Using SPIN Model Checking for Flight Software Verification[1]

Peter R. Glück
NASA Jet Propulsion Laboratory
Pasadena, CA 91109
peter.r.gluck@jpl.nasa.gov

Dr. Gerard J. Holzmann
Computing Science Research
Bell Labs, Murray Hill, NJ 07974
gerard@research.bell-labs.com

Abstract—Flight software is the central nervous system of modern spacecraft. Verifying spacecraft flight software to assure that it operates correctly and safely is presently an intensive and costly process. A multitude of scenarios and tests must be devised, executed and reviewed to provide reasonable confidence that the software will perform as intended and not endanger the spacecraft. Undetected software defects on spacecraft and launch vehicles have caused embarrassing and costly failures in recent years.

Model checking is a technique for software verification that can detect concurrency defects that are otherwise difficult to discover. Within appropriate constraints, a model checker can perform an exhaustive state-space search on a software design or implementation and alert the implementing organization to potential design deficiencies. Unfortunately, model checking of large software systems requires an often-too-substantial effort in developing and maintaining the software functional models.

A recent development in this area, however, promises to enable software-implementing organizations to take advantage of the usefulness of model checking without hand-built functional models. This development is the appearance of "model extractors." A model extractor permits the automated and repeated testing of code as built rather than of separate design models. This allows model checking to be used without the overhead and perils involved in maintaining separate models.

We have attempted to apply model checking to legacy flight software from NASA's Deep Space One (DS1) mission. This software was implemented in 'C' and contained some known defects at launch that are detectable with a model checker. We will describe the model checking process, the tools used, and the methods and conditions necessary to successfully perform model checking on the DS1 flight software.

1. INTRODUCTION

The software applications we are considering in this paper are fundamentally concurrent in nature. Many threads of execution can be active at the same time. When these threads share the same CPU, their executions are interleaved in time. The particulars of the thread interleavings depend on the thread scheduler, and are not predetermined for each run. If all threads perform independent computations the final result of all interleavings will be the same. But this is not necessarily the case if the threads can interact. The behavior of each thread of execution can now depend subtly on the relative timing of events in the system, which is in part determined by the indeterminate thread interleavings. Reliably testing a multi-threaded concurrent system can therefore be exceptionally difficult with traditional means. Two problems prevent the tester from doing a good job in these cases. The first problem is the limited *controllability* in a concurrent system: the tester cannot control the specifics of thread interleavings. The second problem is the limited *observability* of events: when an error is detected it can be very hard to reconstruct the sequence of events that preceded it, to identify the root cause.

Observations such as these have inspired work on the construction of logic model checkers that can be used for a more thorough analysis of concurrent systems behavior. By constructing a model of a system, we can gain complete control over all salient aspects of its execution, and perform a more thorough analysis than otherwise possible. There is a fairly long history of the construction of such model checking systems. In 1978, for instance, Jan Hajek built a system called 'Approver' [Hajek78] that could analyze simple models of data transfer protocols with a heuristic method based on a reachability analysis of the underlying control graphs (i.e., a systematic exploration of the reachable states of the graph). Similarly, the earliest direct predecessor of the SPIN model checking system that we discuss in more detail in this paper dates from 1980 [H81].

[1] 0-7803-7231-X/01/$10.00/© 2002 IEEE
[1] IEEEAC paper #435, Updated Oct 12, 2001

The power of model checking systems based on reachability analysis was extended significantly by two important developments. The first of these was the development of dedicated logics that allow one to effectively reason about the correctness of systems that evolve over time. Several such logics have been developed. The most important one for our work is the *linear temporal logic* (LTL) that was first introduced by Amir Pnueli [P77]. Initially, this logic was only used for pen and paper proofs of relatively small concurrent systems. The second development was the discovery that temporal logic formulae can be translated, even mechanically, into small test drivers that can be used inside a model checking system to automate correctness proofs [VW86], [GPVW95], [EH00].

