Are Practitioners Writing Contracts?

Patrice Chalin

Dept. of Computer Science and Software Engineering,
Dependable Software Research Group, Concordia University
chalin@cse.concordia.ca

Abstract. For decades now, modular design methodologies have helped software engineers cope with the size and complexity of modern-day industrial applications. To be truly effective though, it is essential that module interfaces be rigorously specified. Design by Contract (DBC) is an increasingly popular method of interface specification for object-oriented systems. Many researchers are actively adding support for DBC to various languages such as Ada, Java and C#. Are these research efforts justified? Does having support for DBC mean that developers will make use of it? We present the results of an empirical study measuring the proportion of assertion statements used in Eiffel contracts. The study results indicate that programmers using Eiffel (the only active language with built-in support for DBC) tend to write assertions in a proportion that is higher than for other languages.

Keywords: design by contract, program assertions, empirical study, Eiffel.

1 Introduction

It is generally accepted that there is no silver bullet and that there probably never will be; the challenges faced by software engineers will be alleviated by a combination of techniques. One of the effective ways that software engineers have found to manage the size and complexity of modern-day software systems is to use a modular-design methodology. An appropriate partitioning of a system into modules (e.g., libraries, classes) offers an effective means of managing complexity while providing opportunities for reuse. But when applied to large industrial applications in general and fault-tolerant systems in particular, modular design methods can only be truly effective if module interfaces are rigorously defined.

An increasingly popular approach to interface specification for object-oriented software is design by contract (DBC) [19-21]. Support for DBC is built in to the Eiffel programming language. Although Eiffel is the only active language with integrated support for DBC, researchers are currently busy adding DBC support to other languages. Generally, this added support is achieved by extending a subset of the target language. For example,

- SPARK for Ada [1],
- Spec# for C# [2],
- JACK for JavaCard [5],
- Java Modeling Language (JML) [4], Jass [3], Jcontract [22] for Java.

Are such research efforts justified? Does having built-in support for DBC mean that developers will write contracts? In an attempt to provide initial answers to these questions we undertook an empirical study of the use of contracts in Eiffel. More specifically, we sought to measure the proportion of source lines of code that are assertions because program assertions are the main ingredient of contracts, and they are easy to quantify. Why did we choose Eiffel programs as the subject of our study? Eiffel is the only active programming language with built in support for DBC, and this since its inception two decades ago. Hence, it is the only language for which there is a sufficiently large code base to sample.

In the next section, we explain the relationship between assertions, DBC and behavioral interface specifications. A brief review of Eiffel is also given, thus providing the necessary background for an understanding of the metrics used in the study. An introduction to the study and an explanation of the metrics are given in Section 3. Section 4 provides the study results, and Section 5 discusses threats to validity. We conclude in Section 6.

2 Design by Contract and Eiffel

2.1 Assertions, DBC and Behavioral Interface Specifications

Design by contract (DBC) refers to a *method* of developing object-oriented software defined by Bertrand Meyer [19, 20]. The main concept that underlies DBC is the notion of a precise and formally specified agreement between a class and its clients. Such an agreement, named a *contract* in DBC, is called a *behavioral interface specification* (BIS) in its most general form [26]. Contracts and BISs are built from class invariants, method pre- and post-conditions, (and other constructs) which are expressed by means of *program assertions*.

DBC as a programming language feature refers to a limited form of support for BISs where assertions are restricted to be expressions that are *executable*. Hence, for example, in Meyer's Eiffel programming language an assertion is merely a Boolean expression (that possibly makes use of the special `old` operator[1]). Meyer clearly identifies this as an *engineering tradeoff* in the language design of Eiffel [20]—a tradeoff that we believe is an important stepping stone from the current use of (plain) assertions in industry to the longer-term objective of the industrial adoption of verifying compilers [17]. It is understood that this engineering tradeoff imposes a limit on the expressiveness of Eiffel assertions (e.g. absence of quantifiers[2]) but, at the same time, we also believe that it is precisely this tradeoff that has kept them accessible to practitioners. We stress that it is the individual assertion expressions that are restricted to being executable, not the contracts. Hence, for example, a method contract might not be executable if its postcondition describes properties of the method result rather than how it can be computed.

[1] `old` *e* refers to the pre-state value of *e*, and can only occur in postconditions.

