Report

A comparative study of programmer-written and automatically inferred contracts

Author(s):
Polikarpova, Nadezhda; Ciupa, Ilinca; Meyer, Bertrand

Publication Date:
2012

Permanent Link:
https://doi.org/10.3929/ethz-a-006824877

Rights / License:
In Copyright - Non-Commercial Use Permitted
A comparative study of programmer-written and automatically inferred contracts

Nadezhda 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 of the respective roles of humans and tools. Working on an industrial base of contract-equipped software, we applied Daikon to infer contracts, and compared the results 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 (assertions) as written by programmers; but it only finds about a half of those originally written by programmers.

The study also uncovered strong correlations between the quality of inferred contracts and some other interesting code metrics.

1. Introduction

Embedding contracts (executable specification elements) in software texts yields a number of benefits [14]: contracts provide a basis for program proving 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 [15], JML [13] and Spec# [2]. 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 [4] by Chalin, who shows that Eiffel classes contain a higher proportion of assertions\(^1\) than classes written in languages not supporting DbC and that 97% of assertions 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.

\(^1\)A contract element such as a precondition, postcondition, class invariant or loop invariant consists of a number of clauses, combined using logical conjunction; the term assertion denotes such a clause.
This is not, however, the full story regarding programmer-written contracts, since closer examination (for languages with direct support) shows that they are often incomplete, and sometimes incorrect in the sense of contradicting the implementation or the informal intent [5]. 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 [8], which produces likely program 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 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 contract elements [8, 17, 18].

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 can infer in non-contract-aware languages. 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 programmer-written contracts in existing, production-grade Eiffel code (freely available, so that others can repeat and continue our experiment), and assertions inferred for the same code by a Daikon front-end for Eiffel that we developed. We were in particular investigating answers to the following questions:

- What proportion of programmer-written assertions are implied by the inferred assertions and vice-versa? Is there an inclusion relationship between these two kinds?

- What proportion of inferred assertions are correct and interesting (to the extent that we can assess these partly informal properties)?

- How can assertion 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 inferred assertions are correct (90%) and interesting (65%).

- Assertion inference tools produce around 5 times more correct and interesting assertions than programmers write.

- Assertion inference tools cannot find all programmer-written contracts: such tools infer only about 50% of all programmer-written assertions.

To summarize these results by applying them to a hypothetical representative class for which the programmer had written 13 assertions: the tool would infer 100 assertions, out of which 90 would be correct; 65 of these would also be interesting. 7 of the 13 programmer-written assertions would also appear among the inferred assertions, or follow logically from them.

The experiment results also indicate that the quality of inferred contracts is negatively influenced by the size of the search space: the more program points a class has, for which contracts should be inferred, and the more variables are in scope at each program point, the more incorrect and uninteresting assertions are inferred.

**Overview.** The rest of this presentation is organized as follows. The next section introduces the basic notions of automated assertion inference and the tool that we used in the experiment: CITADEL, a Daikon front-end for Eiffel. Section 3 describes the setup of the experiment. Section 4 presents and analyzes the results, ending with a discussion of the threats to 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, it can posit that the corresponding assertions may hold for all program runs.

The best known tool built on this principle is Daikon [8], used for the present study. This section provides an overview of Daikon, discusses some of the specifics of contract inference for Eiffel programs, and presents CITADEL,
the Eiffel front-end for Daikon 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 specific program point, such as the currently executing object, routine arguments, the return value of a function, attributes of other variables, etc. Daikon maintains a list of assertion templates which it instantiates using such program variables at specified program points, and checks if they hold for the executions of the program through a given set of test cases. As soon as an assertion does not hold for an execution, it is eliminated and not checked again for further executions. [8] presents 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 templates from its list using variables of appropriate types. This results in a list of potential assertions, which are then checked against the variable values recorded in the data trace files.

4. The inferred assertions can be post-processed, for instance by a pretty-printer, and inserted into the original source code as annotations.

Out of these components, only the instrumenter and the postprocessor depend on the programming language in which the original system is written. These two components form a front-end that allows the universal assertion detector to work for software systems written in different languages (and even with data that was generated through other means during program execution).

