

# Logical Foundations of Program Assertions: What do Practitioners Want?

---

**Patrice Chalin**

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

ENCS-CSE Technical Report 2005-005, revision 02

June 2005

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

**Keywords:** assertions, runtime assertion checking, extended static checking, design by contract, survey, industrial practice, logical foundations

**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**

- April 2005 – initial release.
- June 2005 – SEFM'05 pre-release.

## Table of Contents

<b>1. Introduction .....</b>	<b>1</b>
1.1. Background .....	1
1.2. Problem motivating the survey questions .....	1
1.3. Importance .....	2
1.4. Outline .....	2
<b>2. Assertions, RAC and ESC .....</b>	<b>2</b>
2.1. Informal assertions .....	2
2.2. Programming language support for RAC .....	2
2.3. Current RAC tools .....	3
2.4. Current ESC and PV tools .....	3
<b>3. Logical semantics of assertions .....</b>	<b>3</b>
3.1. RAC semantics .....	3
3.2. PV and ESC semantics .....	4
<b>4. Survey .....</b>	<b>4</b>
4.1. Participants .....	4
4.2. Questionnaire .....	4
<b>5. Results .....</b>	<b>4</b>
5.1. Run-time exceptions during expression evaluation .....	4
5.2. Run-time exceptions due to arithmetic overflow .....	5
5.3. Exceptions in assertions during RAC .....	5
5.4. Exceptions in assertions during ESC .....	6
5.5. Use of assertions .....	7
5.6. Demographics .....	7
<b>6. What practitioners want and why .....</b>	<b>7</b>
6.1. Logical foundations .....	7
6.2. Two-valued logic misaligned with programming practice .....	8
6.3. Ignoring errors is bad programming practice .....	8
6.4. Consistency .....	8
<b>7. Three-valued logics .....</b>	<b>8</b>
<b>8. Conclusion .....</b>	<b>9</b>

# Logical Foundations of Program Assertions: What do Practitioners Want?

Patrice Chalin

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

## Abstract

*Industrial use of program assertions for the purpose of Run-time Assertion Checking (RAC) is becoming commonplace. A likely next step in the use of assertions is Extended Static Checking (ESC), an area of active research that promises added benefits to industry. Unfortunately, RAC and ESC tools are not consistent in their interpretation of assertions containing undefined terms. In this paper, we report on the results of a survey in which we asked industrial developers what logical semantics they want program assertions to have, and whether consistency across tools is important. Survey results indicate that developers are in favor of a semantics for assertions that is compatible with their current use in RAC.*

## 1. Introduction

This paper presents the results of an end-user survey of programmers, mostly from industry. The main purpose of the survey was to uncover the preferences of programmers with respect to the logical semantics of program assertions in the context of run-time checking and extended static checking (a form of static program verification). One of the key objectives of formal methods research is industrial adoption. While early program verification systems were crafted as proofs-of-concepts, there appears to have been no end-user consultation during the maturation of this technology. It is well known in software engineering that lack of user involvement can significantly increase the likelihood of project failure. The purpose of the survey was to address this deficiency.

### 1.1. Background

A program assertion is essentially a Boolean expression inserted at a point in a program where it is believed to be true [1]. As early as 1947, Goldstine and von Neumann recognized the challenge of writing

correct code and they introduced into the programmer's toolbox a simple, yet powerful and effective tool: program assertions (or, as they called them, assertion boxes) [2]. Before the start of the 50's Goldstine, von Neumann and Turing had laid out the two main (and complementary) uses of assertions which still prevail today: *Run-time Assertion Checking (RAC)* and the role of assertions in formal *Program Verification (PV)* [2].

Since then, practitioners in most application domains have adopted RAC as a basic tool of the trade encouraged by the simplicity and effectiveness of the technique. On the other hand, research into program verification was headed independently by academics in formal methods (for the most part). Significant advances have been made in program verification in the last decade. This, coupled with the phenomenal growth in raw processing power, has led us to a point of conjunction where PV tools are able to automatically check a nontrivial proportion of the assertions that developers already embed in their programs for the purpose of run-time checking. Some of these tools, which implement a limited though fully automatic form of PV, are said to perform *Extended Static Checking (ESC)*.

As will be confirmed by the survey results, programmers in several application domains have already (willingly) integrated the use of assertions into their development practices. The majority use assertions in conjunction with run-time checking. A natural next step is the more general adoption of ESC tools. This would allow developers to apply these tools to their existing code base (annotated with assertions) and consequently increase their return on investment. Unfortunately, there is an obstacle to this adoption as we explain next.

