Systematically Testing a Real-Time Operating System

Testing large and complex software is an inherently difficult process that must be as systematic as possible to provide adequate reliability and quality assurance. This is particularly true for a complex real-time operating system, in which an ad hoc testing approach would certainly fail to affirm the quality and correctness of the requirements specification, design, and implementation. We discuss applying systematic strategies to the testing of real-time operating system RTOS under development in the Esprit III project 8906 OMI/CLEAR.

Manthos A. Tsoukarellas

Vasilis C. Gerogiannis

Kostis D. Economides

Advanced Informatics Ltd.

Software testing is one of the most significant activities in the software life cycle and must affirm the quality of requirements specification, design, and implementation. A company developing software spends about 40 percent of a project's cost on testing activities.[1] In exceptional cases, such as safety-critical real-time software, the testing phase may cost from three to five times as much as any other phase of the software life cycle. Usually, testing begins early in the development process, as test planning and specification overlap, to a certain degree, with requirements specification.

Testing activities demonstrate that software is consistent with its specifications. Therefore, as test results accumulate, evidence indicating the level of software quality and reliability emerges. In particular, if testing often detects important errors, the software's quality and reliability are probably inadequate and further testing is required. On the other hand, if the discovered errors are minor and easily correctable, then either the level of software quality and reliability is acceptable, or the executed tests are inadequate to reveal software errors. Testing can never assure that a program is correct because undetected errors may exist even after the most extensive testing. Therefore, the common view that a successful test is one that reveals no errors is incorrect, as the following general testing goals emphasize:[2,3]

- Testing is the process of executing a program with the intent of finding errors.
- A good test case has a high probability of finding an undiscovered error.
- A successful test uncovers an undiscovered error.

Aspects of general testing strategies and techniques also apply to real-time software. However, additional requirements and difficulties characterize an effective testing process for real-time software, especially real-time operating systems (RTOSs), for several reasons:[4,5]

- The software contains several modules and decision statements.
- Many modules use the same resources simultaneously.
- The same sequence of test cases may produce different outputs, depending on when the test is performed.
- System errors may be time dependent, only arising when the system and controlled environment are in a particular state that may be impossible to reproduce.
- Finally, reliability, schedule, and performance requirements are usually more critical than those for non-real-time software.

This article focuses on a well-established methodology to test the functional behavior of

0272-1732/95/$04.00 © 1995 IEEE

an actual RTOS. The system is being developed as part of the ESPRIT III project 8906 OMI/CLEAR (Open Microprocessor Systems Initiative/Components and Libraries for Embedded Applications in Real-time).[6] CLEAR RTOS is a scalable executive for embedded applications. It supports multiple configuration levels (from hardware and basic I/O services to higher level ones) to establish the correct balance between efficiency (performance/size) and required services.

The strategy followed for testing CLEAR RTOS is systematic and includes individual function, module, and bottom-up integration testing, using both black-box and white-box disciplines. We chose this approach because not all modules are available from the beginning of testing activities. A systematic way to determine test cases satisfies the requirement for low redundancy (in test cases selection) and high reliability (in test results). In addition, this method incorporates often-used coverage criteria and software complexity metrics.

Issues related to RTOS timing behavior (schedule analysis and performance testing) are beyond the scope of this article. However, Gerogiannis and Tsoukarellas discuss such issues.[7]

Software testing techniques

We base testing specification on methods that guide testers to follow a systematic approach. These methods provide criteria for determining appropriate test cases, to ensure test completeness and guarantee a high probability of discovering software errors. Using these methods, we derive test cases from specifications or by code examination. The test methods corresponding to these approaches are known as black- and white-box testing (see the box, next page).

Software testing strategies

A strategy for software testing incorporates a set of activities organized into a well-planned sequence of steps that finally affirms software quality. The initial important decision in the process is determining who will perform testing. Pressman[1] discusses the various inherent problems associated with allowing software developers to test the product.

> From a psychological point of view, software analysis, design (along with coding) are constructive tasks. However, when testing commences there is a subtle, yet definite, attempt to 'break' what the software engineer has built. From the point of view of the builder, testing can be considered destructive. So, the builder designs tests that will demonstrate that the program works. The role of an independent test group (ITG) is to remove the inherent problems associated with letting the builder test his own product. However, the developer and the ITG have to work closely throughout a software project to ensure that thorough tests will be conducted.

