

Assertions: Instrumenting Safety Critical Code for Maintenance JPL D-15199 *

Robyn R. Lutz and Hui-Yin Shaw
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

November 21, 1997

1 Introduction

NASA's software subsystems undergo extensive modifications and updates over their operational lifetimes. Much of the software safety work completed during the development of a system becomes outdated or falls into disuse during the maintenance phase. In situations of code reuse or redesign, the design assumptions, interface requirements, and other implementation considerations for safety-critical modules must be reconsidered as extensively as in the original development effort. New and existing projects need preservation of safety analysis through builds and reuse.

Assertions can capture and specify some preconditions, postconditions, invariants, and assumptions that are required for the software to function safely. Assertions are widely used and even more widely recommended as a debugging aid. In addition, assertions can provide benefits for preserving software safety information and features during maintenance. These benefits include (1) making safety-critical code more maintainable; (2) supporting a continuous safety assurance process; (3) reducing interface errors (a frequent source of safety-related errors); and (4) providing runtime checking of the system state before critical operations.

2 Software Safety

Leveson defines safety as the freedom from accidents or acceptable losses [15]. She states that "safety is a judgment of the acceptability of risk, and risk, in turn, as a measure of the

*This report describes work for the research project, "Maintaining Software Safety," funded by NASA's Office of Safety and Mission Assurance under UPN 323-08-5H.

probability and severity of harm to human health. A thing is safe if its attendant risks are judged to be acceptable”. When apply to unmanned system, “safety is a judgment of the acceptability of risk, and risk, in turn, as a measure of the probability and severity of harm to the success of the mission. A system which performs the mission is safe if its components risks are judged to be acceptable.”

Software system safety implies that the software will execute within a system context without contributing to hazards. Leveson points out that software can affect system safety in two ways:

1. it can produce output values and timing that lead system to a hazardous state
2. it can fail to recognize or handle hardware failures that it is required to control or to respond to.

Safety-critical software is any software that can directly or indirectly contribute to the occurrence of a hazardous system state [15]. Sommerville, in *Software Engineering*, states that there are two classes of safety-critical software, although the distinction between the two is not always trivial. [26]. The two classes are:

1. Primary, safety critical. Malfunctioning of such software can cause a system malfunction which results in human injury, environmental damage, or mission failure.
2. Secondary, safety critical. Malfunctioning of such software can indirectly result in injury, environmental damage, or mission failure .

Software safety should be treated with attention throughout the software system’s life cycle. Software safety is sometimes discussed as an element of software reliability, but software reliability and software safety are not the same thing. Sommerville states that “although a safety-critical system should be reliable in that it should conform to its specification and should operate without failures, software systems which are reliable are not necessarily safe”. On the other hand, software security and software safety are closely related. Leveson points out that “software safety and software security both deal with threats or risks and involve protection against losses, although the losses involved may be different”. Even so, software safety and security have different focuses. “Security focuses on malicious actions, whereas safety is concerned with well-intended actions.” Because software safety is a distinct quality, it should be treated separately from reliability and security. By doing so, the issues with safety will be open and not be hidden from view.

3 Software Maintenance

Sommerville defines software maintenance as “the process of changing a software system after it has been delivered and is in use” [26]. For an overview of the maintenance process, refer to Sommerville’s *Software Engineering*.

Software maintenance is triggered by a set of change requests from the system users or customers. There are basically three types of software change during the maintenance phase: Corrective, adaptive and perfective. Corrective maintenance refers to fixing reported problems. Adaptive maintenance has to do with change as the result of changing environment.

Perfective maintenance involves implementing new or refined requirements. The process of implementing change is similar for all three.

3.1 Problems in Software Maintenance

Software maintenance is generally considered to be difficult, time consuming, and expensive [2]. Software maintenance involves: 1. Understanding the existing code, 2. Determining whether to make changes, 3. Assessing the impact of modifications, 4. Rebuilding the code after alterations, and 5. Regression testing to validate changes. Common problems during maintenance include:

1. lack of a well-defined process
2. lack of documentation
3. lack of techniques
4. lack of time
5. fault induction
6. lack of experience and knowledge domain
7. lack of motivation

A well-defined process is often lacking in the maintenance phase. The maintenance phase usually receives the least attention by projects and developers alike. The change control and the configuration management process in this phase are often not well-enforced. When this happens, the intended software change and software versions may not be easily traceable and verified. Configuration management errors can be made when new or updated code modules are integrated into unchanged modules. Therefore, a well-defined configuration management process is a must in the maintenance phase. Safety-related changes should be documented in hazard report or safety analysis report. These reports should be tracked through verifications and closures.

