

# Ensuring Continued Mainstream Use of Formal Methods: An Assessment, Roadmap and Issues

---

**Patrice Chalin**

[www.cs.concordia.ca/~chalin](http://www.cs.concordia.ca/~chalin)

Technical Report 2005-001, revision 2  
February 2005

Dependable Software Research Group  
Department of Computer Science and Software Engineering  
Faculty of Engineering and Computer Science  
**Concordia University**

**Keywords:** assertions, design by contract, runtime assertion checking, extended static checking, survey, industrial practice, bottom-up formal methods

**2001 CR Categories:** D.2.1 [Software Engineering] Requirements/ Specifications — languages, tools, JML; D.2.2 [Software Engineering] Design Tools and Techniques; D.2.4 [Software Engineering] Software/Program Verification; D.3.2 [Programming Languages] Language Classifications — Object-oriented languages; F.3.1 [Logics and Meanings of Programs] Specifying and Verifying and Reasoning about Programs.

## Revision History

- 2005, January. Initial release.
- 2005, February. Revision 2.

# Table of Contents

1	Introduction .....	1
2	Background and roadmap .....	2
	2.1 Bottom-up formal methods (and a Grand Challenge) .....	2
	2.2 Assertions, Design by Contract and Behavioral Interface Specifications .....	3
3	Current practice: an assessment .....	4
	3.1 Assertions .....	4
	3.2 Assertions in Eiffel: a survey .....	5
	3.2.1 Eiffel source .....	5
	3.2.2 Metrics .....	7
	3.2.3 Methodology .....	7
	3.2.4 Results .....	8
4	Issues .....	9
	4.1 ESC: Choice of logical foundations .....	10
	4.1.1 ESC .....	10
	4.1.2 Motivating example .....	10
	4.1.3 The problem .....	10
	4.1.4 Consequences .....	11
	4.2 Null references and non-null types .....	13
5	Comments on the survey results .....	13
6	Conclusion .....	14

# Ensuring Continued Mainstream Use of Formal Methods: An Assessment, Roadmap and Issues

Patrice Chalin

Dept. of Computer Science and Software Engineering,  
Dependable Software Research Group, Concordia University  
[www.cs.concordia.ca/~chalin](http://www.cs.concordia.ca/~chalin)

**Abstract.** Some feel that broad industrial adoption of formal methods or techniques has yet to occur. We believe that “bottom-up” formal methods—those that developers put to use directly in their code—are part of common practice, and we present initial survey results in support of this claim. Bottom-up formal methods are rooted in the use of *assertions*. Current use is mainly centered on debugging and testing, but almost forty years of research on assertions allows us to appreciate their greater potential. We identify milestones along a path from current practice to the longer term goal of mainstream use of Floyd/Hoare verifying compilers. In our view, the frontline of adoption today has reached runtime assertion checking of contracts. Our survey of the use of assertions in Eiffel helps provide weight to this opinion. Extended static checking (ESC) of contracts is the most natural and likely milestone after that. Finally, we identify research issues concerning ESC of contracts that will need to be carefully resolved, so that industry will continue to see a clear migration path from current practice to this next milestone, in a way that will maximize investments while minimizing costs.

**Keywords:** assertions, design by contract, runtime assertion checking, extended static checking, survey, industrial practice, bottom-up formal methods.

## 1 Introduction

Motivation for this paper was partly prompted by an underlying current in the discussions held during Industry Day (1-day) open sessions at FME’03 (and partly reflected in [SW03]). The general feeling was that, as formalists, we appear to have insufficient evidence to put forth in regards to one of our long held primary objectives: *widespread* adoption of formal methods, particularly by industry. The present author believes that there has been broad adoption of a particular kind of formal method (FM) that we will call bottom-up methods. In this paper, we lay a likely path (roadmap) for the continued successful migration of bottom-up FMs into industry. As a final milestone on this path we target mainstream use of Floyd/Hoare verifying compilers.

The three principle *contributions* of this paper are as follows. Firstly, we identify the importance of bottom-up methods and present a roadmap for their continued broad adoption by industry. We use this map to help us get our bearings

concerning where current practice is at and what the next steps will be with respect to adoption. Secondly, we present empirical data in support of the hypothesis of a broad use of bottom-up formal methods. Part of this data is from a survey that we have conducted on the use of assertions in Eiffel. As will become clear in the next section, the survey results also serve to confirm that the software development community is ready for the adoption of the next step of bottom-up FMs—use of contracts. Such results are important to researchers making advances in this and related areas. Our final contribution is an identification of issues whose resolution must be carefully thought out failing which we will lose the favor of industry.