Because the contract inference process is based on checking assertion 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 the situation is different, because of its support for Design by Contract — Eiffel developers do indeed include contracts in the programs they write, but these contracts are generally incomplete and, as we have analyzed in previous work [5], 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 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 suites quality**: since the quality of the inferred contracts depends on the test suites used to exercise the system, it can be used to estimate the quality of the test suite.

- **Assisting in static checking** and other non-Eiffel-specific uses.

Our study addresses the first two items on this list; the other two are the topic of ongoing work. In Eiffel, since inferred assertions are used in regular contracts, the types of program points correspond one-to-one to the possible contract types: preconditions, postconditions, class invariants, and loop invariants. The variables to output at each program point correspond to entities that can appear in the respective assertions.
Among these entities are also function return values. Using function results as part of the variables whose values Daikon examines raises two problems. First, executing the functions to get the return values can have side effects. The Daikon front-end for Java solves this problem by requiring that users who turn on the option of including functions in contracts provide a so-called “purity file”, listing all functions which are side-effect free and hence safe to evaluate at runtime as part of the contract inference process. In Eiffel the situation is different, because of the Uniform Access Principle [15], which states that it should be transparent to clients of a class if a zero-argument query is implemented through an attribute or through a function. Hence, to be consistent with this principle, in CITADEL we opted for including both attributes and functions with no arguments in inferred contracts. There is indeed currently no mechanism in the Eiffel language preventing side-effects in functions, but such side-effects are strongly discouraged by the method. Therefore we consider that the added expressiveness of inferred assertions including zero-argument functions outweighs the potential danger of changing system state through function evaluation.

Second, functions are often partial — what happens if a function cannot be called in current context? We solve this problem by checking, before any function evaluation, that the programmer-written precondition of the function holds (assuming that programmers usually write preconditions well). If this is not the case, we just indicate that the function cannot be evaluated by using the special Daikon value “nonsensical”. We also use this value when the target object on which the query should be evaluated is void (null).

The tool we developed to implement this technique, and more generally to provide an Eiffel front-end to Daikon, is called CITADEL (Contract Inference Tool Applying Daikon to the Eiffel Language). The current implementation of the tool supports almost all of Eiffel’s language constructs including the most advanced and recent; this enables it to perform contract inference for realistic, production classes. The main limitations are the tool’s inability to instrument deferred (abstract) features of a class, and external features (whose implementation is in another programming language such as C). Another limitation, directly relevant for this study, is that the tool does not monitor return values of functions with arguments and hence cannot include them in inferred assertions.

3. Experiment setup

Our experiment consisted of running CITADEL on 15 classes, 11 of them taken unchanged from industrial-grade Eiffel libraries and 4 written by computer science students. None of these classes were written especially for the study or modified in any way. We aimed for classes of different size, varying depth in the inheritance hierarchy, with and without loops, and more generally with diverse but clear semantics. Table 1 shows some metrics for the examined classes. The class names appear abbreviated in the figures below; table 1 gives the correspondence between abbreviations and actual class names.

Library classes are from standard Eiffel libraries: EiffelBase, EiffelTime and Gobo, all included in the standard distribution of the most popular IDE for Eiffel (EiffelStudio [1]); they were taken from the current release at the time of experiment (6.1). Most Eiffel applications rely on some or all of these libraries. The 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.

Because we also wanted to include 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 and classes GENEALOGY1 and GENEALOGY2 were implemented as part of a project given in a software engineering course. These two classes had a given common interface, including some contracts. The students were not allowed to change the routine preconditions, but they could add clauses to the other contracts. This circumstance affects the study results as the quality of contracts may be higher than if the contracts had been entirely devised by students.

For each class we manually constructed three 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; (2) a medium-size test suite, containing about 50 calls to every instrumented routine and adding some partition tests that exercise less typical behavior; and (3) a large test suite, which does not differ from the medium one qualitatively, but has 10 times more routine calls (naturally using different inputs); the large test suite hence contains about 500 calls to every instrumented routine.

4. Results and discussion

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

4.1. Quality of the inferred contracts

The definition of inferred contracts quality used in this study involves two measures: the proportion of correct as-
Table 1. Classes used in the experiment.