### 1.2. Problem motivating the survey questions

The semantics of program assertions as given by RAC and PV tools are not consistent with respect to

each other. More precisely, RAC and PV tools disagree on the interpretation of assertion expressions in those situations where run-time evaluation of the assertion expression fails to yield a value. For example, consider the following Java class containing an inlined assertion

```
class C {
    void m(int x) {
        assert 1/x == 1/x;
    }
}
```

A call to `m(0)` while run-time checking is enabled will result in an *exception* being raised (due to division by zero), yet ESC/Java2, a static verification tool for Java, interprets the assertion expression as *true*. This is surprising because usually, when a static analysis tool reports that a certain property holds, then it is expected to hold at run-time for all possible program states (and method invocations)<sup>1</sup>. Although we wrote our example in Java, it could just as well have been written in, say, SparkAda or Spec# with similar results. (Explanations are postponed until Section 3.)

The main purpose of our end-user survey was essentially to ask practitioners: “How do *you* want program assertions to be interpreted?” particularly for those assertions that RAC and PV tools offer differing interpretations for. We will also ask “Should RAC and PV tools agree on their interpretation of assertions?” As we shall see, answers to these questions will help us determine what practitioners would like the logical semantics of program assertions to be.

### 1.3. Importance

As software engineers we are well aware of the importance of customer involvement during the elaboration of requirements. Not doing so will decrease the likelihood of user adoption and increase the likelihood of project failure [3]. As there is currently an increase in the amount of research funds allocated to, and research activity surrounding PV and ESC technology, it is important that the targeted end-users of this technology be consulted.

But should research in assertion-based formal methods (ABFMs) be funded at all? We believe so. The base technique in ABFMs is the ad hoc use of program assertions. More advanced methods include the use of assertions in systematically capturing module contracts—such as function/method preconditions and postconditions as is done in Design by Contract (DBC) [4]. Run-time or extended static checking can be applied to varying degrees in either case. We be-

---

<sup>1</sup> The result could be due to unsoundness issues in ESC/Java2, but this is not the case for the given example.

lieve that ABFMs are one of the most promising avenues for the industrial adoption of formal methods, in particular because they [5]:

- require *minimal* extra *training*, capitalizing on the knowledge and mastery that developers have of programming languages,
- can be applied to new developments as well as retroactively applied to the ever increasing base of legacy code,
- can be *progressively integrated* into existing practices,
- can be applied *partially*, i.e. only to those subsystems where it is needed or justified,
- can yield benefits immediately, and benefits gained are *proportional* to the effort invested.

As was mentioned earlier, programmers in several application domains have already integrated the use of assertions into their development practices.

## 1.4. Outline

A review of assertions, run-time checking and extended static checking is provided in Section 2, followed by a brief discussion of the logical semantics of assertions from the perspective of current RAC and PV tools (Section 3). The survey and survey results are presented in Sections 4 and 5 respectively. We offer a discussion of the survey results in Sections 6 and 7, and conclude in Section 8.

## 2. Assertions, RAC and ESC

### 2.1. Informal assertions

As stated in the introduction, a program assertion is essentially a Boolean expression inserted at a point in a program where it is believed to be true. It is our experience that all programmers (even novices) naturally write *informal* assertions in the form of comments or using if statements that print error messages and debugging information or break to a debugger. We do not consider such informal assertions in this paper. Instead, our attention is focused on program statements explicitly identified as assertions. Support for such program assertions is often built-in to a programming language, provided by an external library, or expressed in the form of special annotations/directives that are processed by RAC or ESC tools.

### 2.2. Programming language support for RAC

Most programming languages offer support for assertions. An assertion statement consists of an assertion expression sometimes with extra arguments

that can be printed to aid in debugging when assertions fail. Examples of the support for assertions include the

- `Assert` compiler pragma in Ada,
- `assert` macro of C and C++,
- `Assert` method of the C# `Trace` class,
- `assert` statement in Java [6],
- `assert`, `requires`, `ensures` and `invariant` clauses of Spec# (a language extension to C#) [7], and
- the Eiffel `check`, `require`, `ensure` and `invariant` statements [4].

Most languages also provide support for run-time assertion checking (RAC). This language feature allows assertion expressions to be evaluated at run-time during the normal course of execution of a program. If an assertion expression evaluates to true, then no further action is taken. If it evaluates to false, then an assertion violation is reported. All of the previously mentioned languages allow assertion checking code to be conditionally compiled into object code or excluded from it. This allows assertion checking to be excluded from production code thus eliminating, if so desired, any overheads in code size or code execution time. In addition, some languages such as Java, allow checking to be enabled or disabled at run-time.

Building upon the basic support for assertions, some companies have developed specialized kinds of assertion (e.g. as reported by Hoare at Microsoft [8]). Variants include: assertions that are kept in production code; assertions whose failure may result in special error logging and/or a behavior other than abrupt termination.

Another specialization in the use of assertions is Design by Contract (DBC) in which assertions can be explicitly identified as preconditions, postconditions or invariants. Eiffel, and more recently Spec#, are the only active programming languages with built-in support for DBC [4]. Eiffel currently only supports run-time assertion checking whereas Spec# has both RAC and ESC support [7].

