

A Historical Perspective on Runtime Assertion Checking in Software Development

Lori A. Clarke

Department of Computer Science
University of Massachusetts
Amherst, MA 01003
USA
clarke@cs.umass.edu

David S. Rosenblum

Department of Computer Science
University College London
London WC1E 6BT
United Kingdom
d.rosenblum@cs.ucl.ac.uk

Abstract


This report presents initial results in the area of software testing and analysis produced as part of the Software Engineering Impact Project. The report describes the historical development of runtime assertion checking, including a description of the origins of and significant features associated with assertion checking mechanisms, and initial findings about current industrial use. A future report will provide a more comprehensive assessment of development practice, for which we invite readers of this report to contribute information.

1 Introduction

The Software Engineering Impact Project is documenting the impact that software engineering research has had on computer science research and on software development practice. The authors of this paper are responsible for documenting the impact of research in software testing and analysis for the Impact Project. One aspect of testing and analysis that has clearly had an impact is the widespread use of *assertions*, particularly for use in automated runtime detection of faults. This report documents the results of our initial assessment of assertions and narrates the history of software engineering research as it relates to the evolution and maturation of runtime assertion checking capabilities in programming languages and software development support tools.

Despite decades of research into powerful software engineering technologies, and despite the continual discovery of tenets of good software engineering practice, software development remains an exceedingly complex endeavor. No matter how

thoroughly a software system's requirements are documented, and no matter how carefully and elegantly the system's design has been constructed, inevitably latent *faults*, or incorrect program statements, are introduced in the system's implementation. These faults may be revealed during various levels of testing, or they may make a more inopportune appearance during field use by end-users. In such situations the faults are typically manifested externally as program *failures*, such as unexpected outputs, or other undesirable outcomes such as a program crash. Such failures provide developers with precious little information for initiating the task of correlating the simple external evidence of failure with the complexity of searching numerous possible locations for the faults that caused them.

Assertions are one of the most useful automated techniques available for detecting faults and providing information about their locations, even for  but do not lead to failures. As described in this report, assertions have a long and distinguished history in the annals of software engineering and programming language design. Initially developed as a means of stating expected or desired program properties as a necessary step in constructing formal, deductive proofs of program correctness, assertions have found many other applications in software engineering over the years, albeit primarily in the later stages of development (particularly in the development and execution of source code). They are an important element of *model checking*, an alternative and actively studied approach to program verification, in which the state space resulting from a program's execution is checked against logical assertions expressing temporal safety and liveness properties

(e.g., SPIN [38]). They are embedded in the type systems of many programming languages that support strong typing of data and objects (e.g., Ada [3, 30]). And they are frequently used in an informal fashion by developers to describe module interfaces more precisely in order to assist understanding by other developers. Yet the application of assertions having the greatest impact on development practice, and the one on which we focus in this report, is their use for automated runtime fault detection, in which formal assertion checks are instrumented into a program for execution along with the program's application logic.

Assertions may be applied for automated fault detection during any activity in which a program is executed, including debugging, testing, and production use. Assertions may be used for secondary purposes, such as documentation or to support static analysis of a program. For the purposes of our assessment, an *assertion capability* comprises at a minimum the following features:

- a high-level language for representing logical expressions (typically Boolean-valued expressions) to characterize invalid program execution states;
- a syntax for associating the logical expressions with a program and applying them to well-defined states of the program;
- a means for automatic translation of the logical expressions into executable statements that evaluate the expressions on the appropriate state or states of the associated program; and
- a predefined or user-defined runtime response that is invoked if the logical expression is violated upon evaluation.

This combination of features has been shown to provide a powerful, flexible, high-level facility for automated fault detection during program execution. Note that according to this description, the time-honored tradition of using “print” statements as debugging instrumentation does not qualify as an assertion capability, since it represents a manual, low-level, and typically ad hoc implementation of assertion-like checks. Note also that exceptions are similar to the above description. Exceptions, however, are often intended to describe how the program execution is to behave when an exceptional, but valid, event occurs [29]. It is often a matter of taste as to how exceptional the events that trigger an exception really are. When execution leads to signaling an exception, corrective actions are defined in an exception handler. The semantics of exception

handling, including the continuation semantics, may alter the flow of control in the program considerably. Assertions, on the other hand, describe program invariants that are by definition expected *always* to hold. When the logical expression in an assertion is found to be false, a fault has occurred. This fault is reported and execution continues or terminates, depending on the severity of the fault. As discussed below, assertion capabilities often have expressive notations for representing an assertion and for indicating the scope or states in which it applies. Exceptions have often been used as a mechanism to implement assertions, as noted below.

Although evidence of institutionalized use of assertions in software development projects is hard to come by, it is well-known that assertions have long been used by seasoned software developers, who eventually come to learn the value of seeding their code with automated defensive checks at the outset of development, thereby avoiding the pain of belated, trial-and-error insertion of print statements during debugging [36]. Indirect evidence for more recent growth in the popularity of assertions among developers is to be found in the proliferation of assertion capabilities for widely-used programming languages, especially C++, Java, and C#. Thus, assertions have had a significant impact on software development for the past two to three decades, and one can quite easily trace the research forbears of the various assertion capabilities developers have used over the years.

The next section of this report presents a brief history of the research ideas that have contributed to the assertion capabilities that are available for use in current development practice. Section 3 summarizes the characteristics of the most widely-used assertion capabilities, and Section 4 describes experimental evaluations about how the use of assertions impacts the software development process. Section 5 concludes with a discussion of future plans for further assessments of assertion capabilities, which will be undertaken after receiving feedback from readers of this report and from the community of researchers and practitioners.

2 A History of the Technology

This section presents a concise history of assertions. The history is organized around the key ideas that arose in the development of assertions.

2.1 Logical Assertions as Characterizations of Behavior

While the use of program instrumentation mechanisms for dumping low-level execution and memory traces is probably as old as computing itself (e.g., Evans and Darley [22]), the origins of formal, logical assertions about program behavior predates computers [27, 90]. Goldstine's and von Neumann's work on reasoning about programs used the term "assertion" for documenting invariants in algorithms. Turing also advocated that assertions be used to document the states that can be associated with various points in a routine and that this information subsequently be useful in determining the correctness of the whole program:

"In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows." [90]

It is interesting that Goldstine and von Neumann as well as Turing used the word "assertion," although there is some speculation that this was not merely happenstance, but that Turing was aware of this earlier work [42]. In 1963, McCarthy's notion of a "mathematical science of computation" [58] also advocated the use of assertions, and in 1966, Naur introduced a similar, but less formal concept, of a "general snapshot" [65]. While McCarthy's ideas enjoyed some early linguistic expression in Algol W [92], it was really the pioneering work of Floyd in 1967 [25] and Hoare in 1969 [35] that served to widely indoctrinate computer scientists in the use of assertions expressed in first-order logic as a means of stating formal constraints on program states in order to support formal reasoning about the correctness of programs.

