferred for that code?

Among the **results**:

- A high proportion of the inferred assertions clauses are correct (90%) and relevant (64%).
- Contract inference tools produce around 5 times more relevant (correct and interesting) assertion clauses than programmers write.
- Contract inference tools cannot find all programmer-written assertions: such tools infer only about 59% of all programmer-written assertion clauses.

To summarize these results by applying them to a normalized example of a hypothetical representative class for which the programmer had written 13 assertion clauses: the tool would infer 100 assertion clauses, out of which 90 would be correct; 64 of these would also be interesting. 8 of the 13 programmer-written assertion clauses would also appear among the inferred assertion clauses, or follow logically from them.

The experiment results also indicate that the quality of contracts inferred for a class is negatively influenced by the number of classes on which the class depends and the number of variables that are in scope at each program point where contracts are inferred.

Overview.

The rest of this paper is organized as follows. The next section introduces the basic notions of automated contract inference and the tool that we used in the experiment. Section 3 describes the setup of the experiment. Section 4 presents and analyzes the results; it ends with a discussion of threats to the validity of generalizations of the results. Section 5 presents related work and section 6 draws general conclusions.

2. DYNAMIC CONTRACT INFERENCE AND ITS APPLICABILITY TO A CONTRACT-AWARE LANGUAGE

Given a set of passing test cases that exercise a system, a dynamic contract inference tool will determine conditions that hold at various program points for the executions of the system through the test cases. Generalizing from these observations, the programmer can conclude that the corresponding assertions may hold for all program runs.

One of the best known tools built on this principle is Daikon [10]. This section provides an overview of Daikon, discusses some of the specifics of contract inference for Eiffel programs, and presents CITADEL, the Daikon front-end for Eiffel which we have developed, and which permitted the experiment reported here.

2.1 Daikon

To infer program properties, Daikon observes values of certain *variables* at specific *program points* during program executions. Interesting program points can be, for instance, routine entries and exits. Variables are different expressions which make sense at a program point, such as the currently executing object[2], routine arguments, the return value of a function[3], attributes of other variables, etc. Daikon maintains a list of templates which it instantiates into assertion clauses, using such program variables at specified program points, and checks if they hold for all executions of the program through a given set of test cases. As soon as an assertion clause does not hold, it is eliminated and not checked again for further executions. Ernst [10] and Perkins et al. [22] present a collection of heuristics that make this simple approach realistic.

Daikon's dynamic contract inference system consists of several components, as shown in figure 1. The main steps involved in the contract inference process are the following:

1. An instrumenter modifies the program source so that, at certain program points, it saves the values of the variables in scope to a data trace file. The instrumenter also produces program point declarations (static information about program points and variables).
2. The instrumented program is exercised through a test suite. Each run of the program results in a data trace file.
3. Daikon instantiates assertion clause templates from its list using variables of appropriate types. This results in a list of potential assertion clauses, which are then checked against the variable values recorded in the data trace files.
4. The inferred assertions can be post-processed, for instance inserted into the original source code as annotations.

Out of these components, only the instrumenter and the post-processor depend on the programming language in which the original system is written. These two components form a *front-end* that

[2]In Eiffel denoted as **Current**

[3]In Eiffel denoted as **Result**

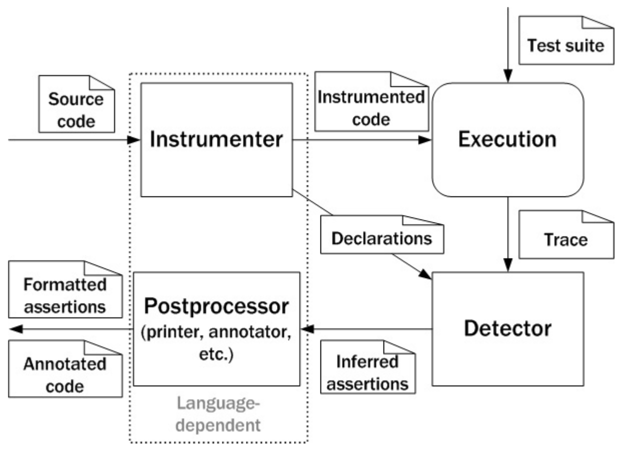


Figure 1: Dynamic contract inference system.

allows the universal assertion detector to work for software systems written in different languages (and even with data that was generated through other means than during program execution).

Because the contract inference process is based on checking assertion clause templates on actual *executions* of a system through a test suite, the inferred contracts reflect properties of both the original software system and the test suite.

2.2 Contract inference in Eiffel

Dynamic contract inference has proved its usefulness for software that lacks programmer-written specifications. In Eiffel developers do include contracts in the programs they write, but, as was mentioned above, these contracts are generally incomplete and sometimes incorrect.

We hence conjecture that dynamic contract inference can be used in Eiffel for the following purposes:

- **Strengthening contracts**, mainly strengthening programmer-written postconditions and class invariants. Strengthening loop invariants and weakening preconditions is also possible, but maybe less interesting for most programmers.
- **Correcting contracts**, in particular strengthening preconditions that failed to capture the full conditions necessary for a routine to work.
- **Improving test suite quality**: since the quality of the inferred contracts depends on the test suite used to exercise the system, the inferred contracts can be used to estimate the quality of the test suite.

Our study addresses the first two items on this list; the last one is the topic of ongoing work.

Since Eiffel has the built-in support for routine pre- and postconditions, class invariants and loop invariants, Eiffel programmers would like to infer contracts at program points that correspond to these four kinds of assertions. The variables in scope at each program point should correspond to expressions that can appear in the respective assertions.

