

New Challenges in Model Checking

Gerard J. Holzmann, Rajeev Joshi, and Alex Groce

NASA/JPL Laboratory for Reliable Software, Pasadena, CA 91109, USA
{Gerard.Holzmann, Rajeev.Joshi, Alex.Groce}@jpl.nasa.gov

Abstract. In the last 25 years, the notion of performing software verification with logic model checking techniques has evolved from intellectual curiosity to accepted technology with significant potential for broad practical application. In this paper we look back at the main steps in this evolution and illustrate how the challenges have changed over the years, as we sharpened our theories and tools. Next we discuss a typical challenge in software verification that we face today – and that perhaps we can look back on in another 25 years as having inspired the next logical step towards a broader integration of model checking into the software development process.

Keywords. Logic model checking, software verification, software reliability, software structure, grand challenge project, flashfile system challenge.

1 Introduction

The idea to build a practically useful tool to check the correctness of program code quite possibly already occurred to the first people who attempted to write code. Not by coincidence, many of those people were mathematicians. Goldstein and Von Neumann took a first step in 1947 when they introduced the notion of an assertion in program design [7] (see also [5]):

For this reason we will denote each area in which the validity of such limitations is being asserted, by a special box, which we call an 'assertion box.'

A series of foundational papers on program analysis and program verification techniques appeared in the sixties and seventies, including seminal work by Robert Floyd, Tony Hoare, and Edsger Dijkstra, that we will not attempt to summarize here. More closely related to the topic of this paper and the theme of this symposium is work that started in the late seventies on model based verification techniques. Among the earliest models used for this purpose were Petri nets and finite automata, initially paired with manual analysis procedures (e.g., [3]). Carl Sunshine described a basic reachability analysis method for automata models in 1975 [20], a variant of which was applied in a verification tool built by West and Zafropulo at IBM [24]. The latter tool attracted attention by uncovering relatively simple defects in trusted international standards for data communication. A favored example of an automaton model from this period was also the simple alternating bit protocol [2]. It is in retrospect remarkable that some of the early verification tools could not yet handle the complexity of this very basic protocol.

Work on what later became the SPIN model checker started in 1980 at Bell Laboratories. This first tool, named *pan*, was also based on an optimized reachability analysis procedure, though initially supported by an algebraic specification formalism [8]. Like the IBM tool, this tool attracted attention within AT&T by successfully uncovering defects in models of trusted telephony software. Pan was restricted to the verification of safety properties, and was therefore not a true *logic* model checking system as intended in [4]. The restriction to safety properties did however allow us to verify models with up to millions of reachable states, although the latter could take a good week of computation on the fastest available hardware at that time. We did not contemplate an extension of the verification system to properties specified in linear temporal logic, and more broadly the set of omega-regular properties, until the late eighties, when available compute power had increased, and our verification techniques had sharpened.

Looking back, we can recognize some patterns in what we considered to be the main obstacles to a broader application of model checking techniques to problems in *software* verification. As obstacles were overcome, new challenges were identified and targeted. The following list sketches some of the deciding issues that influenced the evolution of the SPIN model checker.

1. *Specification formalisms*: The initial challenge, in a period that we can indicate very approximately as 1975-1985, was to find a usable formalism for constructing *models* with verifiable properties. The focus in this period was on the identification of specification formalisms that could facilitate analysis. Ultimately, automata-based models were found to provide the most solid foundation, and much work has since been focused on them. *Trace*, the successor to *pan* and the next step in the evolution of SPIN, dropped *pan*'s process algebra specification formalism in favor of automata models in 1983, leading the way for SPIN to easily conform to the automata-theoretic foundation from [22].
2. *Efficient Algorithms*: The next challenge, between approximately 1985 and 1995, was on developing new data structures and *algorithms* that could improve the range and efficiency of model checking systems. This development produced BDD-based and symbolic verification methods, as well as the partial order reduction methods that are at the core of model checking systems today. Partial order reduction was integrated into SPIN in the early nineties [9].
3. *Model Extraction from Code*: The third challenge, between 1995 and 2005, was to find ways to apply model checking techniques more directly to implementation level code, using software abstraction and *model extraction* techniques. This work led to the extension of the SPIN model checker with support for embedded software in abstract models. This change enabled the application of SPIN to the verification of unmodified, implementation level software for call processing in a commercial voice and data switch, and as such perhaps the first application of formal software verification at this scale

[10]. Similarly, this third challenge led to the successes at Microsoft in the formal verification of device driver code [19], and the work at Stanford on the CMC model checker [17].