[2] This exclusion is due not to the quantifiers per se, but rather to the possibility of allowing quantified expressions with bound variables ranging over arbitrarily large or infinite collections.

How are contracts currently used in practice? A principal use for contracts, other than for documentation, is run-time assertion checking (RAC) [6]. All current systems supporting DBC also support RAC. When RAC is enabled, assertions are evaluated at run-time and an exception is thrown if an assertion fails to evaluate to true. Various degrees of checking can be enabled—e.g. from the evaluation of preconditions only, to the evaluation of all assertions, including preconditions, postconditions, invariants and inline assertions. Enabling RAC during testing, particularly integration testing, is an effective means of detecting bugs in modules and thus can help contribute to the increase in overall system quality.

Of course, for most applications, particularly fault-tolerant, safety- and security-critical systems, it is preferable to be able to guarantee the absence of assertion failures before a component is run. Extended Static Checking (ESC) [11] and Verified DBC (VDBC) [10, 25] tools can be used for this purpose. Such tools attempt to determine the validity of assertions by static analysis. ESC tools exist for Modula-3 and Java [9, 14], and one is currently under development for Eiffel. VDBC tools include Omnibus [25] and PerfectDeveloper [10].

2.2 Eiffel: A Brief Review

A sample Eiffel class taken from the Gobo Eiffel kernel library is given in Figure 1. Lines too long to fit on the page have been truncated and suffixed with ellipses ("…"). Classes optionally begin (and/or end) with an *indexing clause* that offers information about the class. In other languages this is often accomplished by using a comment block. Comments, like in Ada, start with a "`--`" and run until the end of the line. An Eiffel class generally declares a collection of *features* (attributes and "methods"). The given sample class declares only one feature, an *n*-ary exclusive or, `nxor`.

Of main concern to us here are assertions. An *assertion* in Eiffel is written as a collection of one or more optionally tagged assertion clauses. The meaning of an assertion is the conditional conjunction of its assertion clauses [12]. The tags can help readability and debugging since they can be output when the clause is violated [21]. Tags `zero`, `unary` and `binary` adorn lines 40, 41 and 42 of Figure 1, respectively.

An assertion clause is either a

- Boolean expression (as given in lines 40, 41 and 42) or a
- comment (e.g. line 43).

Such comments are called *informal assertions*. Eiffel's Boolean operators consist of the usual negation (`not`), conjunction (`and`) and disjunction (`or`) as well as conditional (i.e. short-circuited) conjunction (`and then`) and disjunction (`or else`). The implication, *a* `implies` *b*, is an abbreviation for (`not` *a*) `or else` *b*. Assertions can contain calls to methods identified as queries. A particular characteristic of a query is that it is not permitted to have side effects [21].

In Eiffel, an assertion can be used to express a

- precondition (introduced by the keyword `require`),
- postcondition (`ensure`),

```
1    indexing
2
3     description:
4
5          "Routines that ought to be in class BOOLEAN"
6
7     library: "Gobo Eiffel Kernel Library"
8     copyright: "Copyright  (c) 2002, Berend de Boer and others"
...   Lines 9 to 11 have been removed.
12
13   class KL_BOOLEAN_ROUTINES
14
15   feature -- Access
16
17    nxor (a_booleans: ARRAY [BOOLEAN]): BOOLEAN is
18              -- N-ary exclusive or
19         require
20              a_booleans_not_void: a_booleans /= Void
21         local
22              i, nb: INTEGER
23         do
24              i := a_booleans.lower
25              nb := a_booleans.upper
26              from until i > nb loop
...                              Lines 27 to 37 have been removed.
38              end
39         ensure
40              zero: a_booleans.count = 0 implies not Result
41              unary: a_booleans.count = 1 implies Result = ...
42              binary: a_booleans.count = 2 implies Result = ...
43              -- more: there exists one and only one `i' in ...
44         end
45   end
```

Fig. 1. Sample Eiffel class (kl_boolean_routines.e)

- class invariant (`invariant`),
- loop invariant (`invariant`),
- inline assertion (`check`)

A sample precondition is given in line 20 of Figure 1. The sample postcondition (lines 40-43) illustrates the use of more than one assertion clause. Assertions in postconditions can contain occurrences of the special operator `old`. For example, the postcondition

```
ensure  count = old count + 1
```

will be true when the post-state value of `count` is one more than the pre-state value of `count`. A `check` is equivalent to an `assert` statement in other languages such a C, C++ and Java.