<table>
<thead>
<tr>
<th>Name</th>
<th>Abbrev.</th>
<th>Source</th>
<th>LOC(^1)</th>
<th>LOCC(^2)</th>
<th>#R(^3)</th>
<th>#A(^4)</th>
<th>#L(^8)</th>
<th>Expr. (^9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGRAL_INTERVAL</td>
<td>C1</td>
<td>Base</td>
<td>469</td>
<td>113</td>
<td>26</td>
<td>11</td>
<td>18</td>
<td>34</td>
</tr>
<tr>
<td>BASIC_ROUTINES</td>
<td>C2</td>
<td>Base</td>
<td>92</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>BOOLEAN_REF</td>
<td>C3</td>
<td>Base</td>
<td>174</td>
<td>72</td>
<td>12</td>
<td>2</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>ST_SPLITTER</td>
<td>C4</td>
<td>Gobo</td>
<td>389</td>
<td>61</td>
<td>14</td>
<td>2</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>COMPARABLE</td>
<td>C5</td>
<td>Base</td>
<td>117</td>
<td>21</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td>DS_TOPOLOGICAL_SORTER</td>
<td>C6</td>
<td>Gobo</td>
<td>487</td>
<td>42</td>
<td>21</td>
<td>1</td>
<td>11</td>
<td>23</td>
</tr>
<tr>
<td>TIME</td>
<td>C7</td>
<td>Time</td>
<td>401</td>
<td>41</td>
<td>27</td>
<td>9</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>GENEALOGY_2</td>
<td>C8</td>
<td>students</td>
<td>1501</td>
<td>132</td>
<td>37</td>
<td>1</td>
<td>55</td>
<td>31</td>
</tr>
<tr>
<td>GENEALOGY_1</td>
<td>C9</td>
<td>students</td>
<td>874</td>
<td>120</td>
<td>37</td>
<td>1</td>
<td>58</td>
<td>18</td>
</tr>
<tr>
<td>FRACTION_1</td>
<td>C10</td>
<td>students</td>
<td>166</td>
<td>64</td>
<td>14</td>
<td>3</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>FRACTION_2</td>
<td>C11</td>
<td>students</td>
<td>156</td>
<td>68</td>
<td>14</td>
<td>3</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>LINKED_STACK</td>
<td>C12</td>
<td>Base</td>
<td>159</td>
<td>41</td>
<td>7</td>
<td>22</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>LINKED_QUEUE</td>
<td>C13</td>
<td>Base</td>
<td>202</td>
<td>34</td>
<td>5</td>
<td>22</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>BI_LINKABLE</td>
<td>C14</td>
<td>Base</td>
<td>138</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>ST_WORD_WRAPPER</td>
<td>C15</td>
<td>Gobo</td>
<td>186</td>
<td>16</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

\(^1\) Lines of code
\(^2\) Lines of contract code
\(^3\) Number of instrumented routines
\(^4\) Number of ancestors
\(^5\) Number of programmer-written precondition clauses
\(^6\) Number of programmer-written postcondition clauses
\(^7\) Number of programmer-written class invariant clauses
\(^8\) Number of programmer-written loop invariant clauses/number of loops inside instrumented routines
\(^9\) Percentage of programmer-written assertions expressible in Daikon’s grammar (assertions that match one of Daikon’s templates)

Assertions and the proportion of relevant assertions, based on the following definitions: an IA is correct if it reflects a property of the source code; 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 of the form `person.age < person.bank_account_number` may be always true for 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 the `TIME` class, 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`, then an assertion `(sorted_items = Void)= (not Result)` is redundant.

- Misplaced — IA that conceptually belong in another program point than where they were inferred. For instance, an assertion `s.count >= 0`, where `s` is instance of `STRING` class, is not a special property of variable `s`, but rather a common property of class `STRING` and should have been placed in its invariant.

While lenient, 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.

Figure 2(a) shows the total percentage of correct IA for each class, for each test suite size. Table 2(a) shows the averages, over all classes, of the percentages of correct IA, for every test suite size. The medium-size test suite generally brings a substantial improvement over the small one. The large test suite only brings a small improvement, if any, over the medium-size one. For 13 of the classes, more than 80% of the assertions inferred for the medium-size and large test suites are correct; this percentage drops under 50% only for one class. For 4 classes, 100% of the assertions inferred for the medium-size and large test suites are correct. For 4 other classes, this proportion exceeds 95%. For the large test suite, the average correctness of IA exceeds 90%.