The theory of logic model checking is by now fairly well established e.g., [CGP00], [H97]. In Section 2 of this paper we first give a brief overview of how logic model checkers such as SPIN work. In Section 3 we show how SPIN models can be extracted mechanically from 'C' program source text and how a verification system can be built in this way. Without this capability the work reported here would not have been possible. This application is further described in Section 4. Section 5 concludes the paper and summarizes our findings.

2. How A Model Checker Works

The input language of the model checker SPIN allows us to build high-level models of distributed systems from three basic components: asynchronous processes, message channels, and data objects, [H97].

Each process in SPIN represents an asynchronous thread of execution. SPIN processes can interact by exchanging messages via channels, or by competing for access to shared data. Two types of message channels are predefined: standard FIFO-buffered channels with a user-defined number of slots, and unbuffered channels that can be used for data exchanges via synchronous handshakes (i.e., the data is passed from one process to another in a single operation, without intermediate storage in a buffer). Data objects can be local to a process, or global to all processes. A small number of basic data types are supported (int, short, byte, bool, and bit), plus a mechanism for defining higher-level data structures from these basic elements. Other arbitrary data-types and structures from C can be handled separately via a mechanism for encapsulated code that we discuss in Section 3.

SPIN requires the definition of systems that are 'closed' to their environment. This means that all potential input sources must be defined as part of the system. In practice this is not a difficult requirement to meet. If the exact behavior of an input source cannot be determined, a conservative approximation can usually be made. For instance, to analyze certain properties of a telephone system with a model checker [HS99] we must include a statement about the possible behavior of the subscribers that can interact with the system. The exact behavior of a telephone subscriber is of course ultimately unknowable, but a conservative model of it can easily be constructed. After all, there are only finitely many things that a telephone subscriber can do with a phone (lift the hook, flash the hook, or lower the hook, and dial one of a small number of possible digits at a time). We can, for instance, readily define a demon that generates a stream of randomly selected events from this small set. If we can prove that the telephone switch cannot be confused by this representation of possible subscriber behavior (regardless of which random sequence is generated) we have accomplished our goal. Note that the demon can do everything that a real subscriber can do, although the reverse is not necessarily true. (The subscriber can, for instance, not go off-hook twice in a row without going on-hook in between.) One way to think of the environment models that are used in model checking to close a system is to consider them a special type of *test driver*, as often used in conventional systems testing.

There are many ways to explain how a model checker actually works, for instance in terms of language-theory, graph-theory, or the theory of ω-automata. We will attempt to avoid excessive formality here by first giving a short explanation of the mechanics of the model-checking process itself, followed by a brief sketch of its theoretic underpinnings.

To effectively analyze a systems model, SPIN begins by computing a state-vector that defines the initial state of the system being modeled. Each process is minimally represented in this state-vector by its program counter. If the process has local variables, the set of current value assignments for those variables is also included in the state-vector. Each global message channel and global variable is also represented: the message channels by their content, and the global variables by their value assignments. Any one particular value assignment of the state-vector defines a *global system state*. We can describe *correctness properties* for the system in terms of instances of the global system state, e.g., state X shall exist before state Y is achieved. A violation of a correctness property implies a defect in the as-built system, the system requirements, or the way the property was specified. The model checker's objective is to compute the minimal number of reachable system states that are necessary to prove or disprove a given correctness property. SPIN allows correctness properties to be defined in terms of either finite or infinite (i.e., cyclic) executions. That is, the tool supports the verification of both *safety* and *liveness* properties (cf. [P77]).

In each system state, any one of the currently active processes might perform an atomic step (we can always chose a level of granularity of process-execution for which this is true). The semantics of the modeling language determine which steps are effectively executable, and which are not. Receiving a message from a channel, for instance, is only possible if the channel is non-empty. Similarly, we can define that sending a message to a channel is only possible

```
int shared = 0;
int *ptr;

void
thread1(void)
{       int tmp;

        ptr = &shared;
        tmp = shared;
        tmp++;
        shared = tmp;
}

void
thread2(void)
{       int tmp;

        if (ptr)
        {       tmp = shared;
                tmp++;
                shared = tmp;
                assert(shared == 1);
        }
}
```

Figure 1 – Example of two asynchronous threads competing for access to a shared variable.

when that channel is non-full. Typically, there will be several processes (threads of execution) that could perform a step at any given system state. Initiating or continuing execution with any one of these processes defines a different interleaving of process executions in time.