Having software documentations consistent with actual software implementation has traditionally been a struggle. Not only the documentation is often lagging behind software implementation during the development phase, when the software goes through changes, these changes are frequently not reflected in the associated documents. This lack of (updated) documentation makes the software impossible to understand and maintain. Hence, the program understanding becomes the a bottleneck in the maintenance phase [23]. Therefore, it is important to enforce the links between the software and its associated documentation.

Software changes are often done as patches. Reasons for this include that the program being maintained was unstructured, the documentation was unavailable or inadequate, there was a lack of time, and/or there was a lack of knowledge in the application domain. It is possible to use techniques, such as logical analysis and code assertions [8], to help alleviate some of these problems.

Time has always been a precious commodity in software development and maintenance. With the lack of time, a problem may not have been well-understood before attempts are

made to correct the problem, and patches may be made instead of finding a more suitable solution. Additionally, corrections may not have been thoroughly verified before re-release, resulting in degraded quality in the software product.

Correction of one fault may initiate an untested path or function in previously integrated and installed systems [27]. Changes may induce new faults, which can trigger further change requests. This problem is especially prominent in programs that are complex or difficult to understand. One remedy is to run a suite of tests periodically to detect any regression problems. Operational profile testing may also be used to monitor the quality of the software and detect problems. Having available up-to-date documentation of software products also helps.

People maintaining the software may be inexperienced and unfamiliar with the application domain and/or the methodology used during the development. Again, the combined use of techniques such as logical analysis and code assertions may help alleviate this problem.

The fact that the maintenance phase is the least favorable phase in the life-cycle, coupled with obstacles in performing maintenance tasks, often causes a lack of motivation in the maintenance teams. Refer to Sommerville for Boehm's suggestions on improving staff motivation [26].

3.2 Cost of Software Maintenance

Maintenance remains the single most expensive software engineering activity [26]. Problems encountered during the maintenance phase as discussed above contribute to the escalating maintenance cost. In some businesses, it has been estimated that 80% of all software expenditure is consumed by system maintenance and evaluation [26]. A Unitech International Corp. study indicates that for every dollar spent on new application development, five dollars were spent for maintaining the application during a five-year period [1]. This cost increases as the systems age.

3.3 Software Safety Considerations in Maintenance

It is important that the problems in software maintenance be addressed when developing and maintaining safety-critical software. Considerations to be made during development and maintenance of safety-critical software include:

1. Have certified development process. Particularly, configuration management is an essential process in the development and maintenance of safety-critical software.
2. Have a good set of design guidelines. Design features to be considered include:
 - Information hiding: Isolate those parts of the system which are safety-critical from other parts of the system
 - Simplicity: Keep the system as simple as possible
 - New attitude: Look at what you do NOT want software to do along with what you want it to do, and assume that things will go wrong [16].

3. Assess safety hazard from system identifications and analysis down to software hazard analysis and specifications, and operational hazard analysis [26]. Use of formal specification and associated verification is helpful in enforcing a detailed analysis of the safety specification. Doing this may reveal potential inconsistencies or omissions which might not otherwise be discovered.
4. Have good coding practice. Refer to Sommerville for more in-depth discussion in this area. Sommerville recommends:
 - As with design, keep the safety-critical software simple.
 - Isolate safety-critical code from the rest as much as possible.
 - Avoid potentially error-prone language features; e.g. low-level features such as pointers, for which their terse style can be hard to understand and, thus, difficult to validate.
 - Exclude the use of language features which are not properly defined.
5. Avoid fault induction. Exercise caution always, from hazard analysis to safety-critical software implementation, testing and maintenance.
6. Conduct thorough validation and reviews. Because safety-critical parts of the software systems are usually relatively small, performing validations using mathematical specifications are well justified [15, 26]. Thorough testing of software safety-critical paths should not be over-looked.
7. Apply software safety techniques. There are various techniques addressing software safety. One of them, the assertions technique, is the topic of this report.

4 Assertions

4.1 Description

An assertion is a Boolean statement in a program's code that makes a claim about the state of the computation [9]. If the assertion is true, execution of the program continues without interruption. If the assertion is false, the program will take some specified action. What actions are possible depend on the programming language and the compiler. Most commonly, an assertion that is not satisfied will cause an abort of the program with an accompanying error message.

For example, the assertion may specify that a certain variable has a value within some range. If the variable's value is, in fact, within the specified range at that point in the program, then the assertion is true. If, contrary to the programmer's or user's expectations, the variable's value is currently outside the specified range, then the fact that the assertion is false will trigger a (language-dependent) response.