4. *Today's Challenge*: This brings us to the next, and current, challenge for work that may well turn out to define the primary emphasis for our work in logic model checking for the period 2005 to 2015. This fourth challenge is to find effective ways to *structure software* such that formal verification techniques, and especially logic model checking techniques, become simpler to use and more effective in identifying potential violations of correctness properties in executable code.

We will devote the remainder of this paper to a description of this new challenge.

2 The New Challenge

The “Grand Challenge in Verification” recently posed by Sir Tony Hoare [23], prompted us to propose a mini-version, which is to design and implement a verifiable file system for non-volatile memory [14]. This mini-challenge was of course not chosen arbitrarily. Space exploration missions need a reliable capability to record data that is either received from earth (e.g., commands and parameters), or to be returned to earth (e.g., telemetry and images). Often a spacecraft is temporarily pointed away from earth to capture an image or take a measurement. The data can only be returned later, sometimes much later, when communication with the Deep Space Network on earth is restored.

The MER Rovers that currently explore the surface of Mars, for instance, use flash memory cards to store critical data. The reliability of hardware components can often be increased by adding nominally redundant backups. The flash memory cards used on spacecraft are special radiation-hardened designs that can be duplicated for redundancy if needed. For software, though, increasing reliability is not nearly as simple to achieve, and a number of mission anomalies related to data storage on flash memory cards can be traced back to software flaws. Curiously, the software used for the management of flash memory cards in missions to date has consisted of off-the-shelf code that was designed and built primarily for use in cameras and home computers, but not for reliable operation in space, resisting hardware failures, power-loss, and sudden reboots. What makes failures in this software so difficult to accept is that a file system is easily one of the best understood modules on a spacecraft in terms of its required functionality. It should be possible to design an ultra-reliable version of this type of software. These observations provided the motivation behind our mini-version of the grand challenge. The real challenge, though, is somewhat broader:

Is there a way to structure software in such a way that the application of logic model checking techniques becomes a trivial exercise?

It is of course all too easy to pose a challenge problem and wait for others to solve it. We have therefore decided take our own challenge and to pursue a full design, a full

verification, *and* a complete implementation of a flight-qualified file system module that can withstand the rigors of space. We have also committed to building the module to standards that satisfy all existing flight software development requirements at JPL. This decision rules out a number of choices for the design and development that otherwise might have been possible. It means, for instance, that the target programming language is most conveniently C (the language most commonly used at JPL for implementing mission critical software), the target operating system VxWorks[®] (a real-time operating system), and the process followed must comply with all reporting and book-keeping requirements for software development at our host institution. Naturally, our desire is to not just comply with the existing process, but to show how it can be exceeded. Our goal is further to chart a course for reliable software development that can later plausibly be followed by non-experts in formal software verification.

At the time of writing, we have completed a first implementation of the file system software that we will use as a reference for our formal verification attempts. The prototype is written to a high standard of reliability, compliant with all JPL coding requirements, as well as conforming to a small set of fairly strict additional coding rules, described in [15]. These additional rules are in part meant to simplify, if not enable, formal verification with logic model checking techniques.

3 Our Plan

We started on our mini challenge in the middle of 2005, initially pursuing three tracks in parallel.