Floyd used *loop invariants* in flowchart representations of programs to represent invariant conditions over the iterations of loops [25]. Hoare generalized the use of assertions to make formal statements about the behavior of individual program statements and compositions of statements, including whole programs [35]. In particular, Hoare defined the famous proof schema $\{P\} S \{Q\}$ (originally written by Hoare as $P \{S\} Q$), where S is a (composition of) statements, P is the *precondition* of S , and Q is the *postcondition* of S . The meaning of this proof schema is that if the environment of S establishes the truth of P in the state immediately preceding the execution of S , and if S executes and terminates, then the execution of S is required to

establish the truth of Q in the state immediately following its execution. Hoare defined axiom schemas for primitive program statements (such as assignment statements) and inference rules for compound statements (such as conditionals and loops) and compositions of statements. Hoare then sketched a proof method whereby a programmer would supply a precondition P and postcondition Q for a program S and then apply the system of proof rules to establish the validity of the statement $\{P\} S \{Q\}$.

Several mechanical program verification efforts arose based on the proof of simple assertions, called *verification conditions*, which are generated as a by-product of applying Hoare-style proof rules [16, 28, 40, 43, 50, 91].

2.2 Specification and Documentation with Assertions

While Floyd and Hoare's primary goal in using assertions was to support formal reasoning about the correctness of programs, programmers soon realized the utility of assertions as a way of documenting programmer intent. This idea, proposed before computers as mentioned above, was articulated nicely by Parnas in a paper on principles for succinctly specifying program elements, where the main goal of specification is aiding the work of human designers rather than supporting mechanical proofs of correctness [70]. Thus began the tradition of using the first-order predicate calculus as a language for stating declarative preconditions, postconditions, loop conditions and other constraints on program states during formal specification and design.

Formal specification languages that were designed to support specification, verification, refinement and analysis of programs at early stages of development and at a high level of abstraction often introduced assertion capabilities. For example, AFFIRM [64] and Euclid [72] included a notation for representing non-executable assertions. The SPARK language, which is based on Ada, includes assertions but expects these to be discharged either through the use of static analysis techniques or, when such analysis is inadequate, through the use of theorem provers [5]. Gypsy [4] included support for assertions (e.g., `assert`, `entry`, and `exit`) that were intended to be verified but, if not verified, could be checked during execution. Alphard [93] described support for specifications that indicated the invariants and preconditions and postconditions of each class (called a "form") that were intended to support verification. Even early versions of the currently

popular specification languages Z [1, 83] and VDM [10, 41] included support for preconditions and postconditions and continue to do so.

In addition to defining and experimenting with formal specification languages, a number of authors articulated comprehensive philosophies of formalized software development. Two of the most influential of these were *A Discipline of Programming*, by [17], and *The Science of Programming*, by Gries [31]. Many developers who have been exposed to these ideas opt for a purely informal use of assertions, in which module pre- and post-conditions are documented strictly for human consumption inside stylized comment regions with varying degrees of formality in the assertion expressions. For instance, the Unified Modeling Language (UML) [67] supports specification of pre- and post-conditions on classes and interfaces through its Object Constraint Language (OCL) [66]. While such constraints conceivably can be exploited at various stages of development for verification or runtime checking, tool support for OCL pre- and post-conditions is still rather limited, and thus it remains primarily a language of informal documentation with assertions.

2.3 Adding Assertions to Programming Languages via Preprocessors

By the mid 70's researchers realized that monitoring assertions during execution offered a simpler and more practical alternative to formal proofs of correctness. Such monitoring was typically achieved by deriving runtime checks from assertions expressed as comment annotations or macros in the program text. In general, assertions have been supported in programming languages in one of two ways—either by using a special preprocessor or by incorporating assertion constructs into the design of a language.

Some of the first preprocessor systems were reported in papers at early conferences on reliable software [85, 86, 94]. These papers described systems for deriving runtime consistency checks from simple assertions. The main goal of these approaches was to evaluate an assertion at runtime whenever the state associated with the assertion was reached during execution. Violation of the assertion triggered some appropriate diagnostic response, either predefined (as with the **assert** macro described below) or programmer-defined (as proposed by Parnas [70]). Hence, this use of assertions involved checking *individual* program executions rather than proving (or disproving) the correctness of *all* program executions.

In Stucki's and Foshee's approach [85, 86], assertions were written as annotations of FORTRAN source code, and an extended version of the FORTRAN validation tool PET (Program Evaluator and Tester) was then used to convert the annotations to embedded runtime checks that were invoked at appropriate times during the execution of the program [86]. Yau and Cheung used the term *self-checking programs* to characterize programs having embedded runtime assertion checks, and they attempted to define a systematic framework for identifying, incorporating and using self-checks on all elements of a software design. They recommended specifying self-checks to check the function, control sequence, and data of a program, and they recommended using self-checks for detecting faults, locating and stopping the propagation of errors, and assisting in the recovery from errors [94]. Yau and Cheung also noted the widespread, albeit ad hoc, use of defensive checking mechanisms in systems of that period (particularly operating systems), as well as their more extensive and systematic use in high-reliability systems such as the AT&T Electronic Switching Systems (ESS). While the notion of self-checking programs remains popular, it has also been used in another context by Blum and others who have studied some of the theoretical aspects of programs that probabilistically check the correctness of their own execution behavior [11].

The Anna annotation system, developed to augment the Ada programming language, is interesting because of the extensive kinds of assertions that it supported [49, 53]. These included:

- *subtype annotations*, for specifying a logical constraint on the set of values belonging to an Ada subtype;
- *object annotations*, for specifying a logical constraint on the values a variable may hold;
- *statement annotations*, for specifying point assertions on the states following statement executions;
- *subprogram annotations*, for specifying pre- and postconditions on subprograms;
- *axiomatic annotations*, for axiomatic or algebraic specification of package behaviors [78];
- *context annotations*, for specifying constraints on the items a compilation unit uses from the other compilation units it imports; and

- *propagation annotations*, for specifying constraints on the way exceptions are propagated within a program.

In addition to these seven kinds of assertions, Anna augmented the Ada expression language with quantifiers, state references, expressions over the heap collections associated with access types (Ada pointer types), and other such features. Nearly all the features of Anna were supported by tools that transformed Anna annotations into corresponding Ada checking functions that would be invoked at appropriate states in the program execution [75, 79, 80]. In addition, Sankar developed a method for runtime checking of algebraic specifications in axiomatic annotations that was supported by incremental theorem proving [78].

The preprocessor category also includes a substantial variety of enhancements to the popular programming languages C and C++. The earliest and still most widely-used of these enhancements is the **assert** macro, which made its first appearance in the C programming language [89].¹ The usual definition of the **assert** macro expands to an if-statement on the negation of the asserted condition, with an abort and core dump of the program resulting if the condition evaluates to false at runtime. Despite the limited expressive power and flexibility of the **assert** macro, it remains the assertion construct of choice for many C and C++ developers, and constructs of similar capability have been added recently to the languages Java [87] and C# [6].