In some programming languages the assertions can be compiled and checked at runtime, making the assertions useful for debugging and operational tests. Often assertions are first written using a formal specification language, pseudocode, or even English, in an attempt

to precisely specify the required behavior before code is produced. Even when assertions are not compilable, they serve as valuable internal program documentation. Headington and Riley give as an example [9]:

```
for (i = 0; i < 50; i++)
    trialCount[i] = 0;
// ASSERT: All trialCount[0..49] == 0
```

Depending on the level of formality, assertions can be used for assuring consistency between the design and the code, for facilitating software maintenance, for concisely specifying proper reuse, or for aiding in proofs of program correctness. Assertions can abstract out and present the key features of the software that the implementation may blur.

The placement of the assertion in the code determines when it executes and, hence, its use. Three main types of assertions are preconditions, postconditions, and invariants.

Preconditions specify what must be true at the time that a caller invokes a function. Such assertions often check that the parameters provided to a function are valid (frequently, not NULL). Postconditions specify what must be true when the function completes its service [9].

Postconditions typically describe parameters that were altered by the function, but not local variables. Preconditions and postconditions are widely used in Object Oriented Development because they capture the contract between the client (the object requesting a service) and the server (the class that provides that service or operation).

The scope of an Invariant assertion is, most commonly, a loop invariant that specifies what must be true every time the program reaches that location in the loop. Such assertions are usually placed just prior to the test for loop exit. Another common type of invariant assertion is the class invariant. This assertion states what is true about the environment of a class instance before and after any operation of that class is invoked [9]. Luckham recommends that invariant assertions “be used locally over a block, loop, or case statement to document that something bad does not happen—or, more positively, that some good properties achieved earlier remain invariant” [18].

4.2 Shipped code vs. debugging code

Assertions are widely recommended for debugging purposes during program testing. They allow erroneous states to be identified as soon as possible, rather than having to backtrack from the final symptoms of failure to the earlier cause. Wilson acknowledges the insight into the paths through the code that assertions can provide, calling them “a trail of breadcrumbs” and describing the placing of an assertion as “putting a stake in the ground” [37]. Joch states, “Use assertions liberally. An assertion is simply a line of code that says, ‘I think this is true. If it isn’t, something is wrong, so stop execution and let me know right away.’ If a value is supposed to be within a certain range, check first. Make sure that pointers point somewhere and that internal data structures are consistent. . . . You will find problems quickly, making them much easier to track down” [11].

More controversial is the issue of whether assertions belong in delivered code, or whether all assertions should be compiled out of code before delivery. Joch recommends compiling assertions out of production code before it enters final testing. Maguire recommends

maintaining two versions of code, one with assertions for debugging and the other without assertions to ship [20]. A “fast and sleek” version is shipped, while the “slow and fat” version, loaded with assertions, is used during testing. Other authors recommend retaining assertions in the production code for faster debugging of software defects found during operations [37].

The choice of which, if any, assertions to keep in the runtime code is made based on the application’s requirements, the consequences of the increased code size, the overhead (assertion checking means decreased performance and increased memory usage), and safety (highly critical software often should not abort if it fails an assertion).

5 Resources

Assertions are a feature of many programming languages, but their capabilities and roles differ somewhat among these languages. We here briefly describe the use of assertions in Eiffel, ANNA, Visual C++, and formal specification languages, and then discuss some related work. The languages were chosen to identify what is possible with assertion handling and to point out some possible directions for the future. Other programming languages also include assertions (see, e.g., [29] regarding Common Lisp). The discussion here is thus representative of assertions’ capabilities, rather than a comprehensive description. Enhanced use of assertions and enhanced assertion-handling techniques offer ways to build and maintain safer software systems in a variety of programming languages.

5.1 Assertions in Specific Programming Languages

5.1.1 Eiffel

Eiffel is an object oriented programming language. It was created by Bertrand Meyer of Interactive Software Engineering in the 1980’s to support the development of safe and robust object oriented software systems [35].

Eiffel includes powerful and flexible assertion-handling capabilities that far exceed those available in most languages. An optional *require* clause in each routine of a class contains the preconditions (constraints on the routine’s parameters) that any potential client of the routine must guarantee. An *ensure* clause at the end of the routine contains any postconditions that relate attribute values before and after the routine is invoked. Class invariants are in an *invariant* clause at the end of the class definition.

An exception is raised in the client’s code if the routine’s preconditions are not satisfied, but in the routine itself if the postconditions or class invariants are not satisfied. Exception handlers can be implemented using the *rescue* clause to report the error, to restore a safe state, and to reinvoke the routine if required.

