Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 49–61
www.elsevier.com/locate/entcs

Thread Verification - An Experience Report

Robert P. Cook[1,2]

Department of Computer Sciences
Georgia Southern University
Statesboro, GA 30460 U.S.A

Abstract

The paper details the author's thread verification experiences with four applications: Linux kernel code, the Red Hat Linux POSIX Thread library, a portable PThread library, which was developed by the author for NASA, and a HandyChecker prototype. Based on the author's experiences, the paper concludes with a list of challenges for the verification community.

Keywords: Verification, model checking, reuse Libraries, POSIX threads, Linux

1 Introduction

The theme of the paper is that there exists a spectrum of analysis techniques between those used to code a concurrent algorithm and those used to verify a concurrent algorithm. Software engineers who ignore verification techniques because they "can't handle my problems" do so at their own risk. To paraphrase Plato, "the unexamined concurrent algorithm is not worth using." [3] The question is "what steps should comprise an examination?"

The paper details the author's thread verification experiences with four applications: Linux kernel code [1], the Red Hat Linux POSIX Thread library [2], a portable PThread library [3], which was developed by the author for NASA, and a HandyChecker prototype. Each of the examples illustrates the futility of relying solely on traditional testing as a verification technique. The discipline of applying a spectrum of verification methods, even if none succeed completely, can expose errors that would be difficult to discover through testing alone.

[1] Email: bobcook@GeorgiaSouthern.edu

[2] Acknowledgements: The Army Research Office (DAAD19-01-1-0473) and NASA for supporting this research.

[3] Socrates, in Plato, Dialogues, Apology

1571-0661 © 2007 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2007.04.006

2 Linux ID Allocation

Linux is a good source for verification examples because the code is posted online and there has been a concerted effort (the Linux Test Project [4]) by IBM and other companies to improve the reliability of the Linux code base through testing. It is unfortunate that a similar project was not undertaken based on verification analysis.

Consider the common problem of generating unique id numbers. Operating systems have a requirement to generate unique ids in a number of contexts. The ids may need to be locally unique or unique in time and space. Microsoft's C++ product, for example, includes a "uuidgen" utility that generates 128-bit ids that are unique in space and time.

For this example, we consider the problem of generating process ids (pids) from a fixed range of positive integers. The requirements are 1) that a given pid come from within the fixed range, 2) that it not be allocated more than once without being freed, 3) that all integers in the range be allocated before returning an error, 4) that a freed id must be in the allocated state, 5) that an id cannot be freed more than once, 6) return 0 if no pid is currently available, and 7) that any solution be thread safe. The first few hundred ids are reserved for daemon processes.

The following solution (Figure 1) [1] is from the Linux operating system kernel/fork.c v2.4.28 ©Linus Torvalds et al and is used to allocate unique process ids. The solution has the advantage that no auxiliary data structure is needed. The algorithm allocates a process id only if it is not already in use by other processes.

The first verification step was to recode the algorithm in Promela for input to the SPIN [5] model checker. Figure 2 lists a Promela code fragment. Promela supports multi-threaded programming with guarded (pre-conditions) statements and channels (message queues) for inter-thread communication. It is interesting that neither channels nor their predecessor, UNIX pipes, have been "codified" in the popular Java and C++ languages. However, in the new Sony/IBM Cell Broadband Engine [9], channels and mailboxes are first-class hardware primitives, so the appearance of these abstractions in programming language syntax may be forthcoming.

The resulting model's execution uncovered two bugs; unfortunately both were in SPIN. One was fixed and one was not. Unfortunately, the latter bug (termed a "feature") resulted in a runtime error that terminated model execution.

The second verification step was to apply a technique that we term "fine-grain concurrency testing". The StarLite [6] programming environment, which was developed by the author, supports a C++ interpreter and a library of thread interfaces. The runtime has the unusual property that the clock period is tied to instruction execution (a clock interrupt is generated every N instructions). As a result, a clock tick could be scheduled as often as every instruction fetch cycle. By scheduling threads based on clock ticks, a fine-grain level of multiplexing can be achieved that would be impossible on a physical machine.