1. The first track is to build a simulation environment for a file system that can reproduce all relevant behavior of the target hardware. A software simulation of the environment will simplify the use of model-driven verification techniques, as outlined in [12]. We have meanwhile completed several versions of this hardware simulation layer, supporting different levels of abstraction. The most accurate simulation module supports a bit-level accurate representation of a typical flash memory card.
2. The second track is to develop a formalization of all relevant requirements, including standard POSIX requirements for the user interface to the file system [18]. As always with requirements specifications, identifying and capturing a representative set of requirements is a non-trivial task. The primary requirements for file systems, for instance, are functional and not temporal in nature, and there are few if any adequate formalisms available for expressing such requirements. We currently plan to capture most requirements of this type as system invariants, and as pre- and post-conditions on basic file system operations.
3. The third track includes the detailed design of the file system itself. This is in principle a white-board design, aspects of which are currently being verified with the SPIN model checker [13] and with the ACL2 theorem-prover [1].

3.1 Constraints

To give a flavor for the design requirements, note that flash-memory is typically logically organized into separately mountable file system partitions (sometimes called volumes), but physically they are organized in pages, blocks, and banks. On a typical NAND flash memory card there may be 2 banks, 1024 blocks per bank, and 32 pages per block, each page able to record 4096 bytes of information.

Pages on NAND-flash devices must always be written in their entirety, in one operation. The information can be read back in portions, but only sequentially and not randomly. A page can be read any number of times without degrading the information that is stored in it, but it should be written only once. After a page has been written, it should be erased before it is reused for new write operations. A page, of course, holds no useful information until it is written. The reason for the single-write requirement is that a page erase operation on flash memory sets all bits on the page to one, and subsequent write operations can only set bits to zero. Once a bit is zero, it can only be reset to one in an erase operation.

To make things more interesting still, pages on a flash memory card can only be erased in multiples of blocks (i.e., 32 pages at a time). This means that there are in principle only three types of operations that can be performed on a flash disk: *read a page*, *write a page*, and *erase a block* of pages. A block, finally, can only be erased a limited number of times, e.g., 100,000 times, and the reliability of the pages in a block degrades with the number of erasures that have taken place. Since we don't want some blocks to wear out long before others, blocks have to be erased and reused in such a way that the wear on all blocks is roughly the same. This process is called *wear-leveling*. Page read- and write-operations, and block erase operations can fail, sometimes intermittently, sometimes permanently. On such a failure, a block may have to be marked as *bad*, to indicate that no further write or erase operations should be attempted on that block. When a block goes bad, the pages in that block can no longer be written or erased, although any correctly written page in the block may still be read.

A first observation about the target design for our file system is that no information can be stored in a fixed location on disk, not even information that is unlikely to change. The wear-leveling requirement means that all stored data may have to move from time to time, so that all blocks can be erased and reused roughly equally. Since stored data is the only information that will survive a reboot, it must also be possible to reconstruct all relevant information about the file system from scratch, without knowing in advance where it is stored on disk. Another requirement, orthogonal to the wear-leveling requirement, is that the consistency of the file system must be maintained at all times, even in the presence of arbitrary reboots or a sudden loss of power. That means that all operations on the file system must be interruptible. No data should be lost or corrupted when the system is interrupted at a random point in its execution. A user of the file system must be able to assume that changes in the stored data are atomic, even if they take multiple page writes to complete. A strong design requirement is that, with very few exceptions, file system operations must either

succeed completely or fail completely, never leaving visible evidence of intermediate states when interrupted.

In the target environment, the file system should also be able to deal with random data errors caused by radiation, which can be particularly severe during solar flares. To give one small example of the problems that this can pose: a so-called Single Event Upset (SEU) in the address register of a flash memory device during a *read_page* operation could alter the page address and result in the wrong page being read, without an error condition being flagged (i.e., the data read from the page can pass a checksum test successfully). The same type of error during a *write_page* operation could result in a page different than the one intended being written, again without detectable error unless special precautions are taken in the design that is adopted.

3.2 Verification Challenges

A key challenge in this project is to provide the ability to prove the integrity of the file system under all types of hardware error, and power-loss scenarios. A model of the flash hardware can capture the relevant assumptions about the lower interface to the hardware, and a model of nominal user behavior can capture our assumptions about the upper interface. This leads to a *sandwich* model of the file system: enclosed between two SPIN models that define the environment in which it is meant to operate and against which it must be verified, as illustrated in Figure 1.