## 2 Background and roadmap

### 2.1 Bottom-up formal methods (and a Grand Challenge)

Approaches to formal methods can be broadly classified into two categories: top-down and bottom-up. Top-down approaches are exemplified by the B method [Abrial96], for example, in which software applications (or their components) are derived stepwise from formal requirements descriptions. In theory, each refinement step can be subject to formal verification so that in the end one has formally correct software. Such developments are rarely done in industry since, e.g., the upfront costs of the approach are seldom justifiable. Bottom-up approaches essentially focus on formal methods that can be put to use by developers directly in their code. At the heart of these bottom-up approaches is the notion of assertion. Program assertions have been an active research topic for almost four decades. They are also a part of the basic toolset of the mainstream developer. For example, Hoare reports that the Microsoft Office suite contains a quarter of a million assertions. Unfortunately at Microsoft, as elsewhere in industry, assertions are mainly used for testing and debugging [Hoare03a]. The full potential of assertions has yet to be exploited.

In recognition of the high potential assertions, Hoare recently revived Floyd’s *verifying compiler* (VC) project as a **Grand Challenge for Computing Research** [Hoare03b]. One of the objectives of this challenge is to expand industrial use of assertions to include formal verification (and model checking) so that by the year 2020, use of verifying compilers will have become mainstream. Industrial adoption of VC technology is likely to proceed in stages as technological prototypes become mature enough to be integrated into commercial compilers. Mainstream use of assertions (and associated tool support) will occur in steps, quite possibly in the following order<sup>1</sup>:

- Testing and debugging. This includes both compile-time and runtime assertion checking (RAC).
- RAC support for Design by Contract (DBC),
- Extended static checking (ESC) support for DBC and, more generally,

---

<sup>1</sup> Though we believe that model checking will also be a feature of future verifying compilers, we do not discuss the topic further in this paper.

- ESC support for Behavioral Interface Specifications (BISs)<sup>2</sup>,
- Complete formal verification of BISs by a verifying compiler.

Of course, no tool will be able to provide automated proofs for all verification conditions generated from general BISs. On the other hand, we anticipate that tools will easily discharge mundane verification conditions while compiling a list of lemmas whose proof require human insight. This is in fact already done by the Spark tools, for example [Barnes03]. We also anticipate that proof carrying code will become common place, both at the byte code level [Necula97] as well for higher-level source code. Proof-carrying source code could offer a means by which BISs could be fully and automatically checked by a verifying compiler.

All modern day programming languages have some form of support for assertions even if only by means of a library rather than direct language support. We agree with Hoare that current practice mainly consists of the use of assertions for testing and debugging. A natural evolution of such a use of assertions is their more systematic application in expressing method pre- and post-conditions as well as class invariants—the essence of DBC. We believe that the frontline of adoption (and tool development) has reached DBC runtime assertion checking. The next step after that will be the use of DBC in extended static checking.

Unfortunately, very few programming languages have built-in support for DBC. To our knowledge, the only language expressly designed to support DBC that is still active is Eiffel [Meyer97]. Researchers are attempting to bring DBC support to other languages. Examples of projects include the following. For Java: the Java Modeling Language (JML) [Burdy+04], Jcontract [Parasoft05], iContract [Kramer98], JASS [BCMW01], jContractor [KHB98], and Handshake [DH98]. Of these only JML and Jcontract are still active. For C#: Spec# (and Boogie) [Barnett+04]; finally, Spark for Ada [Barnes03].

But, is this research in DBC warranted? Does having DBC support in a programming language mean that developers will write contracts or even make use of assertions? We believe that our Eiffel survey results tend to answer these questions affirmatively.

## 2.2 Assertions, Design by Contract and Behavioral Interface Specifications

Before moving on to the heart of the matter, we pause briefly to define the terms (program) assertion, design by contract and behavioral interface specification.

By a program assertion we understand a logical formula that is most often given in the form of a Boolean expression in an underlying programming language<sup>3</sup>. Program assertions can be evaluated at compile-time, or at runtime. They can also be used in formal reasoning about programs or program segments—as advocated, e.g., since the 60's by Floyd and Hoare [Jones03]. Formally, an assertion can be modeled either as a unary or a binary predicate over program

<sup>2</sup> We delay to until the next section a more detailed description of what DBC and BIS are.

<sup>3</sup> We use the term “programming language” quite liberally. E.g. in [HH98], the programming language is essentially a logic.

states. Binary predicates arise in one particular form of assertion expression modeling in which pre-state variables are referred to in method specification post-conditions.

Design by Contract (DBC) refers to a *method* of developing object-oriented software that was defined by Bertrand Meyer [Meyer97]. 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 *contract* in DBC, is called a *behavioral interface specification* (BIS) in its most general form. Contracts, like BISs, are expressed using assertions and take the form of class invariants and method pre- and post-conditions, among others.