There is only one looping construct in Eiffel, and it has the general form given in Figure 2. As was previously mentioned, an assertion can be used to express a loop invariant. Also, of interest is the loop variant: an integer expression that must decrease through every iteration of the loop while remaining nonnegative. That covers the basics of what we need to be able to explain the study metrics.

```
from
  init_instructions
invariant
  assertion
variant
  variant
until
  exit_condition
loop
  loop_instructions
end
```

Fig. 2. Eiffel loop instruction

3 Study

3.1 Objectives and Hypotheses

Given a language like Eiffel, with built-in support for DBC, our objective has been to measure the extent to which developers actually write contracts for their classes. Since program assertions are the basic ingredient of contracts and since it is relatively straightforward to count assertions, we chose this as a basic metric for our study. In addition to counting assertions we will also categorize them by kind—e.g. preconditions, postconditions, etc. vs. ordinary inline assertions. Our main study hypotheses are the following:

> (H1) Developers using a programming language with built in support for DBC will write program assertions in a proportion that is higher than for languages not supporting DBC.
>
> (H2) Furthermore, assertions will be used as part of contracts in a proportion that is higher than their use as inline assertions.

3.2 Projects

During the initial portion of our study we gathered metrics from free Eiffel software, consisting of both free commercial software (such as the sources distributed with EiffelStudio) as well as open source projects. This allowed us to conduct a pilot study during which we fine tuned our metrics gathering tool. This was essential before embarking on the second phase of the study in which we solicited the participation of industry.

In the second phase of our study, we posted announcements in the *EiffelWorld* newsletter [7]—published monthly by *Eiffel Software*, the makers of EiffelStudio—as well as Eiffel mailing lists and bulletin boards, inviting developers of commercial and open source Eiffel applications to contribute to the study. The invitation directed developers to a web site managed by our research group where the purpose of the study is explained and instructions for participation are given. After filling in a consent form, developers are provided with a script to run on their Eiffel code. The script generates a metrics file which participants subsequently upload to the study site. Finally, the identity of submitters is confirmed by means of an acknowledgement e-mail.

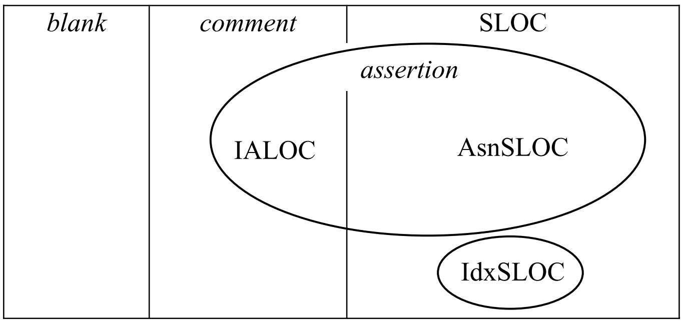

Fig. 3. Categorization of Eiffel LOC

3.3 Definition of Metrics

Our basic metric is a count of Lines of Code (LOC) per class file. As can be seen in Figure 3, each LOC is categorized at the top-level either as a

- blank line, containing at most white space
- comment line, containing a comment possibly preceded by white space
- (physical) Source Line of Code (SLOC) [23].

An illustration of the top-level categorization of the sample Eiffel class of Figure 1 is given in Figure 4.

One of our main statistics is a measure of the proportion of LOC that are assertions. The computation of this ratio is slightly complicated by the existence in Eiffel of informal assertions and index blocks, as we explain next.

An enumeration of the kinds of assertion that are supported by Eiffel is given in Figure 5. Note that we chose to include loop variant expressions as a kind of assertion, since it contributes, like the loop invariant, to the overall specification of the loop instruction.