Assertions are also used to check loop invariants and for debugging (using the *check* clause). Assertion handling supports inheritance in which preconditions of a routine in a descendant class can be weakened and postconditions strengthened. Thus, the descendent class can offer at least the same guarantees of service as its parent class but for a broader collection of inputs. In addition, different levels of assertion handling can be activated so that, for example, all assertion checking is turned on or off, only the precondition checks are turned on, etc.

Eiffel's assertion handling capabilities support the development and continued maintenance of safe software. Wiener states, "Assertion handling promotes a high level of fault detection and safety. It provides a semi-formal basis for doing 'correct' programming and verifying this correctness at run-time. It provides an additional level of self-documentation, informing a client about the formal specifications (preconditions) that must be satisfied in order to use a routine or class" [35]. The emphasis on using assertions to assure robust interfaces among the software components is valid for any language.

5.1.2 ANNA

ANNA is a specification language that extends the Ada language to allow *annotations* to be included and runtime-checked in Ada programs. ANNA stands for ANNotated Ada [18].

Assertions are one kind of statement annotations. They can be used for permanent run-time checks, for temporary run-time checks during testing and debugging, or for formal documentation of the completed program.

Annotations, including assertions, use the formal comment symbol `--|`. An example of an assertion is:

```
S := (3,3);
--| SQUARE (S.X) = 9;
```

[18]. The emphasis in ANNA is on structured program development and on building a framework of assertions that support proofs of the consistency of the annotations and the programs. To this end, Luckham offers rules for choosing and placing assertions that embed each assertion in a larger context. "An assertion should express a subgoal achieved locally towards attaining a higher level of annotation." "Assertions should formalize sufficient information that is true of the computation state at their positions to support assertions at later points in the flow of control."

5.1.3 Visual C++

The programming environment Visual C++ 4.0 supports several flavors of assertions with its extensive on-line documentation and libraries. The ANSI C/C++ *assert* function, which requires the `<assert.h>` header, is available in both the debug and release versions of the library. When linked with a debug version of the runtime library, the Visual C++ assertion handling creates a message box with three buttons: Abort, Retry (e.g., to find out what functions were executing when the assertion failed), and Ignore, to simplify debugging.

If the program uses the MFC (Microsoft Foundation Classes) library, the MFC *ASSERT* macro can be used, e.g.,

```
{ASSERT (p1 == p2);}
```

The `_ASSERT` macro prints a diagnostic message, while the `_ASSERTTE` macro also prints a string representation of the failed expression. The disadvantage is that every expression evaluated by `_ASSERTTE` must be included in the debug version as a string constant, using a perhaps unacceptable amount of memory.

The on-line documentation provided with Visual C++ 4.0 suggests the following uses for assertions:

- To check that arguments are not NULL
- To check that an object's internal state is valid
- To check whether a particular object belongs to a specified class

```
ASSERT ( pObject1 -> IsKindOf (RUNTIME_CLASS (CPerson) ) );
```

- To check that the results of an operation are as intended
- To test for error conditions (for example, memory leaks):

```
ASSERT (ialloc %50 == 0);
```

(where memory is allocated in blocks of 50 bytes).

Assertions can also be useful for avoiding unwarranted assumptions about the software's operating environment. GNU C provides three predefined predicates: *system* (i.e., the operating system and version number), *cpu*, and *machine* (<http://www.cygnus.com/library/cpp/cpp-36.html>). Assertions using these predicates can identify some limitations on the portability of code that is to be reused or updated.

5.1.4 Specification Languages

Model-based specification languages such as Z, Larch, PVS, and VDM can specify a computation based on its precondition (the requirements on the state before the computation) and its postcondition (the desired final state) [6, 12, 14, 28]. Assertions regarding the program state are thus an integral part of the formal specification. Larch has been extended with several behavioral interface specifications to specify modules to be written in particular programming languages. Larch/C++, for example, allows the interface between the abstract model and the particular C++ features to be specified.

5.2 Related Work

Because of the overlap in this effort of maintenance, safety, and assertions (testing), there is a large body of related work. Assertions are mentioned often in the maintenance literature as one of several techniques for improving software's maintainability. For example, the strongest postcondition predicate transformer has been used as the formal basis for translating code into formal specifications during reverse engineering for maintenance [7]. The collaborative ESPRIT project REDO (reverse engineering) reports that "intermediate assertions and plausible loop invariants" have been useful in the reverse engineering process [3].

Assertions are mentioned in the safety literature as one of several techniques for improving software's safety, and are described extensively in the testing literature. Voas and Miller,

for example, have worked to identify those places in the code where traditional testing is unlikely to uncover software faults, but assertions may reveal them. Assertions’s capability to provide insight into intermediate results during debugging is emphasized. They correctly call assertions “windows into the computational state” [32]. Montgomery provides assertion handling as part of the testing toolkit SRLT [21]. SRLT can give detailed stack traces for each thread when an assertion fails.