More powerful assertion capabilities for C were provided by APP [74], C-Patrol [95] and Robust C [24]. Cline and Lea proposed an Anna-like assertion language for C++ called A++ [13], and Maker developed a macro-based assertion capability for C++ called Gnu Nana, as part of the GNU free software project [55]. Assertions via preprocessors have even been added to the popular scripting language Tcl [14].

2.4 Assertions in Programming Languages

The second way of supporting assertions in programming languages is to incorporate assertions directly in the language definition itself, with the

concomitant expectation that these constructs will be recognized and supported by compilers and other development tools for the language. As noted above, experimental programming languages that emphasized verification adopted this approach in the 1970s, including Gypsy [4], Euclid [72], and Alphard [93].² The Turing Language [37], developed in the 80's, combined the interactive aspects of Basic with the simplicity of Pascal and the flexibility of C, but, consistent with its goal of supporting beginning programmers, also included assertion constructs.

The most influential commercial language to support and strongly advocate the use of assertions, and the first object-oriented programming language to do so, is Eiffel, developed by Bertrand Meyer [60, 61]. Meyer ultimately gave the popular characterization *design-by-contract* to the use of assertions in the context of object-oriented programs and argued for the use of assertions as permanent defensive mechanisms for fault detection in programs (including production versions) [59]. The design of Eiffel was influenced strongly by much of the early work described above on formalized program development, including the early work of Floyd and Hoare, as well as Dijkstra's paper on guarded commands [18] and the book *A Discipline of Programming* [17]. Meyer was also aware of many of the languages with assertion constructs (e.g., Alphard [93], Euclid [72], Anna [53]) and the specification approach advocated by Liskov and Guttag [47], and was himself one of the originators of the Z notation [82-84]. Eiffel incorporated assertion constructs into the programming language, including support for preconditions and postconditions, initial values (in postconditions only), loop invariants, class invariants, and a general assert statement. In addition, it defined subtyping rules for assertions within a class hierarchy (including rules for precondition weakening and postcondition strengthening for the methods of a subtype) and provided compiler support for three different levels of assertions [60]. As discussed below, Eiffel was one of the first programming languages to support the use of assertions in object-oriented languages.

¹ Some sources state that the **assert** macro first appeared in the UNIX Sixth Edition, May 1975. However, a search of the UNIX Sixth Edition source tree (downloaded July 26, 2003, from <http://minnie.tuhs.org/UnixTree/V6/>) revealed no **assert** macro definition.

² Also notable in this category is Zuse's Plankalkül, a visionary proposal for a programming notation developed in the mid-1940s that included support for assertions among its many novel ideas 96. Zuse, K. Der Plankalkül, Gesellschaft für Mathematik und Datenverarbeitung, 1972..

2.5 Incorporating Implicit Assertions into Strong Typing

While the languages mentioned above support explicit specification of logical assertions in programs, it is worth noting that the use of strong typing in modern programming languages can be viewed as an attempt to support limited forms of assertion specification via type declarations. For instance, a simple assertion in a FORTRAN program that requires the range of indices for an array to be between 1 and 100 can be easily specified and enforced using an appropriate array type declaration in a modern language such as Ada or Java.

Early tools that automatically checked such “type” information were employed via a separate phase in the software development process, via a preprocessor or a special compiler [76]. The idea of trying to first verify or evaluate assertions statically and then only leaving in runtime checks when such static analysis failed was evident in the programming language Gypsy [4] and in flow analysis tools [68]. This notion was supported in the Ada programming language for such features as array bound violations, which are sometimes but not always statically checkable [3].

2.6 Assertions in Hardware Design Languages

Inspired by the use of assertions in software, there also have been attempts to define assertion capabilities for hardware designs. Mahmood and McCluskey explored the notion of a *watchdog processor*, a coprocessor that monitors hardware instruction execution in order to detect control-flow and memory access errors [54]. And the language VAL (VHDL Annotation Language) used assertion constructs to support the formal specification of VHDL hardware designs [7].

3 Assertion Language Features

Assertion capabilities are available for most common programming languages, including Java, C, C++, C#, Ada, Eiffel, Fortran, Cobol, Basic, and even scripting languages such as TCL. This section describes the most common features associated with these capabilities. **Error! Reference source not found.** presents a general summary of the features of a number of available assertion capabilities, focusing on support for Java and C++. This discussion illustrates how the ideas that originated in software engineering research have become incorporated into programming practice.

3.1 The Scope of Assertions

An assertion includes a Boolean expression, or constraint, that is to be evaluated at an individual program state (i.e., a steady state between the execution of two consecutive program statements or the state immediately preceding or following a program state). It is often useful, however, to associate assertions with higher-level program constructs, which may implicitly require the assertion to be associated with or applied to multiple program states. Usually an assertion’s location in the program or a keyword indicates where the constraint will be evaluated. Commonly used keywords are **precondition**, **pre**, or **require** for preconditions; **postcondition**, **post**, or **ensure** for postconditions; and **assert** or **invariant** for intermediate assertions. A precondition assertion is associated with a method, or procedure, but is meant to be checked before the method is called. Thus, instead of having to state the assertion explicitly at all the locations where a method is called, the assertion capability will make this determination and assure that the constraint is checked whenever one of these locations is encountered during execution. Similarly, a postcondition is also associated with a method and checked at each place the method returns control to the caller.

Another common feature is to allow an assertion for a method to reference a variable’s value immediately before execution of that method. This is frequently used in postconditions, which are often conveniently expressed in terms of values that existed in the state associated with a corresponding (perhaps implicit) precondition. For instance, the postcondition of a *swap* routine that takes two parameters *x* and *y* by reference requires that the post value of *x* equal the pre value of *y* and vice versa. Turing’s early paper about reasoning about programs differentiated between the initial and final value of a program’s variable [90]. Early papers on verification introduced various notations for distinguishing between the value of a variable at the current state and the value of that variable at the precondition state. For example, Manna’s early work [56], based on his thesis, introduced an indexed superscript for each different value, Hantler and King [32] marked the initial value with a superscript prime, and Linger, Mills, and Witt [46] used the subscript naught. These notational conventions found their way into specification languages, where for example VDM [41] incorporated the use of a “hat” to indicate the precondition state and the Z Calculus [84] used primed variable names. Assertion capabilities incorporate similar conventions. Usually the initial values of a variable are denoted by a keyword, such

as **pre**, **in**, or **old**. Some assertion capabilities allow initial values to be referenced only in postconditions. By default, precondition assertions can only reference initial values, so usually no keyword is required.

Several assertion capabilities provide support for global invariants, which must be valid throughout execution. Programming languages that support data abstraction or classes provide an opportunity to describe assertions that are intended to hold at multiple states throughout an execution of a data type or class. For example, Anna supported the specification of subtype annotations, each of which applied to all states in which a value is assigned to a variable of the constrained subtype [49]. For object-oriented languages, invariants are usually associated with a class and checked after execution of any method in the class that could change the value of a variable referenced in the assertion's constraint. For such languages, no assertion violation is reported if a class method invalidates a class invariant's constraint temporarily, as long as the constraint's validity is reestablished before the method returns. Eiffel, for example, supports class invariants that should hold after any invoked method in a class returns.