As was explained in Section 2.2, an assertion can take the form of a source statement (**AsnSLOC**) or a comment. The latter is called an informal assertion (**IALOC**)—see line 43 of Figure 4 for an example. Hence,

AsnLOC = AsnSLOC + IALOC

The lines in Eiffel indexing clauses (identified as **IdxSLOC** in Figure 3), though technically SLOC, merely provide documentation for a class in a manner that is handled by a comment block in other languages. We therefore define an "adjusted SLOC" metric as

AdjSLOC = SLOC – IdxSLOC + IALOC

so we can simply and accurately define the proportion of lines that are assertions as

AsnProp = AsnLOC / AdjSLOC

1	SLOC	idx	**indexing**
2	blank		
3	SLOC	idx	description:
4	blank		
5	SLOC	idx	"Routines that ought to be in class BOOLEAN"
6	blank		
7	SLOC	idx	library: "Gobo Eiffel Kernel Library"
8	SLOC	idx	copyright: "Copyright (c) 2002, Berend de Boer and others"
…	…	…	*Lines 9 to 11 have been removed.*
12	blank		
13	SLOC		**class** KL_BOOLEAN_ROUTINES
14	blank		
15	SLOC		**feature** -- Access
16	blank		
17	SLOC		nxor (a_booleans: ARRAY [BOOLEAN]): BOOLEAN is
18	comment		-- N-ary exclusive or
19	SLOC		**require**
20	SLOC	req	a_booleans_not_void: a_booleans /= Void
21	SLOC		**local**
22	SLOC		i, nb: INTEGER
23	SLOC		**do**
24	SLOC		i := a_booleans.lower
25	SLOC		nb := a_booleans.upper
26	SLOC		**from until** i > nb **loop**
…	…	…	*Lines 27 to 37 have been removed.*
38	SLOC		**end**
39	SLOC		**ensure**
40	SLOC	ens	*zero*: a_booleans.count = 0 implies not Result
41	SLOC	ens	*unary*: a_booleans.count = 1 implies Result = …
42	SLOC	ens	*binary*: a_booleans.count = 2 implies Result = …
43	comment	ens	-- more: there exists one and only one `i' in …
44	SLOC		**end**
45	SLOC		**end**

Fig. 4. LOC categorization for our sample (kl_boolean_routines.e)

Statement	Use to express …	AsnLOC qualifier
`require`	preconditions	`Req`
`ensure`	postconditions	`Ens`
`invariant` (class)	class invariants	`Inv`
`invariant` (loop)	loop invariant	`invL`
`variant` (loop)	loop variant	`varL`
`check`	inline assertion	`chk`

Fig. 5. Kinds of assertion

We will keep separate AsnLOC$_a$ counts for each kind of assertion a (see Figure 5); we note that:

$$\text{AsnLOC} = \text{AsnLOC}_{req} + \text{AsnLOC}_{ens} + \ldots + \text{AsnLOC}_{chk}$$

Table 1. Number of projects, classes and LOC

Project Category	Number of projects	Number of classes	LOC (10^6)	% of total LOC
Proprietary	5	28 149	4.4	55%
Open Source	79	15 986	2.7	33%
EiffelStudio L&S	1	4 373	0.9	11%
Total	85	48 508	7.9	100%

3.4 Metrics Gathering Tool

At first we used the SLOCCount tool [24] as our base. This tool can count physical SLOC for over two-dozen languages—though initially not for Eiffel. Aside from its ability to process many different kinds of languages SLOCCount also does convenient house-keeping tasks such as determining the type of a file (by its extension or content), flagging duplicates and ignoring generated files.

Since our needs were specific to Eiffel source, we eventually chose to use a single Perl script to gather all metrics. The creation of the script did pose some challenges due, e.g., to the various flavors of Eiffel (as supported by different compilers) and inconsistent line endings (Unix, DOS or Mac) sometimes within the same file, as well as the variation in lexical rules used for multi-line string literals.

4 Results

4.1 General

As can be seen from Table 1, the study covered 85 projects totaling 48 508 Eiffel classes and 7.9 million lines of code (MLOC). The projects included applications from the areas of databases, developer tools, finance/HR, games, modeling, middleware, networking, scientific computing, systems software, utility library/toolkits, visualization and web applications. We divided the projects into three categories:

- proprietary (accounting for 55% of the code of the study),
- open source (33%), and the
- library and samples shipped with EiffelStudio 5.5 (11%).

Note that half of the files in the EiffelStudio category consist of open source samples (or what they call "free add-ons"), most of which are provided by GoboSoft—an important contributor of open source Eiffel libraries and tools. Nonetheless GoboSoft add-on files were counted in the EiffelStudio category only. We separated out EiffelStudio (libraries and samples) into its own category because we expected it to have the highest proportion of assertions.