In addition, the methods used to identify meaningful assertions have ties both to the formal methods literature (axiomatic specifications and proofs of correctness [25]) and to the dependency graphs and slicing techniques (e.g., the parse-tree static slicing tools of [17]) used for testing and software maintenance.

Related work of interest investigates how code can be inserted into an operational system to monitor to what degree the system is meeting its performance requirements. This run-time monitoring code, which sometimes looks much like assertions, can alert maintainers when redesign or update to the design parameters is needed [5].

5.3 Examples

The description of assertions above includes several short examples. The associated references provide pointers to additional examples on each topic. In this section an example from [20] is given of debug C code that uses *assert* from the ANSI *< assert.h >* header file to check for null pointers and abort with an error message if it finds one.

```
void *memcpy(void *pvTo, void *pvFrom, size_t size)
{
    byte *pbTo    = (byte *)pvTo;
    byte *pvFrom  = (byte *)pvFrom;

    assert(pvTo != NULL && pvFrom != NULL);

    while (size-- > 0)
        *pvTo++ = *pvFrom++;

    return (pvTo);
}
```

6 Industrial Experience with Assertions

The use of assertions in the industry is not new. In addition to their use in formal verification, assertions also have been recognized as a potentially powerful tool for automatic runtime detection of software faults during debugging, testing and maintenance; and as a permanent defensive programming mechanism for runtime fault detection in production versions of software systems [24].

Areas of assertions applications in software maintenance include:

1. General integrity checks

2. Logical code analysis
3. Understanding of unfamiliar software
4. Run-time check, debugging, and testing
5. Use assertions to validate function arguments and to alert programmers when they do something undefined

An example of the advantage with the use of assertions can be found in the Microsoft's Word (Maguire). In the late 1988, the Word for MS-DOS was delayed in its shipment by 3 months. This delay was due to the fact that a key component used by the Word was a product from Microsoft's application tool group. The Word group had thought that they were going to ship any day. But the application tool group, while believing that they were almost ready, their product kept showing up with bugs. The key difference between the Word group and the tool group here was that the Word software was loaded with assertions and debug code, while the tool group hardly used any assertions in their code. The tool group did not have a good way of detecting and preventing errors in their software.

Another example of integrity checking using assertions can be found in the use of APP. APP is an assertion processing tool. It stands for An Annotation PreProcessor. APP was developed for C programs in UNIX-based development environments by David S. Rosenblum at AT&T [24]. APP includes the use of assert feature of the C language in its assertion constructs. When using APP, assertions are written inside comment regions of a C program, using the extended comment indicators `/*@...@*/`. Each APP assertion specifies a constraint that applies to some state of a computation. This constraint is specified using C's expression language.

APP converts each assertion to a runtime check, which tests for the violation of the constraint specified in the assertion. This tool also provides the option to attach a violation action to the assertion. There is also an optional severity level for each assertion. This can be used to control the amount of assertion checking that is performed at runtime without recompiling the program to add or remove checks.

Rosenblum has applied assertions to check the software function interfaces as well as function bodies. In his experience using APP at AT&T, Rosenblum has found that assertion checks applied during the software system development "automatically revealed a number of serious faults, and the diagnostic information they provided was often sufficient to quickly isolate and remove the faults."

6.1 Logical Code Analysis

Hart worked with the software maintenance teams to determine the feasibility of the logical code analysis in software maintenance [8]. Logical code analysis refers to the determination of the code properties using weakest preconditions and postconditions. By analyzing logical properties of the code and inserting preconditions, postconditions and invariants at appropriate places in the code, the maintenance teams were able to analyze and correct the software problems. Hart points out that this technique provides the maintenance teams a method to isolate defects and to determine code correctness. Additionally, no real understanding of the

application (one of the problems faced by software maintenance) is required to perform the task.

6.2 Understanding Unfamiliar Software

Electris Software Product uses the assertion technique to assist software developers and maintainers to understand unfamiliar software systems [4]. In their product, LibC/Inside works by instrumenting (code assertions) all system interactions at the operating system level, to produce a comprehensive analysis of a software subsystem. This enables the user to trace the execution of dynamically linked software subsystems. Items monitored include: calling address, function name, arguments passed to the function, return value, and any arguments that may be modified. Because library calls often comprise over 90% of the code executed by a program, therefore by using assertions LibC/Inside is able to provide users the understanding of how a subsystem functions.

6.3 Run-Time Checking, Debugging and Testing

As mentioned earlier, some Microsoft products, such as Word, are loaded with code assertions and debugging codes. Experience at Microsoft has shown that the use of code assertions is beneficial and improves the quality of the software product.