### 2.3. Current RAC tools

Other than the run-time checking facilities mentioned in the previous section, there also exist tools that process assertions written as specially formatted comments. One such tool for C is called APP [9]. There exist several RAC tools for Java including the one associated with the Java Modeling Language (JML) [10], as well as Jass [11] and the commercial tool Jcontract [12]. JML supports DBC for Java [13] as well as a more general form of specification called Behavioral Interface Specification (BIS).

### 2.4. Current ESC and PV tools

Current extended static checking tools include:

- Spark for Ada [14],
- Splint for C (formerly LCLINT, a member of the Larch family of languages and tools) [15],
- Boogie for Spec# [7],
- ESC/Java2 [16] and JIVE [17] for Java,
- JACK for JavaCard [18],

Older systems include Penelope for Ada [19], and ESC/Modula-3 [20], the predecessor to ESC/Java [21] and ESC/Java2. Spark is based on a subset of Ada which is enhanced with support for assertions in the form of annotations. ESC/Java2 and JACK use JML as an annotation language; JIVE is being adapted to use JML as well. Splint can only process a very limited form of assertions/annotations, contrary to its predecessor LCLINT, that accepted general BISs (but LCLINT only performed type checking).

There are much fewer tools that can be used to perform complete verification of (sequential) programs. In fact, the only one that we are aware of is the LOOP tool developed at the University of Nijmegen [22]. The LOOP tool can be used to verify JML annotated Java programs. It does so by first translating the code and specifications into PVS theories and then one uses the PVS theorem prover to carry out the proofs [23]. Case studies in the use of LOOP are referenced in [24].

## 3. Logical semantics of assertions

In this section, we explain the logical semantics of program assertions from the perspective of current run-time assertion checkers, and program verification tools. The goal is to provide sufficient background for the understanding of the survey questions.

### 3.1. RAC semantics

It is well understood that the semantics of program assertions, as realized by run-time checkers, essentially corresponds to a three-valued logic [25]; i.e. evaluation of an assertion expression can yield true, false or “undefined/error” (e.g. due to a run-time exception). The law of excluded middle “ $E \vee \text{not } E$ ”, fundamental to two-valued logic, no longer holds (as is well known to programmers).

In three-valued logics, it makes sense to have binary Boolean connectives that are non-strict in their second argument. Such “conditional” or “short-circuited” operators, as they are called in programming language terminology, are also quite familiar to developers—in fact, for C-based languages, these are used almost exclusively. For example, in Java “`&&`” and “`||`” are the

conditional-and and conditional-or operators, respectively. In Ada and Eiffel these are named “and then”, and “or else”. The non-conditional operators, called “logical operators” in Java ([6] §15.22), are “&” and “|”. (These may seem unfamiliar to C developers who are accustomed to seeing these operator names used solely for bitwise conjunction and disjunction.)

### 3.2. PV and ESC semantics

All of the extended static checking and program verification tools mentioned in Section 2.4 interpret assertions as if they were predicates in a classical two-valued logic that models partial function as underspecified total functions [26] as we explain next.

Let  $f$  be a function in such a logic then  $f$  will be defined for all values of its domain and  $f(v)$  will always have a value in the range of  $f$ . If  $f$  is usually undefined at  $v$  then  $f(v)$  will have some unspecified value in the range of  $f$ . As an example, consider the integer division expression  $1/0$  which is usually undefined because the divisor is 0. In the logic under discussion,  $1/0$  will have some integer value, although we do not know which value it is. Note that  $1/0$  and  $2/0$  are not necessarily equal since the division operator is being applied to different arguments,  $(1,0)$  and  $(2,0)$  respectively, and hence these might be mapped to different values. Referring to the sample assertion given in Section 1.2, we can now understand why ESC/Java2 interpreted  $1/x = 1/x$  as true for any value of  $x$ .

One of the advantages of modeling partial functions in this way is that the rules of classical logic with equality can be preserved. Thus, in particular

- equality remains reflexive so that  $1/x = 1/x$  regardless of the value of  $x$ , and
- the law of excluded middle holds, e.g.  $x/x = 1 \vee x/x \neq 1$  for any value of  $x$ .

In the remainder of this article, we will use the term “two-valued logic” to mean “two-valued logic with partial functions modeled as underspecified total functions”.

Finally we note that in two-valued logic, conditional operators are equivalent to their non-conditional counterparts—e.g. “&&” and “&” are synonyms. As a result, `a != null && a.length > 0` becomes logically equivalent to `a.length > 0 && a != null`. Is this something that practitioners want? (We will find out in Section 5.4.)

## 4. Survey

### 4.1. Participants

Initially a few hundred e-mail invitations were sent out by members of our research group asking colleagues and contacts (mainly from industry) to participate in the survey as well as to forward the invitation to their peers. We subsequently broadened our invitation by posting it to programmer news groups and online bulletin boards.