The user behavior, though conceptually simple, can add a surprising amount of complexity. Note that for even very small systems, there are a very large number of possible ways to define directory hierarchies, file contents, and file and directory names. Rename operations can move files arbitrarily between different locations in the directory hierarchy, and seek operations can change where new contents are written to or read from in a file. Files may be truncated, moved, removed, recreated, etc. This means that it is not simple to define a single user model and hope to perform an exhaustive verification against that model. To alleviate some of these problems, different levels of abstraction and different subsets of possible user behaviors can be defined to perform a series of targeted verification runs against a relevant subset of the correctness properties.

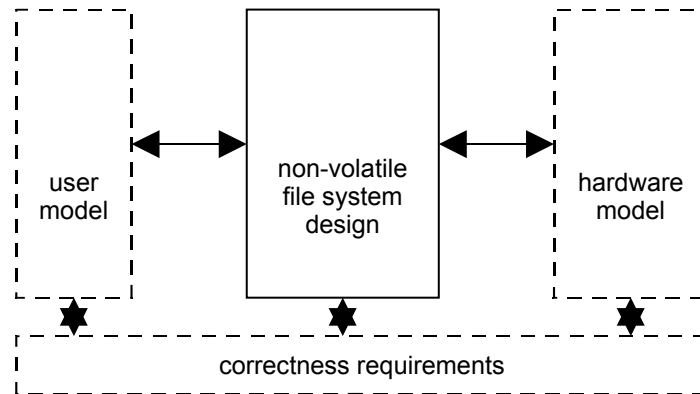


Fig. 1. Sandwich Model for File System Verification

A simple model of user behavior could, for instance, specify the manipulation of a single file by a user, performing random reads and writes to that file, resetting the file pointer to the start of the file at arbitrary points, while the hardware model fails read and write operations arbitrarily. We have been able to demonstrate that such a model can indeed be verified exhaustively, when applied to the prototype software implementation, using model driven verification techniques and fairly straightforward conservative abstractions of the file system state.

A simple version of an abstract model for the flash hardware is shown in Figure 2. This model captures only basic behavior of the hardware, allowing page reads and writes, with the possibility of failure and the sudden appearance of bad blocks, and allowing for a distinction between meta-data (directory information) and regular file data. Meta-data is typically stored with a verified write cycle, which is slower than regular write operations, but lowers the probability of subsequent page read errors. In a verified write operation the information is read back and compared with the original data to make sure it can be retrieved correctly.

A sample correctness property for the file system is that no page is written more than once before it is erased. This property could be expressed in LTL as follows:

$$\square (pw \rightarrow X (\neg pw \cup be))$$

where pw indicates the occurrence of write operation on an arbitrary given page, and be indicates a block erase operation on the block that contains the given page.

A more complete model of the flash hardware will also record page header information; so that one can track which pages are current and which are obsolete and erasable, in the verification of the properties of the file system software. With that

model we should also be able to prove that a block with only obsolete pages and no free pages will eventually be erased for reuse, for instance expressed as:

$$\square (ob \rightarrow \diamond be)$$

where *ob* is true when some given block contains only obsolete pages, and *be* is true when that block is erased.

```

active proctype flash_disk()
{
    byte b, p;
    bool v;

    do
        :: flash?readpage(b,p,_ ) ->
            assert(b < NBL);
            assert(p < PPB);
            if /* non-deterministic choice */
                :: blocks[b].meta[p] != free ->
                    user!success
                :: blocks[b].meta[p] != verified ->
                    user!error
            fi

        :: flash?writepage(b,p,v) ->
            assert(b < NBL);
            assert(p < PPB);
            assert(blocks[b].meta[p] == free);
            if
                :: blocks[b].bad -> user!error
                :: else ->
                    if
                        :: blocks[b].meta[p] = v;
                            user!success
                        :: blocks[b].bad = true ->
                            user!error
                    fi
            fi

        :: flash?eraseblock(b,_,_) ->
            assert(b < NBL);
            if
                :: blocks[b].bad -> user!error
                :: else ->
                    if
                        :: blocks[b].bad = true ->
                            user!error
                        :: erase_pages(b) ->
                            user!success
                    fi
            fi
    od
}

```

Fig. 2. Simplified Model of the Flash Hardware.