Trying all possible interleavings to see which ones can lead to failure would be astoundingly complex. To avoid this SPIN uses a theory of partial order reduction [HP94] to group process executions into equivalence classes. This is done in such a way that each interleaving within a given class necessarily will have identical correctness properties. Therefore, only *one* sequence from each class needs to be inspected by SPIN to achieve the effect of a fully exhaustive inspection of *all* possible executions.

The input language of SPIN is defined in such a way that there can always be only finitely many reachable system states, no matter how the model is defined. There can, for instance, be only finitely many processes, finitely many message channels and finitely many data objects with finite

```
2:                  thread2:[( ptr )]
3: thread1:[ tmp = shared; ]
4: thread1:[ tmp++; ]
5: thread1:[ shared = tmp; ]
6:                  thread2:[ tmp=shared; ]
7:                  thread2:[ tmp++; ]
8:                  thread2:[ shared=tmp; ]
pan: precondition false: (shared==1)
```

Figure 2 – Error scenario generated by Spin of an execution leading to assertion violation.

value ranges, and each process can only do finitely many different things. Although each execution step performed will modify the system state vector, only finitely many distinct state vectors are possible (also because there are only finitely many bits in a state vector). The set of all effectively computable states combined defines the globally reachable *state space* of the model. Note that even in a finite state space there can still be executions that can persist in principle infinitely long if the state space contains cycles.

Clearly, a naïve method of storing such a state space could consume huge amounts of memory, thereby limiting the practical use of this technique. SPIN has several algorithms to avoid this. One method, defined in [HP99], is based on the on-the-fly construction of a minimized recognizer for reachable states – a technique that can be likened to the BDD storage method used in some hardware verification systems [CGP00]. This technique can make it possible to store the state space in an amount of memory that is exponentially smaller than the amount consumed by direct storage. For rare cases where the amount of memory required to complete a correctness check still exceeds available memory, SPIN also has efficient approximation methods built in [H97]. These approximation methods make it possible to analyze even exceptionally large models.

To check the compliance of a system with a logic system property specified in linear temporal logic, SPIN first converts the formula into a test automaton that works much like an observer or monitor of the system executions. While building the system executions, the monitor is consulted at every step to see if violations occurred. If a violation is detected, SPIN displays the exact interleaving sequence leading from the initial system state to the state where the violation was detected. This serves as a counter-example to the correctness claims and facilitates diagnosis of the detected violation.

More formally, a temporal logic property defines a formal *language*. The words in this language define precisely those system executions that satisfy the property. By negating the property, we obtain a language that formalizes all the error sequences for that property, i.e., all the executions that do not satisfy the property. All the feasible (i.e., possible) executions of a SPIN model also define a formal language. We can now compute the intersection of these two languages: the language of the negated property and the language of the model. If the intersection is empty, there is

no feasible execution that violates the original property. If the intersection is non-empty, it contains the execution sequences that show in detail how the original property can be violated. SPIN will report these sequences as counter-examples to the correctness claims. In practice, SPIN does not compute the two languages separately before it computes their intersection: it achieves far greater efficiency by directly computing the intersection product on-the-fly, stopping as soon as the product can be shown to be non-empty

As a small example, consider the C-code shown in Figure 1. The two procedures give the code to be executed by two concurrent threads. The first thread gives the integer pointer variable *ptr* a non-zero initial value, then reads the value of the global variable *shared*, increments that value by one, and assigns the new value back to the variable. The second thread will have no effect if it starts executing before *ptr* was has a non-zero value. If it is found to have a non-zero value, *thread2* will perform the same steps as *thread1*, attempting (incorrectly) to increment the value of *shared* by one. An assertion at the end of *thread2* then checks that the value of the variable equals one. We can check mechanically with SPIN if there exists any thread interleaving for which the assertion might fail. We can do so by extracting a model from each of the two threads and running the SPIN model checker on the resulting code. If we do so, the model checker will produce the error sequence shown in Figure 2 to prove that the assertion can indeed be violated.

