

# Are the Logical Foundations of Verifying Compiler Prototypes Matching User Expectations?

Patrice Chalin

*Dependable Software Research Group (DSRG),*

*Department of Computer Science and Software Engineering, Concordia University, Montréal, Québec, Canada*

**Abstract.** The Verifying Compiler (VC) project proposals suggest that mainstream software developers are its targeted end-users. Like other software engineering efforts, the VC project success depends on appropriate end-user consultation. Industrial use of program assertions for the purpose of Run-time Assertion Checking (RAC) is becoming commonplace. A likely next step on the path to VC adoption is the use of assertions in Extended Static Checking (ESC), a fully automated form of Static Program Verification (SPV). Unfortunately, all current VC prototypes supporting SPV, adopt a semantics which is *unsound* relative to the standard run-time interpretation of assertions. In this article, 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 RAC and SPV 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.

**Keywords.** Assertions; Survey; Industrial practice; Logical foundations; Runtime assertion checking; Static program verification; Extended static checking; Design by contract.

## 1. Introduction

The Verifying Compiler (VC) project is a core component of the Dependable Systems Evolution Grand Challenge—also named GC6 [63, 68]. Such a compiler is meant to be used to automatically prove that a program or program component is correct. Correctness is defined relative to specifications written using *program assertions* (and other redundant annotations). The business case for the Challenge was well articulated by Hoare and Misra during the 2005 IFIP Working Conference on “Verified Software: Theories, Tools, Experiments” [37]: in particular, technological feasibility seems at hand—as was made evident by convincing demos of early VC candidate language and tool prototypes.

Unfortunately, all of these early VC languages adopt an assertion semantics which is *unsound* relative to the standard run-time interpretation of assertions—which we will refer to as the Run-time Assertion Checking (RAC) semantics. One of the key objectives of formal methods research is industrial adoption. While early program verification systems were crafted as proof-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. As a means of addressing this deficiency, this article 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 static program verification.

### 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 [34]. 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) [41]. 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 *Static Program Verification* (SPV) [41].

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 static program verification was headed independently by academics in formal methods (for the most part). Significant advances have been made in static program verification in the last decade. This, coupled with the phenomenal growth in raw processing power, has lead us to a point of conjunction where SPV 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 SPV, are said to perform *Extended Static Checking* (ESC) [26].