4 On Code Structure

One focus of our project is to study if the adoption of specific code and data structures can enable stronger types of verification, and make it easier to apply existing software verification methods. Clearly, code can be written in such a way that most properties of interest become unprovable. It is unfortunately easier to demonstrate this point than it is to show that the opposite is also possible. To achieve the opposite, to write code in such a way that it *can* be verified, takes more planning, but the additional level of effort required may still be relatively small. We adopted strong coding rules for our project, that include a restricted use of pointers, statically verifiable bounds on all loops, absence of dynamic memory allocation, and even the absence of direct and indirect recursion [15].

A loop to traverse a linked list, for instance, can be bounded as follows:

```
SET_BOUND(MAX);
for (ptr = start; ptr != null; ptr = ptr->nxt)
{
    ...
    CHECK_BOUND();
}
```

Where `SET_BOUND` and `CHECK_BOUND` are macros. The first macro call will initialize a predefined loop variable to the boundary value that should never be exceeded. The second macro call decrements the variable value and asserts that the result remains positive. The protection here is against infinite loops, so the precise bound is often not that important, as long as it is a finite number. (Nested loops should be rare in high integrity code, and are handled separately since they will require us to track more than one loop bound.)

Assertions are also handled differently. The standard assertion definition from the `C assert.h` library is not really adequate for our purposes. An abort on assertion failure is rarely the right response in embedded systems, and in most, if not all, cases processing should also *not* continue normally when an assertion fails. Instead, some type of corrective or recovery action should be initiated to handle the unexpected situation. We therefore define an assertion as a Boolean pseudo-function that normally returns *true*, but will optionally print a diagnostic message when it fails and then return *false*, so that the caller of the assertion can take the required corrective action. With some C preprocessor magic, this can be written as follows:

```
#define ASSERT(e) ((e) ? (1) : \
    output("%s:%d assert(%s) failed\n",
    __FILE__, __LINE__, #e), 0)
```

The macro definition makes use of the predefined preprocessor names `__FILE__` and `__LINE__` to print the location of the assertion in the source files, and of the C preprocessor operator `#` to reproduce the text of the failing assertion. Assertions, then are *always* used as expressions in a conditional and *never* as standalone statements.

The use of an assertion to defend against a null-pointer dereference could for instance be written as follows:

```
if (!ASSERT(ptr != NULL))
{
    return ERROR;
}
```

This forces the programmer to think about the corrective action that would be needed in case the assertion fails.

For embedded code, where there is typically no mechanism for printing output, the assertion can be redefined *after* testing with the variant:

```
#define ASSERT(e) (e)
```

which maintains the protection and the original functionality of the assertion (so that all test results remain valid), but removes the diagnostic output on failure.

The rules we adopted allow us to derive bounds on memory and stack use, and to prove the finiteness of all file system operations. Without recursion, the function call graph is acyclic and can be analyzed with traditional logic model checking techniques. We use, for instance, the *uno* static analyzer [11,21] to generate the function call graph for the software, and convert it with a small awk-script into a SPIN model, that is automatically annotated with relevant operations (e.g., semaphore operations). Then we use the SPIN model checker to prove additional properties of the code such as proper locking orders, bounds on stack use, and absence of direct or indirect recursion.

Importantly, the data structures for the prototype file system are organized in such a way that it becomes easy to set up a connection with the model checker for model-driven verification runs. Just two data structures (a mount table and a partition structure) hold all state information that must be tracked in the model checking process, and the required tracking statements are trivially defined – possibly even mechanically derivable from the source code. It is also relatively straightforward, thanks to these structuring conventions, to set up abstraction functions for the model checker over the relevant state data, that will allow us to exploit, for instance, symmetry abstractions. We further make sure that the level of atomicity in the source code, enforced through VxWorks semaphores, matches the level of granularity that the model driven verification method handles best (i.e., function level atomicity).