3. MODEL EXTRACTION

Until recently, the only way to use model checking in the verification of a software application was to first construct an accurate model of that application, formulate the correctness properties, and then to check that the properties hold for the model. If a property turns out not to be satisfied this could have multiple causes. The most innocuous is when the property itself is inaccurate and has to be modified. It is also possible that the model is inaccurate, due to an incomplete understanding of the application. By learning more about the application, the model can be adjusted and the verification repeated. Another problem can be that the model is accurate for some version of the application, but was not kept up to date with further evolutionary changes in the application that may affect its correctness properties. This can be a difficult problem to tackle, especially in cases where the model seems to satisfy the requirements. It can take days or weeks to construct an accurate model of an application, even for skilled users of model checkers, so it is likely that the model is never quite up to date. Results of violations reported by the model checker are all too easily discarded by the application builders, who know that the real application has evolved.

Many of these problems can be avoided when it becomes possible to mechanically extract verification models from application source code. A first such method was recently developed for the SPIN model checker [HS99], and has been applied to a number of case studies, including the exhaustive verification of the source code for the call processing module in a complex telephone switching system [HS00]. It is this method that we have applied to the verification of some of the source code modules of the DS1 spacecraft.

The SPIN model extractor (called FEAVER) works much like the front-end of a standard compiler. In our case we use a compiler front-end for the language C [KR88]. A parser constructs a parse-tree representation of the program, which is converted into a standard control-flow graph. This control-flow structure for each procedure of interest is cast in the modeling language of the SPIN model checker, but the statements and expressions remain native C-code. A relatively small extension of the SPIN model checker [H00] allows for the inclusion of basic statements and expressions from the applications as embedded C-code inside the models. Data can similarly be encapsulated as embedded data inside SPIN models. The model is analyzed as before, this time executing embedded C-code fragments for the individual atomic steps in the execution of each asynchronous process thread, but with the control-structure determined by SPIN. The model checker retains complete control over thread interleaving and the search strategy. It keeps track of the system state as before with a state-vector, but this time the state-vector can include embedded data-declarations that are lifted directly from the application source.

The most important benefit of the model extraction process is that it is virtually instantaneous: it can be repeated at will, whenever the source code of the application changes. Revisions of the application source code can thus be tracked from day to day with model checking runs, without requiring an addition intellectual investment to construct up to date models for each new version of the application.

The effort to build an accurate verification model now shifts to the construction of a *test harness*. The test harness is a small definitions file that is used by the model extractor to decide which portions of the source code are to be converted into model fragments, and how these fragments are to be combined in the model system. At the tester's discretion, large portions of the application can be excluded from the model-checking process and replaced with simple stubs. This is especially beneficial in cases where the test with the model checker is run on a platform where not all components of the actual application system are available.

(E.g., hardware interfaces, platform specific libraries, etc.)

The construction of the test harness itself undeniably also requires some skill, but once mastered it is not time consuming. Typically, several days worth of editing time on an application translates into minutes of work to inspect and where necessary update the test harness, after which a verification suite with the model checker can be repeated.

4. APPLICATION TO DS1

We have applied model extraction and model checking to portions of legacy flight software from NASA's Deep Space One (DS1) mission. This software was implemented in C and contained some known defects at launch that are detectable with a model checker. Our objective in this trial was to consider how we could reproduce a specific known defect mechanically with the model extractor and model checker.

In a first study we looked at the downlink packet handler module from DS1, consisting of about 12 C source files and 18 header files. There are 1162 lines of code in the header files and 5166 lines of code in the C source files.

The first problem we had to solve was that the code is written to execute under VxWorks, with direct access to a specific hardware interface to the spacecraft. The model extractor and model checker run standalone on a generic Unix or Windows environment, without access to the spacecraft hardware. No workstation simulation environment was available for this code. If one had been available, it would have been relatively easy to use the simulated hardware within the test-harness we constructed.