3.2 Boolean Expressions

The Boolean expression that represents the constraint of an assertion is usually written in a notation that is consistent with the programming language where the assertion capability is being employed. Thus, an assertion capability for C will use C syntax to express the constraint. There are two common exceptions to this restriction, quantification and hidden functions.

Quantification is syntactic sugar that makes it easy to indicate that an assertion is intended to hold for all (i.e., universal quantification) or at least one (i.e., existential quantification) of the elements in some collection. Some languages such as Eiffel were deliberately designed not to include such language support in the interest of keeping the language simple. However, Meyer feels that the lack of quantification has meant that Eiffel programmers rarely use Eiffel's loop invariants [62]. In Eiffel, quantification can be explicitly encoded in methods that could be invoked in the Boolean expression. Most other assertion capabilities provide direct support for quantification using keywords such as **forall**, **all**, **exists**, and **some**.

Usually the Boolean expression can reference any variable or method that is visible in the scope where the assertion appears. Thus, a precondition can usually reference any of the variables or methods that can be referenced at the start of a method. Similarly,

a postcondition can reference any variable or method that is visible right before the method returns, although as noted above, postcondition assertions can typically reference the values of variables at the onset of the call.

To express an assertion, sometimes the programmer needs to "break" the abstraction associated with an object and reference information about the state that is not readily available via the object's access methods. To support this, some assertion capabilities allow "hidden" functions to be defined for a class and then used in assertions about that class. With such assertion capabilities, usually the hidden functions can only be referenced in assertions within the scope of the class where the hidden function is defined. Support for hidden functions is reminiscent of similar capabilities provided in specification languages such as OBJ [88].

3.3 Inheritance

Most of the assertion capabilities for C++ and Java support the inheritance of assertions associated with a class. Assertion checking, however, becomes quite complex in the presence of inheritance [48]. The most common approach is for each method to form the disjunction of the preconditions of each of the parent classes and the conjunction of the postconditions of the parent classes for that method. Eiffel provides more options but requires that preconditions remain the same or be weaker than subclass preconditions and that postconditions remain the same or be stronger than the subclass postconditions. Findler and Felleisen document many of the problems underlying support for inheritance in several current assertion capabilities [23].

3.4 Automatic Suppression of Assertions

A key concern with the use of runtime checking of assertions is the extent to which they interfere with the performance, and even semantics, of the programs they check. In particular, assertion checks consume object code space and execution time, both of which could be significant for large numbers of assertions or highly complex assertion checks. In industrial practice, assertion checking is frequently suppressed in production versions of software. For instance, in C programming environments the availability of checks for the **assert** macro are often made conditional on the absence of a non-debug indicator (such as the macro **NDEBUG**). Some industrial development organizations, however, retain assertion checking in their production code and

request users to forward assertion violations back to the organization.

Assertion capabilities tend to provide support for enabling or disabling assertions, although often this support is rather limited. All of the assertion preprocessors allow either *all* the assertions in a source file to be enabled or *all* to be disabled. Typically, the user either compiles the unprocessed source file, which treats all the assertions as comments and thus as disabled, or compiles the output file produced by the assertion preprocessor, which has enabled all the assertions by translating them into executable source statements.

Some assertion capabilities provide static mechanisms for selecting the classes or packages that should have their assertions enabled. For preprocessor systems, these directives indicate which classes or packages are (or are not) to be preprocessed. Similarly, many of the language-based assertion capabilities for C, C++, and Java allow the user to provide command line directives to indicate the classes, files, or packages that should have their assertions enabled or disabled.

Instead of relying on an indication of which components to enable for assertion checking, an alternative approach is to enable or disable assertions based on a specified *severity level*. For instance, the tool APP allows the association of programmer-specified severity levels with individual assertions [74]. For a particular program execution, the maximum severity level to be checked can be set via a runtime parameter, so that assertion checks can be completely included, completely suppressed, or selectively included up to a certain level of severity. Eiffel offers an alternative approach where different types of assertions can be selected, including preconditions only (the default), no assertions, or all assertions [59].

3.5 Assertion Violation Processing

During execution, if the Boolean expression that forms the constraint is found to be false, then the assertion violation must be signaled. When this occurs, some assertion capabilities abort immediately, some report the violation and then continue execution, and some either abort or continue based on the type of the assertion. In programming languages that support both an exception mechanism and assertions, exceptions are the favored mechanism for signaling a runtime violation of an assertion (such as the use of the exception `ANNA_ERROR` in Anna, the exception `AssertionError` in Java, and Gautron's assertion capability for C++ [26]). Many of the

assertion capabilities rely on the programming environment's runtime debugging capabilities for displaying the call stack at the time of an assertion violation.

It is worth noting that the original designers of Ada chose not to incorporate assertion facilities into the language because it was felt that Ada's exception constructs would provide sufficient support for constraint checking and handling. In contrast, Meyer saw the need for both explicit contracts and exceptions in the design of Eiffel, feeling that the latter strongly impacts programming style and thus should be viewed as a tool of last resort for constraint checking [60, 61].

3.6 Assertions Based on Formalisms Other Than First-Order Logic

Given the power and convenience of first-order logic assertions as a tool for runtime checking of programs, people were quick to try to adapt forms of specifications developed originally for program verification. Notable examples of this are the ways in which *temporal logic* specifications have been adapted for runtime checking.

Temporal logics were introduced to provide a means for specifying and verifying concurrent programs and programs that exhibit a high degree of non-determinism [57, 71]. A temporal logic formula typically constrains multiple program states at different points in time, requiring or disallowing the existence of one state before the occurrence of a later state.

While temporal logic formulas typically express constraints over all infinite futures of a program, a refinement of temporal logic called *interval logic* was introduced to allow for the specification of temporal constraints over bounded intervals of time [19, 81]. The finite bounding produced by interval logic made them ideal for *a posteriori* checking of runtime behaviors, and hence people began exploiting interval logic as an alternative or complementary form of assertion to be checked in concurrent programs. When used for runtime checking, the interval logic expressions are typically formulated in terms of *program events*, rather than the *state predicates* that are the basis of temporal logic. Bates and Wileden carried out some of the earliest work along these lines, with their use of *event-based behavioral abstraction* (EBBA) for debugging concurrent Ada programs [8]. Luckham and others built on the approach of Bates and Wileden with the definition of TSL (Task Sequencing Language) for explicit specification of event-based behavioral constraints on

concurrent Ada programs [34, 51]. This work formed the basis for their software architectural description language and simulation system Rapide [52].

These alternative forms of assertion checking provide a great deal of expressive power and fault-detection power. To date their primary application has been for stating properties to be checked in model checking approaches to verification (e.g., [9, 38, 39]).