The breakdown (partitioning) of LOC into SLOC, blank lines and comments is given in Table 2. We see that 74% of LOC are physical source lines of code. On average, the classes in our study contained 163 LOC (120 SLOC). The table also

Table 2. Breakdown of LOC into SLOC, blank and comment lines

	SLOC	blank	comment	Total	IdxSLOC	IALOC	AdjSLOC
LOC (10^6)	5.8	1.3	0.83	7.9	0.25	0.014	5.6
% of total LOC	74%	16%	10%	100%	3.2%	0.17%	71%
Average	120	26	17	163	5	0.3	115

Table 3. Assertion metrics by kind

	Assertion kind, $a \rightarrow$	require	ensure	class inv	loop inv	loop var	check	Total
(a)	AsnLOC_a	138 960	111 420	19 794	745	705	8 563	280 187
(b)	AsnLOC_a/AdjSLOC	**2.5%**	2.0%	0.35%	0.013%	0.013%	0.15%	**5.0%**
(c)	AsnLOC_a/AsnLOC	**50%**	40%	7.1%	0.27%	0.25%	3.1%	100%
(d)	max AsnLOC_a/AsnLOC	**56%**	49%	52%	11%	5%	33%	-
(e)	avg. AsnLOC_a / file	**2.9**	2.3	0.4	0.0	0.0	0.2	5.8
(f)	no. of statements (stmt)	83 712	69 144	8 671	412	694	7 005	169 638
(g)	avg. AsnLOC_a / stmt	1.6	1.6	**2.3**	1.8	1.0	1.2	-
(h)	max AsnLOC_a/ stmt	30	**84**	**79**	12	3	25	-
(i)	IALOC_a	1595	9 752	1 742	104	5	558	13 756
(j)	IALOC_a/AdjSLOC	0.03%	**0.17%**	0.03%	0.00%	0.00%	0.01%	0.25%
(k)	IALOC_a/AsnLOC	0.57%	**3.5%**	0.62%	0.04%	0.00%	0.20%	4.9%
(l)	count (e/=Void)	63 003	22 187	9 672	9	0	2 811	97 682
(m)	% (e/=Void)	45%	20%	49%	1.2%	0.00%	33%	35%

provides the value of AdjSLOC, namely 5.6 MLOC, which is defined to be the number of SLOC excluding indexing clause lines but including informal assertions (cf. Section 3.3). This adjusted SLOC count is the valued used in measuring the proportion of assertions.

4.2 Assertion Metrics

The metrics concerning assertions are summarized in Table 3. We highlight some of the most interesting results. For ease of reference, we have labeled the rows of the table from (*a*) to (*m*). Looking at the Total column for rows (*a*) and (*b*) we see that there were 0.28 MLOC of assertions. Hence, out of the 5.6 MLOC of adjusted SLOC previously mentioned, overall 5.0% of the LOC were assertions.

Row (*c*) of Table 3 gives the distribution of assertions by kind, which is also graphically illustrated in Figure 6. Assertions are mostly used to document preconditions (50%), postconditions (40%) and class invariants (7.1%). Few loop invariants and variants are given, though both of these appear almost as frequently relative to each other. The low frequency of loop invariants and variants may be a testimony to the high degree of challenge associated with writing useful loop invariants and variants. Remarkably only 3.1% of the assertions (0.15% of the overall AdjSLOC) were inline checks.

Recall that the various kinds of assertion statement can contain more than one assertion line. The average number assertion lines per statement (*g*) ranges from 1.0 to 2.3, while the average number of assertions per file (*e*) is 5.8. While preconditions occur most frequently, class invariants have the largest number of assertions

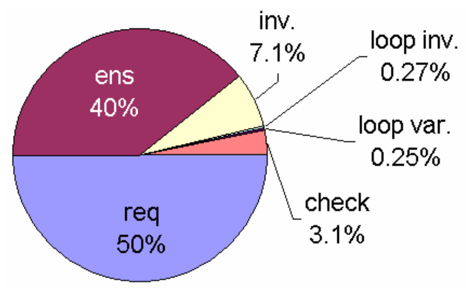

Fig. 6. Distribution of assertions by kind (all project categories)

per statement (2.3). This suggests that class invariants, when written, express more complex conditions since on average, it requires twice as many assertions to express a class invariant than a precondition. The maximum number of assertions per clause (*h*) can be fairly large, e.g. up to 79 LOC for a class invariant and 84 LOC for a postcondition.