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<sup>4</sup>). Meyer clearly identifies this as an *engineering tradeoff* in the language design of Eiffel [Meyer97]—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 adoption of verifying compilers. It is understood that this engineering tradeoff imposes a limit on the expressiveness of Eiffel assertions (e.g. absence of quantifiers<sup>5</sup>) but, at the same time we believe that it is precisely this tradeoff that has kept them accessible to practitioners.

### 3 Current practice: an assessment

#### 3.1 Assertions

To what extent are assertions used in practice? Hoare reports that approximately 1% of the Microsoft Office suite code consists of assertions, that is roughly 0.25 million lines of code (MLOC) out of a total of 25MLOC [Hoare00, Hoare03a].

We are in the early stages of conducting a general survey of the use of assertions in open source and proprietary (“closed source”) projects. The overall results of this study will be published elsewhere however, preliminary figures indicate, for example, that overall usage of assertions in Java is less than 1%.

An interesting question to ask is: how would these results compare to those for a language supporting DBC? As mentioned in Section 2, attempts are underway to add DBC support to languages not originally designed with such support. Are these efforts worthwhile? Does having built-in support for DBC mean that developers will write contracts or even assertions in general? In an attempt to provide some answers to these questions we focused our initial survey efforts on Eiffel projects.

---

<sup>4</sup> “old” operators can only occur in postconditions; “`old e`” refers to the pre-state value of  $e$ .

<sup>5</sup> 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.

indexing	1
	2
description:	3
	4
"Routines that ought to be in class BOOLEAN"	5
	6
library: "Gobo Eiffel Kernel Library"	7
copyright: "Copyright (c) 2002, Berend de Boer and others"	8
license: "Eiffel Forum License v2 (see forum.txt)"	9
date: "\$Date: 2003/02/07 12:49:18 \$"	10
revision: "\$Revision: 1.2 \$"	11
	12
class KL_BOOLEAN_ROUTINES	13
	14
feature -- Access	15
	16
nxor (a_bool_eans: ARRAY [BOOLEAN]): BOOLEAN is	17
-- N-ary exclusive or	18
<b>require</b>	19
a_bool_eans_not_void: a_bool_eans /= Void	20
local	21
i, nb: INTEGER	22
do	23
i := a_bool_eans.lower	24
nb := a_bool_eans.upper	25
from until i > nb loop	26
-- Lines 27 ... 37 removed	27
end	28
<b>ensure</b>	39
zero: a_bool_eans.count = 0 implies not Result	40
unary: a_bool_eans.count = 1 implies	>> 41
Result = a_bool_eans.item (a_bool_eans.lower)	
binary: a_bool_eans.count = 2 implies	>> 42
Result = (a_bool_eans.item (a_bool_eans.lower) xor	>>
a_bool_eans.item (a_bool_eans.upper))	
-- more: there exists one and only one 'i' in	>> 43
a_bool_ean.lower..a_bool_ean.upper so that	>>
a_bool_ean.item (i) = True	
end	44
end	45

Figure 1, Sample Eiffel class (kl\_boolean\_routines.e)

### 3.2 Assertions in Eiffel: a survey

In this section we describe our Eiffel survey and its results. We begin by very briefly reviewing Eiffel syntax before explaining the metrics gathered during the survey. A description of our methodology and survey results is provided after that.

#### 3.2.1 Eiffel source

A sample Eiffel class taken from the Gobo Eiffel kernel library is given in Figure 1. Some of the lines were too long to fit on the page and hence their content has been wrapped (and indented to aid in readability) at those points marked with >>.

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). Our sample class declared only one feature, an *n*-ary exclusive or, *nxor*.

Of main concern to us in this paper 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 conjunction (connected by the **and** operator<sup>6</sup>) of its assertion clauses. The tags can help readability and debugging (since they can be printed when the clause is violated) [Mitchell+02]. Tags **zero**, **unary** and **binary** adorn lines 40, 41 and 42 of Figure 1, respectively.

An assertion clause is either a

- boolean expression (e.g. line 40) or,
- comment (e.g. line 43).

As will be noted later we will count such comments as *informal assertion clauses*, or simply informal assertions. Boolean operators consist of the usual negation (**not**), conjunction (**and**), and disjunction (**or**). Eiffel also has conditional, i.e. short-circuited, conjunction (**and then**) and disjunction (**or else**). An implication operator *a implies b* is an abbreviation for (**not a**) **or else b**.

In Eiffel, an assertion can be used to express a

- precondition (introduced by the keyword **require**),
- postcondition (**ensure**),
- class invariant (**invariant**),
- loop invariant (**loop invariant**),
- check (**check**)

A sample precondition is given in lines 19-20 of Figure 1. The sample postcondition (lines 39-43) 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 pre-state value of **count** is one less than the post-state value of **count**. A **check** is equivalent to an **assert** in other languages such as Java and C++.