3.7 Comparison Table

As summarized in Figure 1, many language-based assertion capabilities, for example Gnu Nana for C and C++ and the assertion capability found in Java 1.4, seem to rely on the language's native capabilities, and thus the programmer's ability to use those capabilities, to provide many of the features provided in preprocessor-based assertion capabilities. In particular, language-based assertion systems tend not to provide support for quantification, initial values, or class invariants. For instance, the assertion systems for C and C++ often rely on library macro capabilities for saving the values of variables and then referencing those values.

In contrast, the commercial system Jcontract [69], provided by Parasoft, augments Java with assertion capabilities and seems to be one of the more sophisticated systems available. It provides support for preconditions, postconditions, and class invariant assertions as well as quantification. In Jcontract, the assertion statement is nicely integrated with the language's inheritance, exception, and debugging capabilities. When integrated with the Parasoft test management system, Jtest, it provides a supportive environment for selecting and deselecting assertions, for reviewing the results from single and multiple executions, for reporting assertion execution coverage, and even for test data generation to exercise the assertions.

4 Empirical Evaluation of Assertion Capabilities

On the whole, practitioners in industry seem to regard assertions as a useful component in their arsenal of debugging tools. However, there have been only a few empirical studies of the effectiveness of assertions at detecting or preventing program faults.

Leveson and others performed an empirical comparison between assertion checks and voting mechanisms in programs [44]. While the results of the study demonstrated a high degree of effectiveness

of assertions, the study has been criticized for its exclusive use of student programmers and small program subjects.

Rosenblum carried out a case study on a more significant C program subject that had been written by him and another researcher [74]. The assertions that were written for the program detected a high percentage of the discovered faults in the program, and additionally Rosenblum categorized the assertions into a number of general categories that can be used by future developers who want to reap the benefits of this experience in their own development efforts.

Typke and colleagues performed an experimental comparison of two assertion preprocessor, APP and Jcontract, in terms of their ability to aid software maintenance and extension tasks [63]. They found that the use of assertions both reduced the effort needed for the tasks and made the effort more predictable.

5 Runtime Assertions in Current Practice

Assertions seem to have widely infiltrated common programming practice. Although not universally used, there are assertion capabilities for most current programming languages, and there exists evidence that the use of assertions is a supported practice for many companies and projects. Furthermore, as discussed in Section 2.2, there have been a number of influential works advocating formalized program development, a by-product of which is the use of assertions in an *informal* fashion in module interface documentation developed for human consumption.

Although the extent of use is hard to quantify, assertions are indeed widely used in practice. For instance, Cusumano and Selby report on the widespread use of assertions in debugging and testing at Microsoft, where assertions are used primarily to check developers' assumptions about global program state ([15] pp. 300–301, 334). Hoare reports further anecdotal data that about 250,000 lines of the source code of Microsoft Office is assertions, representing roughly 1% of the source code [36]. Chalin surveyed a number of software projects to determine the density of assertion statements in source lines and reported an average assertion density of 3.27% in the surveyed proprietary projects, 5.10% in the surveyed open source projects, and 6.42% in the surveyed Eiffel projects [12]. Papers and presentations at practitioner-oriented venues, such as Quality Week[73], provide additional anecdotal testimonials on the benefits of using assertions in practice.

	Pre and Post Conditions	Quantification	Pre-State Values	Global Assertions	Language Integration	Severity Levels	Enabling/Disabling Assertions via severity levels	Debugging Support	Hidden Functions	Inheritance	Notes	Web Site (as of July 28, 2003)
App	yes (assume, promise, return)	yes	yes	no	enhanced C preprocessor	yes	no	C debugger	no	no		http://www.research.att.com/s/w/tools/reuse/ widely available
C/C++	no	no	no	no	assert macro in C library, third-party preprocessor tools	no	no	C/C++ debugger	no	no		
C#	no	no	no	no	assert method in class Debug	no	conditional compilation	C# debugger	no	subclass inherits any non-overridden methods in superclass that contain asserts	.NET's Base Class Library (BCL) supports similar facility for C#, C++ and VBA	http://msdn.microsoft.com/vcs/harp/
Eiffel	yes (require and ensure)	no, except through explicit coding in Eiffel	yes, old prefix refers to initial value	Class invariant	integrated with Eiffel					define rules for subcontracting. Can keep or weaken the precondition and keep or strengthen the post condition		http://www.eiffel.com/
GNU Nana	really just an assertion, but can use the keywords require or ensure (from Eiffel) or I or N (not I)	yes, A(Any), E(Exists), C(Count of the number that satisfies the expression), E1(Exist only 1), S(Sum of all the expression values), P(Product of all the expression values).	yes, using ID macro	no (unless user does the programming)	somewhat integrated with C in terms of includes and macros; goal is to provide library support for Eiffel like assertions	yes, on, off, and if guards are true	using L.h for logging can enable and disable messages	C debugger	no	no	can be run during execution or only with the debugger; The 'eiffel.h' library is intended to provide a similar setup to Eiffel in the C++ language.	http://www.gnu.org/directory/lis/c/nana.html
lContract	yes (@pre, @post)	yes	@pre used in a post condition	yes, @invariant (or @inv) over a class; checked before and after execution of each method in that class	preprocessor for Java	no, but can turn off preprocessor at the command line	a separate tool for this has been developed elsewhere	Java call stack	uses Java's hidden functions	can inherit assertions from interfaces and superclasses	not easy to install or use	http://www.reliable-systems.com/tools/Contract/Contract.htm
Jass	yes (require and ensure); also check for anytime, loop invariant, and invariant	yes	yes, old prefix refers to initial values of attributes (but not parameters) Somewhat awkward to use	Yes, invariants are associated with a class and are checked before and after each method call in a class	C/C++, IDL and Java. For Java, preprocessor takes a Jass file and creates a Java file (can run .jass or .java)	2 levels: contract and warning; set at preprocessor time by class. Jass also has nothing and interface(status) interface check		Uses Java IDE to display the call stack	yes	subclass inherits from the parent class (Assertions are inherited when the assertion (in this case, an invariant) is put globally on a class, or when inherited methods have assertions in them.)	traces assertions supports sequences of method calls; does not integrate with the IDE, supports rescue blocks, has a variant to indicate that the loop counter is always decremented. The Post-condition can contain three special constructs: Old, Result, changeonly. Site last updated 23 November 2001	http://csd.informatik.uni-odensburg.de/~jass/
Java 1.4	no	only through explicit programming	no	no	assert keyword plus AssertionError exception class	no	set by runtime command at package and class level	call stack w/ line numbers	uses Java's hidden functions	subclass inherits from the parent class		http://java.sun.com/2se/1.4.2/
Jcontract	yes (@pre, @post), also exceptional returns (@throws exception-name)	yes	yes (@pre in post-condition)	class invariants, which must be true after executing any method of the class	extended Java compiler, with several runtime handlers provided; default handler logs assertion violation and continues execution	no	some control by type of assertion, plus enabling and disabling via choice of compiler (i.e., instrumentation or without)	integrated with Jtest	uses Java's hidden functions	yes, with behavioral subtyping rules	nice GUI support	http://www2.parasoft.com/jsp/products/home.jsp?product=Jcontract&itemid=30
JML	yes (requires, ensures), also exception returns (signals exception-type)	yes (forall, lexists) plus others (vmin, vmax, \sum, \product, \num of)	yes (old)	Class invariants, which must be true after executing any method of the class	extended Java compiler, creates Java byte code	no	via choice of compiler	can run with the Java debugger	yes (variables, methods, classes and interfaces that are visible only to the specification)	yes, including behavioral interfaces (with behavioral subtyping rules enforced as in Eiffel)		http://www.cs.iastate.edu/~lba/vers/JML/
JMSAssert	yes (@pre, @post \$ret == a, and @inv(ariant))	no, but supported through JMScript, @macro foreach(elm in array) assertPost(elm != null);	\$prev	no	preprocessor for Java uses exceptions to report assertion violations	no, unless scripts are written using JMScript	no	Java call stack	Java hidden functions		provides a GUI, limited support. No scoping, no assertions within functions or on local variables.	http://www.mmsindia.com/JMSAssert.html