If x is a variable at some program point and f is an attribute of the class of x, then $x.f$ is also added to the set of variables in scope at that program point. This is called variable *unfolding*. Unfolding is typically done up to a fixed small number of iterations (1 or 2).

It is possible to unfold a variable not only through attribute access, but also through calls to pure (side-effect free) functions. The Daikon front-end for Java provides such an option, requiring that the user supply a “purity file”: a list of functions in the system that are side-effect free and hence safe to use in unfolding. In Eiffel, the purity of functions, though not enforced by the compiler, is strongly encouraged. Therefore we opted for using functions with no arguments in unfolding by default. The user still can specify individual functions as impure through a command-line option. During this study we had to use this option only once, which shows that Eiffel programmers indeed mostly write pure functions.

There is one more problem with using functions for unfolding: what happens if a function is not applicable in the current context? For Eiffel functions this problem can be solved by checking, before any evaluation of $x.f$, that the programmer-written precondition of f holds (assuming that programmers usually write correct preconditions). If this is not the case, the front-end uses a special Daikon keyword to indicate that variable $x.f$ cannot be evaluated.

The Daikon front-end we developed is called CITADEL (Contract Inference Tool Applying Daikon to the Eiffel Language). The current implementation of the tool supports almost all Eiffel’s language constructs; this enables it to perform contract inference for realistic, production classes.

One limitation of the tool, directly relevant for this study, is that calls to functions with arguments are currently not used in the unfolding process and thus cannot appear in the inferred contracts. Another important limitation is the tool’s inability to instrument deferred (abstract) and external class members (members of Eiffel classes implemented in another programming language, typically C); hence contracts cannot be inferred for such members.

3. EXPERIMENT SETUP

Our experiment consisted of running CITADEL on 25 classes, 15 of them taken from industrial-grade Eiffel libraries, 4 — from an application that models public transportation in a city and another 6 classes written by Computer Science students. None of these classes were created especially for the study or modified in any way. We used classes of different size in terms of various code metrics and with diverse but clear semantics. Table 1 shows some metrics for the examined classes.

14 out of 15 library classes come from the standard Eiffel libraries version 6.1 (the current version at the time of the experiment), as indicated in the “source” column of table 1: EiffelBase, EiffelTime and Gobo, all included in the standard distribution of the most popular IDE for Eiffel (EiffelStudio [2]). Most applications written in Eiffel rely on some or all of these libraries. Class *MML_DEFAULT_PAIR* comes from a Mathematical Model Library, developed as part of a PhD dissertation. Library classes are highly reusable and presumably the effort spent to ensure their quality is accordingly high. In particular, library classes are usually equipped with relatively high quality contracts.

To diversify the scope of the experiment we also considered application classes: we used 4 classes from Traffic [3], a graphical application that models and visualizes the public transportation system of a city. This application was developed at ETH Zurich and is used in introductory Computer Science courses.

Because we also wanted to include in the experiment code written by less experienced programmers, we added classes created by students of Computer Science at ETH Zurich: classes *FRACTION1* and *FRACTION2* were implemented as assignments in an introductory programming course; classes *GENEALOGY1* and *GENEALOGY2* were implemented as part of a project given in a software engineering course; classes *NODE* and *EDGE* come from a graph library implemented as a Master project. For the first four classes some assertion clauses were inherited from ancestor classes or predefined in the assignment, which affects the study results as the quality of the contracts may be higher than if the contracts had been devised entirely by students.

Since none of the examined classes came with test suites, we had to write tests ourselves in order to allow Daikon to infer assertions. Because it is known that characteristics of the test suite such as coverage or number of executions of the tested elements influence Daikon's results, we constructed two test suites of different sizes: (1) a *small test suite*, containing approximately 10 calls with different random inputs to every instrumented routine, exercising the most typical behavior of the class, and (2) a *large test suite*, containing about 50 calls to every instrumented routine; this test suite achieves branch coverage and uses category-partitioning to cover different behaviors and achieve a high level of coverage of the object states.

All the studied classes as well as tests and the CITADEL tool itself are freely available online [1], so that the experiment can be reproduced and extended. Note however that the results for each particular run of contract inference can vary slightly because of the random nature of the tests.

4. RESULTS AND DISCUSSION

The results appear below grouped by the main questions under investigation: assessing the quality of the inferred assertion clauses (IA) in absolute terms (§4.1) and comparing the IA to programmer-written assertion clauses (PA) (§4.2). Since the test suite has a significant influence on the results and two different suites were used for each class, we report the quality measures for each test suite separately.

In the following we use box plots [25] to concisely represent the experimental results through their 5-number summary: the lowest value, the first quartile, the median, the third quartile, and the highest value.

4.1 Quality of the inferred contracts

The definition of the quality of inferred contracts used in this study involves two measures: the proportion of correct assertion clauses and the proportion of relevant assertion clauses, based on the following definitions: an IA is *correct* if it reflects a property of the source code (rather than a property of the test suite); it is *relevant* if it is correct and expresses a property that is interesting. An IA is said to be uninteresting if it follows one of four patterns:

- Relation between unrelated variables — IA involving variables whose relation is purely accidental. For example, an assertion clause of the form *person.age* < *person.bank_account_number* may always be true for a certain implementation, but most likely uninteresting.
- Equality of constants — IA that are trivially true because they refer to constants. E.g. *time1.hours_in_day* = *time2.hours_in_day*, where *time1* and *time2* are instances of class *TIME*, which has a constant attribute *hours_in_day*.
- Redundant — IA that are trivially implied by other IA at the same program point (where the implication does not depend on knowledge of the source code). As an example of a trivial implication, if it is already inferred that (*sorted_items* /= **Void**)= **Result**[4], then an assertion clause (*sorted_items* = **Void**)= (**not Result**) is redundant.