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.

---

<sup>6</sup> As opposed to the conditional conjunction operator “and then” which is mentioned later.

```

from
  initialization_instructions
invariant
  assertion
variant
  variant
until
  exit_condition
loop
  loop_instructions
end

```

**Figure 2, Eiffel loop instruction**

That essentially covers the basics of what we need to be able to explain the metrics.

### 3.2.2 Metrics

Our basic metric is a count of Lines of Code (LOC) per class file. Each LOC is classified as either:

- blank line, containing at most white space, or
- comment line, containing a comment possibly preceded by white space, or
- (physical) Source Line of Code (SLOC) [Park92].

Roughly speaking our goal is to count the number of LOC that are assertions (AsnLOC) so as to be able to determine their proportion relative to the total SLOC.

Our overall count of AsnLOC will be computed from the total SLOC that are assertions as well as the total LOC that are informal assertions (IALOC)—i.e. assertions given in the form of comments. We count informal assertions because we believe that they are just as important as formal assertions in documenting contracts. In measuring the proportion of LOC that are assertions we will use the following formula:

$$\text{total(AsnLOC)} / \text{total(AdjSLOC)}$$

where

$$\text{total(AdjSLOC)} = \text{total(SLOC)} + \text{total(IALOC)} - \text{total(IdxSLOC)}$$

IdxSLOC is a SLOC that occurs in an indexing clause. We omit IdxSLOC lines because these lines merely provide documentation for the class in a manner that is handled by a comment block in other languages. We will keep separate AsnLOC counts for preconditions, postconditions, class invariants, checks clauses and loop variants and invariants. This will allow us to determine the proportion of assertions used in each of these categories. We will also collect specialized metrics such as the number of assertions of the form *e /= Void*. Their purpose will be explained in Section 4.2.

### 3.2.3 Methodology

Initially we used the SLOCCount tool [Wheeler05] 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

Project Category	Number of files	LOC	SLOC	Proj.Cat.% (SLOC)
Proprietary	18584	2653032	2033945	51%
Open Source	10657	1761849	1307893	33%
Eiffel 5.5	4840	954001	663281	17%
<b>Total</b>	<b>34081</b>	<b>5368882</b>	<b>4005119</b>	<b>100%</b>

Figure 3, General metrics by project category

	LOC	SLOC	blank	comment	IdxSLOC
<b>Total</b>	5368882	4005119	817623	546140	170743
<b>% LOC</b>	100.00%	74.60%	15.23%	10.17%	3.18%

Figure 4, General metrics (all categories)

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. (We did borrow techniques from SLOCCount—like using md5 sums to flag duplicate files in different projects or directories.) 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 (typically Unix, DOS or Mac) sometimes in the same file.

### 3.2.4 Results

Overall we surveyed 81 projects totaling 34081 Eiffel class files, 5.4 million lines of code and 4.0 million source lines of code. We divided the projects into three categories:

- proprietary,
- open source and
- library and samples shipped with ISE Eiffel Studio 5.5.

Note that half of the files in the Eiffel 5.5 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. The proportion of SLOC per project category is given in Figure 3. Figure 4 provides the overall distribution of LOC into SLOC, blank lines and comments. The IdxSLOC is the proportion of SLOC that occur in indexing blocks.

Metrics concerning assertions are given in Figure 5. Overall, 4.39% of the LOC are assertions. Of this over 50% are used in preconditions, 36% postconditions, and 6.5% class invariants. Few loop invariants and variants are given, though both of these appear as frequently, relative to each other. We note that a very small proportion of assertions are given in the form of comments; i.e. overall, only 3.82% of assertion LOC are informal assertions. The maximum number of assertions per clause type can be fairly large (up to 35 LOC for a class invariant). The average number of assertions per clause type ranges from 1.0 to 2.4.

	require	ensure	inv. class	inv. loop	var. loop	check	Total
<b>AsnLOC</b>	89468	61267	10882	332	325	6206	168480
<b>LOC/AdjSLOC</b>	2.33%	1.60%	0.28%	0.01%	0.01%	0.16%	4.39%
<b>LOC/AsnLOC</b>	53.10%	36.36%	6.46%	0.20%	0.19%	3.68%	100.00%
<b>IALOC</b>	1129	3710	996	91	0	502	6428
<b>IALOC/AdjSLOC</b>	0.03%	0.09%	0.02%	0.00%	0.00%	0.01%	0.16%
<b>IALOC/AsnLOC</b>	0.67%	2.20%	0.59%	0.05%	0.00%	0.30%	3.82%
<b>No. of clauses</b>	53677	39550	4614	203	324	5139	103507
<b>Count(e /= Void)</b>	39742	14484	5571	7	0	2177	61981
							Average or max.
<b>Max AsnLOC size</b>	30	24	35	7	2	14	35 (max)
<b>Average size</b>	1.7	1.5	2.4	1.6	1.0	1.2	1.6
<b>% (e /= Void)</b>	44.42%	23.64%	51.19%	2.11%	0.00%	35.08%	36.79%