Figure 1. Comparison of Current Assertion Capabilities.

One of the most notable systematic uses of assertions in a large software development project is the use of *craft asserts* as defensive checks for invalid data values in AT&T's 5ESS Switching Systems software [2]. These asserts complement the extensive use of *audits* in 5ESS, which periodically check the system state associated with call processing and invoke a variety of responses upon detecting an invalid state, such as generating a fault message on an administrator console or terminating a phone call in progress.

Meyer reports a number of interesting insights gained through well over a decade of successful commercial promulgation of Eiffel [62]. Eiffel has on the order 10,000 users, with a few hundred companies using the language on a large scale. The language is used primarily for large-scale mission-critical financial applications, with defense, aerospace and health-care being important additional sectors for mission-critical use. Many of these systems use Eiffel in conjunction with other programming languages, with Eiffel used for programming the application core and to provide an architectural framework for system development. Eiffel was initially conceived as a component library project rather than a programming language project. Thus, Eiffel programmers who write no assertions of their own are still able to benefit greatly from Eiffel's contract features when they use Eiffel's extensively contracted component libraries.

The success of Eiffel's contract features in commercial development practice has led Meyer to feel that, contrary to accepted wisdom, programmers do not shy away from formalism in software development. Instead, he feels that the main barriers to the use of assertions in development practice are the lack of assertion features in the definition of many commonly used programming languages (with third-party language add-ons such as Jcontract not able to ensure the same level of semantic consistency and continuity), and to excessive schedule pressures from managers who are unwilling to let their engineers develop software with the care that effective use of assertions requires. Companies that do use contracts heavily quickly learn to appreciate their value and view the contracts as corporate assets.

Roman Salvador, vice president of research and development at Parasoft, reported that it was difficult to sell Jcontract, the assertion support system, as a stand-alone tool and that in the future Jcontract capabilities would be included with the popular Jtest tool, which has thousands of users. Jtest provides test coverage information as well as test generation capabilities. When Jcontract assertion capabilities are

combined with the Jtest test generation capabilities, the system tries to find test cases to violate postconditions and invariants. He reported that another significant benefit of this combination of capabilities is that the preconditions are useful in restricting the generated test cases to values in the developer's domain of interest [77].

6 Future Work

The study and promulgation of assertions remains an active endeavor for researchers and practitioners alike. One of the most promising recent developments in research with assertions is the automatic *discovery* of likely program invariants [21], and the related technique of correlating failure data with execution history data from field installations of software systems to help isolate program faults [20, 33, 45]. While the automated reporting of failure data from the field is already a staple of many commercial software systems, albeit in rudimentary form (as evidenced by periodic requests to the user for consent to report data from Microsoft Windows XP, Apple Mac OS X, and other systems), it remains to be seen how well some of the more sophisticated approaches will be able to scale for sufficient impact on development practice.

While the focus of this report has been on the origins of assertion capabilities found in modern programming languages and development tools, we are also working on a comprehensive assessment of the use of assertions in development practice. We invite and strongly encourage readers of this report and others in the software engineering research and software development communities to contact the authors and to contribute information for this assessment. Such information might include anecdotes about ad hoc use of assertions by individual developers, systematic use of assertions in large-scale development projects, documentation of assertion use as organizational best practices or within organizational development process definitions, and historical events in which assertion violations played an important role in revealing faults in critical software systems. Much of this information is difficult to come by since it is to be found primarily in the undocumented lore of software development practice rather than well-documented in the research literature.

Acknowledgments

This article has been developed under the auspices of the Impact Project. The aim of the project is to

provide a scholarly study of the impact that software engineering research—both academic and industrial—has had upon practice. The principal output of the project is a series of individual papers covering the impact upon practice of research in several selected major areas of software engineering. Each of these papers is being published in ACM TOSEM. Additional information about the project can be found at <http://www.acm.org/sigsoft/impact/>.

This article is based upon work supported by the U.S. National Science Foundation (NSF) under award number CCF-0137766, the Association of Computing Machinery Special Interest Group on Software Engineering (ACM SIGSOFT), the Institution of Electrical Engineers (IEE), and the Japan Electronics and Information Technology Association (JEITA). Any opinions, findings and conclusions or recommendations, expressed in this publication are those of the authors and do not necessarily reflect, of the NSF, ACM SIGSOFT, the IEE, or JEITA.

We are grateful to Bertrand Meyer for taking the time to discuss the influences on his notion of design-by-contract, the early development of Eiffel, and the impact of Eiffel on development practice. We thank Patrice Chalin for pointing out several useful references and providing us with data on industrial use of assertions. We thank Roman Salvador from Parasoft for sharing information about the industrial uses of Jcontract and Jtest. We also thank the members of the Software Engineering Impact Project, and particularly the project historian, Prof. Michael S. Mahoney of Princeton University, for the many insights they have provided us in the writing of this report.