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

RAC and SPV 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;
    }
}
```

Evaluation of the call `m(0)` while run-time checking is enabled will result in an *exception* being raised, due to division by zero. Yet, for example, LOOP, a static program verifier for Java [64], would interpret 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. Although we wrote our example in Java, it could just as well have been written in e.g., SPARKADA [2] or Spec# [3] with similar results. (This problem of unsoundness of assertion semantics in SPV tools will be discussed at greater length in Section 6.4.)

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 SPV tools offer differing interpretations. We will also ask “Should RAC and SPV 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

We believe that assertion-based formal methods are one of the most promising avenues for the industrial adoption of formal methods, in particular because they [12, 15]:

- 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 the benefits gained are *proportional* to the effort invested.

These are key points easing managerial buy-in, which is an essential prerequisite to successful adoption [10, 61]. Programmers in several application domains have already integrated the use of assertions into their development practices, though current use of assertions appears to be “ad hoc” for the most part. In contrast, more disciplined methods include the use of assertions in systematically capturing contracts—e.g., by means of function/method preconditions and postconditions as is done in Design by Contract (DBC) [51]. Run-time or extended static checking can be applied to varying degrees in either case.

A strong belief in the importance of assertion-based formal methods is also shared by the international research community; the mobilization of researchers in response to Tony Hoare’s Verifying Compiler Grand Challenge is testimony of this [35]. As was mentioned earlier, with the official kickoff meeting for the VC Grand Challenge having been recently held [37], we believe that the survey reported in this article is timely. The lack of end-user consultation during the development of VC prototypes, and the ensuing unsoundness of the assertion semantics in such tools, places the VC GC at risk—we will address this point at greater length in Section 4.1.

## 1.4. Outline

The remainder of this article is organized as follows. A review of assertions, run-time checking and static program verification is provided in Section 2, followed by a brief discussion of the logical semantics of assertions from the perspective of current RAC and SPV 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 Section 6, and conclude in Section 7.

## 2. Assertions, RAC and SPV

### 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 [34]. 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 article. 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, ESC or SPV tools.

### 2.2. RAC

#### 2.2.1. “Native” 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 [27],
- `assert`, `requires`, `ensures` and `invariant` clauses of Spec# (a language extension to C#) [3], and
- the Eiffel `check`, `require`, `ensure` and `invariant` statements [51].

All of these 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 [33]). Variants include: special assertions that are kept in production code; assertions whose failure may result in special error logging and/or a behavior other than abrupt termination.

### 2.2.2. Assertions in stylized comments

A common strategy used to deal with programming languages that fail to provide sufficiently rich support for assertions, is to extend the language in a manner that is fully backwards compatible: assertions are written inside stylized comments. The advantage of extending the language in this way is that it allows standard compilers to continue processing source files even if these are annotated with specifications. One such language extension for C is called APP [56]. APP supports assertion-based specification constructs inside of `/*@ ... @*/` comments. Assertions in APP, which can even contain a basic form of quantifiers, can be used to specify function pre- and post-conditions as well as inline assertions. Tool support for APP (essentially based on a C preprocessor) allows assertions to be embedded inside C code for the purpose of run-time checking.

## 2.3. Design by Contract

Design by Contract (DBC) refers to a *method* of developing object-oriented software defined by Bertrand Meyer [51]. The main concept that underlies DBC is the notion of a precise and formally specified agreement between a class and its clients. Such an agreement, named a *contract* in DBC, is called a Behavioral Interface Specification (BIS) in its most general form [66]. Contracts and BISs are built from class invariants, method pre- and post-conditions, (and other constructs) which are expressed by means of assertions. DBC as a programming language feature refers to a limited form of support for BISs where assertions are restricted to be expressions that are *executable*. While this does not necessarily entirely preclude quantifiers, most DBC languages do not support quantifiers. It is important to note that it is the individual assertion expressions that are restricted to being executable, not the contracts. Hence, for example, a method contract might not be executable if its postcondition describes properties of the method result rather than how it can be computed.

Eiffel is the only active programming language with built-in support for DBC [51]. Eiffel currently only supports run-time assertion checking; in fact, use of the term “design by contract” is taken to imply RAC-only support. When contracts are subject to static verification, we speak of Verified DBC (VDBC) [18, 65]. An active topic of research is the definition of language extensions for certain mainstream languages that add support for DBC (and RAC). For example,

- Spec# for C# [3],
- Java Modeling Language (JML) [8, 49], Jass [5], and Jcontract [53] for Java.

SPARKADA is an extension for (a subset of) Ada that supports VDBC along with other features such as flow analysis [2]. Actually, to varying degrees, all of these language extensions support BIS features such as: frame properties for method contracts and specification-only fields.

## 2.4. Current ESC and SPV tools

Current extended static checking tools include:

- Splint for C [22] (formerly LCLINT [23], a member of the Larch family of languages and tools [30]),
- SPARK Examiner for SPARKADA [2],
- ESC/Java2 [44] and Jive [52] for Java,
- JACK [9] and Krakatoa [50] for JavaCard,
- Boogie for Spec#,

Older systems include Penelope for Ada [29], and ESC/Modula-3 [19], the predecessor to ESC/Java [26] and ESC/Java2. ESC/Java2 and JACK use JML as an annotation language; Jive is being adapted to use JML as well. The KeY tool has also been recently adapted to support JML as a “constraint language” for UML class diagrams [1, 21]. Splint can only process a limited form of assertions, contrary to its predecessor LCLINT, which accepted general BISs. On the other hand, while Splint supports fully automated verification (ESC), LCLINT only performed type checking.

There are much fewer tools that can be used to perform complete verification of (sequential) programs written in mainstream languages. In fact, the only one that we are aware of is the LOOP tool developed at the Radboud University Nijmegen [64]. 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 verification proofs [58]. Case studies in the use of LOOP are reported by Jacobs and Poll [38]. Finally, we mention the Omnibus [65] and Perfect developer [18] tools that support RAC, VDBC and SPV (in the case of Omnibus) but for their own proprietary object-oriented languages.

### 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 static 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 operational semantics of program assertions, as realized by run-time checkers, naturally corresponds to a three-valued logic [46]. Hence, evaluation of an assertion expression can yield

- true: e.g.  $3 < 5$ ,
- false: e.g.  $3 > 5$ ,
- undefined/error.

The latter can be due to a run-time exception, as would arise in the evaluation of  $3/0 < 5/0$ , or even non-termination. The law of excluded middle “ $E \vee \neg E$ ”, fundamental to two-valued logic, no longer holds, as programmers are well aware.

In three-valued logics, it makes sense to have binary Boolean connectives that are non-strict in their second argument (called McCarthy connectives). 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 [27, §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. Thus, in Java,

`true || E`  
will evaluate to *true* for any *E*, whereas  
`true | E`

is equivalent to “ $E \mid \text{true}$ ” and hence may evaluate to *true*, if *E* is *true* or *false*, and undefined if *E* fails to yield a truth value.

#### 3.2. SPV and ESC semantics

All of the static program verification and extended static checking tools mentioned in Section 2.4 interpret assertions as if they were predicates in a classical two-valued logic that models partial function as under-specified total functions as we explain next. (This approach is attributed to Gries and Schneider [28].)

Let  $f : D \rightarrow R$  be a partial function with domain  $\text{dom } f$ , then for all  $v$  in  $D$  outside of  $\text{dom } f$ ,  $f(v)$  is assumed to have some unspecified value from  $R$ . 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 the LOOP tool 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, but before doing so, we examine the more significant issue of unsoundness of assertion semantics in all current SPV tools.

### 3.3. Unsoundness

Since type systems are very familiar, we begin by explaining the basic concepts of specifications, correctness and soundness relative to these simpler formal systems. We feel that this provides a strong appreciation for the nature of the unsoundness in assertion semantics. Consider the following Java method:

```
int add(int i, String s) {
    return i + s;
}
```

Ignoring types for a moment, this method can be compiled into the following JVM code

```
iload_1      // push first parameter onto the stack
aload_2      // push second parameter onto the stack
iadd         // add top two arguments on the stack
ireturn      // return the result
```

though attempts to execute this method will result in a run-time error because the `iadd` operation does not expect its second argument to be a string. Thankfully, such errors can be statically detected and reported by a standard Java compiler. This is because the compiler contains a type checker that implements the rules of a *sound* type system; i.e. if the checker reports that a program is type correct, then *all* runs of the program will be free of run-time type errors. (There are a few situations where Java's type system is inadequate for guaranteeing *static* type safety. Of course, overall type safety is ensured by the use of run-time checks in these cases.)

We agree with Pierce that (static) type systems are “the most popular and best established lightweight formal methods” [54]. Hence, when developers write their programs (e.g. in Java) they are in fact at the same time annotating them with type *specifications*. For example, the specification of the `add` method is provided by its signature. The type checkers inside compilers of statically typed languages like Java can automatically prove that a program is *correct* with respect to its specification. In the case of our example: for the body of `add` to be correct with respect to its type specification it must commit to returning an `int` under the assumption that it is given an `int` and a `String` as arguments. Of course, the type checker cannot guarantee that the body of `add` will satisfy its type specification and hence, it reports a compile-time error.

We can easily adapt `add` to allow the compiler to check it:

```
int add(int i, String s) {
    return i + Integer.parseInt(s);
}
```

This version of `add` is correct with respect to its type specification. Unfortunately, its execution can still result in a run-time exception if the string argument does not represent an integer. There are no type annotations that we could add that would allow the compiler to rule out the possibility of a run-time error. This highlights the limited expressiveness of the specifications that type systems permit. Language designers purposefully restrict expressiveness of type systems in order to ensure that type correctness can not only be checked but done so efficiently.

While type specifications are not sufficiently expressive, we can provide a method contract for `add` using assertions written in the Java Modeling Language (JML) that rules out the possibility of a run-time exception:

```
/*@ requires Integer.parseable(s);
   @ ensures \result == i + Integer.parseInt(s);
   @ signals (Exception e) false; // forbid exceptions
   */
int add(int i, String s) {
    return i + Integer.parseInt(s);
}
```

ESC/Java2, for example, can easily check that `add` meets its specification [45], and furthermore, it will check that all callers provide a string argument representing a valid integer.

As was the case with type checkers, SPV tools can guarantee that programs will be free of specification violations, provided it embodies a *sound* assertion semantics. Unfortunately, all current SPV tools define an assertion semantics that is unsound. That is, there are cases where a checker claims that a program is correct when in fact an assertion violation would be reported at run-time if the program were executed. As a slight build up to our motivating example of unsoundness, consider the following assertion:

```
assert a[0] == a[1];
```

This assertion will be true at run-time precisely when `a` is a non-null reference to an array of length 2 or more for which the first two elements are equal. These are the same assumptions that an SPV tool or VC would need in order to statically infer that the given assertion was valid. Now suppose that a programmer had inadvertently mistyped the second index, giving:

```
assert a[0] == a[0];
```

At run-time, this assertion will be true precisely when `a` is a non-null reference to an array of length 1 or more. On the other hand, because partial functions are modeled as underspecified total functions (cf. Section 3.2), all current SPV tools would treat this assertion as equivalent to *true*.

### 3.4. Soundness and the Verifying Compiler

Strictly speaking, soundness is not necessary. After all, useful technologies like Extended Static Checking (ESC) have traditionally been unsound by design [19]—the main motivating factor being to maximize the utility and efficiency of the tool, at the expense of possibly missing some errors and/or reporting false positives. We are seeing a change in philosophy in next generation ESC tools such as Spec# which have soundness as a prime objective: “The Spec# compiler uses a combination of static-analysis techniques and run-time checks to guarantee soundness of the language” [3]. Naturally, soundness is a requirement for the future Verifying Compiler as well [35]—how else are software engineers to have confidence in the applications they build if the VC is unsound? In the next section we will be examining whether soundness is something that industrial VC end-users would want.

## 4. Survey

### 4.1. Motivation—A Software Engineering Perspective

The Verifying Compiler project is a major undertaking of the international community, with an estimated development span of one to two decades. Empirical data clearly indicate that the probability of failure of a software project rapidly increases with project complexity [39, 48], size and development time [59]. Hence, technological issues aside, the VC project is, by its very nature, at high risk of failure. It is clear that Software Engineering best practices will need to be rigorously applied in order to help mitigate risks.

The well cited Standish Group CHAOS research reports have, since 1994, accumulated data from hundreds of thousands of software projects. The reports are “aimed at providing an understanding of the scope of application software development failures; the major factors that cause these projects to fail; and recognizing the key ingredients that can reduce failures” [60]. Year after year, the top two factors reported as contributing to project success (and hence the lack of these factors contributing to project failure) are:

- end-user involvement, and
- management support.

These factors, along with the related factor of *end-user acceptance*, are also the conclusions arrived at by numerous other studies—e.g. [25, 32, 47, 57, 62]. Key requisites to achieving these success factors are

- a good understanding of a customer’s current way of doing business and
- a practical (ideally staged) integration strategy adapted to current business practices that minimizes integration costs and maximizes return-on-investment (ROI) [61].

The VC scope is clear [67]: its targeted end-users are

“real programmers” writing code for normal commercial, industrial or open source software using mainstream languages.

While it is recognized that current software written using mainstream languages will offer a challenge to verification efforts, restricting the initial VC scope in this way is essential to gaining broad acceptance from industry—asking industry to discard its code base and start from scratch is not an option [35]. Hence, it is under this setting that we embarked on our programmer survey: i.e., driven by our belief in the Verifying Compiler Grand Challenge, the formidable size of the undertaking, and motivated by software engineering best practices.

### 4.2. Hypothesis

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 static program verification (SPV). Our main hypotheses were the following:

- The majority of industrial developers do not want SPV tools to interpret partial functions as underspecified total functions (as is the case in all current SPV tools).
- The majority of industrial developers want SPV tools to adopt an interpretation of assertions that is consistent (i.e. sound) relative to current RAC semantics.

### 4.3. Participants

Initially a few hundred e-mail invitations were sent out by members of the Dependable Software Research Group (DSRG), 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.

### 4.4. 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.

## 5. Results

Over 281 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 (Section A)

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 (of course, `_[_]` also has the program state  $\sigma$  as an implicit argument). Restated in programming terms, the exception can be seen as reporting a violation of the operator’s *precondition*.

The purpose of the first question (see Fig. 1) was to determine the relative importance that programmers give to this exception reporting mechanism in the context of expression evaluation. The majority of respondents chose “very important” or “important”, with a mean between these two answers—see Table 1. Three percent (3%) of respondents remarked that it would be even better if such errors were reported at compile time.

**A. Run-time errors/exceptions during 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**?

Choose one of: *very important* (2), *important* (1), *neutral* (0), *undesirable* (-1).