These problems motivated the strategy adopted in the OMI/CLEAR project, in which two independent groups perform development and test. The groups cooperate closely to achieve a high-quality final product. In general, the OMI/CLEAR project's testing strategy consists of three phases: individual function, module, and integration testing.[8]

Individual function testing. This strategy focuses on each individual function. Testing a function in an isolated manner means that we do not attribute operations performed by calling other functions to the function under test. In this case, the description of the calling function should identify the called functions. In the beginning, we apply the black-box technique to test each function's interface according to the corresponding input/output specifications. Afterwards, we apply the white-box method to test the paths in the function's source code. During this phase, drivers and/or stubs may have to be constructed for each individual function test. A driver accepts the test case data as input and passes it to the function. A stub, which we place in the code because some functions are not yet available, simulates functions that are immediately subsequent (lower level) in the control flow to the tested function.

Module testing. Next, we combine the already tested functions to compose modules (tightly coupled functions), using either the top-down or bottom-up method. These two approaches (described in the following Integration testing section) apply to module testing if we exchange the term *function* for *module,* and *module synthesis* for *integration process.* That is, the integration of functions into modules is similar in concept to the integration of modules into larger ones to construct the entire program.

In module, and subsequently, integration testing, we use the black-box technique because the main purpose of module and integration testing is to uncover interfacing errors.[1] In addition, the code to be tested tends to be very large as we form and integrate modules, so white-box testing becomes extremely complicated.

The essential difference between testing the individual function and testing the module of a function that calls other functions is that in the first strategy, testing extends only to the calls to other functions; it does not account for the operation of those functions. On the other hand, the second strategy checks the module as a whole, all the way to the operations performed by the called functions.

Integration testing. Third, we integrate the already tested modules into larger ones. There are two approaches to merging modules during this phase: nonincremental and incremental.

The nonincremental or "big bang" approach tests a program by testing modules independently and then combines all the modules together to test the program as a whole. This approach does not facilitate revisions since we cannot easily isolate errors.

Black- and white-box testing

Black-box (functional) testing does not consider the internal structure and behavior of the program,[9] but examines only its input-output behavior. Black-box testing methods are based on the functional requirements specification of the software and determine whether the software behaves as specified. We construct test data from the specifications and, in general, there are three methods for deriving the appropriate test cases.

The equivalence partitioning method divides the input space of a program into equivalence classes to minimize the number of possible test cases. Test cases are considered adequate even when choosing only one value from each class, since all the values of a class exercise the same functionalities and are considered equivalent. A good test case reveals a class of errors that might otherwise require execution of many cases before observing the general error. Therefore, this method achieves a low level of redundancy in test case selection.

Boundary value analysis complements equivalence partitioning and leads to the selection of test cases that exercise boundary values; that is, values on and near the boundaries of input equivalence classes. In many cases, these values are responsible for erroneous program behavior.

Random testing selects test cases either randomly or by sampling the distribution of all possible input values. It is usually used during the final testing stages.

White-box (structural) testing examines the program structure and derives test cases from program logic.[1] We observe the procedural detail closely and test logical paths (decisions and loops, for example) throughout the software. Unlike black-box testing, which identifies mostly interface errors, white-box testing produces test cases that guarantee

- traversal of all independent paths within a module at least once;
- exercise of all logical decisions on their true and false sides;
- execution of all loops at their boundaries and within their operational bounds; and
- exercise of internal data structures.

The flow graph[10] is a common graphical representation of the control flow. A flow graph consists of nodes representing one or more program statements that execute in sequences and arcs (called edges) that represent the flow of control. These edges are analogous to flow chart arrows. A flow graph depicts all theoretically possible paths in the program, even if some of them cannot be traversed because of the specific combinations of conditions in the program's decision statements.

A flow graph consists of a single start node and one or more end nodes. Any other node lies on at least one path between the start node and at least one end node. A flow graph differs from a standard low-level flowchart in that it emphasizes decision and branch statements as the graph nodes—the critical points in a program's control logic.