Figure 2(b) shows the total percentage of relevant IA for each class, for each test suite size. Table 2(b) shows the averages, over all classes, of the percentage of relevant IA, for each test suite size. Again, the medium-size test suite generally brings an improvement over the small one, but for 4 classes smaller percentages of relevant assertions are
Table 2. Averages for the two measures of the absolute quality of the inferred contracts.

(a) Averages of the percentages of correct inferred assertions.

<table>
<thead>
<tr>
<th>Small TS</th>
<th>Medium TS</th>
<th>Large TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop invariants</td>
<td>79.52%</td>
<td>90.47%</td>
</tr>
<tr>
<td>Preconditions</td>
<td>46.30%</td>
<td>80.18%</td>
</tr>
<tr>
<td>Postconditions &amp; class invariants</td>
<td>68.04%</td>
<td>89.95%</td>
</tr>
<tr>
<td>Total</td>
<td>67.09%</td>
<td>89.01%</td>
</tr>
</tbody>
</table>

(b) Averages of the percentages of relevant inferred assertions.

<table>
<thead>
<tr>
<th>Small TS</th>
<th>Medium TS</th>
<th>Large TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop invariants</td>
<td>62.86%</td>
<td>69.96%</td>
</tr>
<tr>
<td>Preconditions</td>
<td>32.64%</td>
<td>52.58%</td>
</tr>
<tr>
<td>Postconditions &amp; class invariants</td>
<td>57.24%</td>
<td>68.83%</td>
</tr>
<tr>
<td>Total</td>
<td>52.57%</td>
<td>65.80%</td>
</tr>
</tbody>
</table>

found through the medium and large test suites than through the small one. The percentages of relevant assertions vary widely, from less than 20% to 100%; section 4.3 will discuss relation to code metrics and possible explanations. On average, around 65% of IA are relevant for the medium and large test suites.

Discussion

Overall, these results show that Daikon can infer many relevant assertions for test suites of a certain size and quality.

The most frequent reasons for generation of uninteresting assertions are the use of complex numeric functions and comparisons between unrelated variables. Another is that some clauses do not characterize the program point where they are inferred. This is most often the case for loop invariants. As a solution, some additional filtering techniques can be used, such as filtering out, in function postconditions, all clauses that do not contain a reference to the returned value, or filtering out in loop invariants the clauses that do not contain variables used in the loop body. Furthermore, the number of uninteresting IA can also be reduced by introducing advanced rules for suppressions: for instance, the invariant of class \( A \) can suppress assertions about variable \( x \) of type \( A \) in its clients.

4.2. Inferred assertions vs. programmer-written assertions

The first measure we use to compare inferred to programmer-written assertions is recall, the proportion of the PA that are also IA, or implied by IA. We distinguish two variants: recall of PA expressible in Daikon’s grammar, or expressible recall; 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, for every class and every test suite size. Tables 3(a) and 3(b) show the averages of the expressible and total recall over all classes.

While the expressible recall is higher than 0.9 for 6 out of the 15 classes for the large test suite, the total recall exceeds 0.9 only for 1 class for the large test suite. This is also reflected in the averages: 0.86 for the expressible recall and 0.51 for the total recall for the medium test suite. These values show that not all PA are inferred by CITADEL, not even all expressible ones.

It is also interesting to note that for all classes containing programmer-written loop invariants, the expressible recall is 100% for all test suites for these loop invariants. The same holds for the total recall, with the exception of class \( \text{FRACTION}1 \), for which the total recall is 67% for all test suite sizes. So overall the recall for loop invariants is very high, but the low number of programmer-written loop invariants 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 il-
(a) Expressible recall.

Figure 3. The expressible recall and the total recall.

Table 3. Averages of the expressible and the total recall.