The only change in the algorithm for StarLite testing was to reduce the "last reserved pid" from 300 to 4 and the "pid maximum" from 32767 to 15. An algo-

```
int last_pid=LAST_R_PID;
spinlock_t lastpid_lock = SPIN_LOCK_UNLOCKED;
static int get_pid() {
        static int next_safe = PID_MAX;
        struct task_struct *p;
        int pid, beginpid;
        spin_lock(&lastpid_lock);
        beginpid = last_pid;
        if ((++last_pid) & 0xffff8000) {
                last_pid = LAST_R_PID;
                goto inside;
        }
        if (last_pid >= next_safe) {
inside:         next_safe = PID_MAX;
                read_lock(&tasklist_lock);
repeat:         for_each_task (p) {
                        if (p->pid == last_pid) {
                                if (++last_pid >= next_safe) {
                                        if (last_pid & 0xffff8000) {
                                                last_pid = LAST_R_PID;
                                        }
                                        next_safe = PID_MAX;
                                }
                                if (last_pid == beginpid) {
                                        next_safe = 0;
                                        read_unlock(&tasklist_lock);
                                        spin_unlock(&lastpid_lock);
                                        return 0;
                                }
                                goto repeat;
                        } // if p->pid==
                        if (p->pid > last_pid && next_safe > p->pid) {
                                next_safe = p->pid;
                        }
                } //for each
                read_unlock(&tasklist_lock);
        } //if last_pid >= next_safe
        pid = last_pid;
        spin_unlock(&lastpid_lock);
        return pid;
}
```

Fig. 1. Linux Allocate Process ID (get_pid)

```
 1 #define MAX 4                  /* file ex.2 */
 2 proctype A(chan in, out)
 3 {     byte mt; /* message data */
 4             bit   vr;
 5 S1:    mt = (mt+1)%MAX;
 6             out!mt,1;
 7             goto S2;
 8 S2:    in?vr;
 9             if
10               :: (vr == 1) -> goto S1
11               :: (vr == 0) -> goto S3
12               :: printf("MSC: AERROR1\n") -> goto S5
13             fi ;
14 S3:   out!mt,1;
15           goto S2;
16 S4:   in?vr;
17           if
18               :: goto S1
19               :: printf("MSC: AERROR2\n"); goto S5
20             fi ;
21 S5:   out!mt,0;
22           goto S4
23 }
```

Fig. 2. Sample Promela Code

rithm with $N = 15$ (or less) can be inductively proved valid for larger values of N. An interesting thread verification question is deciding when an inductive proof is possible and when the problem has been optimally reduced in size.

On the first StarLite run, two problems were exposed. Both were "obvious", but neither occurred to the author a priori. First, the algorithm "reads" the state of the task list then releases the lock. At that point, the invariant "new pid not in list" becomes problematic. If *last_pid* cycles before the old allocation is stored in the task list, the same id could be multiply allocated. The second error occurs after *last_pid* counts past PID_MAX. The "last reserved pid" is allocated instead of that value plus one. The third, and final problem was discovered based on the author's experience, but it could also be detected mechanically. Once *last_pid* is "locked", there is no reason to manipulate it as a global variable in the loop, which induces a performance penalty in multiprocessor environments.

After Linux v2.4.28, the pid management algorithm was upgraded to dynamically allocate pages using a bitmap as the id-allocation data structure. The following two code snippets (Figure 3) illustrate classic "race" conditions. Since page allocation is time consuming, the first race ($map \rightarrow page$) was probably exposed and compensated for, although in a non-optimal fashion.

The second race condition ($map \rightarrow nr\_free$) only covers three or four instruc-

```
if (unlikely (!map->page)) {
        long page = get_zeroed_page(GFP_KERNEL);
        /*Free the page if someone raced with us */
        spin_lock(&pidmap_lock);
        if (map->page)
                free_page(page);
        else
                map->page = (void *)page;

        spin_unlock(&pidmap_lock);
        if (unlikely (!map->page))
                break;
}
```

```
if ( likely (atomic_read(&map->nr_free))) {
        do {
                if (! test_and_set_bit ( offset , map->page)) {
                        atomic_dec(&map->nr_free);
                        last_pid = pid;
                        return pid;
                }
                offset = find_next_offset (map, offset);
                pid = mk_pid(map, offset);
        }
}
```

Fig. 3. Linux alloc_pidmap Code Snippets

tions so its exposure probability was low. An error occurs when multiple threads find nr_free equal to one. Only one caller gets the free id while the remaining threads search a full bitmap.