[4]/= is the Eiffel "not equal" operator; **Void** denotes detached references, also called `null` in other programming languages.

Table 2: Averages of correctness and relevancy.

(a) Averages of the percentages of correct inferred assertion clauses.

	Small TS	Large TS
Loop invariants	80%	90%
Preconditions	50%	84%
Postconditions & class invariants	72%	91%
Total	70%	90%

(b) Averages of the percentages of relevant inferred assertion clauses.

	Small TS	Large TS
Loop invariants	63%	70%
Preconditions	31%	50%
Postconditions & class invariants	60%	69%
Total	56%	64%

- Misplaced — IA that conceptually belong in another program point than where they were inferred. For instance, an assertion clause *s.count* >= 0 inferred in the precondition of a routine having *s* of type *STRING* as argument and where *count* is an attribute containing the number of characters in the string, is not a special property of variable *s*, but rather a common property of all instances of class *STRING* and should have been placed in its invariant.

While maybe not restrictive enough, this definition of relevancy has the advantage that it can be assessed objectively. Another option would have been to ask a developer or maintainer of the tested code to rate relevancy, but this was not possible for this case study. Hence we judged both contract correctness and relevancy manually, using the criteria listed above.

It is possible to implement partial elimination of uninteresting assertion clauses following these patterns in the Daikon front-end. This is part of future work, but at the time of the experiment assertion clauses that match the patterns listed above were not suppressed by CITADEL.

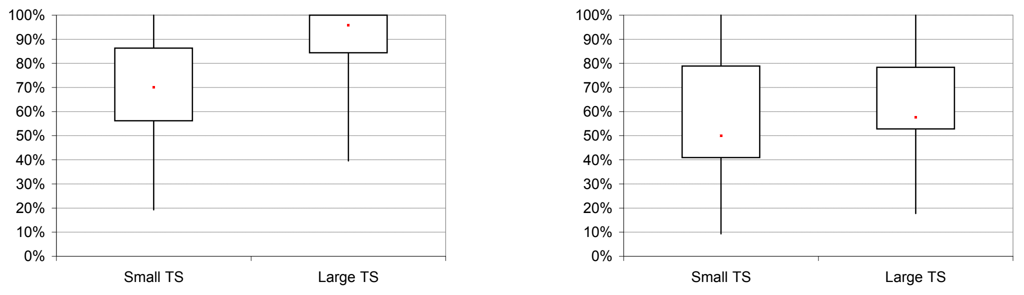
Figure 2(a) shows through box plots the 5-number summary of the percentage of *correct* IA for the classes used in the experiment. Table 2(a) shows the averages, over all classes, of the percentage of correct IA. The large test suite brings a substantial improvement over the small one. For 21 of the classes, more than 80% of the assertion clauses inferred for the large test suite are correct; this percentage is under 50% only for one class. For 8 classes, 100% of the assertion clauses inferred for the large test suite are correct. Section 4.4 discusses possible reasons for variations in IA correctness over different classes.

Figure 2(b) shows the percentage of *relevant* (correct and interesting) IA. Table 2(b) shows the averages, over all classes, of the percentage of relevant IA. Again, the large test suite generally brings an improvement over the small one, but for 6 classes smaller percentages of relevant assertion clauses are found through the large test suite than through the small one. This can happen when some correct but uninteresting assertion clauses, inferred for the large test suite, are not reported for the small test suite, because in this test suite there are not enough observations of these assertion clauses holding for system executions to pass Daikon's statistical confidence checks.

Table 1: Classes used in the experiment.

		Class size					Programmer-written assertion clauses					
Name	**Source**	**LOC**[1]	**R**[2]	**A**[3]	**S**[4]	**L**[5]	**PA**[6]	**Pre.**[7]	**Post.**[8]	**Inv.**[9]	**LInv.**[10]	**Expr.**[11]
BASIC_ROUTINES	EiffelBase	92	6	2	0	0	7	0	7	0	0	43%
BI_LINKABLE	EiffelBase	141	8	4	0	0	8	0	6	2	0	88%
BOOLEAN_REF	EiffelBase	174	12	3	0	0	25	6	16	3	0	48%
COMPARABLE	EiffelBase	117	7	3	0	0	21	7	13	1	0	33%
INTEGER_INTERVAL	EiffelBase	469	26	12	0	5	61	18	34	9	0	57%
LINKED_QUEUE	EiffelBase	202	5	23	1	2	34	4	3	27	0	29%
LINKED_STACK	EiffelBase	159	7	23	3	2	41	7	8	26	0	44%
TIME	EiffelTime	401	27	10	16	0	40	14	17	9	0	85%
DS_TOPOLOGICAL_SORTER	Gobo	487	21	2	10	9	42	11	23	8	0	69%
ST_COPY_ON_WRITE_STRING	Gobo	141	8	3	14	0	27	5	21	1	0	70%
ST_SPLITTER	Gobo	387	14	3	24	6	61	24	32	3	2	61%
ST_WORD_WRAPPER	Gobo	186	6	3	14	3	16	5	7	4	0	100%
UT_CHARACTER_FORMATTER	Gobo	164	6	4	9	0	8	6	2	0	0	100%
UT_VERSION	Gobo	272	14	4	0	0	45	12	29	4	0	89%
MML_DEFAULT_PAIR	MML	89	5	6	1	0	10	1	7	2	0	20%
TRAFFIC_BUILDING	Traffic	209	12	5	9	0	32	11	14	7	0	97%
TRAFFIC_COLOR	Traffic	120	7	2	0	0	39	18	15	6	0	100%
TRAFFIC_ROAD	Traffic	188	9	3	54	0	28	10	11	7	0	82%
TRAFFIC_TAXI	Traffic	198	9	8	16	0	32	14	7	11	0	94%
FRACTION1	Students	166	14	4	0	1	22	8	10	1	3	86%
FRACTION2	Students	156	14	4	0	1	21	8	10	1	2	90%
GENEALOGY1	Students	874	37	2	23	4	76	58	18	0	0	33%
GENEALOGY2	Students	1501	37	2	10	4	86	55	31	0	0	26%
EGDE	Students	188	8	2	0	0	27	7	18	2	0	81%
NODE	Students	125	6	5	0	0	22	4	15	3	0	91%
Average		288	13	6	8	1	33	13	15	5	0	69%
Total		7206	325	142	204	37	831	313	374	137	7	-