<table>
<thead>
<tr>
<th></th>
<th>Small TS</th>
<th>Medium TS</th>
<th>Large TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop invariants</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Preconditions</td>
<td>0.72</td>
<td>0.97</td>
<td>0.98</td>
</tr>
<tr>
<td>Postconditions &amp; class invariants</td>
<td>0.68</td>
<td>0.81</td>
<td>0.80</td>
</tr>
<tr>
<td>Total</td>
<td>0.67</td>
<td>0.86</td>
<td>0.86</td>
</tr>
</tbody>
</table>

(b) Total recall.

<table>
<thead>
<tr>
<th></th>
<th>Small TS</th>
<th>Medium TS</th>
<th>Large TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop invariants</td>
<td>0.89</td>
<td>0.89</td>
<td>0.89</td>
</tr>
<tr>
<td>Preconditions</td>
<td>0.54</td>
<td>0.73</td>
<td>0.74</td>
</tr>
<tr>
<td>Postconditions &amp; class invariants</td>
<td>0.42</td>
<td>0.49</td>
<td>0.48</td>
</tr>
<tr>
<td>Total</td>
<td>0.41</td>
<td>0.51</td>
<td>0.51</td>
</tr>
</tbody>
</table>

Table 4 shows the average ratios of the relevant IA to PA for all test suite sizes.

For loop invariants, which programmers rarely write in practice, the ratios are very high. A striking difference exists between ratios for preconditions and those for postconditions and class invariants. With the small test suite, CATADEL finds fewer relevant preconditions than programmers write; for the medium and large test suites, it finds only marginally more preconditions than written by programmers. This factor is significantly higher for postconditions and class invariants: CATADEL finds about 5 times more relevant assertions 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 must not check them); postconditions and class invariants have no such immediate benefit, so developers tend to neglect them. Another possible explanation has to do with the way in which assertion inference tools work, namely with the dependency between their results and the quality of the test suites they use: while finding the postconditions from given preconditions is easy (it only requires executing the routines), finding the correct preconditions is much harder, as the test cases have to exercise enough calling situations, which requires inter-procedural rather than intra-procedural cover. Thus, it is hard to come up with good test cases for inferring preconditions, and this affects the performance of the tool. This observation also relates to a future development of this work, already noted: using CATADEL to assess the quality of test suites.

We also calculated the proportion of program points where no PA exist, but for which there are relevant IA. Program points are routine entry and exit (corresponding to pre- and postconditions), loops (for loop invariants), and one point per class for the class invariant. So the number of program points per class is 2 * number_of_processed_routines + number_of_loops + 1.

Illustrated in figure 4, which shows a comparison of the number of PA and the number of relevant IA for the different test suite sizes.
Table 4. Averages of the ratios of relevant inferred assertions to programmer-written assertions.

<table>
<thead>
<tr>
<th></th>
<th>Small TS</th>
<th>Medium TS</th>
<th>Large TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop invariants</td>
<td>12.44</td>
<td>13.28</td>
<td>14.11</td>
</tr>
<tr>
<td>Preconditions</td>
<td>0.63</td>
<td>1.39</td>
<td>1.61</td>
</tr>
<tr>
<td>Postconditions &amp; class invariants</td>
<td>4.55</td>
<td>5.77</td>
<td>5.45</td>
</tr>
<tr>
<td>Total</td>
<td>4.50</td>
<td>5.88</td>
<td>5.85</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Small TS</th>
<th>Medium TS</th>
<th>Large TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop invariants</td>
<td>0.68</td>
<td>0.70</td>
<td>0.70</td>
</tr>
<tr>
<td>Preconditions</td>
<td>0.02</td>
<td>0.03</td>
<td>0.05</td>
</tr>
<tr>
<td>Postconditions &amp; class invariants</td>
<td>0.28</td>
<td>0.33</td>
<td>0.35</td>
</tr>
<tr>
<td>Total</td>
<td>0.20</td>
<td>0.23</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Figure 5 shows the percentage of program points without PA but for which relevant assertions were inferred. (The numbers in this figure include loop invariants. The results excluding loop invariants are not significantly different, because the examined classes contain very few loops.) This percentage varies considerably, from 0% for class COMPARABLE to over 50% for the large test suite for class BASIC_ROUTINES. 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-inference tools can indeed produce relevant assertions for program points for which programmers did not write any assertions.