Figure 5, Metrics concerning assertions (all project categories)

Project Category	SLOC	AdjSLOC	AsnLOC	AsnLOC / AdjSLOC
Proprietary	2033945	1956184	63983	3.27%
Open Source	1307893	1253721	63971	5.10%
Eiffel 5.5	663281	630899	40526	6.42%
<b>Total</b>	<b>4005119</b>	<b>3840804</b>	<b>168480</b>	<b>4.39%</b>

Figure 6, Proportion assertion LOCs per project category

A noteworthy proportion of assertions include subexpressions of the form *e /= Void* asserting that a given reference is not *Void* (i.e. null). This number is over 50% for class invariants, but 37% overall. We will comment further on these numbers in the next section.

Finally, in Figure 6 we show how the proportion of LOC that are assertions (AsnLOC) varies according to the project category. As might be expected, the Eiffel category has the highest proportion, 6.4%, followed by open source projects<sup>7</sup> and then proprietary code with a little over 5% and 3% respectively. Even in the worse case of the three, the proportion of AsnLOC is over three times more than what was reported by Hoare for the Microsoft Office Suite.

## 4 Issues

In commerce it is said that the “customer is king<sup>8</sup>”—a concept that we attempt to instill in our junior software engineers as well. As formalists, one of our main customers is industry, and one of their key objectives is *return on investment* (ROI). To be successful, we must cater to this need. We should also keep in

<sup>7</sup> Recall that the open source category excludes GoboSoft software (since it is counted in the Eiffel 5.5 project category).

<sup>8</sup> Or Queen, or simply Royal.

mind that industry tends to be conservative, and rightly so since there is considerable risk associated with jumping on what may be perceived as an unproven technological trend or fad. (The dot com collapse reminds us of this, as many companies, both large and small, suffered the consequences of having rapidly shifted resources into dot-com related development projects that never provided ROI.)

Most industries will welcome new technologies that enable them to

- capitalize on current assets—such as personnel skills and software code base;
- while minimizing risks and costs inherent in adopting a new technology.

Upfront costs for specialists and training of personnel are often cited as a key dissuasion for top-down methods [SW03]. Particularly welcome by industry are those methods that can be

- retrofitted into legacy code as well as used on new developments,
- applied progressively with ROI that is proportional to the investment.

We believe that bottom-up methods have exhibited these qualities so far. But what of the forthcoming step along this bottom-up path: i.e. the transition in use of assertions from runtime assertion checking (RAC) to extended static checking (ESC)?

## 4.1 ESC: Choice of logical foundations

### 4.1.1 ESC

Extended static checking involves the use of theorem proving technology in the interpretation of code and its embedded assertions with the objective of statically detecting program faults. Being able to statically identify and resolve faults usually reduces the number of exceptions generated by the code at runtime. Ridding programs of (avoidable) runtime exceptions is appealing to industry at large but particularly to those developing safety and security critical applications.

### 4.1.2 Motivating example

Consider the following assertion in which `a` is an array:

$$\text{sum}(a) / a.\text{count} > n \quad (1)$$

What should its meaning be if `a` is `Void` (i.e. a null reference) or `a` contains no elements? The RAC answer is clear: an exception would be reported at runtime. In ESC, this issue reduces to the choice of semantics adopted for partial functions. Since logical connectives can be viewed as (Boolean) functions, a corollary to this is the choice of two- vs. three-valued logic.

### 4.1.3 The problem

The main problem that has arisen is that all of the ongoing research into ESC is based on classical two-valued logic with total functions. For example, the Simplify prover uses a first-order two-valued logic; it is the prover behind ESC/Modula-3, ESC/Java [Flanagan+02] and more recent experimental support

for Spark [Ireland04]. This choice seems to have been motivated by practical reasons: almost all theorem proving technology uses such logical foundations. But is this the right choice for industry—i.e. *our* customers?

As reported in this paper alone, we have cited over 418 thousand assertions. All of these assertions are currently interpreted using the operational semantics of some underlying programming language. This semantics is most closely modeled by a three-valued logic with partial functions such as LPF [JM94]<sup>9</sup>. Hence, it may come as a surprise to current practitioners that an ESC tool would consider the following two assertions logically equivalent to true

$$f(x) = f(x) \tag{2}$$

$$a.item(i) = 3 \text{ or } a.item(i) \neq 3 \tag{3}$$

regardless of possible exceptions that could be generated. Let us suppose that the method  $f$  generates an exception when  $x = 0$ , then in a two-valued logic  $f(0)$  will have a legal, though arbitrary value in the range type of  $f$ . Hence, no matter which value it is,  $f(0)$  will be equal to  $f(0)$ <sup>10</sup>. Assertion (3) is an instance of the law of excluded middle. Exceptions can arise if  $a$  is `Void` or  $i$  is an invalid index for the array  $a$ . Similar arguments can be given for (3) to explain why it is always considered true.