3 Red Hat POSIX Threads

In 2004, the author received a summer fellowship at the NASA Kennedy Space Center to modify the Red Hat NPTL library to support the POSIX Threads real-time features. The task required an investigation into the implementation of the Linux kernel futex [7], which is probably the most under-studied synchronization primitive in modern operating system history.

A futex is a fast user-space mutex. It is fast because the "busy" test is performed with non-privileged instructions. The operating system kernel is invoked only for queuing and delay.

Unfortunately, the NPTL library is tightly bound to both Linux internals and the futex implementation. It proved infeasible to modify both the library and the

Linux kernel in the time allowed.

The author then spent part of that summer analyzing the correctness of selected modules of the NPTL code. NPTL was first released in 2002 with an extensive test suite. The code was released and in use for almost two years before the error was discovered by the author. Figure 4 lists the implementation (with an error) of the "get-read-lock" method for read/write locks.

Even before the SPIN model for rwlock was completed, it was obvious that there were problems with the implementation. Just the act of performing a verification review can be instructive! In the case of rdlock, a reader must wait (writer preference) if a writer is queued. At this point, the readers-queued count is incremented. However, it is never decremented!! This bug was acknowledged by Red Hat and fixed in the next release.

A second problem, which was discovered by model checking, occurred because the data-lock is released to perform a futex-wait on a writer, and then the lock is reacquired. This leads to a scrambling of all readers from the futex queue when reentering the data-lock-queue, leading to possible infinite overtaking or starvation problems.

4 HandyChecker Prototype

As mentioned earlier, it proved infeasible to modify the Red Hat POSIX Thread implementation to support the POSIX real-time features; therefore, a new library was designed. In addition to the real-time requirement, it was deemed advisable for the new library to be as portable as possible. At the minimum, for example, it could serve as a "reference" implementation. The source code and test results for the library are posted [3] on the web.

Coincident with the library's implementation by the author, it was decided to develop a prototype HandyChecker verification application. There were several goals. First, the checker had to work with C/C++ syntax. Second, the prototype had to support an automatic, or semi-automatic, annotation of C/C++ code to enable any program to be checked. Third, the prototype had to produce trails for discovered errors. Fourth, the state-space search had to be encapsulated in a simple enough fashion to support the replacement of the initial "brute-force" evaluation with more sophisticated methods.

HandyChecker's implementation is based on two mappings. First, a program's variables are renamed as structure elements. Secondly, the code is partitioned into code "strips". Each strip contains either an interference point (Read-Write or Write-Write) involving a global variable or a point that represents a control flow decision. The code strips have the nice property of piece-wise composition as long as the components have been verified. Thus, a module hierarchy can be checked by levels. Figure 5 lists the code strips (before transformation) for the barrier implementation in the portable PThreads library. Note that access/test steps, as in B, are decomposed into two entries "b,IF..." in the strip figure.