However, code assertions do take up system resources. Assertions work very well in environments where memory and timing constraints aren't a paramount issue. For systems where timing is critical and/or memory is scarce (such as embedded, real-time systems), the use of assertions cannot be amply applied. To overcome this problem, Williams is able to apply assertions as part of a toolset for embedded systems [36]. In his CodeTEST, William combines a hardware probe head with software code instrumentation to test running code in the embedded systems.

The CodeTEST's ability to test running code in the target system is a passive hardware probe that monitors program execution without consuming target resources. The hardware probe head monitors the processor bus and detects "tags" that have been inserted into the object code. The instrumenter inserts test-point instructions into the source code that compile into the tags in the object file. These tags are recognized by the probe head as signals to collect test data. With this method, the use of target memory and the impact on the timing of the target system are minimized.

6.4 Performance Analysis

Pure Atria uses its patented assertion technique, Object Code Insertion (OCI), in its performance analysis tool, Quantify. The OCI assertion technique enables Quantify to measure the execution speed of software applications and automatically identify portions of application code that slow down execution speed. With this, the user can quickly detect performance bottleneck at the source, enhancing the effort to improve performance.

6.5 Memory Usage and Monitoring

Again, Pure Atria uses its patented OCI technique to provide a memory usage and monitoring tool. Purify provides the user the ability to monitor memory usage and provide a warning when memory violation takes place. By inserting checking and logging instructions into the object code around every memory access at run-time, Purify is capable of detecting and reporting memory related errors, sometimes even before the failure takes place. Example of faults detected include reading and writing past the boundary of an array, memory leakage, and use of uninitialized memory. Memory errors are normally hard to find and have long latency period [33]. Pure Atria's code assertion technique enables software developers and maintainers the ability to remove memory errors quickly.

7 Safety Assertions

We are interested here in the advantages that assertions pose for preserving software safety information and features during maintenance. Evolutionary or incremental development with successive builds adding greater functionality (as on DS-1) are included in this discussion.

7.1 Role of Assertions

1. *Assertions make safety-critical code more maintainable.*

Wilson points out that assertions can notify you when changes made elsewhere in a program, perhaps years after initial development, violate the contract for an existing interface [37]. Assertions can serve as reminders during code inspections and code reviews [26]. During regression testing, the existence of assertions assists in verifying the updated code [15]. Maguire sums up the role of assertions as, "Assertions are forever."

The role of assertions in the maintenance phase thus encompasses testing and debugging of updated code, internal program documentation, continued compatibility between design and source code, safety proofs of updated code, and high-level specification of the client/server contract.

2. *Assertions can support a continuing safety assurance process.*

Sommerville states that assertions can capture and preserve the output of the safety analysis, providing a link between the safety requirements and the implementation. "In safety critical systems, the assertions should be generated from the safety specification rather than the system specification. They are intended to assure safe behaviour rather than behaviour which conforms to the specifications" [26].

3. *Assertions can reduce interface errors, historically a source of safety-related errors [19, 26].*

The emphasis on specifying the preconditions and postconditions of the software modules, and the notion of establishing a contract between the software providing a service and the software invoking the service are powerful mechanisms for ensuring the safety

of program interfaces. Sommerville gives as an example the software for an insulin pump. This software performs assertion checking on its parameters to ensure that the increments of insulin that it is instructed to deliver are within safe bounds [26].

4. *Assertions can provide runtime checking of system state before critical operations* [26].

This is usually performed by validating the arguments passed to every safety-critical function. Once hazardous states have been identified through the safety analysis, checks can be added to the code to preclude entrance to those states. Maguire suggests that during testing, assertions be used for redundancy checking of critical outputs. For example, in the debug version of Excel, a complex algorithm is double checked by a second, different algorithm. An assertion fires if the results of the two calculations differ [20].

7.2 Risks and Limitations

Discussions of assertions include several warnings about the risks of the improper use of assertions. These fall into the following categories:

- **Incorrect assertions.** Assertions that are incorrect can lead to incorrect software, or even software that contributes to hazardous system states. Leveson points out that providing access to the information introduces the possibility of corrupting the information [15].
- **Unclear assertions.** Maguire makes the point that assertions that are unclear should always be commented, so that the tester and maintenance programmer are certain what the assertion is claiming and why.
- **Assertions in shipped code.** The risks of degraded performance and excessive memory usage has been discussed above. In addition, since failed assertions often abort the program's execution, the tradeoffs between increased reliability and possible aborts caused by runtime assertions should be carefully considered.
- **Inconsistency between shipped and debug versions.** If two versions, one with assertions and one without assertions, are maintained, then inconsistency can be avoided by using, e.g., the C preprocessor to conditionally include the assertions [20].
- **Side effects.** "Assert should not disturb memory, initialize data that would otherwise be uninitialized, or cause any other side effects." Note that avoiding side effects sometimes means forgoing assertions that could be included, but that would involve side effects, in the interest of simplicity. The Visual C++ documentation gives as an example that `ASSERT (numMols++ > 0);` changes the value of `numMols`. In addition, the documentations points out that debug and release versions will have different results because the side effects occur only in the debug version (with assertions enabled). Side effects may also lead to unpredicted results in cases in which some assertions are deleted for optimization reasons, while others remain enabled.