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

- The strengthening factor of IA over PA $\alpha_1$ reflects how much stronger PA become when IA are added. It is calculated by adding the numbers of relevant IA and PA, subtracting the number of IA implied by PA, and dividing the result by the number of PA:
  $$\alpha_1 = \frac{\text{relevant IA} + \text{PA} - \text{IA implied by PA}}{\text{PA}}$$

- The strengthening factor of PA over IA $\alpha_2$ reflects how much stronger IA become when PA are added. It is calculated by adding the numbers of relevant IA and PA, subtracting the PA implied by IA, and dividing the result by the number of relevant IA:
  $$\alpha_2 = \frac{\text{PA} + \text{relevant IA} - \text{PA implied by IA}}{\text{relevant IA}}$$

Figure 5. Percentage of program points where there are no programmer-written assertions, but relevant assertions were inferred.

Table 6. A comparison of the strengthening factors.

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

<table>
<thead>
<tr>
<th></th>
<th>Small TS</th>
<th>Medium TS</th>
<th>Large TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop invariants</td>
<td>12.56</td>
<td>13.39</td>
<td>14.22</td>
</tr>
<tr>
<td>Preconditions</td>
<td>1.11</td>
<td>1.13</td>
<td>1.24</td>
</tr>
<tr>
<td>Postconditions &amp; class invariants</td>
<td>5.31</td>
<td>6.13</td>
<td>5.87</td>
</tr>
<tr>
<td>Total</td>
<td>5.17</td>
<td>6.06</td>
<td>6.03</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Small TS</th>
<th>Medium TS</th>
<th>Large TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop invariants</td>
<td>1.03</td>
<td>1.03</td>
<td>1.03</td>
</tr>
<tr>
<td>Preconditions</td>
<td>2.16</td>
<td>1.22</td>
<td>1.19</td>
</tr>
<tr>
<td>Postconditions &amp; class invariants</td>
<td>2.15</td>
<td>1.42</td>
<td>1.43</td>
</tr>
<tr>
<td>Total</td>
<td>1.65</td>
<td>1.20</td>
<td>1.20</td>
</tr>
</tbody>
</table>

Values strictly greater than 1.00 for these factors mean that strengthening occurs.

Table 6(a) shows the averages for $\alpha_1$ for loop invariants, preconditions, postconditions and class invariants, and the averages for $\alpha_1$ for all assertions. It shows that IA can strengthen PA, but the strengthening factors for preconditions are generally much lower than those for postconditions and class invariants.

Table 6(b) shows the averages for $\alpha_2$ for loop invariants, preconditions, postconditions and class invariants, and the averages for $\alpha_2$ for all assertions. It shows that PA can strengthen IA, but to a lesser degree than the other way around.
Discussion

Daikon is good at inferring simple and frequently used assertions such as `an_argument /= Void` or `an_attribute = an_argument`, but at present it cannot compete with humans on higher levels of abstraction.

Daikon frequently infers theorems — assertions that follow logically from other assertion — 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; an example is not is_empty in the postcondition of the routine `put` of class `STACK`, which pushes an element on the stack.

The results show that inferred assertions can be used to strengthen programmer-written contracts (postconditions and invariants) and that inferred assertions can be sometimes used to correct existing contracts (strengthening preconditions). Still, not all PA are inferred or implied by the IA; and at about 23% of program points the programmer found nothing to specify while Daikon could infer relevant assertions. So although inferred assertions strengthen programmer contracts to a greater extent than conversely, automated assertion inference does not find a superset of the PA.

A methodological observation is appropriate here. 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. Assertion 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. So even if an contract inference tool could find all assertions written by programmers, such a tool would not completely replace the manual work of writing specifications, but should only serve to strengthen existing contracts upon completion of the implementation.

4.3. Correlations

In trying to establish which properties of the classes may influence inferred contracts quality, the study examined correlations between class metrics and the quality of contracts inferred for each class. All correlations listed below were computed for the experiment results using the Pearson product-moment correlation coefficient and calculated for the set of assertions inferred for the medium-size test suite, including the generated loop invariants. We define as strong correlations those for which the correlation coefficient is between 0.7 and 1.0 for positive correlations and -1.0 and -0.7 for negative correlations. We define as medium correlations those for which the correlation coefficient is between 0.5 and 0.7 for positive correlations and -0.7 and -0.5 for negative correlations.