### 4.2. Questionnaire

Participants were invited to complete the survey questionnaire online via the research group’s secured site (<https://www.dsrg.org>). Participants were first asked to complete a consent form in which they provided their name, company/institution affiliation and e-mail address. To ensure at least a minimal level of authenticity, an acknowledgment e-mail was sent to respondents. This e-mail contained a link (with an embedded unique id) which would allow the receiver to confirm that it was indeed him/her who filled in the survey.

The survey questionnaire begins by asking general questions about the respondents’ perceived importance of the exception reporting mechanism in the context of expression evaluation. It then presents Boolean expressions containing partial functions and ask the respondent how he/she believes the expressions should be interpreted under given situations. The questionnaire contained 12-15 questions distributed among five sections, each section devoted to a particular topic. Almost all questions had an associated text box in which respondents could add comments in the form of free text. The questionnaire was dynamically generated, hence only relevant questions were presented to respondents (as will be detailed next).

## 5. Results

Over 200 respondents successfully completed and acknowledged their survey entries. We present the survey results by following the organization of the questionnaire.

### 5.1. Run-time exceptions during expression evaluation

All programmers are aware that an attempt to access the array element `a[0]` when `a` is null will result in a null dereference exception being reported. (In some languages like C, the term “run-time error” is more

commonly used than “exception”. Unless specified otherwise we will be using the term “exception” liberally to include run-time errors reported by some form of signaling mechanism.) In mathematical terms we can understand that an attempt is being made to apply the partial function/operator of array lookup `_[_]` to arguments outside its domain<sup>2</sup>. Restated in programming terms, the exception can be seen as reporting a violation of the operator’s *precondition*.

The purpose of the first question was to determine the relative importance that programmers give to this exception reporting mechanism in the context of expression evaluation:

**A.1. For expressions in general, how would you rate the programming language feature of reporting an error/exception at run-time when an operator’s precondition is violated?**

Respondents were given the choice of the following answers: *very important*, *important*, *neutral*, *undesirable*. The majority of respondents choose “very important” or “important” with a mean between these two answers—see Table 1. Of the respondents who answered “undesirable”, some cited efficiency as a justification for not wanting run-time exceptions to be reported in a given application domain (such as embedded controllers).

Table 1. Responses to Questions A.1, B.1

Question	very important (2)	important (1)	neutral (0)	un-desirable (-1)	Mean
A.1	54%	34%	9%	3%	1.4
B.1	24%	42%	28%	5%	0.8

## 5.2. Run-time exceptions due to arithmetic overflow

Unfortunately almost all languages that have evolved from C have inherited what some would consider a deficiency: arithmetic overflows are silently ignored<sup>3</sup>. A recent exception is C# which supports two arithmetic modes: checked and unchecked, with the later being the default. In checked mode, arithmetic overflows are reported as exceptions that can be caught and handled by user code. At a minimum, such a feature gives the programmer a choice of dealing with overflows or not.

<sup>2</sup> Of course, `_[_]` also has the program state  $\sigma$  as an implicit argument.

<sup>3</sup> Actually, the ANSI C standard states that reporting overflow is implementation dependent; it is our experience that very few compilers support the reporting of overflow.

```
int f(...) {
    int sum = 0;
    ...
    assert(sum > 0);
    ...
}

int g(int a[], int n)
/* Here is a precondition for g */
/*@ require(a != NULL && n > 0); @*/
{
    int sum = 0;
    ...
    /*@ assert(sum > 0); @*/
    ...
}
```

(a) Basic assertions as supported by C-based or C-like languages

(b) Assertions in specially formatted comments

Figure 1. Sample assertions

**B.1. How would you rate this programming language feature of enabling arithmetic overflows to be reported as errors/exceptions?**

This question is complementary to A.1 in that for programmers of C-based languages, it allows us to measure programmers’ responses for a feature that they do not currently enjoy.

The mean response to this question is a little less than “important” (see Table 1). This is a lower rating than for question A.1; a common explanation given for the lower rating was that such a feature would be useful only if it could be selectively enabled (as is the case in C# for example).

## 5.3. Exceptions in assertions during RAC

The first question of this section asked “do you know what an assertion is?” If not, the respondent was directed to examples (Figure 1) and a short explanatory text. All but one respondent answered yes.

The next question gets us closer to the heart of the matter and asks, in general, what should be done if an exception is raised during the evaluation of an assertion expression.

**C.2. What should be done during run-time assertion checking if an error/exception is reported during the evaluation of an assertion expression (such as `a[0] > 0` when `a` is null)?**

Possible answers were:

- Interpret the expression as **true**.
- Interpret the expression as **false**.
- Report an **error/exception**.
- Other (provide details).

Over 80% of respondents choose “error/exception” as indicated in Table 2.