In the current HandyChecker prototype, only checking is implemented. Strip

```
int __pthread_rwlock_rdlock (pthread_rwlock_t *rwlock) {
        int result = 0;
        lll_mutex_lock (rwlock->__data.__lock);
        while (1) {
                /* Get the rwlock if there is no writer... */
                /* ... and if either no writer is waiting or we prefer readers.
                   */
                if (rwlock->__data.__writer == 0
                    && (!rwlock->__data.__nr_writers_queued
                    || rwlock->__data.__flags == 0)) {
                        /* Increment the reader counter. Avoid overflow. */
                        if (unlikely(++rwlock->__data.__nr_readers == 0)) {
                                /* Overflow on number of readers. */
                                --rwlock->__data.__nr_readers;
                                result = EAGAIN;
                        }
                        break;
                } /*if rwlock->__data.__writer */
                /* Make sure we are not holding the rwlock as a writer. */
                if (unlikely(rwlock->__data.__writer
                    == THREAD_GETMEM (THREAD_SELF, tid))) {
                        result = EDEADLK;
                        break;
                }
                /* Remember that we are a reader. */
                if (unlikely(++rwlock->__data.__nr_readers_queued == 0)) {
                        /* Overflow on number of queued readers. */
                        --rwlock->__data.__nr_readers_queued;
                        result = EAGAIN;
                        break;
                }
                int waitval = rwlock->__data.__readers_wakeup;
                /* Free the lock. */
                lll_mutex_unlock (rwlock->__data.__lock);
                /* Wait for the writer to finish. */
                lll_futex_wait (&rwlock->__data.__readers_wakeup, waitval);
                /* Get the lock. */
                lll_mutex_lock (rwlock->__data.__lock);
        } /* while 1 */
        /* We are done, free the lock. */
        lll_mutex_unlock (rwlock->__data.__lock);
        return result;
}
```

Fig. 4. Linux glibc MPTL rwlock

```
int _pthread_barrier_wait( p_thread_barrier_t *barrier) {
        long i, j;
        assert ( barrier != NULL);          /* A */
        j = barrier->cycle;
-------------------------------------------------------
        i = InterlockedDecrement (&barrier->queued);
        if ( i > 0 ) {
                Lock(&barrier->q[j]);
                BlockSelf(&barrier->q[j]);  /* B */
                return 0;
        }
-------------------------------------------------------
        assert (i == 0);
        i = barrier->count;                 /* C */
        barrier->queued = i;
-------------------------------------------------------
        barrier->cycle ^= 1;                /* D */
        assert(LookHead(&barrier->q[barrier->cycle]) == NULL);
-------------------------------------------------------
        for (i-- ; i != 0; i--) {           /* E */
                pthread_t  p;
                p = RemoveHead(&barrier->q[j]);
                assert ( p != NULL );
                Run(p);
        }
        assert ( i == 0 );
        assert( LookHead(&barrier->q[j]) == NULL );
        return PTHREAD_BARRIER_SERIAL_THREAD;
}
```

Fig. 5. HandyChecker Code Strips (before transformation)

identification and variable transformation are performed manually. Further, parallel searching is not implemented.

All state variables are collected into a single struct that contains a control-information struct, a global-variable struct and one struct (containing the local variables) for each procedure. Recursion is not currently supported. An example follows.

```
typedef struct {
        p_thread_barrier_t *barrier;
        long i, j;
        int function_return_value;
} prWait;

typedef struct {
        Controls control;        /* checker data structures */
        Globals global; /* pointers to global variables */
        prWait wait;             /* local variables */
} State;
```

Fig. 6. State Structure

Each strip is coded as a function of one argument of type "void *". Preprocessor #define macros are utilized to transform variable references into the equivalent State structure reference. Only C++ classes that have a copy constructor can be checked because the state has to be duplicated at every decision point.

Further, any primitives that suspend (queue) threads are simply marked as "to queue". The checker looks for these markers at every step to remove those threads from the "ready" list. This transformation is necessary because the checker is a single thread; blocking in a nested procedure would block checking. This restriction could be lifted if a separate thread were used to evaluate each "state". However, that option was not explored.

Figure 7 lists the barrier code after transformation. The current prototype's implementation assumes that a thread can only be in one queue at a time. Queues are assigned numbers in each State structure's control information. The Strip Code Array in the figure contains information for the checker on the "result" of each strip's execution. The IF selector is followed by three arguments, which are the strip to execute and array indices for the IF_TRUE (Suspend/Exit) and IF_FALSE (CDE...) cases.

Figure 8 & 9 list the sample output from HandyChecker for the barrier example. There are two threads (0 and 1). Note that it is possible for the "serial" thread to "beat" earlier threads attempting to block. The result is a program fault where none should exist. Figure 1 lists an interpretation of the error state.

HandyChecker is just a prototype that takes a step along the path to "proof of concept". There is much work yet to be done. For example, the implementation could be improved by adding semantic hints to reduce the number of search paths.