[1] Lines of code
[2] Number of instrumented routines
[3] Number of ancestors (superclasses); note that Eiffel allows multiple class inheritance
[4] Number of suppliers (see section 4.3)
[5] Number of loops inside instrumented routines
[6] Number of programmer-written assertion clauses
[7] Number of programmer-written precondition clauses
[8] Number of programmer-written postcondition clauses
[9] Number of programmer-written class invariant clauses
[10] Number of programmer-written loop invariant clauses
[11] Percentage of programmer-written assertion clauses expressible in Daikon's grammar (assertion clauses that match one of Daikon's templates)



(a) Percentage of correct inferred assertion clauses.

(b) Percentage of relevant inferred assertion clauses.

Figure 2: Correctness and Relevancy.

The percentages of relevant assertion clauses vary widely, from less than 20% to 100%; section 4.4 discusses the relation between the percentage of relevant assertion clauses and code metrics and possible explanations. On average, around 64% of IA are relevant for the large test suite.

Discussion

Overall, these results show that Daikon can infer many relevant assertion clauses for test suites containing sufficiently many calls to the routines of the classes under test and achieving high coverage of the possible states for instances of these classes.

The most frequent reasons for the generation of uninteresting assertion clauses are assertion misplacement and comparisons between unrelated variables. Most of the misplaced assertion clauses reside in loop invariants. A possible reason is that loop invariants have very specific semantics (describing how the loop iteratively achieves the goal of computation), which is hard to formalize and encode in the tool. As a solution, some heuristics can be introduced, such as filtering out the clauses that do not contain variables used in the loop body. Implementing various techniques for variable comparability analysis and suppressing assertion clauses at one program point by clauses inferred at another one [10] would most likely significantly reduce the number of irrelevant IA.

It is likely that a developer examining the correct and interesting inferred assertion clauses would find some of them more important than others. We did not investigate the question of how many inferred assertion clauses would be classified by a developer as "important" or "worth adding to the code", because for such a subjective decision the input of a creator or maintainer of the code would be necessary and we did not have the possibility of involving such a person in the study.

4.2 Inferred contracts vs. programmer-written contracts

The first measure we use to compare inferred to programmer-written contracts is *recall*, the proportion of the PA that are also inferred or implied by the IA. We distinguish between the recall of PA expressible in Daikon's grammar, or *expressible recall*, and the recall of all PA, which we refer to as *total recall*.

Figure 3(a) shows the expressible recall and figure 3(b) the total recall. Tables 3(a) and 3(b) show the averages of the expressible and total recall over all classes. The results show that not all PA are inferred by CITADEL, not even all expressible ones: the average of the expressible recall is 86% and the average of the total recall is 59% for the large test suite. While the expressible recall is higher than 90% for 12 out of the 25 classes for the large test suite, the total recall exceeds 90% only for 2 classes.

It is also interesting to note that for all classes containing programmer-written loop invariants, the expressible recall is 100% for both test suites for these loop invariants. The same holds for the total recall, with the exception of class *FRACTION1*, for which the total recall is 67% for both test suites. So overall the recall for loop invariants is very high, but the low number of programmer-written loop invariant clauses in the code we examined suggests special care in generalizing this result.

Programmer-written and inferred contracts can also be compared based on the numbers of clauses they contain. In general, the number of relevant IA is much higher than the number of clauses in programmer-written contracts, as illustrated in figure 4, which shows the ratios of relevant IA to PA, and in table 4, which shows the averages of these ratios.

For loop invariants, which programmers rarely write in practice, the ratios are very high.

Table 3: Averages of expressible and total recall.

(a) Expressible recall.

	Small TS	Large TS
Loop invariants	100%	100%
Preconditions	61%	89%
Postconditons & class invariants	77%	85%
Total	69%	86%

(b) Total recall.

	Small TS	Large TS
Loop invariants	89%	89%
Preconditions	48%	72%
Postcondition & class invariants	54%	59%
Total	50%	59%

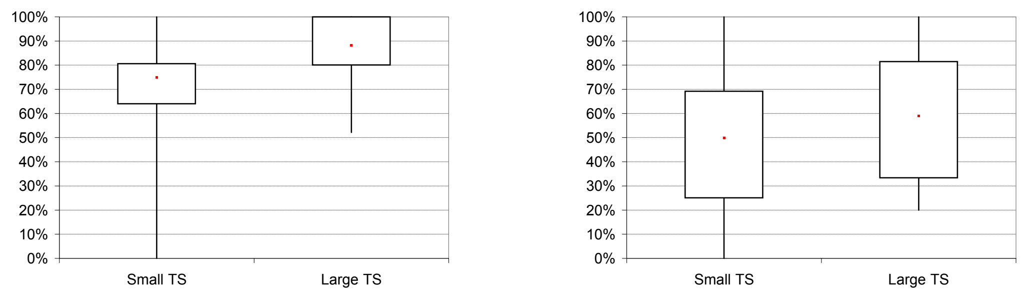
A striking difference exists between ratios for preconditions and those for postconditions and class invariants. With the small test suite CITADEL finds fewer relevant preconditions than programmers write; for the large test suite, it finds only marginally more preconditions than written by programmers. This factor is significantly higher for postconditions and class invariants: with the large test suite CITADEL finds about 5 times more relevant assertion clauses in these categories than programmers write.