We note that a very small proportion of assertions are given in the form of comments. Overall, only 0.25% of the AdjSLOC and 4.9% of assertion LOC are informal assertions (*j*), (*k*). Informal assertions are used most frequently in postconditions (3.5% of AsnSLOC$_{ens}$). We expect this to be the case either because (i) some aspect of the postcondition may be too complex to express as an assertion—e.g. it may require quantifiers—or, (ii) developers do not want the overhead of full postcondition evaluation during run-time checking and choose express as comments those predicates that would be too computationally intensive.

A noteworthy proportion of assertions include subexpressions of the form *e* `/= Void`, stating that a given reference is not `Void` (i.e. null). This number is close to 50% for class invariants and 35% overall (*m*). These figures provide some weight to the choice made by a number of language designers and static analysis tools (such as Splint [13]) which consider a reference type declaration to be non-null by default. In fact, we recently completed a more detailed study that indicates that well over 50% of reference type declarations in Java are meant to be non-null [8]. In the newly released ECMA Eiffel standard, the notions of attached and detachable types are introduced. An identifier of an attached type is guaranteed to always be bound to an object, i.e., it cannot be `Void`/null. The standard mandates that types are attached by default; to indicate a detachable version of a type *T* one prefixes the type name with a question mark: ?*T* [12].

What was the distribution of AsnProp? A little over half (52.4%) of the classes in the study contained no assertions. We note that a class without assertions can still have a contract, since subclasses inherit contracts from their superclasses (but detecting and quantifying such implicit contracts is outside the scope of this study). The distribution of the files with a nonzero AsnProp is given in Figure 7. The highest proportion of files (11%) had an AsnProp in the range 2.5% to 5%. A third of the files had an AsnProp between 0 and 12.5%. Figure 8 shows the number of projects with an average proportion of assertions in a given range. Two projects had no assertions, while the majority of projects had between 1.5% and 7% of assertions per adjusted SLOC.

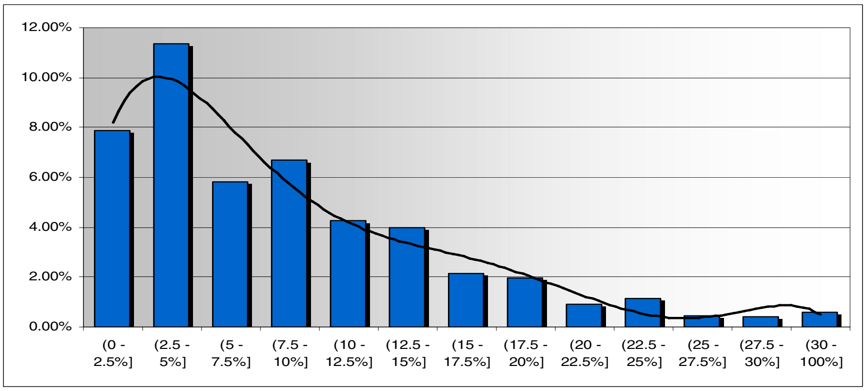

Fig. 7. Percentage of files with AsnProp in a given range

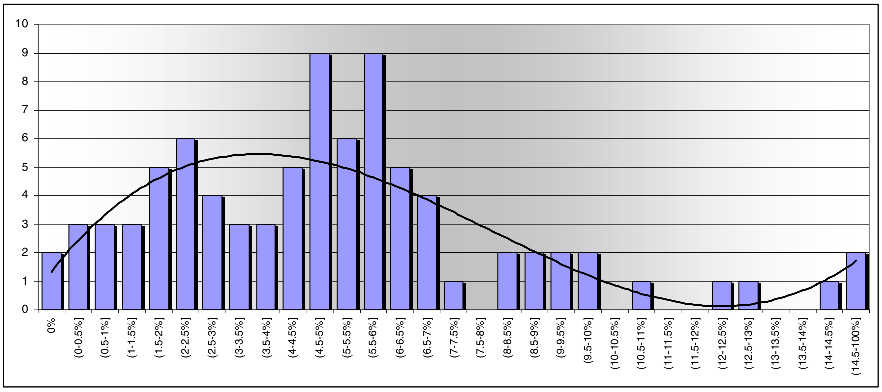

Fig. 8. Number of projects with AsnProp in a given range

Table 4 shows how the proportion of LOC that are assertions varies by project category. As might be expected, the EiffelStudio category has the highest proportion, 6.7%, followed by open source projects and proprietary code with 5.8% and 4.2%, respectively. (Recall that the open source category *excludes* GoboSoft software because it is counted in the EiffelStudio project category.)