5 Testing?

To perform an initial check of the working of our prototype implementation, we have relied on a number of methods that include strong static source code analysis, and randomized differential testing [16]. These more conventional testing methods serve not only for us as designers to gain confidence in our initial prototype, they also help to win the trust of colleagues who may need to be convinced of the added value of a more formal verification effort of the same code. We have not attempted to generate

implementation level code from high level design models (e.g., Spin models), although we may revisit that decision later once the full design and its verification challenges are thoroughly understood.

A few more words on the random differential testing method we used may be of interest. For these tests, one or more reference systems are needed to serve as a judge of the validity of operations that are performed on the system under test. Fortunately, for a standard file system, reference implementations are readily available. As part of our tests, we ran randomly generated usage scenarios on the flash file system, comparing against reference file systems on Solaris, Linux, and Cygwin. Perhaps not surprisingly, we found defects not only in our own prototype software but also in some of the widely-used reference implementations. The Linux implementation proved to be the most reliable, and was used for the majority of our tests. A test fails in this setup when the file system created by the module under test differs from that created on the reference system when given the same sequence of POSIX operations. Each such test failure is inspected manually and in cases of doubt the official POSIX requirements are consulted to determine which implementation is at fault: the prototype file system or the reference system. The test harness used in these tests randomly injects simulated hardware faults, such as bad blocks and sudden reboots. Because these faults cannot easily be reproduced on the reference system, our integrity requirement was that the module under test either completes an operation fully, matching the result of the completed operation on the reference system, or fails the operation completely, matching the state of the reference system before the operation is executed, but never creating an intermediate or a corrupted state. Long error scenarios found through this method are minimized with a method based on Zeller's delta-debugging system [6] and all error scenarios found are preserved in a regression test suite.

The differential test method is easily automated. The random tests run in principle non-stop in the background on our machines. Even at that pace, there is of course no hope that these tests can be exhaustive, yet our experience so far is that new defects in the code are found quickly.

There is an opportunity to replace at least part of the differential test system with a stronger Spin driven verification system, using models along the lines of what is shown in Figure 2, where we replace the random choices from the tester with non-deterministic choices that are controlled by the model checker. Our real challenge here is to make the model driven software verification methods work as easily, and be as effective as the original differential test method.

A switch to the model driven verification method will allow us to formulate and verify more complicated correctness properties in linear temporal logic, to perform the verifications more systematically, and has the potential for a significantly greater accuracy in catching requirements violations.

6 Summary

In this paper we have posed a challenge that illustrates where we believe the new frontier in the application of logic model checking techniques rests today. To realize the full potential of logic model checking techniques for software verification, we will need to find ways to structure code in such a way that verification becomes easier. We believe that the main potential in this area is in the application of model-driven verification techniques. To give substance to these ideas, we have described a specific challenge problem that we, and we hope many others, will try to solve fully in the coming years. The domain in which we have phrased this problem is that of space exploration, but it could be any other application domain where the correctness of software is critically important. Like many organizations, JPL and NASA already have strict requirements for the development of mission-critical software components. Still, anomalies that can be traced to software defects do occur, and have on occasion led to mission failures. Stronger types of software verification are therefore essential to reach higher levels of software reliability.

JPL today lists among its strategic goals in software development the adoption of formal methods for software design, an increased use of model-driven software verification techniques, and routine application of logic model checking techniques to mission software by the year 2013. The work we have sketched is an attempt to realize at least some of these goals.

Acknowledgements. We are grateful to Len Day for building the bit-level accurate software simulator for flash hardware, and to Cheng Hu for his initial formalization of the POSIX requirements for our file system.

The research 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.