A hypothesis to explain this difference is that developers using contract-aware languages view the various kinds of contracts in a different light: they take care to specify preconditions accurately, because preconditions make implementing routines easier (preconditions can be assumed, and the routine need not check them); postconditions and class invariants have no such immediate benefit and developers tend to neglect them. In other words, writing preconditions makes it easier to write code, while writing postconditions makes code easier to use and makes faults in the code easier to detect. The results seem to indicate that programmers care less about these code quality measures (ease of use and correctness) than about the ease of implementation.

One reason why inferred assertion clauses outnumber the programmer-written ones is that Daikon frequently infers theorems — assertion clauses that follow logically from other assertion clauses — while programmers almost never write them (except occasionally in class invariants). Nontrivial theorems, whose proofs require knowledge of source code semantics, can be useful for better understanding of the software. For example, in a routine that pushes an element on a stack the programmer would typically write a postcondition, stating that the number of elements is increased by one and that the pushed element is now on top. The tool would additionally infer that the stack is never empty after calling this routine, which shows an interesting relation between the number of elements and emptiness.

Another reason is that many inferred postcondition clauses describe what is *not* changed by a routine (in the form $x =$ **old** x)[5] — so called "frame properties", which programmers almost never write. If there were a special notation for specifying frame properties in Eiffel (such as the "modifies" or "assignable" clauses of Spec# [4] and JML [16]), then programmers would likely use this notation and such IA would not strengthen the programmer-written

[5]In Eiffel **old** is used in postconditions to refer to the value of an expression before the routine execution.



(a) Expressible recall. (b) Total recall.

Figure 3: Expressible recall and total recall.

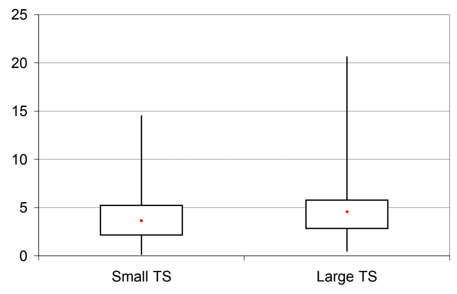


Figure 4: Ratios of relevant inferred assertion clauses to programmer-written assertion clauses.

Table 4: Averages of ratios of relevant inferred assertion clauses to programmer-written assertion clauses.

	Small TS	Large TS
Loop invariants	12.4	13.3
Preconditions	0.6	1.2
Postconditions & class invariants	4.8	5.6
Total	4.0	4.9

specification.

We also calculated the proportion of program points where no PA exist, but for which there are relevant IA (figure 5). This proportion varies considerably, from 0% for class *COMPARABLE* to 46% for the large test suite for class *UT_CHARACTER_FORMATTER*. The averages for loop invariants, preconditions, and postconditions and class invariants (shown in table 5) show again that programmers write more preconditions than postconditions and class invariants, and that they write very few loop invariants. Naturally, these results are highly dependent on the number of contracts written by developers, which vary with the class and author of the code, so it is hard to generalize from them. They do show, however, that contract

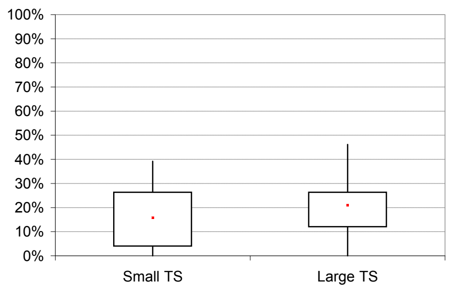


Figure 5: Percentage of program points with relevant inferred assertions and no programmer-written assertions.

inference tools can indeed produce relevant assertions for program points for which programmers did not write any assertions.

We define two factors showing how PA and IA complement each other:

- The *strengthening factor* α_1 *of IA over PA* reflects how much stronger PA become when IA are added:

$$\alpha_1 = \frac{\text{Relevant IA} + \text{PA} - \text{IA implied by PA}}{\text{PA}}$$

- The *strengthening factor* α_2 *of PA over IA* reflects how much

Table 5: Averages of percentage of program points with relevant inferred assertions and no programmer-written assertions.

	Small TS	Large TS
Loop invariants	68%	70%
Preconditions	1%	4%
Postconditons & class invariants	25%	28%
Total	17%	19%

Table 6: A comparison of the strengthening factors.

(a) Averages for the strengthening factor of inferred assertion clauses over programmer-written assertion clauses.

	Small TS	Medium TS
Loop invariants	12.6	13.4
Preconditions	1.1	1.1
Postconditions & class invariants	5.4	5.9
Total	4.6	5.1

(b) Averages for the strengthening factor of programmer-written assertion clauses over inferred assertion clauses.