References

- Abrial, J.-R., Schuman, S.A. and Meyer, B. Specification Language. in McKeag, R.M. and MacNaghtam, A.M. eds. *On the Construction of Programs*, Cambridge University Press, New York, 1980, 343-410.
- Allers, J.A., Huizinga, A.H., Kukla, J.A., Sipes, J.D. and Yeh, R.T., No. 5 ESS-strategies for Reliability in a Distributed Processing Environment. in *13th International Symposium on Fault-Tolerant Computing*, (Milan, Italy, 1983), 388-391.
- ALRM83. Reference Manual for the Ada Programming Language, United States Department of Defense, Washington DC, 1983.
- Ambler, A.L., Good, D.I., Browne, J.C., Burger, W.F., Cohen, R.M., Hoch, C.G. and Wells, R.E., Gypsy: A Language for Specification and Implementation of Verifiable Programs. in *ACM Conference on Language Design for Reliable Software*, (Raleigh, North Carolina, 1977), 1-10.
- Amey, P. and Chapman, R., Industrial Strength Exception Freedom. in *Proceedings of the ACM SIGAda International Conference on Ada*, (Houston, Texas, 2002), ACM, 1-9.
- Archer, T. and Whitechapel, A. *Inside C#*. Microsoft Press, Redmond, WA, 2002.
- Augustin, L.M., Gennart, B.A., Huh, Y., Luckham, D.C. and Stanculescu, A., VAL: An Annotation Language for VHDL. in *International Conference on Computer-Aided Design*, (1987), IEEE Computer Society, 418-421.
- Bates, P.C. and Wileden, J.C. High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach. *Journal of Systems and Software*, 3. 255-264.
- Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R. and Majumdar, R., The Blast Query Language for Software Verification. in *Proceedings of the 11th International Static Analysis Symposium* (2004), Springer-Verlag, 2-18.
- Björner, D. and Jones, C.B. *The Vienna Development Method: The Meta-Language*. Springer-Verlag, 1978.
- Blum, M. and Kannan, S. Designing Programs that Check Their Work. *Journal of the ACM*, 42 (1). 269-291.
- Chalin, P. Ensuring Continued Mainstream Use of Formal Methods: An Assessment, Roadmap and Issues. Group, D.S.R. ed., Concordia University, 2005.
- Cline, M.P. and Lea, D., Using Annotated C++. in *C++ at Work Conference*, (1990).
- Cook, J., Assertions for the Tcl Language. in *Fifth Tcl Workshop*, (Boston, MA, 1997), 73-80.

15. Cusumano, M.A. and Selby, R.W. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. Free Press, New York, NY, 1995.
16. Deutsch, L.P. *An Interactive Program Verifier*, University of California, Berkeley, Berkeley, CA, 1973.
17. Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.
18. Dijkstra, E.W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18 (8). 453–457.
19. Dillon, L.K., Kutty, G., Moser, L.E., Melliar-Smith, P.M. and Ramakrishna, Y.S. A Graphical Interval Logic for Specifying Concurrent Systems. *ACM Transactions on Software Engineering and Methodology*, 3 (2). 131-165.
20. Elbaum, S. and Diep, M. Profiling Deployed Software: Assessing Strategies and Testing Opportunities. *Transactions on Software Engineering*, 31 (4). 312-327.
21. Ernst, M.D., Cockrell, J., Griswold, W.G. and Notkin, D. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27 (2). 1-25.
22. Evans, T.G. and Darley, D.L., On-Line Debugging Techniques: A Survey. in *AFIPS Fall Joint Computer Conference*, (1966), 37-50.
23. Findler, R.B., Latendresse, M. and Felleisen, M., Behavioral Contracts and Behavioral Subtyping. in *Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, (Vienna, Austria, 2001), 229-236.
24. Flater, D.W., Yesha, Y. and Park, E.K. Extensions to the C Programming Language for Enhanced Fault Detection. *Software-Practice and Experience*, 23 (6). 617-628.
25. Floyd, R.W., Assigning Meaning to Programs. in *Symposium on Applied Mathematics*, (New York, 1967), American Mathematical Society, 19-32.
26. Gautron, P., An Assertion Mechanism Based on Exceptions. in *Fourth C++ Technical Conference*, (1992), USENIX Association, 245-262.
27. Goldstine, H.H. and Von Neumann, J. Planning and Coding Problems for an Electronic Computing Instrument. in Taub, A.H. ed. *Jon von Neumann, Collected Works*, Pergamon Press, London, England, 1963, 80-235.
28. Good, D.I., London, R.L. and Bledsoe, W.W. An Interactive Program Verification System. *IEEE Transactions on Software Engineering*, SE-1. 59-67.
29. Goodenough, J.B. Exception Handling: Issues and a Proposed Notation. *Communications of the ACM*, 18 (12). 683–696.
30. Gosling, J., Joy, B. and Steele, G. *The Java(TM) Language Specification*. Addison-Wesley, 1996.
31. Gries, D. *The Science of Programming*. Springer-Verlag, New York, 1981.
32. Hantler, S. and King, J. An Introduction to Proving the Correctness of Programs. *ACM Computing Surveys*, 8 (3). 331-353.
33. Haran, M., Karr, A., Orso, A., Porter, A. and Sanali, A., Applying Classification Techniques to Remotely-collected Program Execution Data. in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (Lisbon, Portugal, 2005), ACM SIGSOFT, 146 - 155
34. Helmbold, D.P. and Luckham, D.C. Debugging Ada Tasking Programs. *IEEE Software*, 2 (2). 47-57.
35. Hoare, C.A.R. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12 (10). 576-580.
36. Hoare, C.A.R. Assertions: A Personal Perspective. *IEEE Annals of the History of Computing*, 25 (2). 14 - 25.
37. Holt, R.C. and Cordy, J.R. The Touring Programming Language. *Communications of the ACM*, 31 (12). 1410-1423.

38. Holzmann, G.J. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23 (5). 279-294.
39. Holzmann, G.J. *The SPIN Model Checker*. Addison-Wesley, 2004.
40. Igarashi, S., London, R.L. and Luckham, D.C. Automatic Program Verification I: A Logical Basis and its Implementation. *Acta Informatica*, 4. 145-182.
41. Jones, C.B. *Systematic Software Development Using VDM*. Prentice-Hall International, 1990.
42. Jones, C.B. The Early Search for Tractable Ways of Reasoning about Programs. *IEEE Annals of the History of Computing*, 25 (2). 26-49.
43. King, J.C. A Program Verifier, Carnegie Mellon University, Pittsburgh, PA, 1969.
44. Leveson, N.G., Cha, S.S., Knight, J.C. and Shimeall, T.J. The Use of Self Checks and Voting in Software Error Detection: An Empirical Study. *IEEE Transactions on Software Engineering*, SE-16 (4). 432-443.
45. Liblit, B., Aiken, A., Zheng, A. and Jordan, M., Bug Isolation via Remote Program Sampling. in *Programming Language Design and Implementation*, (San Diego, CA, 2003), ACM SIGPLAN, 141-154
46. Linger, R.C., Mills, H.D. and Witt, B.I. *Structured Programming: Theory and Practice*. Addison-Wesley, 1979.
47. Liskov, B. and Guttag, J. *Abstraction and Specification in Program Development*. MIT Press, 1986.
48. Liskov, B.H. and Wing, J.M. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16 (6). 1811-1841.
49. Luckham, D.C. *Programming with Specifications: An Introduction to Anna, a Language for Specifying Ada Programs*. Springer-Verlag, 1990.
50. Luckham, D.C., German, S.M., vonHenke, F.W., Karp, R.A., Milne, P.W., Oppen, D.C., Polak, W. and Scherlis, W.L. Stanford Pascal Verifier User Manual, Department of Computer Science, Stanford University, 1979.
51. Luckham, D.C., Helmbold, D.P., Bryan, D.L. and Haberler, M.A., Task Sequencing Language for Specifying Distributed Ada Systems (TSL-1). in *PARLE—The Conference on Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, (1987), Springer-Verlag, 444-463.
52. Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D. and Mann, W. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21 (4). 336-355.
53. Luckham, D.C. and vonHenke, F.W. An Overview of Anna: a Specification Language for Ada. *IEEE Software*, 2 (2). 9-24.
54. Mahmood, A. and McCluskey, E.J. Concurrent Error Detection Using Watchdog Processors—A Survey, Computer Systems Laboratory, Stanford University, 1985.
55. Maker, P.J. GNU Nana User's Guide, Version 2.4, School of Information Technology, Northern Territory University, 1998.
56. Manna, Z. Properties of Programs and the First-Order Predicate Calculus. *Journal of the ACM* 16 (2). 244-255.
57. Manna, Z. and Pnueli, A. Verification of Concurrent Programs: The Temporal Framework. in Boyer, R.S. and J. S. Moore eds. *The Correctness Problem in Computer Science*, Academic Press, London, 1981, 215-273.
58. McCarthy, J., Towards a Mathematical Science of Computation. in *IFIP Congress 62*, (1963), North-Holland.
59. Meyer, B. Applying 'Design by Contract'. *IEEE Computer*, 25 (10). 40-51.
60. Meyer, B. Eiffel: A Language and Environment for Software Engineering. *Journal of Systems and Software*, 8 (3). 199-246.