White-box testing determines a finite number of paths (white-box test cases). Afterwards, the successful execution of these paths according to the appropriate test inputs will cover the flow graph. It is important to determine the degree of graph coverage that affects the global testing coverage. White-box testing methodology consists of four phases.

First, we construct the flow graph directly from the source code. Second, we select a finite set of paths of the flow graph according to one or more coverage criteria. The most common ones (from the weakest to the strongest) are

- Statement coverage requires all statements in the graph to be executed at least once.
- Node coverage requires the test to encounter all decision node entry points.
- Branch coverage requires the test to encounter all exit branches of each decision node. This criterion is considered to provide the lowest acceptable confidence level for the white-box approach. It includes the previous two criteria, since it requires executing every statement and encountering every node by exercising each branch in a program.
- Path coverage requires all possible paths to be executed. This is the strongest but least practical criterion since the combination of all individual paths increases exponentially with the number of decision statements. In addition, infinite loops make the number of possible paths unbounded, so the method considers equivalent all paths that differ only in the number of loops.

Third, we generate test cases. This is the most complicated phase and concerns the determination of the test inputs that cause execution of the previously selected paths. Last, we execute the program using the test cases and compare the actual output to the expected (specified) output.

The incremental approach tests a module in combination with the set of previously tested ones. This constructs and tests the final program incrementally and systematically. Such an approach emphasizes testing the interfaces among the combined modules. The main incremental-integration approaches are top down and bottom up.

Top down. We integrate modules by moving downward through the control hierarchy, beginning with the main control module (main program). The integration process takes five steps:

1. We use the main control module as a test driver and substitute stubs for all modules directly subordinate to the main control module.
2. Depending on the integration approach selected (that is, depth or breadth first), we replace subordinate stubs one at a time with actual modules.
3. We conduct testing.
4. After completing each set of tests, a real module replaces another stub.
5. Testing continues with regression testing (that is, conducting all or some of the previous tests) to ensure that the testing process has not introduced new errors.

The process continues from step 2 until we have built the entire program structure. This strategy should be used with top-down design. Strict top-down testing can be difficult because it may be virtually impossible to produce a program stub that accurately simulates a complex function.

Bottom up. This method involves testing modules starting at the lower levels of modules and moving upward in the hierarchy. As we integrate modules from the bottom up, processing required for modules subordinate to a given level is always available, eliminating the need for stubs. On the other hand, we must construct drivers to present lower level modules with appropriate input. Bottom-up integration calls for

1. combining low-level modules into clusters that perform a specific software subfunction;
2. writing a driver to coordinate test case input and output;
3. conducting cluster testing; and
4. removing the drivers and then moving upward in the program structure (by combining clusters).

The advantages of top-down testing constitute the disadvantages of bottom-up testing and vice versa. In general, the lack of stubs in bottom-up testing makes test case design easier.

We discuss other aspects of real-time software testing in the Host, target, and behavioral testing box.

Host, target, and behavioral testing

Usually, real-time software testing involves host and target computers.[11] The latter is the real-time system controlling the activities of an ongoing process, while the former constructs programs for the target and is usually a commercially available computer. Such a computer usually contains a cross compiler and/or cross assembler, a linking loader for the target, and an instruction level simulator. The characteristic phases of typical real-time software testing are host and target testing.

Host testing's goal is to reveal errors in software modules. Most of the techniques we use for testing on a host computer are the same as for non-real-time applications. The full system is rarely tested on the host, as we can discover only logical, and not timing, errors. An instruction level simulator may detect some target-dependent errors and errors in support software (for example, in the compiler target-code generator or assembler) on the host.

In target testing, we conduct individual function testing first, followed by module and integration testing, which is sometimes performed using an environment simulator to drive the target computer. An environment simulator is an automated replication of the external system world and is of most use in testing real-time applications in which a specific environment exists and has been specified.

A commonly used practice in testing a real-time system is to examine the system's reaction to a single event (behavioral testing). After testing a single execution path, we can then introduce multiple events of the same class without introducing events of any other class. The process continues to test a single class at a time, and then progresses to more than one class of events occurring simultaneously, in random order and with random frequency. At this stage, we should introduce new event tests gradually so that we may localize system errors.