	Small TS	Large TS
Loop invariants	1.0	1.0
Preconditions	2.3	1.4
Postconditions & class invariants	1.7	1.3
Total	1.5	1.2

stronger IA become when PA are added:

$$\alpha_2 = \frac{\text{PA} + \text{Relevant IA} - \text{PA implied by IA}}{\text{Relevant IA}}$$

Values strictly greater than 1.00 for these factors mean that strengthening occurs. These factors are complementary to recall, in the sense that they show how much the assertion clauses present in only one set (either that of the programmer-written assertion clauses or that of the inferred assertion clauses) can strengthen the ones from the other set.

Table 6(a) shows the averages for α_1 for loop invariants, preconditions, postconditions and class invariants, and the averages for α_1 for all assertions. It shows that IA can strengthen PA, but the strengthening factor for preconditions is generally much lower than that for postconditions and class invariants.

Table 6(b) shows the averages for α_2 for loop invariants, preconditions, postconditions and class invariants, and the averages for α_2 for all assertion clauses. It shows that PA can strengthen IA, but to a lesser degree than IA strengthen PA.

Discussion

The results show that inferred assertion clauses can be used to strengthen programmer-written contracts (postconditions and invariants) and sometimes even to correct existing contracts (strengthening preconditions). At about 19% of program points the programmer found nothing to specify while Daikon could infer relevant assertions. Still, not all PA are inferred or implied by the IA. So although inferred assertion clauses strengthen programmer contracts to a greater extent than conversely, automated contract inference does not find a superset of the PA.

Daikon is good at inferring simple and frequently used assertion clauses such as *an_argument* /= **Void**[6] (in preconditions) or *an_attribute* = *an_argument* (in postconditions of "setter" routines). The experiment results show that around 97% of the relevant IA use one of a few simple assertion clause templates: $x \sim y$ (where x is a variable, y is a variable or constant and $\sim$ is one of the relations $=, \neq, <, \leqslant, >, \geqslant$) or one of two kinds of implications that Daikon supports.

[6]Such assertion clauses are currently indeed very common in Eiffel contracts. In future they will likely become less important, since in the latest versions of the Eiffel compiler void-call safety is built into the type system.

It is tempting to draw the conclusion that at present the tool cannot compete with humans on higher levels of abstraction [10]. However, on closer examination it turns out that "abstract" assertions are mostly not expressible by the tool, because they contain functions with arguments, which are not used in the unfolding process. For about 54% of the inexpressible assertion clauses the reason was that they contained a call to a function with one argument. This means that even including functions with only one argument into the unfolding process can bring a substantial improvement in expressibility. Another 19% of the PA are inexpressible because of Daikon's very restrictive templates involving implications, while programmer-written contracts often contain implications.

These results indicate that for an object-oriented language like Eiffel the expressive power of contracts lies rather in their ability to contain calls to arbitrary side-effect free functions from the system, than in complicated syntax.

Here are some illustrative examples of programmer-written assertion clauses which Daikon could not infer, taken from class *LINKED_STACK* (which inherits them from *LINKED_LIST*):

- Assertion clauses not expressible in Daikon's grammar:
  - *occurrences* (*v*)= **old** (*occurrences* (*v*))+ 1 in the postcondition of a routine that pushes an element on a stack, which means that the number of occurrences of the pushed item is increased by one; this property is not expressible, because it uses function *occurrences*, which has an argument.
  - *is_empty* **implies** *off* in the invariant, meaning that in an empty stack the internal cursor cannot point to a valid position; the property is not expressible because it involves an implication not supported by Daikon.
  - **not** (*after* **and** *before*) in the invariant, saying that the internal cursor cannot be at the same time after and before any valid position; this property can be represented as (*after* **implies not** *before*)**and** (*before* **implies not** *after*) and thus could be inferred with better support for implications.
  - *index* <= *count* + 1 in the invariant, stating that the internal cursor cannot go beyond the last valid position further than by one; this property is not expressible because $x \leqslant y + c$ (with x and y being variables and c a constant) is not among Daikon's assertion clause templates.
- Assertion clauses (taken from the class invariant) expressible in Daikon's grammar, but still not inferred by the tool:
  - *extendible*, which says that a new element can always be added to the stack; this property most likely was not inferred because it didn't pass Daikon's statistical check.
  - *index_set*.*count* = *count*, stating that the number of elements in the stack is the same as the number of elements in the set of indexes which are valid for the underlying list; this property was not inferred, because it requires unfolding the variable **Current** twice (**Current**.*index_set*.*count*), while in the experiment the maximum number of unfolding iterations was set to 1.

4.3 Comparative analysis of the results for the different kinds of classes

As there were three groups of classes participating in the experiment (library classes, application classes and classes written by students), it is interesting to find out whether there are significant differences in the contract inference results between them. Our intuition was that contract strengthening should show more evidently in application classes than in library classes and even more in classes written by students because of the difference in quality of the original contracts. Indeed the average strengthening factor α_1 for postconditions and class invariants is 4.9 for library classes vs. 8 for application classes (all results in this section are given for the large test suite); however, the average for student classes — 7.1 — is not as high as was expected.

Another interesting observation is that the percentage of PA expressible in the tool for library classes is much lower than for application classes (62% vs. 93%), which indicates that in library classes programmers tend to write more complex and abstract contracts.

Another reason for differences than the quality of the original contracts seems to be the varying level of difficulty of writing good unit tests. In particular, we found writing tests for library classes much easier than for application classes because the former have less coupling and are intended for modular use. To capture this informal observation we introduce the following metric to measure the coupling of a class *C*: the number of *suppliers* of *C* (classes that *C* uses both directly and through other classes), which are not also suppliers of class *ANY*[7]. The average coupling factor for library classes turned out to be 6.1 vs. 16 for application classes, which supports our intuition. This difference influences the contract inference results: correctness (91% for library classes vs. 68% for application classes) and expressible recall (88% vs. 65%). The difference in recall is especially significant for preconditions (98% vs. 48%), which is quite intuitive, since inferred preconditions are influenced by the test suite more directly that other kinds of contracts. This is also reflected in the significant difference in the α_2 factor for preconditions: programmer-written preconditions strengthen inferred ones by a factor of 1.1 in library classes vs. 2.3 in application classes.