**Definition of precondition**

The integer division  $m/n$  is **undefined** when  $n$  is 0. A **precondition** is the condition under which it is meaningful to invoke an operator, function or method. E.g. integer division  $m/n$  has precondition  $n \neq 0$ . It is an error to invoke an operator when its precondition does not hold (i.e. when the precondition is violated). Most languages report an error/exception in this case.

**Examples**

- `a[0]` will raise a “null pointer” exception when `a` is null.
- `sum/n` will raise a “division by 0” when `n` is 0.

Fig. 1. Survey question, Section A.

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

Question	very important (2)	important (1)	neutral (0)	undesirable (-1)	Mean
<b>A.1</b>	<b>59%</b>	32%	6%	2%	1.5
<b>B.1</b>	27%	<b>44%</b>	24%	5%	0.9

**B. Run-time errors/exceptions and arithmetic overflow**

Some languages like C# allow arithmetic overflow to be reported as an exception that can be caught and handled by user code.

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

Choose one of: *very important (2), important (1), neutral (0), undesirable (-1).*

**Definition of arithmetic overflow**

Since integral types like `int` have a fixed precision, some arithmetic operations can yield a result that exceeds the precision of `int`.

**Example**

16-bit integers can represent the values in the range -32768 to 32767 inclusive; therefore `32767+1` will result in an overflow because 32768 is outside the range of values.

**Fig. 2.** Survey question, Section B.

In support of this, we are witnessing the emergence of null-pointer analysis, e.g. in popular Java IDEs, that allows the *static* detection of potential null dereferences. Most of these systems are based on the work of Fähndrich and Leino [24] who propose a simple way of retrofitting the type systems of object-oriented languages to support distinguished nullable and non-null types.

Of the respondents who answered “neutral” or “undesirable”, 24% cited efficiency as a justification for not wanting run-time exceptions to be reported—particularly in certain application domains such as embedded controllers.

**5.2. Run-time exceptions due to arithmetic overflow (Section B)**

Unfortunately almost all languages that have evolved from C have inherited what some would consider a deficiency: arithmetic overflows are silently ignored. The ANSI C standard states that reporting of overflows is implementation dependent, though few compilers actually support the reporting of overflow [43, §A7]. A recent exception is C# which defines 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.

Question **B.1** 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), which is a lower rating than for question **A.1**. A common (7%) 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 (Section C)**

The first question of Section C, given in Fig. 3, asked respondents whether they knew what program assertions were. If not, respondents were directed to examples and a short explanatory text. Ninety-five percent (95%) of respondents answered yes. Unless specified otherwise, for the remainder of this article, the results presented will be relative to the answers of respondents who knew what assertions were.

The next question (**C.2**) 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. Eighty-one percent (81%) of respondents chose “error/exception” as indicated in Table 2.

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 vs. outside the context of an assertion. Seventy percent (70%) of respondents answered that expressions should be given the same interpretation. In fact,

### C. Errors/exceptions in program assertions (questions were presented on separate pages)

A **program assertion** is essentially a boolean expression inserted at a point in a program where it is believed to be true.

#### C.1. Do you know what program assertions are? (yes/no)

If you have used assertions, please answer the following questions based on your experiences in using assertions.

#### 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)?

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

#### 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**.

#### Examples

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

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

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

(b) Assertions in specially formatted comments

#### Definition

Support for **program assertions** is provided for most languages either directly in the language (using an assert statement or macro as in function  $f$ ), by means of an external library or a separate preprocessing tool (that often accept assertions as specially formatted comments like in function  $g$ ).

Run-time Assertion Checking (RAC), is a common use of assertions in which assertion expressions are evaluated at run-time during the normal course of execution of a program.

Fig. 3. Survey questions, Section C.

this is the case for all of the systems supporting run-time assertion checking that were mentioned in Sections 2.2 and 2.3, except for JML (because the JML compiler *attempts* to emulate an interpretation of assertions that matches classical two-valued logic—we will return to this briefly in Section 6.4). 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.

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

Question	true	false	error/exception	other
<b>C.2</b>	2%	9%	<b>81%</b>	8%
<b>D.1</b> <EXPR>				
t    (nullRef[0] > 0)	<b>74%</b>	2%	21%	3%
(nullRef[0] > 0)    t	8%	6%	<b>82%</b>	4%
nullRef[0] > 0	1%	7%	<b>89%</b>	3%
t   (nullRef[0] > 0)	5%	6%	<b>85%</b>	3%

#### D. Errors/exceptions in assertions during Static Analysis (SA)

**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.

**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**? [<EXPR> are provided in Table 2 along with responses]

##### Example

The following illustrates sample output from a Static Analysis tool. It is reporting that the call to `average` at line 10 violates the precondition of `average`.

<pre> 1  int average(int a[], int n) { 2    /*@ require(a != NULL &amp;&amp; n &gt; 0); @*/ ...    ... 5  } 6  7  void main() { 8    int b[10]; 9    ... 10   int v = average(b, -1); 11   ... 11  } </pre>	<pre> ----- Line 10: precondition violation       int v = average(b, -1);                     ^ Associated declaration: line 2:       /*@ require(a != NULL &amp;&amp; n &gt; 0 ...                     ^ ----- </pre>
---	--

**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)

##### Definitions

The state of a program (i.e. **program state**) at any given moment is determined by the value of all variables and objects in existence that that moment. When we speak of a program state  $S$  in the context of the static analysis of an expression  $E$  we mean it in the following way: assuming that the program is in a state  $S$ , what would the value of  $E$  be?