61. Meyer, B. *Eiffel: The Language*. Prentice Hall, 1991.
62. Meyer, B. Personal communication. Clarke, L. and Rosenblum, D. eds., 2005.
63. Müller, M.M., Typke, R. and Hagner, O., Two Controlled Experiments Concerning the Usefulness of Assertions as a Means for Programming. in *International Conference on Software Maintenance*, (Montreal, Canada, 2002), 84–92.
64. Musser, D.R., Abstract Data Type Specification in the Affirm System. in *Specifications of Reliable Software*, (Cambridge, MA, 1979), IEEE Computer Society, 47-57.
65. Naur, P. Proof of Algorithms by General Snapshots. *BIT*, 6. 310–316.
66. OMG. OCL 2.0 Specification, Version 2.0, Object Management Group, Framingham, MA, 2005.
67. OMG. Unified Modeling Language (UML) Version 1.4.2, Object Management Group, Framingham, MA, 2001.
68. Osterweil, L.J. Using Data Flow Tools in Software Engineering. in Muchnick and Jones eds. *Program Flow Analysis: Theory and Application*, Prentice-Hall, Englewood Cliff, N. J., 1981.
69. Parasoft. Jcontract, 2004.
70. Parnas, D.L. A Technique for Software Module Specification with Examples. *Communications of the ACM*, 15 (5). 330-336.
71. Pnueli, A., The Temporal Logic of Programs. in *Eighteenth Symposium on Foundations of Computer Science*, (Providence, RI, 1977), 46-57.
72. Popek, G.J., Horning, J.J., Lampson, B.W., Mitchell, J.G. and London, R.L., Notes on the Design of Euclid. in *ACM Conference on Language Design for Reliable Software*, (Raleigh, North Carolina, 1977), 11-18.
73. QualityWeek, in *International Quality Week*, (San Francisco, CA, 1987 - 2006), Software Research Institute.
74. Rosenblum, D.S. A Practical Approach to Programming with Assertions. *IEEE Transactions on Software Engineering*, 21 (1). 19-31.
75. Rosenblum, D.S., Sankar, S. and Luckham, D.C., Concurrent Runtime Checking of Annotated Ada Programs. in *Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, (New Delhi, India, 1986), Springer-Verlag, 10–35.
76. Ryder, B.G. The PFORT Verifier. *Software --- Practice and Experience*, 4. 359-378.
77. Salvador, R. Personal communication with VP of Research and Development at Parasoft. Clarke, L. and Rosenblum, D. eds., 2006.
78. Sankar, S., Run-Time Consistency Checking of Algebraic Specifications. in *The 4th Software Testing, Analysis and Verification Symposium*, (1991), ACM SIGSOFT, 123–129.
79. Sankar, S. and Rosenblum, D.S. Runtime Checking and Debugging of Formally Specified Programs. *ACM Computing Surveys*, 23 (1). 125–127.
80. Sankar, S., Rosenblum, D.S. and Neff, R.B., An Implementation of Anna. in *Ada in Use: The Ada International Conference*, (1985), Cambridge University Press, 285–296.
81. Schwartz, R.L., Melliar-Smith, P.M. and Vogt, F.H., An Interval Logic for Higher-Level Temporal Reasoning. in *2nd Symposium on Principles of Distributed Computing*, (1983), ACM SIGOPS, 173–186.
82. Spivey, J.M. *Introducing Z: A Specification Language and its Formal Semantics*. Cambridge University Press, Cambridge, MA, 1988.
83. Spivey, J.M. *The Z Notation A Reference Manual*. Prentice Hall International Series in Computer Science, Englewood Cliffs, NJ, 1992.
84. Spivey, J.M. *The Z Notation: A Reference Manual*. Prentice Hall International, Englewood Cliffs, NJ, 1989.

85. Stucki, L.G., Automatic Generation of Self-Metric Software. in *IEEE Symposium on Software Reliability*, (1973), IEEE Computer Society, 94-100.
86. Stucki, L.G. and Foshee, G.L., New Assertion Concepts in Self-metric Software Validation. in *1975 International Conference on Reliable Software*, (1975), 59-71.
87. Sun Microsystems, I. Programming with Assertions, 2002.
88. Tardo, J. and Goguen, J.A., An Introduction to OBJ: a Language for Writing and Testing Algebraic Specifications. in *Specifications of Reliable Software*, (Cambridge, MA, 1979), 170-189.
89. Thompson, K. and Ritchie, D.M. *UNIX Programmer's Manual*. Murray Hill, NJ, 1979.
90. Turing, A., Checking a Large Routine. in *Conference on High Speed Automatic Calculating Machines*, (Cambridge, UK, 1949), 67-69.
91. Wegbreit, B. Constructive Methods in Program Verification. *IEEE Transactions on Software Engineering*, SE- 3. 193-209.
92. Wirth, N. and Hoare, C.A.R. A Contribution to the Development of Algol. *Communications of the ACM*, 9 (6). 413--431.
93. Wulf, W.A., London, R.L. and Shaw, M. An Introduction to the Construction and Verification of Alphard Programs. *IEEE Transactions on Software Engineering*, SE-2 (4). 253-265.
94. Yau, S.S. and Cheung, R.C., Design of Self-Checking Software. in *International Conference on Reliable Software*, (1975), ACM and IEEE Computer Society, 450-457.
95. Yin, H. and Bieman, J.M., Improving Software Testability with Assertion Insertion. in *International Test Conference*, (1994), 831-839.
96. Zuse, K. *Der Plankalkül*, Gesellschaft für Mathematik und Datenverarbeitung, 1972.