We noted earlier that the model checker requires us to define a closed system model, with all inputs and environment parameters included. Fortunately, the environment model does not have to be nearly as perfect as a full-blown simulation environment: it only needs to tell us which inputs and events in the environment might affect the run of our chosen software module. It need not detail the precise conditions under which those inputs or events are

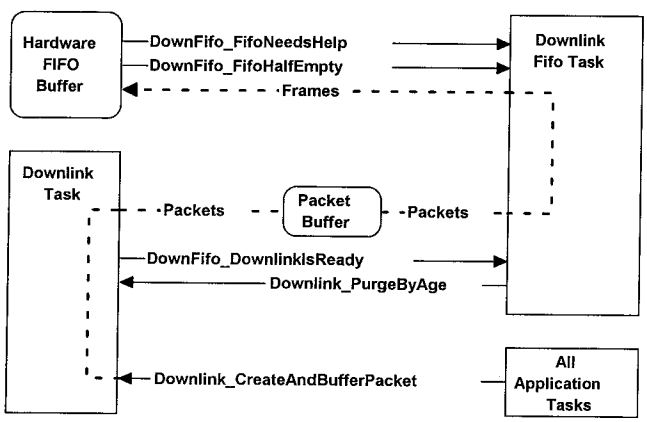

Figure 3 – DS1 Downlink telemetry packet handling

triggered. (Similar to the phone subscriber emitting input events for a telephone switch.) To close the downlink module for the test with SPIN, we wrote a small library of stubs, randomly emitting possible responses (such as 'success' or 'failure') when specific hardware or software functions from VxWorks or from the spacecraft hardware were invoked. This stub-library is 1343 lines of C. Of that total, 981 lines are print statements for informational purposes, so only 352 lines are functionally necessary.

The FEAVER test harness for this application is just 146 lines of text. The first thing that the test harness defines is the set of data objects in the application that hold basic state information. The model checker uses this information to construct the reachable state-space for the system during its search for defects. The test-harness also defines two asynchronous threads of execution: a DownFifo controller thread and the DownLink handler thread. These two threads each read and dispatch messages in priority order. The last component defined in the test-harness is the main test driver that emits a random stream of valid input commands for the module to handle during the test. (The model checker has a built-in notion of non-determinism that guarantees that the verification results hold for not just one, but all possible random streams that could be generated by the test drivers.)

The code for the main control threads in the system can be derived from the source text of the application via model extraction, but we chose to write these modules in a few lines of SPIN code instead, to gain more control over possible variations of the priority handling mechanism.

In a first test, we left the remainder of the code in native C, to be invoked directly by the main controller threads without further instrumentation. The compiled test code, generated by SPIN, is linked with the stub-library to give an executable system that performs the search for errors under the control of the model checker.

The property that we knew up-front could fail was that when the Downlink Purge command was given, there could be a particular error scenario in the code that would prevent it from being successfully executed. This can be formalized in a simple LTL requirement on the value of a variable, say v:

$$[] (v > 0 \rightarrow <> v = 0)$$

The symbols [] and <> are temporal operators in LTL, and the symbol $\rightarrow$ stands for standard logical implication (i.e., p $\rightarrow$ q means whenever p is true q must also be true). In English, this requirement reads: "it is always ([]) the case that whenever the value of v becomes greater than zero (v>0) eventually (<>) its value must return to zero at least once (v=0)." The variable we are interested in tracking with

this requirement is `Downlink_waitingToPurge`.

Our intent was to first perform the test for this property with the test harness as defined above. In this first case, all code is executed by invoking the C source code procedures directly without further instrumentation by model extraction. Then we would replace modules in the call chain one by one with extracted models (with the individual execution steps of each extracted procedure now under the control of the model checker) until the requirement violation would reveal itself.

In fact, two variations of the error scenario were immediately discovered with the initial definition of the test harness. (That is, with only the main controller threads instrumented, but with all the procedures executed from source directly without detailed interleaving control of the model checker.) With the help of the counter-examples provided by SPIN, the two variations identified were traced back to the following possible causes.

First, the loss of a message sent into a full message buffer could cause the purge command to be lost, leaving the module in a state where the command had been issued but not executed. This was precisely the known defect which we were seeking.