Correctness and relevancy of IA have strong negative correlations, -0.95 and -0.69 respectively, to the total number of IA.

These two measures also have strong negative correlations to the number of integer zero-argument queries in a class (-0.81 and -0.73 respectively). Integer queries with no arguments increase Daikon’s assertion search space significantly, because Daikon has many assertion templates for integer variables, some of these templates involving relations between 2 or 3 variables. The strong positive correlation (0.76) between the number of integer queries with no arguments and the total number of IA also shows this.

The number of immediate routines of a class (routines implemented in the class itself, not inherited) also correlates negatively to the correctness (-0.59) and relevancy (-0.55) of the IA. Since this number determines the number of program points where assertions can be inferred, it is another source of increase in the size of the search space for assertions. The positive correlation (0.62) between the total number of IA and the number of immediate routines of a class also shows this.

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

4.4. 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.

Furthermore, the study only involved unit testing library classes; testing entire applications may produce different results. Examining classes from applications written by other developers is difficult mainly because the semantics of such classes is often clear only to the programmers who originally wrote the code, making it harder to devise good test cases. The solution is to involve application developers, which was not done for the present study.

As shown both by the results of this study and of previous investigations [18, 9], the quality and size of the test suite have a strong influence on the quality of the inferred contracts. We ran the experiment for three 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 Eiffel front-end we developed for it.

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 [17] showed that Daikon produces, even from relatively small test suites, assertions that are consistent and sufficient for automatic verification (proving the absence of runtime errors) with very little change. Nimmer and Ernst [18] also showed that test cases that mainly exercise corner cases are not suited for assertion inference. Gupta et al. [9] 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 assertions than random.

A study of users’ experience with Daikon [18] showed that using Daikon neither speeds up nor slows down users trying to annotate programs with contracts, but improves recall (how many assertions from 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 assertions as support for finding others. More than half the users found removing incorrect assertions 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 assertion 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 [19] tool uses contracts inferred by Daikon as filters for invalid inputs and as automated oracle. Xie and Notkin [22] 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 [7] 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 [6]. Substra [23] generates integration tests based on Daikon-inferred constraints on component interfaces.

DIDUCE [10] 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. [20] developed the Carrot assertion detector which uses the same principles as Daikon, but has a different implementation. Other work [11, 12] investigates dynamic inference techniques for algebraic specifications.

Some of the ideas developed in academic research on assertion inference were also adopted by industry. AgitatorOne [3], 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 Axion Meister tool [21] 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 inferred assertions are correct (reflect true properties of the source code): around 90% for the medium and large test suites.

- A high proportion of inferred assertions are also relevant (correct and interesting): around 65% for the medium and large test suites.

- Assertion inference can be used to strengthen programmer-written contracts, as shown by a strengthening factor averaging 6 for the medium and large test suites.

- Assertion inference cannot find all contracts written by programmers; this is evidenced by an average recall value of 0.5, meaning that only around half of the programmer-written assertions are part of the inferred assertions or implied by them.

- The quality of inferred contracts decreases with the size of the search space: the more zero-argument integer queries and the more routines a class contains, the more incorrect and uninteresting assertions are inferred. (Queries with no arguments are key elements
inferred assertions and the number of routines directly influences the number of program points where assertions can be inferred.)

These results suggest that assertion inference cannot completely replace the manual work of writing contracts, nor should it: contracts are a support in software development starting from its very first phases of requirements engineering and analysis, and the task of writing them should not be postponed to the stage when a full implementation of the system is already available. Nevertheless, at this stage assertion inference tools can be used to strengthen the existing programmer-written contracts, resulting in more accurate specification. This comes at a price: test suites are necessary for exercising the system and programmers still need to sort out the irrelevant assertions (around half of all generated ones).

Future work includes improving both Daikon and its Eiffel front-end, based on the insights gained through this study. A promising idea that we intend to explore is “push-button inference”: using automated testing tools such as AutoTest [16] to generate the test suites necessary for the assertion inference instead of handmade tests. Another direction for future work is to explore the use of assertion 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.

References