Fig. 4. Survey questions, Section D.

## 5.4. Exceptions in assertions during static analysis (Section D)

We chose to phrase **D.1** (Fig. 4) using the slightly more general term “static analysis” (i.e. static program verification) rather than “extended static checking”. 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 this same interpretation in the context of 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, and in the questionnaire, this expression we headed by the following explanation:

The next sample uses Java’s logical-or (rather than conditional-or). The logical-or operator always evaluates both of its arguments. Ada, C# and Eiffel have a similar logical-or.

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 synonymous (as is the case in two-valued logic—cf. Section 3.2). Question **D.2** generalizes **D.1**. Some believe **D.2** is superfluous on the grounds that consistency is obviously desirable, yet it should be noted that *none* of the languages currently supporting both RAC and SPV offer a consistent semantics—including JML, SPARKADA and Spec#. Technically, the question should have been phrased in terms of soundness (of SPV relative to the RAC semantics), but we felt that use of the term consistency was preferable for a general developer audience. Seventy-one percent (71%) of respondents answered “yes”; 40% of those who provided extra comments gave simplicity or consistency as a justification. One respondent wrote: “[consistency avoids] special cases; helps me remember how things work”.

<p><b>D. Errors/exceptions in assertions during Static Analysis (SA)—continued</b></p> <p>Assume that &lt;EXPR&gt; in the function h is replaced by one of the &lt;EXPR&gt; given in the table below. For each expression, answer the following question.</p> <p><b>D.4. During static analysis</b> how should each of the following assertion expressions, &lt;EXPR&gt;, be interpreted when <b>a and b are null</b>?</p> <p><b>Why the strange &lt;EXPR&gt;?</b></p> <p>Of course, the given assertion expressions would not be used in practice. They have been chosen to help us better understand how you wish potential errors to be treated during static analysis.</p>	<pre>int g(int m) {     return m/m; } void h(int a[], int b[]) {     assert(&lt;EXPR&gt;)     ... }</pre>
--	---

Fig. 5. Survey questions, Section D, continued.

Table 3. Responses to Question D.4.

Question	true	false	error/exception	other
<b>D.4</b> <EXPR>				
a[0] == a[0]	18%	7%	72%	3%
a[0] == b[0]	11%	8%	77%	4%
a[0] == b[1]	6%	10%	79%	5%
g(a[0]) == a[0]/a[0]	8%	9%	80%	4%
a[0] == 0    a[0] != 0	10%	11%	72%	8%

The respondents who answered something other than “error/exception” to **C.2**, or “no” to either **C.3** or **D.2** (i.e. 43% of respondents) were asked to complete question **D.4**. We anticipated that respondents who e.g., 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 3, 18% 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 11% of respondents recognized this. As was explained in Section 3.2, the third expression would have an undetermined value in two-valued logic since the array access operator is being applied to different arguments. The next expression involves a function `g` (see Fig. 5). 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 (Section E)

Survey participants were asked about their use of assertions in Section E of the questionnaire (Fig. 6). The distribution of answers to **E.1** from all respondents is shown in Fig. 7; the mean is 3.1, slightly more than “regularly”. Seventy-three percent (73%) of the respondents answered that assertions were used at their institution (**E.3**). In those cases where a assertions were not used, the respondent was asked why. A breakdown of the reasons why assertions are not used is given in Table 4.

The distribution in the use of assertions relative to RAC and ESC is given in Table 5. Note that we adjusted the count in the number of uses of ESC to include only those for which an ESC tool was named. The refined distribution in the non-RAC and non-ESC uses of assertions (last row of Table 5), is given in Table 6. Assertions are used in a broad range of applications such as:

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 / pre-processors / IDEs, speech / image / vision (and other real-time signal processing), telecommunications / network software, various “freeware” applications.

**E. Errors/exceptions in assertions during Static Analysis (SA)**

**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).

**E.2.** What is the main application domain of the software developed/maintained at your institution?

**E.3.** Are program assertions used at your institution? (yes/no)

**(The following questions are presented only if the answer to E.3 was “yes”.)**

Choose a representative product or set of products developed/maintained at your institution and that make use of assertions. Then answer the following questions.

**E.4.** What are assertions used for? Choices: run-time assertion checking (RAC); extended static checking (ESC); other.

**E.5.** Which tools, libraries or languages are used to support your use of assertions?

The following questions are optional.

**E.6.** (If applicable) Why are assertions not used for **run-time assertion checking**?

**E.7.** (If applicable) Why are **static analysis** tools not used?

**E.8.** Please indicate the top three languages that this product(s) is/are written in. Choices: Ada, C, C++, C#, Eiffel, Java, other.

**E.9.** On average, for every 100 lines of code, how many of these lines are assertions? Please answer this question for the top three languages previously indicated.

Fig. 6. Survey questions, Section E.

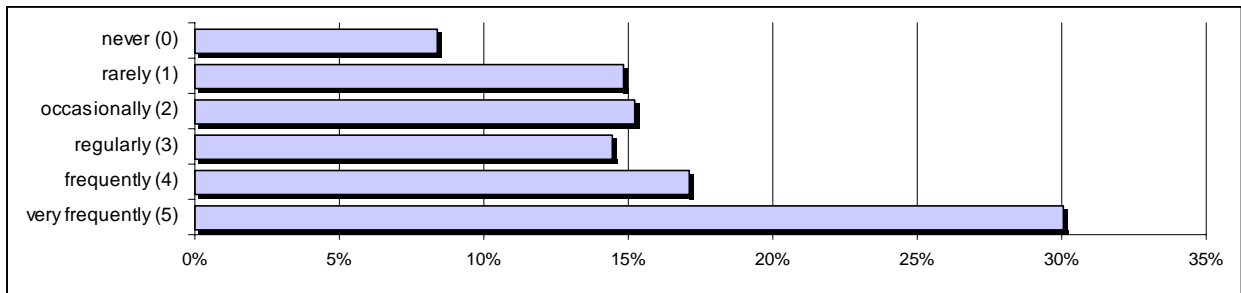


Fig. 7. Responses to Question E.1

The top RAC tools used are:

- assertions facility provided by the programming language (84%),
- custom facilities (5%),
- \*Unit (mainly JUnit) (4%).

JML and Jass were also mentioned (1%). Of the 9 explicitly named ESC tools we find:

- Splint [22]
- Spark Examiner [2]
- ESC/Java2 [17]
- VDM tools [40]
- Nice [7]

Splint was named twice and all others once, leaving us with three proprietary (unnamed) tools. A distribution of the top five programming languages used in conjunction with assertions is given in Fig. 8. Respondents estimated that for a typical project, between 1.5% and 5.2% 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 [11].

Table 4. Why assertions are not used

lack of developer training	22%
informal assertions used instead	21%
inadequate language/tool support	18%
do not know why	13%
not needed	7%
too much effort	6%
other	12%

Table 5. Use of assertions

RAC	95%
ESC	5%
other	9%

Table 6. Other uses for assertions

debugging	39%
unit testing	28%
documentation	11%
rest	22%

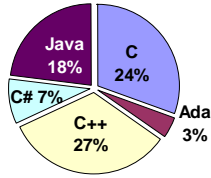