From the point of view of RAC, (2) and (3) are actually equivalent to “ $f$  is defined at  $x$ ” and “ $a$  is not `Void` and it contains an item at index  $i$ ”, respectively. As a final example consider

$$a.count = b.count \tag{4}$$

For a RAC system this generates an exception if (i)  $a$  is `void`, (ii)  $b$  is `void`, or (iii) both are `void`. It is interesting to note that for these three situations, (4) is provable in current ESC systems when both  $a$  and  $b$  are `void`. Clearly, there is a mismatch between current research directions in ESC and current practice using RAC.

#### 4.1.4 Consequences

Some of the disadvantages of the current choice of logical foundation for ESC tools include:

- Practitioners will have to be versed in *two* logical systems. Managing one logical system is already a challenge for the majority of practitioners and students—as is exemplified by ongoing debates on the role of mathematics in computer science and software engineering education e.g. [Devlin03]. The need to learn and use two logical systems will have an impact on
  - costs, e.g. due to training, and,
  - productivity. The impact on productivity should be apparent to anyone who has developed software in two mostly similar but subtly different

---

<sup>9</sup> Of course, in a logic for assertions, we are not interested in modeling all details of the operational semantics of expressions from the underlying programming language. Undesirable properties like side-effects are excluded. (Though it becomes a proof obligation to demonstrate that queries used in assertions are side-effect free.)

<sup>10</sup> For the sake of presentation, we are assuming here that the feature (method)  $f$  is functional.

languages—e.g. C++, Java or C#. The minor differences in language semantics often gives rise to subtle bugs.

- One of the objectives of joint RAC / ESC technologies is to generate runtime checking code only when ESC is unable to prove that a given assertion holds. This is considered an optimization and should not affect the behavior of the code. But should an ESC tool exclude runtime checking code for either of the assertions (2) and (3)? Doing so would give rise to a difference in behavior: i.e. without the optimization, RAC might cause an exception to be generated under some circumstances (e.g. `a is Void`).
- In an attempt to alleviate the previous problem, some have suggested that RAC semantics be retrofitted to conform to the ESC semantics.
  - This is what has been done in JML [CL02]. Unfortunately, the JML RAC, like any other such tool, can only approximate the ESC semantics. For example, consider the case of an arbitrary assertion containing `f(...)` as a subexpression: `... f(x)... = ... f(y)...`. Because partial functions applied outside of their domain yield arbitrary values, it becomes impractical for the RAC to determine the (two-valued) truth value of this expression. Hence, it must rely on heuristics to approximate the ESC semantics [CL02].
  - As a result of these and similar issues, the code generated by the RAC to support a two-valued logic is considerably more complex and significantly larger than it would otherwise be. For example the increase in JML RAC code size is at least an order of magnitude.
- Treatment of conditional logical operators is unintuitive. In Eiffel, the conditional form of conjunction and disjunction are “`and then`” and “`or else`”, respectively (in Java they are `&&` and `||`). These operators evaluate their second operand only if necessary to establish the value of the expression: e.g.

`a /= Void and then a.count > 5` (5)

will be false when `a is Void`; on the other hand

`a /= Void and a.count > 5` (6)

will cause an exception to be generated. In current ESC semantics (like ESC/Java [Leavens+03]) conditional logical operators are treated like their corresponding non-conditional forms: thus (5) and (6) would be given the same meaning (i.e. false when `a is Void`). What is more unexpected is that both of these assertions are also logically equivalent to:

`a.count > 5 and then a /= Void` (7)

which will clearly surprise most practitioners.

On the other hand, what would be some of the advantages of aligning more closely with current practice (in the use of RAC) and adopting a three-valued logical foundation for ESC?

- Practitioners would have only one logical system to master and it is the one they are already most familiar with (because they use it whenever they write

code containing assertions and general boolean expressions). Consequently, this would help reduce training costs associated with adopting ESC.

- RAC and ESC semantics remain compatible. Thus RAC implementations remain rather straightforward. The RAC / ESC optimization previously mentioned can (in theory) be safely performed without worry of changing the runtime behavior of the code.