5 Challenges

- While the Linux kernel has efficient algorithms and a wonderful organization factored by architecture, it fails in terms of object-oriented design. An object-oriented implementation would facilitate verification in parts, if not in the whole.
- It would be interesting to have a Linux Verification Project in order to compare

```
int A(void *x) {
        L(x)->wait.j = L(x)->wait.barrier->cycle;
        return NEXT_STEP;
}
```

```
int B(void *x) {
        L(x)->wait.i =
        InterlockedDecrement (&L(x)->wait.barrier->queued);
        if ( i > 0 ) {
                assert (CONTROL(x)->queue == EMPTY)
                CONTROL(x)->toQueue = L(x)->wait.j;
                L(x)->wait.function_return_value = 0;
                return IF_TRUE;
        }
return IF_FALSE;
}
```

```
int C(void *x) {
        assert  (L(x)->wait.i == 0);
        L(x)->wait.i = L(x)->wait.barrier->count;
        L(x)->wait.barrier->queued = i;
        return NEXT_STEP;
}
```

```
        int D(void *x) {
        L(x)->wait.barrier->cycle ^= 1;
        return NEXT_STEP;
}
```

```
int E(void *x) {
        for (L(x)->wait.i-- ; L(x)->wait.i != 0; L(x)->wait.i--) {
                Wakeup (L(x)->wait.j);
        }
        assert  (L(x)->wait.i == 0);
        assert (isEmpty(L(x)->wait.j));
        L(x)->wait.function_return_value =
        PTHREAD_BARRIER_SERIAL_THREAD;
        return NEXT_STEP;
}
```

STRIP CODE ARRAY

0, A, IF, B, 8, 6, SUSPEND, EXIT, C, D, E, RESUME, EXIT

[0 1 2 3 4 5 **if true** 7 8 9 10 11 12]

Fig. 7. HandyChecker Code Strips (after transformation)

,0A,1A,1IF,0IF,1Suspend,0C,0D,0E,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,0IF,0C,1Suspend,0D,0E,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,0IF,0C,0D,1Suspend,0E,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,0IF,0C,0D,0E,1Suspend,0Resume,0EXIT,1EXIT,OK
,0A,1A,1IF,0IF,0C,0D,0E,0Resume, ERR
,0A,1A,1IF,0IF,0C,0D,0E,1Suspend,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,0IF,0C,0D,1Suspend,0E,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,0IF,0C,1Suspend,0D,0E,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,1Suspend,0IF,0C,0D,0E,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,0IF,1Suspend,0C,0D,0E,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,0IF,0C,1Suspend,0D,0E,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,0IF,0C,0D,1Suspend,0E,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,0IF,0C,0D,0E,1Suspend,0Resume,0EXIT,1EXIT,OK
,0A,1A,1IF,0IF,0C,0D,0E,0Resume, ERR
,0A,1A,1IF,0IF,0C,0D,0E,1Suspend,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,0IF,0C,0D,1Suspend,0E,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,0IF,0C,1Suspend,0D,0E,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,0IF,1Suspend,0C,0D,0E,0Resume,1EXIT,0EXIT,OK
,0A,1A,0IF,1IF,0Suspend,1C,1D,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,0Suspend,1D,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,1D,0Suspend,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,1D,1E,0Suspend,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,1D,1E,1Resume, ERR
,0A,1A,0IF,1IF,1C,1D,0Suspend,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,0Suspend,1D,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,0Suspend,1C,1D,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,0Suspend,1IF,1C,1D,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,0Suspend,1D,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,1D,0Suspend,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,1D,1E,0Suspend,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,1D,1E,1Resume, ERR
,0A,1A,1IF,0IF,0C,0D,0E,1Suspend,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,0IF,0C,0D,1Suspend,0E,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,0IF,0C,1Suspend,0D,0E,0Resume,1EXIT,0EXIT,OK
,0A,1A,1IF,0IF,1Suspend,0C,0D,0E,0Resume,1EXIT,0EXIT,OK
,0A,1A,0IF,1IF,0Suspend,1C,1D,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,0Suspend,1D,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,1D,0Suspend,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,1D,1E,0Suspend,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,1D,1E,1Resume, ERR