Table 4. Proportion of AsnLOCs per project category

Project Category	SLOC (10^6)	AdjSLOC (10^6)	AsnLOC (10^6)	AsnLOC / AdjSLOC
Proprietary	3.3	3.2	0.13	4.2%
Open Source	1.9	1.8	0.11	5.8%
EiffelStudio L&S	0.62	0.59	0.04	6.7%
Total	5.8	5.6	0.28	5.0%

5 Threats to Validity

5.1 Internal Validity

The most significant potential source of error is in the measurement of metrics because the metrics are gathered by a script that uses keyword-based pattern matching rather than a true Eiffel parser. This was deemed the only practical approach because study samples were written in several different variants of Eiffel; with the variability being due to differences in the language as supported by different compilers or even to changes in the language introduced over time. Since none of the current Eiffel compilers support all variants, it seemed utterly impractical to attempt to build our own parser that would.

Due to the manner in which Eiffel makes use of keywords to delimit code blocks that can contain assertions, a keyword-based pattern matching approach turned out to be not only feasible but also (seemingly) quite accurate. Our confidence was boosted by the use of an inclusive test suite and by the fact that a comprehensive set of sanity checks have been build into the script—we have run the script on over 5 million LOC without it reporting errors.

Another aspect which could have biased the study results would be for a file's data to have been counted more than once. This would be likely to occur when the code of an open source library was used in multiple projects. To guard against this, the script used to compute the study metrics was also designed to generate a 32 bit hash code for each file based on the file content. In computing the final statistics we retained at most one file with the given hash code.

5.2 External Validity

Were the projects used in the study representative of typical Eiffel software? In the first phase of the study we obtained projects from SourceForge and other sites dedicated to open source Eiffel software. Our only selection criterion was for projects to appear to be active; we believe that this is reasonable. In the second phase of the study, we solicited contributions from the Eiffel community. This resulted in 10 submissions, half of which were proprietary, though this half contributed 55% of the LOC for the study. With respect to the threat to validity, our main concern is whether the volunteered projects would have a proportion of assertions that is higher than average, hence unfairly contributing support towards our hypothesis. This cannot be ascertained, but we note that the proportion of assertions for proprietary code (4.2%) was in fact less than that for open source code (5.8%) and that all but five of the open source projects were chosen by us in phase one. It is clear though, that the relatively small size of the Eiffel user community, as compared, say, to that of C or C++, may also have some bearing on the study results—e.g. lesser variability.

Could similar results be expected to hold for other languages supporting DBC? One might argue that those who write applications in Eiffel have chosen Eiffel over other programming languages precisely because of its built-in support for DBC. Hence, the proportion of developers who are willing to write contracts may be higher in the case of Eiffel than for another programming language. Even if this was the case, the results offer the promise that such developers may well choose to adopt another programming language if DBC support were adequate.

6 Conclusion

In previous work, we were able to establish that the industrial use of assertions is fairly widespread [6]. The present study focuses on the use of assertions in Eiffel, the only active language supporting the disciplined use of assertions in specifying contracts, i.e. Design by Contract (DBC). Overall, 5.0% of the studied code consisted of assertions. Ninety-seven percent of these assertions were used in contracts rather than inline assertions (confirming our hypothesis H2). We are not aware of any other empirical studies that measure the use of assertions, but *estimated* figures are available. For example, Hoare estimates that 1% of the Microsoft Office Suite LOC are assertions [15, 16]. Participants of a survey that we recently conducted offered estimates with a mean of 3.2% [6]. The results of the study reported here, allow us to confirm (H1) that Eiffel classes contain program assertions in a proportion that is higher than the use of assertions in programming languages not supporting DBC. In our opinion, this is good news for those researchers currently striving to add DBC support to other languages.