References

- [1] M. Kaufmann, P. Manolios, and J Strother Moore, *Computer-Aided Reasoning: An Approach*, <http://www.cs.utexas.edu/users/moore/publications/acl2-books/car/index.html>. Kluwer Academic Publishers, July 2000.
- [2] K.A. Bartlett, R.A. Scantlebury, P.T. Wilkinson, A note on reliable full-duplex transmission over half-duplex lines. *Comm. ACM*, Vol. 12, No. 5, pp. 260-265.
- [3] G.V. Bochmann, Finite state description of communication protocols, Publ. 236, Dept. d'Informatique, University of Montreal, July 1976.
- [4] E.M. Clarke, E.A. Emerson, and A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, *ACM Trans. On Programming Languages and Systems*, Vol. 8, No. 2, April 1986, pp. 244-263.
- [5] L.A. Clarke, D.S. Rosenblum, A historical perspective on runtime assertion checking in software development, *ACM SIGSOFT Software Engineering Notes*, Vol. 31, Issue 3, May 2006, pp. 25-37.

- [6] <http://www.st.cs.uni-sb.de/dd/> on delta debugging techniques.
- [7] Herman H. Goldstein and John Von Neumann, *Planning and coding problems for an electronic computing instrument*. Part II, Volume 1. Princeton, April 1947.
In: John von Neumann, *Collected Works*, Vol. V, p. 92. *Design of computers, theory of automata and numerical analysis*. Ed. A.H. Taub. Pergamon Press, New York, 1963.
- [8] G.J. Holzmann, *PAN: a protocol specification analyzer*, Tech Report TM81-11271-5, AT&T Bell Laboratories, March 1981.
- [9] G.J. Holzmann and D. Peled, An improvement in formal verification. *Proc. 7th Int. Conf. on Formal Description Techniques (FORTE)* 1994, pp. 197-211. Chapman & Hall.
- [10] G.J. Holzmann, and M.H. Smith, Automating software feature verification, *Bell Labs Technical Journal*, Vol. 5, No. 2, pp. 72-87, April-June 2000.
- [11] G.J. Holzmann, Static source code checking for user-defined properties, *Proc. 6th World Conference on Integrated Design & Process Technology (IDPT)*, Pasadena CA USA, June 2002.
- [12] G.J. Holzmann, R. Joshi, Model-driven software verification, *Proc. 11th SPIN Workshop*, Barcelona, Spain, April 2004, Springer-Verlag, LNCS 2989, pp. 77-92.
- [13] G.J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, 2003.
- [14] G.J. Holzmann, R. Joshi, "A mini grand challenge: build a verifiable file-system" (position paper), *Grand Challenge in Verified Software – Theories, Tools, Experiments*, Zurich, Switzerland, October 2005.
- [15] G.J. Holzmann, The Power of Ten: rules for developing safety critical code, *IEEE Computer*, June 2006.
- [16] W.M. McKeeman, Differential testing for software, *Digital Technical Journal*, Vol. 10, No. 1, pp. 100-107, December 1998.
- [17] M. Musuvathi, D.Y.W. Park, A. Chou, D.R. Engler, D.L. Dill, CMC: A pragmatic approach to model checking real code, *Proc. Fifth Symposium on Operating Systems Design and Implementation*, OSDI, Dec. 2002.
- [18] <http://www.opengroup.org/>
- [19] Ball, S.K. Rajamani, Automatically Validating Temporal Safety Properties of Interfaces, *SPIN 2001 Workshop on Model Checking Software*, LNCS 2057, May 2001, pp. 103-122.
- [20] C.A. Sunshine, C.A., *Interprocess Communication Protocols for Computer Networks*, Ph.D. Thesis 1975, Dept. of Computer Science, Stanford Univ., Stanford, CA.
- [21] <http://spinroot.com/uno/>
- [22] M. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification, *Proc. 1st Annual Symposium on Logic in Computer Science*, LICS 1986, pp. 332-344.

[23] *Grand Challenge in Verified Software – Theories, Tools, Experiments (VSTTE)*, Zurich, Switzerland, October 2005, <http://vstte.ethz.ch/>

[24] C.H. West, and P. Zafiropulo, Automated validation of a communications protocol: the CCITT X.21 recommendation, *IBM J. Res. Develop.*, 1978, Vol. 22, No. 1, pp. 60-71.