4.4 Correlations

In trying to establish which properties of the classes may influence the quality of the inferred contracts, in this study we also examined correlations between class metrics and the quality of contracts inferred for each class. More specifically, we looked for correlations between the following factors:

- Code metrics of the examined classes: number of lines of code, number of routines implemented, number of ancestors, coupling (as introduced in section 4.3), number of queries[8] with no arguments (also separately for numeric and boolean queries), number of PA, percentage of assertion clauses expressible in Daikon's grammar;

- Metrics of the IA: total number of IA, correctness, relevancy, expressible recall, total recall, the ratio of relevant IA to PA, strengthening factors α_1 and α_2.

All correlations listed below were calculated using the Pearson product-moment correlation coefficient for assertion clauses (including those in loop invariants) inferred for the large test suite.

[7] *ANY* in Eiffel is the root of the class inheritance hierarchy, similar to `Object` in Java.
[8] Attributes and functions

Correctness and relevancy of IA have negative correlations, -0.91 and -0.62 respectively, to the total number of IA.

These two measures also have negative correlations to the number of numeric zero-argument queries in a class (-0.66 and -0.60 respectively). Numeric (integer and real) queries with no arguments increase Daikon's assertion search space significantly, because Daikon has many assertion clause templates for numeric variables, some of these templates involving relations between 2 or 3 variables. The positive correlation (0.69) between the number of numeric queries with no arguments and the total number of IA also shows this.

All these results suggest that the increased assertion search space has a negative influence on the correctness of the IA and a slightly weaker negative influence on their relevancy.

On the other hand, the total number of IA has strong positive correlations to the ratio of relevant inferred IA to PA and the strengthening factor of IA over PA (0.72 and 0.71 respectively). These correlations indicate that with more assertion clauses inferred, despite of a large proportion of them being irrelevant, the number of relevant ones also increases.

Correctness and relevancy also have negative correlations to coupling, -0.58 and -0.52 respectively, which seems to confirm the intuition that the worse quality of unit tests for classes with higher coupling results in lower correctness and relevancy of assertions inferred from these tests.

4.5 Threats to generalization

Probably the biggest threat to generalization of these results is the limited number of classes examined in the experiment. We selected classes written by programmers with various degrees of experience, classes having different semantics and sizes in terms of various code metrics, but naturally their representativeness is limited. Specifically, the relatively low number of application and student classes examined in the experiment suggests special care in generalizing the comparative results discussed in section 4.3. Furthermore, the relatively small size of some of the examined classes may also be a threat to validity of the results.

The study only involved unit testing separate classes; testing entire applications may produce different results.

As shown both by the results of this study and of previous investigations [20, 11], the quality and size of the test suite have a strong influence on the quality of the inferred contracts. We ran the experiment for two different test suites for each class, but test suites of other sizes and with other characteristics might lead to different results.

Since we could not discuss the IA with the developers of the classes used in the study, we judged the correctness of the IA based on the implementation and we used a fixed set of rules for determining which IA are interesting and which not, as explained in section 4.1. The results might have been different had the original developers performed the classification.

Other factors likely to influence the results and providing caution against hasty generalization are the technical characteristics of Daikon and of the front-end we developed for it, and specifically the potential faults in these tools.

5. RELATED WORK

Several studies on Daikon-inferred contracts have been performed, but we are not aware of any studies comparing these to contracts written by programmers independently of the tool.

Some studies investigate the effect of the test suite on Daikon-inferred contracts. Nimmer and Ernst [19] showed that Daikon produces, even from relatively small test suites, assertion clauses that are consistent and sufficient for (proving the absence of runtime errors with very little change. Nimmer and Ernst [20] also showed that test cases that mainly exercise corner cases are not suited for contract inference. Gupta et al. [11] and Harder et al. [13] showed that existing code coverage criteria (branch coverage, definition-use pair coverage) do not provide test suites that are good enough for invariant detection, but test suites that satisfy these traditional criteria produce more relevant assertion clauses than random.

A study of users' experience with Daikon [20] showed that using Daikon neither speeds up nor slows down users trying to annotate programs with contracts, but improves recall (how many assertion clauses from the intended specification do finally appear in contracts). Half of the users participating in their study considered Daikon to be helpful, especially because they could use the generated assertion clauses as support for finding others. More than half the users found removing incorrect assertion clauses easy. That study showed how developers can use Daikon as support in the assertion-writing process; in the present study there was no interference between running the contract inference tool on the code and the manual process of writing contracts, since we wanted to investigate the contributions of each approach to providing classes with high-quality executable specifications.

A substantial amount of work uses Daikon-inferred contracts as support for automated testing. The Eclat [21] tool uses contracts inferred by Daikon as filters for invalid inputs and as an automated oracle. Xie and Notkin [26] developed the operational violation approach, which uses Daikon to infer likely assertions and automatically generates tests, verifying the inferred assertions. Tests violating these assertions are presented to users for examination, since they exercise behavior that the tool has not seen before. The DSD-Crasher tool [9] employs Daikon for inferring contracts, exports these contracts as JML contracts, and uses these to guide the input generation of the Check'n'Crash tool [8]. Substra [27] generates integration tests based on Daikon-inferred constraints on component interfaces.