- Not for error conditions. This is, in fact, an area of some controversy. Maguire, for example, recommends that assertions never be used to test for possible error conditions. Assertions are for illegal preconditions that should never occur if the program is working correctly [20]. The latter is appropriately tested via an assertion; the former requires error handling of conditions that may arise during operations. Anna, on the other hand, identifies exceptional states by means of assertions in the code which, if not satisfied, raise exceptions. Wilson uses a broad definition of assertions that includes exception handling. He recommends “self-correcting” assertions where the user never needs know that an error occurred [37]. Maguire points out that, at least for debugging, the tester always wants to know that an anomaly occurred, even if the code recovers gracefully from the anomaly.

8 Future Work

Future work will focus on how to identify assertions that can enhance software safety. The application domain will be the software under development for the Deep Space 1 spacecraft on the New Millennium Project. Leveson indicates that the leaf nodes of a software fault tree analysis are appropriate sources for assertions [15]. Assertions can be used to confirm that the conditions under which failure recovery software should be initiated actually exist. In addition, assertions can identify some hazardous software states (for example, situations in which control decisions will be made based upon out-of-range values). Adding appropriate assertions for runtime checking during testing of safety-critical conditions may reduce hazards in maintained software.

Another possible source of assertions is the formal specifications of the requirements or design in terms of preconditions and postconditions. A requirements analysis tool from the Naval Research Laboratory, SCR*, contains an assertion dictionary that allows additional claims about the required relationships among the software’s variables to be stated [10]. Finally, flight rules often exist to preclude reaching hazardous states that have been identified during the development process. It may be possible to extract useful assertions from the flight rules that can be incorporated into the code to check for these hazardous states.

The new EDCS (Evolutionary Design of Complex Software) initiative [31] can be expected to generate additional research and applications in the area of maintaining critical systems. In particular, the work by Richard Julig on “Evolution Based on Precise Semantic Design Records,” by Nancy Leveson and Francesmary Modugno on “Specification and Design Support for Software Evolution,” and by Leon J. Osterweil, Lori Clarke, Debra Richardson, and Michael Young on “Perpetual Testing” may provide additional results that relate to our topic.

9 Conclusion

Assertions are a powerful tool for abstraction and for preservation of information internal to the source code. They can assist in maintaining code consistency with the safety design by detecting some unintended approaches to hazardous states. Assertions have demonstrated

their value most clearly in preventing the corruption of interfaces by checking that the precondition/postcondition contracts among the software components continue to be satisfied. In this, they provide a ready match with object oriented development.

Acknowledgments

The first author thanks Martin S. Feather for his helpful comments on an earlier draft of portions of this paper.

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial or noncommercial product, process, or service by name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government, the Jet Propulsion Laboratory, or the California Institute of Technology.

References

- [1] R. Abi, "Software Maintenance: tools and techniques", *System Development*, Vol. 8, no. 8 pp., 3-6, 1988
- [2] Samuel Ajila. "Software Maintenance: An Approach to Impact Analysis of Objects Change." *Software Practice and Experience*, Vol. 25(10), October 1995. pp. 1155-1181.
- [3] Jonathan P. Bowen, Peter T. Breuer, and Kevin C. Lano, "A Compendium of Formal Techniques for Software Maintenance," to appear in the *BCS/IEEE Software Engineering Journal*.
- [4] Electris Software Ltd. <http://www.electris.com/>
- [5] Steve Fickas and Martin S. Feather, "Requirements Monitoring in Dynamic Environments," *Proceedings of the 2nd IEEE International Symposium in Requirements Engineering*, IEEE Computer Science Press, 1995, pp. 140-147.
- [6] *Formal Methods Specification and Verification Guidebook for Software and Computer Systems, vol. II*, NASA Office of Safety and Mission Assurance, draft copy, Oct., 1996.
- [7] Gerald C. Gannod and Betty H. C. Cheng, "Strongest Postcondition Semantics as the Formal Basis for Reverse Engineering," *Automated Software Engineering*, 3, 139-164 (1996).
- [8] J. M. Hart. "Experience with Logical Code Analysis in Software," *Software Practice and Experience*, Vol. 25(11), November 1995, pp. 1243-1262.
- [9] Mark R. Headington and David D. Riley, *Data Abstractions and Structures Using C++*, D. C. Heath and Company, 1994.