Fig. 8. Use of assertions, by language

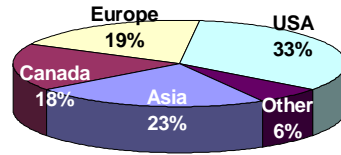


Fig. 9. Work location profile

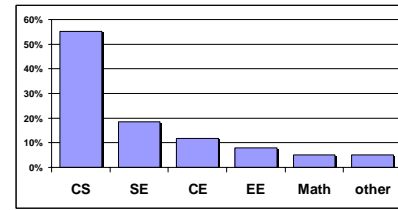


Fig. 10. Respondent degrees

## 5.6. Demographics

Respondents worked in industry (82%), academia (21%) or other (7%)—e.g. government, self-employed, or retired. Some worked in more than one location, hence the total of the previous percentages is more than 100. The distribution of the workplace of respondents is given in Fig. 9. On average, respondents had 11.2 years of programming experience and the highest programming-related degree—completed or in progress—was a bachelors (47%), a masters (31%), Ph.D. (10%) or diploma (3%), while 10% had no programming-related degree. Fig. 10 illustrates the distribution of degrees by kind (the abbreviations are for Computer Science, Software Engineering, and Computer Engineering, Electrical Engineering).

## 5.7. Threats to validity

One of the main threats to (internal and external) validity has to do with the “selection” of respondents. The ideal would have been a random sampling from the pool of all industrial programmers, but this is clearly impractical. As was explained in Section 4.3, participants were recruited on a voluntary basis by sending general invitations. Originally, invitations were sent to institutions via representatives—i.e., colleagues and contacts (mainly from industry) of members of our research group. Subsequently, invitations were posted to news groups and online bulletin boards. It is possible that this would have resulted in a biased sampling, but we do not believe the bias, if any to be significant. In particular we note that the 281 respondents represented 216 different “organizations” (i.e. self-employed, or belonging to an institution).

### 5.7.1. Internal validity

One threat specific to internal validity would have been for two or more participants to discuss the survey and its answers before completing it. While a simple warning could have been added to the survey introductory text, such a factor did not occur to us at the time. We believe that a mutual influence between respondents would have a low rate of incidence, especially between different organizations. Also, with an average of 1.3 respondents belonging to the same “organization”, this makes mutual influence somewhat unlikely.

### 5.7.2. External validity

Can the results be taken as representative of programmers in general? One possibility, other than the comments made above with regards to the method used to solicit participants, is that the views of geographically diverse populations would be different. We note that there is a higher representation of North American developers (51%) as compared to any other region of the world. As an sample of the variability consider the following response profiles:

Question	North American developers	Non North American developers	Mean
<b>D.1</b> , <code>nullRef[0] &gt; 0</code>	94% (for error/exception)	84% (for error/exception)	89%
<b>D.2</b>	72%	71%	71%

Hence, while the study hypotheses get stronger support from North American developers, the support from other regions is sufficiently strong as well.

## 6. What practitioners want and why

In this section, we comment on what practitioners want (as indicated by the survey results), as well as some of the reasons that may be motivating this preference.

### 6.1. Logical foundations

While some authors have argued that an interpretation of assertions consistent with two-valued logic is adequate in the context of program specification and verification [31], the survey results suggest otherwise. The laws of classical logic do not hold for assertion expressions in the context of imperative mainstream 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 but in either case a bug has been exposed). Even when presented with an instance of the law of excluded middle (see the last expression of **D.4** in Table 3), respondents favored interpreting it as an error/exception (72%) or even false (11%) rather than true (10%). Hence, the survey results clearly indicate that practitioners want SPV tools to adopt a logical semantics 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.

Several logics exist which support this point of view, most of which are a kind of three-valued logic. Barringer, Cheng and Jones have explored various formulations of three-valued logic for use in the context of program specification and verification [4], finally settling upon what has become known as the Logic of Partial Functions (LPF), the logic underlying the Vienna Development Method (VDM) [40]. LPF adopts a choice of logical operators (called Strong Kleene) that are non-strict and monotonic [16]. 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 [16], conditional binary logical 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 [11], 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 correctness 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.

While we noted, at the start of this section, that the laws of classical logic do not hold for assertion expressions in the context of most programming languages, it is obvious that the evaluation of 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.

We emphasize that the survey results support an *interpretation of assertions* that is consistent with a three-valued logic. No statement is being made concerning the *necessity* of using a three-valued logic in the provers underlying SPV tools. It is well known that two-valued logics are sufficient to model three valued logics—e.g. [42, 46]. In the remaining subsections we explore some of the top factors that may have contributed to the preference in logical foundations expressed by practitioners.

### 6.2. Exceptions help detect errors

A partial function that occurs in a program must not be applied to values outside its domain—at best, because it is senseless, at worst because bad things can happen, such as non-termination. One tool used in defensive programming to guard against the misuse of a function is to throw an exception when illegal arguments are supplied to it. Such a practice can help in detecting programming errors as close as possible to their point of occurrence.

### 6.3. Ignoring exceptions 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. Hence the common recommendation in programming is “Don’t ignore exceptions” — Item 47 of Bloch’s *Effective Java* [6].

### 6.4. Consistency (soundness)

As was reported in the previous section, 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 no current programming/specification language that has support for both RAC and SPV actually provides a consistent semantics across tools. To be technically accurate: we can only speak of consistency relative to a fixed valuation of a quantifier free assertion. While we used the term “consistency” in the survey, what respondents actually want is soundness of SPV tools. (Completeness is certainly desirable, but Gödel’s incompleteness theorem reminds us that it cannot be attained for sufficiently rich logics.)