DIDUCE [12] is another tool which infers contracts from program executions. DIDUCE is built on the same principles as Daikon, but can operate in two modes: the training mode and the checking mode. In the training mode, the tool infers assertions from executions of the system, by starting out with the most restrictive conditions and relaxing them as if finds states that violate them. The checking mode is an extension of the training mode, in the sense that in the checking mode, when an assertion violation occurs, DIDUCE also reports the violation, in addition to relaxing the assertion in question.

Pytlik et al. [23] developed the Carrot assertion detector which uses the same principles as Daikon, but has a different implementation. Other work [14, 15] investigates dynamic inference techniques for algebraic specifications.

Some of the ideas developed in academic research on contract inference were also adopted by industry. AgitatorOne [5], previously called Agitator, implements a Daikon-like approach for inferring assertions. Users have the option of promoting these inferred assertions to contracts included in the program or discarding them. The Axiom Meister tool [24] developed at Microsoft Research uses symbolic execution for finding routine contracts for .NET programs.

6. CONCLUSIONS

From the experiment results we can draw the following conclusions:

- A high proportion of the inferred assertion clauses are correct (reflect true properties of the source code): around 90% for the large test suite.
- A high proportion of the inferred assertion clauses are relevant (correct and interesting): around 64% for the large test suite.
- Contract inference can be used to strengthen programmer-written specifications, as shown by a strengthening factor averaging at 5.9 for postconditions and class invariants and at 13.4 for loop invariants for the large test suite.
- Contract inference cannot find all assertion clauses written by programmers; this is evidenced by an average recall value of 59%, meaning that only a bit more than half of the programmer-written assertion clauses are also inferred or implied by the inferred assertions.
- The quality of inferred contracts decreases with the growth of coupling between classes and the size of the assertion search space: the more suppliers and zero-argument numeric queries a class has, the higher the percentage of incorrect and uninteresting inferred assertion clauses.

These results suggest that contract inference cannot completely replace the manual work of writing assertions. Nor should it: in the Design by Contract software development method, the manual process of writing contracts starts already before the implementation work and can expand until after the implementation is finished, while contract inference tools can only be used when the implementation is ready. Only relying on such a tool to produce contracts loses all the benefits of writing contracts from software analysis and design through implementation. Nevertheless, when a complete or partial implementation of the system is ready, contract inference tools can be used to strengthen the existing programmer-written assertions, resulting in more accurate specification.

Future work includes improving both Daikon and its front-end for Eiffel, based on the insights gained through this study. A promising idea that we intend to explore is "push-button inference": using an automated testing tool to generate the test suites necessary for the contract inference instead of handmade tests. Another direction for future work is to explore the use of contract inference for estimating test suite quality, based on the idea that the quality of inferred contracts is indicative of the quality of a test suite, which should relate to the test suite's fault-revealing capability.

Acknowledgements

We thank Michael Ernst for the support provided during the implementation of CITADEL and for his valuable feedback on the work presented here and on a previous draft of this paper. We also thank Cristiano Calgano, Andreas Leitner and Alexander Pretschner for insightful comments on the presentation of this work.

7. REFERENCES

[1] CITADEL webpage. `http://se.inf.ethz.ch/people/polikarpova/citadel.html`.

[2] EiffelStudio. Eiffel Software. `http://www.eiffel.com/`.

[3] Traffic. `http://traffic.origo.ethz.ch/`.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*. Springer, 2004.

[5] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 169–180, New York, NY, USA, 2006. ACM Press.

[6] P. Chalin. Are practitioners writing contracts? In *Springer LNCS 4157*, pages 100–113, 2006.

[7] I. Ciupa, B. Meyer, M. Oriol, and A. Pretschner. Finding faults: Manual testing vs. random+ testing vs. user reports. In *Proceedings of ISSRE (International Symposium on Software Reliability) 2008*, 2008.

[8] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proc. 27th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 422–431. ACM, May 2005.

[9] C. Csallner and Y. Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 245–254. ACM, July 2006.

[10] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, Aug. 2000.

[11] N. Gupta and Z. V. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *ASE 2003: Proceedings of the 18th Annual International Conference on Automated Software Engineering*, pages 49–58, Montreal, Canada, October 8–10, 2003.

[12] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM.

[13] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *ICSE'03, Proceedings of the 25th International Conference on Software Engineering*, pages 60–71, Portland, Oregon, May 6–8, 2003.

[14] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In L. Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference*, Darmstadt, July 2003. Springer.

[15] J. Henkel and A. Diwan. A tool for writing and debugging algebraic specifications. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 449–458, Washington, DC, USA, 2004. IEEE Computer Society.

[16] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

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

[18] B. Meyer. *Object-Oriented Software Construction, 2nd edition*. Prentice Hall, 1997.

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

[20] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: an empirical evaluation. *SIGSOFT Software Engineering Notes*, 27(6):11–20, 2002.

[21] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, Glasgow, Scotland, July 25–29, 2005.

[22] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 23–32, Newport Beach, CA, USA, November 2–4, 2004.

[23] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *Proceedings of the Workshop on Automated and Algorithmic Debugging 2003*, 2003.

[24] N. Tillmann, F. Chen, and W. Schulte. Discovering likely method specifications. In *ICFEM*, pages 717–736, 2006.

[25] J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, Reading, MA, 1977.

[26] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 40–48, Oct. 2003.

[27] H. Yuan and T. Xie. Substra: A framework for automatic generation of integration tests. In *1st Workshop on Automation of Software Test (AST 2006)*, pages 64–70, Shanghai, China, May 2006.