Thus, it would seem that to better cater to the needs of mainstream practitioners, ESC will have to be founded on a logic that is “backwards compatible” with industry’s current use of RAC: hence, a three-valued logic with partial functions. Experiences in conducting proofs in such logics have been reported, e.g. [Bicarregui98] but there is as yet no full support in a modern theorem prover. To this end, our research group and others have begun experiments in the use of Isabelle [Paulson94] to support three-valued logics, building on earlier work such as [AF97].

## 4.2 Null references and non-null types

Languages should be designed to be safe, flexible and allow reasonable conciseness. Safety as a language feature means that simple errors should be rather easy to detect. Hence, for example, an ESC system reporting that “ $f(x) = f(x)$ ” can be undefined—due to an attempted application of  $f$  outside its domain—is safer than a system that would trivially dismiss it as true (as would be the case when a two-value logic is used).

An interesting result of the Eiffel survey is the measure of the number of assertions of the form  $e \neq \text{Void}$ . Roughly 44% of assertions in preconditions contain subexpressions of this form, 51% of class invariants and 37% overall for all kinds of assertions. We believe that this clearly indicates a need for non-null types as have been introduced, for example in Spec#, the super set of C# supporting assertions [Barnett+04]. In Spec#, if  $A$  is a reference type then  $!A$  denotes the non-null type corresponding to  $A$ . Having such a language mechanism would result in more concise programs. In fact, as a safety feature, it would seem more appropriate to assume that a reference type denotes non-null references unless indicated otherwise. This is the semantics adopted by the Splint checker, a lint/ESC tool for C [Evans03]: to indicate that a pointer variable can be null, one must annotate its declaration with `@null`. We contrast this with the `nonnull` annotation in JML. It has been our experience in a recent case study of the use of ESC/Java2, that more than 50% of method parameters and return types required `nonnull` annotations [RC04].

## 5 Comments on the survey results

We concede that the Eiffel survey sampling is not very large by industrial standards (5.4MLOC), but we are hopeful that it is somewhat representative. As mentioned earlier, our survey is still ongoing—both for Eiffel and other languages. On the other hand, we do anticipate that the use of Eiffel will be significantly less than the use of, e.g. C or C++. The relatively small size of the

Eiffel user community may also have some bearing on the survey results—e.g. a lesser variability.

## 6 Conclusion

There is an important class of formal methods (FMs) centered on assertions that we have called bottom-up methods. One of our claims was that bottom-up methods are a part of mainstream practice. In support of this we reported data (by Hoare) of the use of assertions in the Microsoft Office suite, as well as initial results from our own survey. Our survey data 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). Before conducting the survey we asked: does having language support for DBC mean that practitioners will make use of it? Overall, 4.4% of the (physical) SLOC of the surveyed projects were assertions. The results for the category of projects consisting solely of proprietary code was 3.3% which is roughly 2% more than what was reported by Hoare for the Office suite. In our opinion, this is good news for those researchers currently striving to add DBC support to other languages. “Build it and they will use it.”

The next milestone on our bottom-up adoption roadmap consists of the application of extended static checking (ESC) to contracts. To this end we have identified a key issue for which we believe current ESC research efforts are going astray: the logical foundations of ESC systems. To our knowledge, all current systems are based on classical two-valued logic. Unfortunately, this semantics is incompatible with the needs and expectations of industry. If the results reported in this paper are any indication, then there is a large number of assertions currently being used in RAC and interpreted with what amounts to a three-valued logic like VDM’s Logic of Partial Functions (LPF) [JM94]. ROI is a strong driving force in industry hence if we are to ensure the continued successful adoption of bottom-up methods like DBC ESC, we must choose a logical foundation that is compatible with the operational semantics of current RAC systems. Certainly LPF based proof systems are feasible (e.g. [Bicarregui98]), but will they be practical? This and related research topics are being investigated in our research group as well as others.

We also noted how the survey results give weight to the concept of non-null types as used in Spec# [Barnett+04], for example. As a language safety feature, we even suggest that reference types represent non-null references by default, an interpretation adopted by Splint [Evans03].

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. It will then become a smaller step to reach the full expressiveness of behavioral interface specifications (BISs). Of course, BISs are not the entire picture either; future generation verification compilers are likely to include support for model checking as well as BISs.

## Acknowledgments

We would like to thank Joe Kiniry for answering our questions concerning Eiffel and for helping suggest individuals to contact for the survey; Peter Grogono for offering feedback and challenging questions concerning earlier drafts of this work; Frederic Rioux for offering feedback as well as supplying us with the most of the open source projects.