Table 2. Responses to Questions C.2, D.1, D.4

Question	true	false	error/ except.	other
<b>C.2</b>	0%	9%	<b>81%</b>	9%
<b>D.1</b> <EXPR>				
t    (nullRef[0] > 0)	<b>73%</b>	1%	18%	7%
(nullRef[0] > 0)    t	6%	5%	<b>84%</b>	4%
nullRef[0] > 0	1%	5%	<b>91%</b>	3%
t   (nullRef[0] > 0)	4%	5%	<b>87%</b>	3%
<b>D.4</b> <EXPR>				
a[0] == a[0]	16%	7%	<b>75%</b>	3%
a[0] == b[0]	7%	8%	<b>80%</b>	4%
a[0] == b[1]	3%	8%	<b>84%</b>	5%
g(a[0]) == a[0]/a[0]	6%	9%	<b>82%</b>	3%
a[0] == 0    a[0] != 0	8%	10%	<b>74%</b>	7%

**C.3.** For any given boolean expression  $E$ , consider evaluating  $E$  at run-time (with **run-time assertion checking** enabled) when  $E$  is used as (a) an assertion expression, (b) an if statement condition. Should the result of evaluating  $E$  always be the same in both cases? That is, in cases (a) and (b):

- they will *both* interpret the assertion as **true**, or
- they will *both* interpret it as **false** or
- they will *both* report an **error/exception**.

The purpose of **C.3** was to determine, from the respondent’s point of view, if there should be any difference in the semantic interpretation of a Boolean expression when used as an assertion expression vs. outside the context of an assertion. 72% of respondents answered yes. Some who answered “no” commented that assertion expressions should not have side-effects (since this would result in different program behavior when assertions are enabled vs. when they are not).

The majority of respondents who answered “no” did so because they remarked that assertion statements and if statements are meant to be used for different purposes, i.e. *after* the expression is evaluated these statements will have different effects. Unfortunately this points out that the intent of the question was not clear enough for these respondents.

## 5.4. Exceptions in assertions during ESC

The question in this section was preceded by the following explanation: “Static Analysis (SA) tools can be used to detect program assertion violations without running the code in a way that is similar to how compilers detect type errors in programs”. (We chose to phrase the question using the slightly more general term “static analysis” rather than “extended static checking”.)

**D.1.** During **static analysis** how should each of the following assertion expressions, <EXPR>, be interpreted when **nullRef** is a **null** array reference and, **t** is **true**?

The sample <EXPR> along with the profile of responses are given in Table 2. The first expression involves a conditional-or operator with its first argument being true. In such a case the value of the second argument is irrelevant and the overall expression evaluates to true. The majority of respondents chose “true”. The next most popular answer was “error/exception” which appears to have been chosen, in some cases, because the reader was unaware that “||” is a conditional-or in C-like languages. All other expressions in **D.1** would result in a null pointer exception if evaluated during run-time checking; most respondents were in favor of the same interpretation in during static analysis.

The first expression, i.e. “t || (nullRef[0] > 0)”, is the same as the second, but with its arguments permuted. The last expression has the same arguments as the first but uses Java’s non-conditional disjunction. Based on the choice of responses for these three expressions we see that developers do not want conditional and non-conditional operators to be treated as synonyms (as is the case in two-valued logic—cf. Section 3.1). The next question generalizes **D.1**:

**D.2.** For any given assertion expression  $E$  and program state  $S$ , should **run-time assertion checking (RAC)** and **static analysis (SA)** always agree on the same interpretation of  $E$ ? That is, should

- they *both* interpret the assertion as **true**, or
- they *both* interpret it as **false**, or
- RAC will report an **error/exception** and SA will interpret the assertion as being in error.

**D.3.** What is your main reason for answering yes or no to the previous question? (optional)

(Some believe this question is superfluous on the grounds that consistency is obviously desirable, yet it should be noted that *none* of the current languages supporting both RAC and ESC offer a consistent semantics—including JML, SparkAda and Spec#.) 73% of respondents answered “yes”, several providing consistency as the main reason. One respondent wrote: “[consistency avoids] special cases; helps me remember how things work”.

The respondents who answered something other than “error/exception” to **C.2**, or “no” to either **C.3** or **D.2**, were asked to complete the following question:

**D.4.** During **static analysis** how should each of the following assertion expressions, <EXPR>, be interpreted when **a** and **b** are **null**?

We anticipated that the respondents who answered no to **D.2** might have in mind the modeling of partial functions by underspecified total functions. The `<EXPR>` of **D.4** were chosen so as to highlight issues that might arise under such an interpretation of partial functions.