Fig. 8. HandyChecker Code Output (Threads 0 & 1)

```
,0A,1A,0IF,1IF,1C,1D,0Suspend,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,0Suspend,1D,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,0Suspend,1C,1D,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,0Suspend,1IF,1C,1D,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,0Suspend,1D,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,1D,0Suspend,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,1D,1E,0Suspend,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,1D,1E,1Resume, ERR
,0A,1A,0IF,1IF,1C,1D,0Suspend,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,1C,0Suspend,1D,1E,1Resume,0EXIT,1EXIT,OK
,0A,1A,0IF,1IF,0Suspend,1C,1D,1E,1Resume,0EXIT,1EXIT,OK
```

Fig. 9. HandyChecker Code Output (Threads 0 & 1) Contd

0A 1A 0IF 1IF 1C 1D 1E 1Resume ERR	
Thread 0	**Thread 1**
0A j = barrier→cycle; j == 0	
	1A j = barrier →cycle; j == 0
0IF i = InterlockedDecrement(barrier→queued); i == 1	
	1IF i = InterlockedDecrement(barrier→queued); i == 0
	1C i = barrier →count; barrier→= count; i == 2
	1D barrier → cycle ^= 1;

	IE
	for (i−−; i != 0; i−−) {
	pthread_t p;
	p = RemoveHead;

Thread 1 encounters a violation of the invariant that N-1 threads are queued. Thread 0 changed the global state to indicate that it was blocked when, in fact, it is not blocked.	

Table 1
Barrier Error Interpretation

the results to what is now a mature LTP effort.

- POSIX Threads needs to have a reference implementation and needs to be revived as a living standard.
- Every POSIX Threads implementation should be verified as well as tested.
- According to Enea TekSci, "DO-178B [8] is perhaps the most stringent standard in the world. If it flies commercially, it must be DO-178B certified". However,

its software process requirements are so time-consuming and complex that programmer productivity may average less than 100 lines/month. Every point of entry and exit in a program must have been invoked at least once in testing, every decision in the program must have taken all possible outcomes at least once, and each condition in a decision must have been shown to independently affect that decision's outcome. Is DO-178B testing guaranteed to uncover any error in a sequential program? A multi-threaded program? What is the research community's perception of current DO-178B Software Verification Plans?

- Current model checkers are too closely tied to a model language. A "checking" algorithm library should be constructed to facilitate experimentation with the infrastructure independent of language issues.
- What is the current state of model-checking benchmarks? What are the performance and quality criteria?
- The OpenMP standard utilizes "pragma"s to annotate code. Each "pragma" has implicit assertions associated with it. Can model checkers verify any of them?
- The Sony/IBM Cell Broadband Engine presents a complex challenge to programmers and to program verifiers and optimizers.
- Is it possible to standardize an XML intermediate representation for input to model checkers?
- Java and Ada broke new ground with first-class syntax notation to support concurrent programming. What is the next step?
- Is it time to discard two-dimensional programming notations in favor of multi-dimensional XML annotations that document a program's refinement from requirements to target machine language?

References

[1] http://lxr.linux.no/source/kernel/fork.c

[2] http://sourceware.org/cgi-bin/cvsweb.cgi/libc/nptl/?cvsroot=glibc

[3] http://bcook.cs.georgiasouthern.edu/pthreads/

[4] http://ltp.sourceforge.net/

[5] Holzmann, Gerard J., The SPIN Model Checker, Addison-weseley, (2003)

[6] Cook, Robert P., The StarLite Operating System, in: *Operating Systems for Mission-Critical Computing,* edited by K. Gordon, P. Hwang, A. Agrawala, IOS Press, (1992) 2-10.

[7] Franke, Hubertus, Rusty Russell, Matthew Kirkwood, Fuss, Futexes and Furwocks: Fast User-level Locking in Linux. *Proceedings Ottawa Linux Symposium*, (2002).

[8] Hilderman, Vance, Certifying an RTOS to DO178B: Tips & Tales, *The Open Group Real-Time and Embedded Systems Forum*, Amsterdam, Netherlands, (Oct 2001).

[9] http://www-128.ibm.com/developerworks/power/cell/