Lack of consistency would also require that practitioners 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 [20]. The need to learn and use two logical systems will have an impact on cost (e.g., due to training) and productivity. The impact on productivity should be apparent to anyone who has developed software in two *syntactically similar* but (sometimes *subtly different*) languages—e.g., C++, Java or C#. The minor differences in language semantics often give rise to subtle bugs.

Consistency (soundness) could be achieved by adapting the semantics of run-time checkers to conform to classical logic, but as is pointed out by Hoare, this is unrealistic in the context of imperative languages [36, §9.3]. As has been commented elsewhere [13], 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, so that the *same* values are consistently used throughout the (run-time) interpretation of a program’s specification. Furthermore, attempts to do this (in the JML run-time checker, for example) results in an unnecessary increase in instrumented code size [55].

## 7. Conclusion

In this article, we have presented the results of a programmer survey that confirms that a large number of application domains make use of program assertions for the purpose of run-time checking. We have also noted that static program verification (SPV) technology is reaching a level of maturity that makes it feasible to automatically validate a nontrivial number of program assertions, possibly even though 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.

The timing also seems appropriate for the official launch of the Verifying Compiler project. But, while a Verifying Compiler holds the promise of helping engineers write more dependable software, we note that all current VC prototypes adopt a semantics for assertions that is unsound. This is clearly unacceptable for a Verifying Compiler.

Empirical studies clearly indicate that larger projects tend to fail more often; due to the grand nature of the VC project it is at high risk of failure. With the hope of mitigating project risks, we followed software engineering best practice and consulted with the targeted VC end-users asking them what kind of logical foundations they wanted for assertions. While the importance of conducting empirical studies as a part of the Dependable Systems Evolution Grand Challenge efforts has been recognized [67, §4.2], the study reported in this article is, to our knowledge, the first such study.

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 programming languages—i.e., application of a partial function to arguments outside its domain is an error;
- is consistently adopted across SPV 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 have discussed some of the reasons why a two-valued

logical interpretation of assertions would be counter-intuitive for programmers. Finally, we noted that SPV and ESC tool designers are free none-the-less to make use of provers built for classical logic to realize assertion semantics.

Guided by the survey results, we are currently finalizing the formal definition of an alternate semantics for JML [13, 14] while experimenting with its realization in the JML compiler [55]. Our next objective will be to implement the new semantics in ESC/Java2.

## 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, Stuart Thiel in particular. Preliminary survey results were presented at the Third International Conference on Software Engineering and Formal Methods (SEFM'05)—we are also thankful for the helpful remarks provided by the SEFM referees.

## References

- [1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt, "The KeY Tool," *Software and System Modeling*, vol. 4, pp. 32-54, 2005.
- [2] J. Barnes, *High Integrity Software: The Spark Approach to Safety and Security*: Addison-Wesley, 2003.
- [3] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview," presented at International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004), Marseille, France, 2004.
- [4] H. Barringer, J. H. Cheng, and C. B. Jones, "A Logic Covering Undefinedness in Program Proofs," *Acta Informatica*, vol. 21, pp. 251-269, 1984.
- [5] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim, "Jass -- Java with Assertions," *Electronic Notes in Theoretical Computer Science*, vol. 55, pp. 103-117, 2001.
- [6] J. Bloch, *Effective Java Programming Language Guide*: Addison-Wesley, 2001.
- [7] D. Bonniot, "The Nice programming language," 2005.
- [8] 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)*, vol. 7, pp. 212-232, 2005.
- [9] L. Burdy, A. Requet, and J.-L. Lanet, "Java Applet Correctness: A Developer-orient Approach," presented at International Symposium of Formal Methods Europe, 2003.
- [10] P. Chalin, "On the Language Design and Semantic Foundation of LCL, a Larch/C Interface Specification Language." Ph.D. Thesis: Concordia University, Department of Computer Science, Montreal, Quebec, 1995.
- [11] P. Chalin, "Are Practitioners Writing Contracts?," in *Workshop on Rigorous Engineering of Fault Tolerant Systems*. Newcastle, UK, 2005.
- [12] P. Chalin, "Ensuring Continued Mainstream Use of Formal Methods: An Assessment, Roadmap and Issues," Dependable Software Research Group, Department of Computer Science and Software Engineering, Concordia University, ENCS-CSE TR 2005-001, 2005.
- [13] P. Chalin, "Reassessing JML's Logical Foundation," in *7th Workshop on Formal Techniques for Java-like Programs (FTJP'05)*. Glasgow, Scotland, 2005.
- [14] P. Chalin, "De-risking the Verifying Compiler Project: Recovering Soundness," Dependable Software Research Group, Department of Computer Science and Software Engineering, Concordia University, ENCS-CSE-TR 2006-001, 2006.
- [15] P. Chalin, P. Grogono, and T. Radhakrishnan, "Identification of and solutions to shortcomings of LCL, a Larch/C interface specification language," in *FME'96: Industrial Benefit and Advances in Formal Methods*, vol. 1051, M.-C. Gaudel and J. Woodcock, Eds., 1996, pp. 385-404.
- [16] J. H. Cheng and C. B. Jones, "On the usability of logics which handle partial functions," in *3rd Refinement Workshop, Springer Workshops in Computing Series*, 1991, pp. 51-69.
- [17] D. R. Cok and J. R. Kiniry, "ESC/Java2: Uniting ESC/Java and JML, Progress and Issues in Building and Using ESC/Java2," in *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004*. Marseille, France: Springer, 2005.
- [18] D. Crocker, "Safe Object-Oriented Software: The Verified Design-By-Contract Paradigm," in *Practical Elements of Safety: Proceedings of the 12th Safety-Critical Systems Symposium*. Birmingham, UK: Springer, 2004.