As indicated in Table 2, 16% of respondents believed that `a[0] == a[0]` should be true when `a` is null—consistent with an interpretation in two-valued logic. On the other hand `a[0] == b[0]` would also be true under a two-valued logic when `a` and `b` are null because the expression simplifies to `null[0] == null[0]` which is true; yet only 7% of respondents recognized this. As was explained in Section 3.2, the third expression would have an undetermined value in classical logic since the array access operator is being applied to different arguments. The next expression involves a function `g` defined as follows

```
int g(int m) {
    return m/m;
}
```

It is unclear how the expression `g(a[0]) == a[0]/a[0]` should be interpreted under a two-valued logic. An issue of concern is the following: since `a[0]`—the argument to `g`—is undetermined, should we still attempt to interpret `g` at `a[0]`? The last `<EXPR>` of **D.4** is an instance of the law of excluded middle. The majority of respondents chose “error/exception” for all expressions (including the law of excluded middle).

## 5.5. Use of assertions

**E.1.** *Is the use of assertions a part of your regular programming practice? For example, when you write code you also write assertions: very frequently (5), frequently (4), regularly (3), occasionally (2), rarely (1), never (0).*

Responses were 36%, 17%, 14%, 11%, 12% and 8% respectively, with a mean of 3.3. 76% of the respondents answered that assertions were used at their institution (**E.3**). Of these, 97% use assertions for run-time checking, 20% extended static checking, 11% mentioned other uses, e.g. unit testing and documentation (**E.4**).

Assertions are used in a broad range of software categories including (**E.2**): business/finance, entertainment, medical, military, security, software tools, and systems software. Specific domains mentioned were: air traffic control / aerospace, database software, document management, embedded software (including real-time controllers), enterprise applications (including web-based for billing, account management, etc.), games (including 3D real-time interactive), robotics,

scientific computing / numerical analysis, simulation, compilers / preprocessors / IDEs, speech / image / vision (and other real-time signal processing), telecommunications / network software, various “freeware” applications.

Assertion tools used included: for the most part, assertion support provided by/with the programming language or custom versions thereof (**E.5**). The top three programming languages used in conjunction with assertions were C++, C, and Java. Respondents estimated that for a typical project, between 1.4% and 5.0% of the lines of code (LOC) consisted of assertions. (These numbers are consistent with a separate quantitative study that we have conducted on the use of assertions in Eiffel programs [27].)

For those institutions not using assertions, the majority commented that informal assertions (using if statements) and other mechanisms were used instead (e.g. specialized logging/error reporting).

## 5.6. Demographics

Respondents worked in industry (77%), academia (26%) or other (9%)—e.g. government, self-employed, or retired. Respondents were working in the United States (46%), Canada (23%), Europe (23%), Asia (3%), Australia (3%) as well as Africa, South America and other places (6%). Some worked in more than one location, hence the total of the previous percentages is more than 100.

On average, respondents had 13.3 years of programming experience and the highest programming-related degree—completed or in progress—for the majority was a bachelors (45%), a masters (28%), or Ph.D. (10%) while 12% had no programming-related degree. Over 50% of respondents with a degree, had a degree in Computer Science, 17% in Software Engineering, and 6% each in Computer and Electrical Engineering.

## 6. What practitioners want and why

In this section, we explain what we believe practitioners want, as well as some of the reasons that may be motivating this preference.

### 6.1. Logical foundations

When presented with an instance of the law of excluded middle (see the last expression of **D.4** in Table 2), respondents favored interpreting it as an error/exception (74%) or even false (10%) rather than true (8%). Hence, the survey results clearly indicate that practitioners want PV tools to adopt a logical se-

antics for assertions that is consistent with the RAC tools that they are currently using. That is, they are in favor of an interpretation of assertions in which partial functions and undefinedness are modeled directly in a manner that is compatible with the operational semantics of programming languages.

We emphasize that the survey results support an *interpretation of assertions* that is consistent with a three-valued logic. Of course, no statement is being made concerning the necessity of using a three-valued logic in the provers underlying PV tools. It is well known that two-valued logics are sufficient to model three valued logics—e.g. [25, 28].

In the remaining subsections we explore some of the top factors that may have influenced the choice of practitioners.

## 6.2. Two-valued logic misaligned with programming practice

The laws of classical logic do not hold for assertion expressions in the context of most programming languages. For example,  $1/x == 1/x$  will not be interpreted as true if  $x$  is 0, instead a “division by 0” exception will be raised. Exceptions raised under such circumstances signal the presence of an error in the program; e.g. the programmer must have believed that  $x$  could not be 0 if  $1/x$  was to be evaluated (of course the belief could have been wrong; in either case a bug has been exposed). Exceptions have been recognized as a useful tool in helping to detect such programming errors as close as possible to their point of occurrence.

(Note: it is obvious that the evaluation of program expressions can result in side-effects, thus easily contravening the laws of logic. It is well understood by programmers that assertions, as well as any “debugging” code, must be free of side-effects. Of course, any assistance by tools in detecting potential side-effects would be welcome.)

## 6.3. Ignoring errors is bad programming practice

Rephrased in programming terms, modeling a partial function as an underspecified total function essentially amounts to catching exceptions raised by the partial function and ignoring or masking them by returning an arbitrary legal value. This makes it much more difficult to locate the origin of an error. It is also known to be bad programming practice, e.g. Item 47 in [29], “*Don’t ignore exceptions*”.