Second, a persistent stream of high-priority messages could postpone the execution of the purge command indefinitely beyond its point of issue.

The model checker finds both scenarios in a few seconds of runtime.

Each additional level of detail in specifying the system to be tested increases the size and scope of the global system state-space to be searched by the model checker. This includes, for example, adding procedures, messages, or processes. Although the model checker employs numerous techniques to be efficient, these additional details cannot but increase the execution time of the model checker. Further investigation of the DS1 downlink area for additional defects is possible, but was not performed in this effort due to resource constraints.

Sequencing Module. In another test, we applied SPIN to a different module of the DS1 code. This module contains the controllers for up to eight sequencing engines that can execute commands in the spacecraft uploaded from the ground. This module was significantly smaller than the DownFifo and Downlink module considered before. Only 1894 lines of C code (both source and header files) define the sequence controllers, to which we added about 600 lines of source text to define the stub-library (about half of what we needed for the first application). The larger part of the stub library was imported without change from the first study.

The test harness for this application is 141 lines of text, very similar to the size of the first application. It identifies the data to be tracked as state information, two controllers and a test driver that randomly emits sequence activation, deactivation, deletion, and status-list commands.

The controllers intercept errors where, for instance, an attempt is made to activate an already active sequence or to deactivate an inactive sequence. A simple property to check is that within a finite (though unspecified) amount of time after an activation command on an inactive sequence, that sequence is indeed activated. The model checker can quickly show that this property is not satisfied for the sequence module. The reason for this is in the particular way that the validity of a command is checked and then executed. The controller, upon receiving an activation command will first check if the sequence specified is already active. It will reject the request and issue a warning if this is the case. If inactive, the command is passed on to the sequence machine for execution. The execution of the command, however, happens in a different thread. So it is possible for a *deactivation* command to have been checked and passed on for execution, but not yet executed, when a subsequent command for re-*activation* is received. When the deactivation command has not yet been executed, the re-activation command will be rejected, with a warning issued.

While somewhat pathological, it is not inconceivable that this could occur. Let us for the moment assume that the spacecraft operators would not to try to deactivate and then reactivate the same sequence in a very short period of time. What of autonomous on-board functions, such as fault protection? Suppose, for example, that spacecraft fault protection is engaged in a fault recovery which results in a sequence being activated. Then, suppose that a second, more critical fault is detected, and that the fault response might include using the same sequence. It is conceivable that fault protection may issue a command to deactivate the sequence (in fact, fault protection typically wants *all* sequences deactivated when reconfiguring the spacecraft), and then want to re-activate the very same sequence. Could this occur with an unfortunate timing that would prevent the sequence from actually becoming active? That is not clear, but the present design of the sequencing module certainly does not *preclude* that possibility. This could be regarded as a latent defect. The commands to deactivate and reactivate may look completely valid upon inspection–yet they can fail under certain subtle conditions that are influenced by thread interleavings and event timings.

It was somewhat surprising to us that in this application we could identify a potential problem in the code without

producing a detailed system model using extracted components from the source code of the application via the model extraction capability. A rudimentary top-level model, making direct calls on the source code modules as written sufficed. Clearly though, for a truly thorough check, model extraction will be essential in building the test harness. The errors revealed by the coarser models we used would necessarily also be uncovered by a more detailed model, but the reverse is not necessarily true: the more thorough model could in principle reveal a greater number of errors.

5. Conclusion

In this work we have (1) used a model checker to detect a known error in the launch version of the DS1 spacecraft flight software, (2) discovered a second scenario under which a similar error could occur, and (3) discovered a third case in the DS1 sequencing module where a rare race condition could cause a sequence to fail to become active. In doing so, we have taken the first steps towards learning how to apply model-checking techniques to the verification of spacecraft flight software. These steps include

- Defining and describing *correctness properties* (from requirements and design specifications)
- Constructing a *test harness* to interact with the module(s) being checked
- Analyzing and interpreting results for plausibility and criticality.