- [10] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw, "SCR*: A Toolset for Specifying and Analyzing Requirements," *COMPASS 95*, June 25-29, 1995, Gaithersburg, MD, pp. 109-122.
- [11] Alan Joch, "How Software Doesn't Work: Nine Ways to Make Your Code More Reliable," *Byte*, Dec 1995, pp. 49-60.
- [12] Cliff B. Jones, *Systematic Software Development Using VDM*, Prentice Hall, 1990.
- [13] Kevin Lano and Howard Haughton, *Reverse Engineering and Software Maintenance, A Practical Approach*. McGraw-Hill International, 1994.
- [14] Gary T. Leavens, "An Overview of Larch/C++: Behavioral Specifications for C++ Modules," *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, ed. Haim Kilov and William Harvey, Kluwer Academic Publishers, 1996, pp. 121-142.
- [15] Nancy G. Leveson, *Safeware, System Safety and Computers*, Addison-Wesley, 1995.
- [16] N. Leveson. "Software Safety." Software Engineering RICIS Symposium, California University, Irvine, July, 1987.
- [17] Panos E. Livadas and Stephen Croll, "System Dependence Graphs Based on Parse Trees and their Use in Software Maintenance," U of Florida CIS Dept.
- [18] David Luckham, *Programming With Specifications, An Introduction to Anna, A Language for Specifying Ada Programs*, Springer-Verlag, 1990.
- [19] Robyn Lutz, "Targeting Safety-Related Errors During Software Requirements Analysis," *The Journal of Systems and Software*, vol. 34, Sept., 1996, pp. 223-230.
- [20] Steve Maguire, *Writing Solid Code*, Microsoft Press, 1993.
- [21] Todd Montgomery, Software Research Lab Testing Toolkit (<http://research.ivv.nasa.gov/projects/SRLT>), 3/21/96.
- [22] Pure Atria. <http://www.pureatria.com/textonly/index.html>
- [23] C. F. Radley, "Software Safety Progress in NASA (Final Report)." Presented at the Safety Through Quality Conference, Oct. 1995.
- [24] David S. Roseblum, "A Practical Approach to Programming With Assertions," *IEEE Transactions on Software Engineering*, Vol. 21, No. 1, January 1995, pp. 19-31.
- [25] Martina Schollmeyer, Hanan Lutfiyya and Bruce McMillin, "An Algorithm for Generating Executable Assertions for Fault Tolerance," CSC-92-01, Dept. of Computer Science, Univ. of MO at Rolla, March 21, 1992.
- [26] Ian Sommerville, *Software Engineering*, 5th ed., Addison-Wesley, 1996.

- [27] D. Sparkman, "Techniques, Processes, and Measures for Software Safety and Reliability," May 30, 1992, Version 3.0. Lawrence Livermore National Lab..
- [28] J. Michael Spivey, *The Z Notation: A Reference Manual*, Prentice Hall, 1992.
- [29] Guy L. Steele, Jr. *Common Lisp*. Digital, 1990.
- [30] Bjarne Stroustrup. *The C++ Programming Language*, 2nd edition. Addison Wesley, 1991.
- [31] Will Tracz, "Evolutionary Design of Complex Software (ECDS) Kick Off Workshop Summary," *Software Engineering Notes*, vol. 21, no. 5, Spet 1996, pp. 40-42.
- [32] Jeffrey M. Voas and Keith W. Miller, "Putting Assertions in Their Place," em Proceedings Fifth International Symposium on Software Reliability Engineering, Nov 1994, pp. 152 - 157.
- [33] Wei-lun Kao, Ravishankar K. Iyer, Dong Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults," *IEEE Transactions on Software Engineering*, Vol. 19, No. 11, November 1993, pp. 1105-1118
- [34] M. S. Wetherholt, C. F. Radley, "Implementing Software Safety in the NASA Environment," National Aeronautics and Space Administration, Lewis Research Center. Presented at the Safety Through Quality Conference, June 1994.
- [35] Richard Wiener, *Software Development Using Eiffel*, Prentice Hall PTR, 1995.
- [36] Tom Williams, "Tools to test and verify code running in embedded systems, *Computer Design*, v 34 n12 Dec 1995, pp. 54-56.
- [37] Rodney C. Wilson, *Software Rx, Secrets of Engineering Quality Software*, Prentice-Hall, 1997.