- [19] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe, "Extended Static Checking," Compaq Systems Research Center, Research Report 159, December 1998.
- [20] K. Devlin, "Why Universities Require Computer Science Students To Take Math," *CACM*, vol. 46, pp. 36-39, 2003.
- [21] C. Engel and A. Roth, "KeY Quicktour for JML," [www.key-project.org](http://www.key-project.org), 2006.
- [22] D. Evans, "Splint User Manual," Secure Programming Group, University of Virginia June 5 2003.
- [23] D. Evans, J. Gutttag, J. Horning, and Y. M. Tan, "LCLint: a tool for using specifications to check code," in *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering (SIGSOFT'94)*: ACM Press, 1994, pp. 87-96.
- [24] M. Fähndrich and K. R. M. Leino, "Declaring and Checking Non-null Types in an Object-Oriented Language," in *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA'03*: ACM Press, 2003, pp. 302-312.

- [25] M. R. Fish and J. A. Turner, "Understanding the Process of Information Technology Implementation," in *Americas Conference on Information Systems*. Baylor University, Dallas, 2002.
- [26] 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, pp. 234-245, 2002.
- [27] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 2nd ed: Addison-Wesley Professional, 2000.
- [28] 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.
- [29] D. Guaspari, C. Marceau, and W. Polak, "Formal verification of Ada programs," *IEEE Transactions on Software Engineering*, vol. 16, pp. 1058-1075, 1990.
- [30] J. V. Guttag and J. J. Horning, *Larch: Languages and Tools for Formal Specification*: Springer-Verlag, 1993.
- [31] R. Hahnle, "Many-Valued Logic, Partiality, and Abstraction in Formal Specification Languages," *Logic Jnl IGPL*, vol. 13, pp. 415-433, 2005.
- [32] HIMSS, "HealthCare CIO Results: Key Trends Index," in *15th Annual Leadership Survey*: Healthcare Information and Management Systems Society, 2004.
- [33] C. A. R. Hoare, "Assertions: Progress and Prospects," 2001.
- [34] C. A. R. Hoare, "Assertions: A Personal Perspective," *IEEE Annals of the History of Computing*, vol. 25, pp. 14-25, 2003.
- [35] C. A. R. Hoare, "The Verifying Compiler: A Grand Challenge for Computing Research," *JACM*, vol. 50, pp. 63-69, 2003.
- [36] C. A. R. Hoare and J. He, *Unifying Theories of Programming*: Prentice Hall, 1998.
- [37] T. Hoare and J. Misra, "Vision of a Grand Challenge project," in *IFIP Working Conference - Verified Software: Theories, Tools, Experiments*. Zurich, Switzerland, 2005.
- [38] B. Jacobs and E. Poll, "Java Program Verification at Nijmegen: Developments and Perspective," presented at International Symposium on Software Security - Theories and Systems (ISSS 2003), 2003.
- [39] C. Jones, *Applied Software Measurement: Assuring Productivity and Quality*, Second ed: McGraw Hill, 1996.
- [40] C. B. Jones, *Systematic Software Development using VDM*, 2nd ed: PHI, 1990.
- [41] C. B. Jones, "The early search for tractable ways of reasoning about programs," *IEEE Annals of the History of Computing*, vol. 25, pp. 26-49, 2003.
- [42] C. B. Jones and C. A. Middelburg, "A Typed Logic of Partial Functions Reconstructed Classically," *Acta Informatica*, vol. 31, pp. 399-430, 1994.
- [43] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, second ed: Prentice Hall, 1988.
- [44] J. R. Kiniry, "ESC/Java2," 2005.
- [45] J. R. Kiniry, P. Chalin, and C. Hurlin, "Integrating Static Checking and Interactive Verification: Supporting Multiple Theories and Provers in Verification," in *International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*. Zürich, Switzerland, 2005.
- [46] 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.
- [47] K. A. Kuhn and D. A. Guise, "From Hospital Information Systems to Health Information Systems. Problems, Challenges, Perspectives," *Methods Inf. Med.*, vol. 40, pp. 275-287, 2001.
- [48] D. Kulak and E. Guiney, *Use Cases: Requirements in Context*, Second ed: Addison-Wesley, 2003.
- [49] G. T. Leavens and Y. Cheon, "Design by Contract with JML," Draft paper 2005.
- [50] C. Marché, C. Paulin-Mohring, and X. Urbain, "The Krakatoa tool for certification of Java/JavaCard programs annotated in JML," *Journal of Logic and Algebraic Programming*, vol. 58, pp. 89-106, 2004.
- [51] B. Meyer, *Object-Oriented Software Construction*, 2nd ed: Prentice-Hall, 1997.
- [52] J. Meyer and A. Poetzsch-Heffter, "An architecture for interactive program provers," presented at Tools and Algorithms for the Construction and Analysis of Systems, 2000.
- [53] Parasoft, "Jcontract product page," 2005.
- [54] B. C. Pierce, *Types and Programming Languages*: MIT Press, 2002.
- [55] F. Rioux, "Effective and Efficient Design by Contract for Java," in *Dependable Software Research Group (DSRG), Faculty of Engineering and Computer Science, Department of Computer Science and Software Engineering*. Montréal, Québec: Concordia University, 2006.
- [56] D. S. Rosenblum, "A Practical Approach to Programming With Assertions," *IEEE Transactions on Software Engineering*, vol. 21, pp. 19-31, 1995.
- [57] C. Sauer, "Deciding the Future for IS Failures: Not the Choice You Might Think," in *Rethinking Management Information Systems*, W. Curie and R. Galliers, Eds.: Oxford University Press, 1999, pp. 279-309.
- [58] SRI International, "The PVS Specification and Verification System."
- [59] Standish Group, "CHAOS: A Recipe for Success," The Standish Group International, Inc 1999.
- [60] Standish Group, "CHAOS Third Quarter Research Report," The Standish Group International, Inc 2004.
- [61] D. C. Stidolph and J. Whitehead, "Managerial Issues for the Consideration and Use of Formal Methods," presented at International Symposium of Formal Methods Europe (FME'03), Pisa, Italy, 2003.
- [62] A. Taylor, "IT Projects: Sink or Swim?," in *Computer Bulletin*: British Computer Society (BCS), 2000.
- [63] UKCRC, "Grand Challenges for Computer Research," UK Computing Research Committee (UKCRC) 2006.
- [64] J. van den Berg and B. Jacobs, "The LOOP compiler for Java and JML," presented at Tools and Algorithms for the Construction and Analysis of Software (TACAS), 2001.
- [65] T. Wilson and S. Maharaj, "Omnibus: A clean language for supporting DBC, ESC and VDBC," in *3rd International Conference on Software Engineering and Formal Methods (SEFM'05)*. Koblenz, Germany: IEEE Computer Society Press, 2005.
- [66] J. M. Wing, "Writing Larch Interface Language Specifications," *ACM Trans. Program. Lang. Syst.*, vol. 9, pp. 1-24, 1987.
- [67] J. C. P. Woodcock, "Dependable Systems Evolution: A Grand Challenge for Computer Science (proposal)," May 26 2003.
- [68] J. C. P. Woodcock, "Grand Challenge 6: Dependable Systems Evolution," 2006.