## 6.4. Consistency

The top reason for having run-time checking and static analysis agree on the interpretation of assertions, is consistency. As one respondent put it: “Anything other than complete agreement will inevitably lead to confusion” and a few commented that “... To do otherwise would violate the principle of least surprise.” Yet, as was noted earlier, no current programming/specification language that has support for both RAC and PV actually provides a consistent semantics across tools.

Developers want PV tools to adopt a semantics that is consistent with the RAC tools that they are currently using. Of course, consistency could be achieved by adapting the semantics of run-time checkers to conform to the two-valued logical interpretation used in static checkers. As has been commented elsewhere [5], such a semantics could be approximated at best. (The main challenge is in implementing a scheme for generating and caching the “unspecified/arbitrary” values of partial functions applied to arguments outside their domains.) Furthermore, attempts to do this in the JML run-time checker, for example, generally result in an increase in instrumented code size that is at least an order of magnitude [5].

## 7. Three-valued logics

Barringer, Cheng and Jones have explored various formulations of three-valued logic [30], finally settling upon what has become known as the Logic of Partial Functions (LPF), the logic underlying the Vienna Development Method (VDM) [31]. LPF adopts a choice of logical operators (called Strong Kleene) that are non-strict and monotonic [32]. While such a choice of operators may be suitable for a foundation of LPF, strict and conditional operators (also referred to as Weak Kleene and McCarthy operators respectively) will also be required. These will be necessary if we are to accurately reason about the logical connectives of Ada, Eiffel and C-based languages since all of these languages support both conditional and non-conditional operators. Additionally, respondents indicated their preference for a strict interpretation of non-conditional operators (see the last expression of **D.1**, in Table 2).

As pointed out by Cheng and Jones [32], conditional operators enjoy fewer properties (such as commutativity and distributivity) than their non-conditional counterparts, but this is something that end users may need to experience first-hand: e.g. if the use of non-conditional operators (such as Java’s “[ ]”) allows ESC tools to prove more properties automatically, then

practitioners may be more inclined to use them. Habits will not be changed unless there is sufficient motive to do so. In a separate survey [27], we have noticed that Eiffel programmers, for example, make use of non-conditional operators more frequently than their conditional counterparts. This may stem from the fact that Eiffel conditional operators—whose syntax is borrowed from Ada—are longer to write: e.g. “or else” vs. simply “or”. No matter what the reason, the end result may well be that it will be easier to (automatically or manually) verify the validity of Eiffel contracts than, say, JML contracts. In the end, the use of conditional operators in programs will not disappear. Hence we will need a logic suitable for reasoning about them.

## 8. Conclusion

In this paper we have presented the results of a programmer survey that indicates that a large number of application domains make use of program assertions for the purpose of run-time checking. We have also noted that program verification (PV) technology is reaching a level of maturity that makes it feasible to automatically validate a nontrivial number of program assertions that are already being used for run-time checking. Hence this gives rise to a unique opportunity for many institutions to further capitalize on their investment of having previously annotated their code with assertions.

Unfortunately all current PV tools (including the more specialized Extended Static Checkers) interpret assertions as if they are predicates in a classical two-valued logic. Based on our experiences gained while working in industry, we hypothesized that this would go against the expectations of most practitioners. One of the main goals of the formal methods community is industrial adoption, hence it has become imperative to ask software developers from industry “how do you want static analysis tools to interpret assertions?” The two main results of the survey are that practitioners want a semantics for assertions that

- directly models partial functions in a manner that is compatible with the operational semantics of most programming languages;
- is consistently adopted across PV and RAC tools (with the former being adapted to conform to the latter).

Hence practitioners are in favor of an interpretation of program assertions that is compatible with a three-valued logic—possibly like VDM’s Logic of Partial Functions. We discussed some of the reasons why a two-valued logical interpretation of assertions would be counter-intuitive for programmers. We also clarified that PV and ESC tool designers are free none-the-

less to make use of provers built for classical logic to realize assertion semantics.

## 9. Acknowledgments

We are grateful to the many developers who accepted our invitation to participate in the survey as well as to the following individuals for their helpful comments on the many drafts of the survey questionnaire: Gary Leavens, Peter Grogono, David Cok, and Joe Kiniry. We thank the members of the DSRG for reviewing the questionnaire and helping setup and test the survey site. We are also thankful for the helpful remarks provided by the anonymous referees; these have contributed to improving the clarity and intent of the paper.