We expect that developers will be inclined to increase their use of assertions as other tools that process assertions and contracts become more mature and widely known—e.g. tools like JmlUnit that can automatically generate test oracles from JML specifications [18]. By design, DBC restricts the expressiveness of assertions by requiring that they be executable. We believe that this moderation in expressiveness is what will allow DBC to be more easily adopted by industry at large. It will then become a smaller step to reach the full expressiveness of behavioral interface specifications (BISs).

References

[1] J. Barnes, *High Integrity Software: The Spark Approach to Safety and Security*. Addison-Wesley, 2003.

[2] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview". In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean editors, *Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, Marseille, France, 2004, vol. 3362 of *LNCS*. Springer, 2004.

[3] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim, "Jass—Java with Assertions", *Electronic Notes in Theoretical Computer Science*, 55(2):103-117, 2001.

[4] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An Overview of JML Tools and Applications", *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212-232, 2005.

[5] L. Burdy, A. Requet, and J.-L. Lanet, "Java Applet Correctness: A Developer-Oriented Approach". *Proceedings of the International Symposium of Formal Methods Europe*, 2003, vol. 2805 of *LNCS*. Springer, 2003.

[6] P. Chalin, "Logical Foundations of Program Assertions: What do Practitioners Want?" *Proceedings of the Third International Conference on Software Engineering and Formal Methods (SEFM'05)*, Koblenz, Germany, September 5-9, 2005. IEEE Computer Society Press, 2005.

[7] P. Chalin, "DbC and assertions in Eiffel: participants needed for quantitative research survey", *EiffelWorld Electronic Newsletter*, 32(2), 2006.
[8] P. Chalin and F. Rioux, "Non-null References by Default in the Java Modeling Language". *Workshop on the Specification and Verification of Component-Based Systems (SAVCBS'05)*, Lisbon, Portugal, Sept., 2005. ACM Press, 2005.
[9] D. R. Cok and J. R. Kiniry, "ESC/Java2: Uniting ESC/Java and JML". In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean editors, *Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, Marseille, France, March 10-14, 2004, vol. 3362 of *LNCS*, pp. 108-128. Springer, 2004.
[10] D. Crocker, "Safe Object-Oriented Software: The Verified Design-By-Contract Paradigm". *Practical Elements of Safety: Proceedings of the 12th Safety-Critical Systems Symposium*, Birmingham, UK, February, 2004. Springer, 2004.
[11] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe, "Extended Static Checking", Compaq Systems Research Center, Research Report 159. December, 1998.
[12] ECMA International, "Eiffel Analysis, Design and Programming Language", ECMA-367. June 2005.
[13] D. Evans, "Splint User Manual", Secure Programming Group, University of Virginia. June 5, 2003.
[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java". *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, June, 2002, vol. 37(5), pp. 234-245. ACM Press, 2002.
[15] C.A.R. Hoare, "Assertions: Progress and Prospects", http://research.microsoft. com/~thoare, 2001.
[16] C. A. R. Hoare, "Assertions: A Personal Perspective", *IEEE Annals of the History of Computing*, 25(2):14-25, 2003.
[17] C. A. R. Hoare, "The Verifying Compiler: A Grand Challenge for Computing Research", *JACM*, 50(1):63-69, 2003.
[18] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs, "JML: Notations and Tools Supporting Detailed Design in Java", in *OOPSLA 2000 Companion, Minneapolis, Minnesota*, 2000, pp. 105-106.
[19] B. Meyer, "Applying Design by Contract", *Computer*, 25(10):40-51, 1992.
[20] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice-Hall, 1997.
[21] R. Mitchell and M. Jim, *Design by Contract, by Example*. Addison-Wesley, 2002.
[22] Parasoft, "Jcontract product page", www.parasoft.com, 2005.
[23] R. Park, "Software Size Measurement: A Framework for Counting Source Statements", CMU, Software Engineering Institute, Pittsburgh CMU/SEI-92-TR-20, 1992.
[24] D. A. Wheeler, "SLOCCount", www.dwheeler.com/sloccount, 2005.
[25] T. Wilson and S. Maharaj, "Omnibus: A clean language for supporting DBC, ESC and VDBC". *Proceedings of the Third International Conference on Software Engineering and Formal Methods (SEFM'05)*, Koblenz, Germany, September 5-9, 2005. IEEE Computer Society Press, 2005.
[26] J. M. Wing, "Writing Larch Interface Language Specifications", *ACM Trans. Program. Lang. Syst.* , 9(1):1-24, 1987.