When a property violation is discovered, it is important to determine the cause of the violation. Was there really a serious violation of the requirements or expected behavior? Or perhaps the test inputs were unrealistic, or the violated property was improperly specified? There are several reasons why a model checker might discover a problem, and not all of them are due to actual defects in the software product. In our case, it took a bit of initial effort to define some *correctness properties*, but once the initial set was established, it became much easier to see what to do and how to do it. Many *correctness properties* are common to multiple modules within the system and are therefore reused throughout the system.

The same was true of the *test harness*. But we found that we were able to reuse significant portions of the test harness from one module to another.

Although this may appear to be a substantial amount of work, it is really no different from the engineering required to do conventional testing. The tester still needs to think about what will have to be tested (i.e., formulate the required correctness properties) and how you will interact with the test article (*test harness*). The difference is that model checking gives the tester the opportunity to detect race conditions, deadlocks, and other interleavings that may occur only sporadically, if at all, during conventional testing. Model checkers operating on as-built code-extracted models can also verify software much more quickly than human testers, and should prove useful in regression testing to show that new concurrency defects have not been introduced by modifications to existing software. We believe that model checkers are a useful and effective tool in the software testers toolchest.

6. Acknowledgements

The work described in this paper was performed jointly at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration, and at Lucent Technologies - Bell Labs Research. Grateful acknowledgement is made to JPL's Center for Space Mission Information and Software Systems for their support to this effort.

7. References

[CGP00] E.M. Clarke, O.Grumberg, D.A. Peled, *Model Checking*, MIT Press, Jan.2000.

[EH00] K. Etessami, G.J. Holzmann, Optimizing Buchi Automata, *Proc. 11th Int. Conf. on Concurrency Theory, CONCUR 2000*, August 2000.

[GPVW95] R. Gerth, D. Peled, M. Vardi, and P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, *Proc. Conf. On Protocol Specification, Testing, and Verification*, Warsaw, Poland 1995, Chapman and Hall, pp. 3-18.

[Hajek78] J. Hajek, Automatically verified data transfer protocols, *Proc. 4th ICCC*, Kyoto, 1978, pp. 749-756.

[H81] G.J. Holzmann, Pan, a protocol specification analyzer, AT&T Bell Laboratories *Technical Memorandum*, TM81-11271-5, 1981.

[HP94] G.J. Holzmann, and D. Peled, An Improvement in Formal Verification, *Proc. Formal Description Technique*, Chapman Hall, Berne Switzerland, October 1994, pp. 197-211.

[H97] G. J. Holzmann, The Model Checker Spin, *IEEE Trans. on Software Engineering*, Vol. 23, No. 5, pp. 279-295, May 1997.

[HP99] G.J. Holzmann and A. Puri, A Minimized Automaton Representation of Reachable States, *Software Tools for Technology Transfer*, Springer Verlag, Vol. 2, 3, pp. 270-278, November 1999.

[HS99] G. J. Holzmann, and M.H. Smith, Software model checking - Extracting verification models from source code, *Formal Methods for Protocol Engineering and Distributed Systems*, Oct. 1999, Kluwer Academic Publ., pp. 481-497.

[HS00] G.J. Holzmann, and M.H. Smith, Automating Software Feature Verification, *Bell Labs Technical Journal*, Special Issue on Software Complexity, April 2000.

[H00] G.J. Holzmann, Logic Verification of ANSI-C Code with Spin, *Proc. 7th Int. Spin Workshop on Model Checking Software, SPIN2000*, Stanford Univ. CA., Sept. 2000, Springer Verlag, LNCS 1885, pp. 131-147.

[KR88] B.W. Kernighan, and D.M. Ritchie, *The C Programming Language*, 2nd Edition, Prentice Hall, Englewood Cliffs, N.J., 1988.

[P77] A. Pnueli, The temporal logic of programs, *Proc. 18th IEEE Symposium on Foundations of Computer Science*, 1977, Providence, R.I., pp. 46-57.

[SPIN] The Spin tool is available at: http://netlib.bell-labs.com/netlib/spin/whatispin.html.

[VW86] M.Y. Vardi, and P. Wolper, An automata-theoretic approach to automatic program verification, *Proc. Symp. on Logic in Computer Science*, Cambridge, June 1986, pp. 322-331.