## References

- [Abrial96] J.-R. Abrial. *The B Book – Assigning Programs to Meanings*. Cambridge University Press. 1996.
- [AF97] S.Agerholm and J.Frost. “An Isabelle-based Theorem Prover for VDM-SL”, In Proceedings of the International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97), Springer-Verlag, LNCS 1275, August 1997.
- [Barnes03] John Barnes. *High Integrity Software: The Spark Approach to Safety and Security*. Addison-Wesley, 2003.
- [Barnett+04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. *The Spec# programming system: An overview*. CASSIS 2004 post-proceedings.
- [BCMW01] D.Bartetzko, F. Clemens, M. Möller, and H. Wehrheim. “Jass – Java with Assertions.” *Electronic Notes in Theoretical Computer Science* 55(2), 2001.
- [Bicarregui98] J. Bicarregui, editor. *Proof in VDM: Case Studies*, Springer-Verlag, FACIT series, March 1998.
- [Burdy+04] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino and Erik Poll. An overview of JML tools and applications. In *International Journal on Software Tools for Technology Transfer (STTT)*, 2004.
- [CL02] Cheon, Y. and G. T. Leavens, A runtime assertion checker for the Java Modeling Language (JML). In H. R. Arabnia and Y. Mun eds., Proceedings of *International Conference on Software Engineering Research and Practice (SERP'02)* pp. 322–328, 2002.
- [Devlin03] Keith Devlin. Why Universities Require Computer Science Students To Take Math. *CACM* 46(9):36-39, September 2003.
- [DH98] Andrew Duncan and Urs Hölzle. *Adding contracts to Java with Handshake*. Technical Report TRCS98-32, University of California, Santa Barbara. December 9, 1998.
- [Evans03] David Evans. *Splint User Manual*. Secure Programming Group, University of Virginia. June 5, 2003. [www.splint.org](http://www.splint.org).
- [Flanagan+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static

- checking for Java. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI-02)*, ACM SIGPLAN 37(5):234–245, June 2002.
- [HH98] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
- [Hoare00] C.A.R. Hoare, *Assertions Progress and prospects*. Presentation available from [research.microsoft.com/~thoare](http://research.microsoft.com/~thoare).
- [Hoare03a] C.A.R. Hoare, Assertions: a personal perspective. *Annals of the History of Computing*, IEEE 25(2):14-25, April-June 2003.
- [Hoare03b] C.A.R. Hoare, The Verifying Compiler: A Grand Challenge for Computing Research. *JACM*, 50(1):63-69, 2003.
- [Ireland04] Andrew Ireland. An Automatic Debugger for SPARK. Project proposal available from [www.macs.hw.ac.uk/~air](http://www.macs.hw.ac.uk/~air). 2004.
- [JM94] C.B. Jones and C.A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta informatica*, 31(5):399-430, 1994.
- [Jones03] Cliff B. Jones. The early search for tractable ways of reasoning about programs. *IEEE, Annals of the History of Computing*, 25(2):26–49, 2003.
- [KHB98] Murat Karaorman, Urs Hölzle, and John Bruno. *jContractor: A Reflective Java Library to Support Design By Contract*. Technical report, Department of Computer Science, University of California, December 1998.
- [Kramer98] Reto Kramer. iContract—the Java Designs by Contract tool. In *Proc. Technology of Object-Oriented Languages and Systems, TOOLS 26*. IEEE Press, 1998.
- [Leavens+03] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (eds.), *Formal Methods for Components and Objects*, pp. 262-284. LNCS 2852, Springer 2003.
- [Meyer97] Bertrand Meyer. *Object-Oriented Software Construction*. 2<sup>nd</sup> edition. Prentice Hall, 1997.
- [Mitchell+02] Richard Mitchell, Jim McKim. *Design by Contract, by Example*. Addison-Wesley, 2002.
- [Necula97] George Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, January 1997.
- [Park92] R. Park, “Software Size Measurement: A Framework for Counting Source Statements.” CMU/SEI-92-TR-20, Software Engineering Institute, Pittsburgh, PA, 1992.

- [Paulson94] L.C. Paulson. *Isabelle - a generic theorem prover*. Number 828 in LNCS. Springer-Verlag, 1994. With contributions by Tobias Nipkow.
- [Parasoft05] Jcontract product page available at [www.parasoft.com](http://www.parasoft.com).
- [RC04] F. Rioux and P. Chalin. *Improving the Quality of Web-based Enterprise Applications with Extended Static Checking: A Case Study*. Dependable Software Research Group, Concordia University. ENCS-CSE TR 2004-004.
- [SW03] Donna C. Stidolph and James Whitehead. Managerial Issues for the Consideration and Use of Formal Methods. In Stefania Gnesi, Keijiro Araki, and Dino Mandrioli (eds.), *FME 2003, International Symposium of Formal Methods Europe*, Pisa, Italy, Sept. 8-14, 2003, Proceedings. pp. 170-186.
- [Wheeler04] David A. Wheeler, [www.dwheeler.com/sloccount](http://www.dwheeler.com/sloccount).