## 10. References

- [1] C. A. R. Hoare, "Assertions: A Personal Perspective," *IEEE Annals of the History of Computing*, pp. 14-25, April-June, 2003.
- [2] C. B. Jones, "The early search for tractable ways of reasoning about programs," *IEEE Annals of the History of Computing*, vol. 25, no. 2, pp. 26-49, 2003.
- [3] D. Leffingwell and D. Widrig, *Managing Software Requirements: A Use Case Approach*, 2nd ed. Object Technology Series. Addison-Wesley, 2003.
- [4] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice-Hall, 1997.
- [5] P. Chalin, *Ensuring Continued Mainstream Use of Formal Methods: An Assessment, Roadmap and Issues*, Dependable Software Research Group, Concordia University, ENCS-CSE TR 2005-001, 2005.
- [6] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 2nd ed. Addison-Wesley Professional, 2000.
- [7] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview." In *Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, Marseille, France, LNCS, vol. 3362, 2004.
- [8] C. A. R. Hoare, *Assertions: Progress and Prospects*, <http://research.microsoft.com/~thoare>, 2001.
- [9] D. S. Rosenblum, "Towards a method of programming with assertions." In *Proceedings of the 14th International Conference on Software Engineering*, pp. 92–104, May, 1992.
- [10] 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)*, 2004.
- [11] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim, "Jass -- Java with Assertions," *Electronic Notes in*

- Theoretical Computer Science*, vol. 55, no. 2, pp. 103-117, 2001.
- [12] *Jcontract*, product page at <http://www.parasoft.com>.
- [13] G. T. Leavens and Y. Cheon, *Design by Contract with JML*, Draft paper (<http://www.jmlspecs.org>), 2005.
- [14] J. Barnes, *High Integrity Software: The Spark Approach to Safety and Security*. Addison-Wesley, 2003.
- [15] D. Evans, *Splint User Manual*, Secure Programming Group, University of Virginia (<http://www.splint.org>), June 5, 2003.
- [16] *ESC/Java2*, [secure.ucd.ie/products/opensource/ESCJava2](http://secure.ucd.ie/products/opensource/ESCJava2).
- [17] J. Meyer and A. Poetzsch-Heffter, "An architecture for interactive program provers." In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, vol. 1785, pp. 63-77, 2000.
- [18] L. Burdy, A. Requet, and J.-L. Lanet, "Java Applet Correctness: A Developer-orient Approach." K. Araki, S. Gnesi, and D. Mandrioli, Eds. In *Proceedings of the International Symposium of Formal Methods Europe*, LNCS, vol. 2805, 2003.
- [19] D. Guaspari, C. Marceau, and W. Polak, "Formal verification of Ada programs," *IEEE Transactions on Software Engineering*, vol. 16, no. 9, pp. 1058-1075, September, 1990.
- [20] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe, *Extended Static Checking*, Compaq Systems Research Center, Research Report 159. ([citeseer.ist.psu.edu/detlefs98extended.html](http://citeseer.ist.psu.edu/detlefs98extended.html)), December, 1998.
- [21] 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)*, vol. 37, no. 5, pp. 234-245, June, 2002.
- [22] J. van den Berg and B. Jacobs, "The LOOP compiler for Java and JML." In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, LNCS, vol. 2031, pp. 299-312, 2001.
- [23] *The PVS Specification and Verification System*, <http://pvs.csl.sri.com>.
- [24] B. Jacobs and E. Poll, "Java Program Verification at Nijmegen: Developments and Perspective." In *Proceedings of the International Symposium on Software Security - Theories and Systems (ISSS 2003)*, LNCS, vol. 3233, pp. 134-153, 2003.
- [25] B. Konikowska, "Two Over Three: A Two-Valued Logic for Software Specification and Validation Over a Three-Valued Predicate Calculus," *Journal of Applied Non-Classical Logics*, vol. 3, pp. 39-71, 1993.
- [26] D. Gries and F. B. Schneider, "Avoiding the Undefined by Underspecification," in *Computer Science Today: Recent Trends and Developments*, vol. 1000, J. v. Leeuwen, Ed.: Springer-Verlag, 1995, pp. 366-373.
- [27] P. Chalin, "Are Practitioners Writing Contracts?" In *Proceedings of the Workshop on Rigorous Engineering of Fault Tolerant Systems*, Newcastle, UK, July, 2005.
- [28] C. B. Jones and C. A. Middelburg, "A Typed Logic of Partial Functions Reconstructed Classically," *Acta Informatica*, vol. 31, no. 5, pp. 399-430, 1994.
- [29] J. Bloch, *Effective Java Programming Language Guide*. The Java Series. Addison-Wesley, 2001.
- [30] H. Barringer, J. H. Cheng, and C. B. Jones, "A Logic Covering Undefinedness in Program Proofs," *Acta Informatica*, vol. 21, no. 3, pp. 251-269, 1984.
- [31] C. B. Jones, *Systematic Software Development using VDM*, 2nd ed. Computer Science Series. PHI, 1990.
- [32] J. H. Cheng and C. B. Jones, "On the usability of logics which handle partial functions." C. Morgan and J. C. P. Woodcock, Eds. In *Proceedings of the 3rd Refinement Workshop*, Springer Workshops in Computing Series, pp. 51-69, 1991.