CLEAR RTOS description

To describe the basic characteristics of CLEAR RTOS,[6] we first logically divide the system into two parts:

- The high-level part is independent of underlying hardware and consists of two layers (system calls, which are accessible by the user, and internal functions). Applications may interact with RTOS through different profiles, according to specific required services and size limitations. These profiles are the basic system and Posix-compliant profiles.[12] (Posix refers to IEEE Std 1003.1, Portable Operating System Interface for Computer Environments.)
- The common, low-level part consists of hardware-dependent functions on which we can implement the high-level machine-independent ones as a library.

This scheme allows RTOS to run over a variety of processors as only the common low-level functions have to be ported to each target. Example processors are the ARM6, ARM (Advanced RISC Machines) Limited's general-purpose, 32-bit RISC microprocessor, and SGS-Thomson's ST9, a 8-/16-bit microcontroller.

The architecture of the OMI/CLEAR real-time libraries supports two execution modes: user mode, which does not allow access to the hardware, protected memory areas, or registers; and system or supervisor mode, which the system uses to perform all crucial operations. In the double-execution mode, tasks are only allowed to access the system in a controlled way through a system call, which involves a trap, a change in the processor's priority, and a stack switch. This method increases system security but affects execution speed. On the other hand, application programmers may select a single-mode scheme in which all parts of the application (even system-independent ones) execute in system mode.

Finally, RTOS offers a variety of services for scheduling (including conventional, real-time, priority-based, and preemptive scheduling algorithms such as FIFO, round robin, and rate monotonic), task management, synchronization (using semaphores or events), memory management, intertask communication (via signals or ports), input/output, and interrupt handling, to name a few.

Each group of services supports different configuration levels, which allows the user to customize services and avoid wasting space and time, as the system does not allocate memory or link code for unwanted resources. For example, if an application does not require buffer pools, selecting the corresponding configuration level and rebuilding the system will exclude all code related to buffer pools. Users can follow the same procedure to select scheduling services; five configuration levels are available:

- Explicit scheduling—The running task explicitly reactivates another task. This option does not support the system clock.
- Priority scheduling—The scheduler works on a priority basis, handling tasks residing in multiple priority queues, and is activated by explicit calls to resume and suspend. This option also does not support the system clock.
- Complete scheduler—This option supports a complete time slicing and round-robin scheme.
- Special behaviors—Users can dynamically define scheduling policies, even on a per-task basis (preemption can be disabled, for example).
- Task accounting—An accounting mechanism traces the number of times a task has been scheduled and the number of clock ticks it has been running.

RTOS testing methodology

Our methodology uses the single-mode scheme to allow direct access to the whole system. Tests cover the basic system and common low-level functions on both the ARM6 and a DOS-based host computer (using the emulator from the JumpStart development environment). We have also tested the Posix functions on a DOS-based host under simulation, as the ST9 was unavailable at the time of test. As a result, we have not tested certain Posix and common low-level functions specific to the ST9.

The checklists we produced contain explicit references to different configuration levels and options with different expected program behaviors only when the specifications contain such references. In all other cases, we selected the most complete system configuration to test all of the source code applicable to the ARM6 target processor. In the source-code listings for the white-box tests, boxes enclose sections of code corresponding to a particular configuration option.

The strategy for testing CLEAR RTOS includes individual (unit) function testing, as well as incremental bottom-up module and integration testing. We first identify all functionalities from the specifications and form checklists,[6,8] then classify each function and consequently all its functionalities into one of the following functional areas: system management, scheduler, task management, interrupts, memory management, semaphores (Posix mutual-exclusion mechanisms), message queues (Posix mailboxes), input/output, time management, signals, or events.

Module and integration testing follows test case design and the individual testing of each function. Every module is integrated into the system when ready. Meanwhile, a module's functionalities (if needed) are temporarily replaced by an appropriate stub or driver to ensure correct coordination among modules. Our method adopts an incremental bottom-up approach because the development is bottom-up, from low- to high-level parts, and testing and development activities must proceed, to a certain degree, in parallel. This strategy makes test case design easier and finds errors in critical modules earlier.

We have not considered the pure top-down approach suitable since our project's timing constraints require proving the feasibility and practicality of the most critical modules (the scheduler, for example) early. Sometimes, a lower level module is not available when we are testing higher level ones that need it. This is why we use a modified bottom-up technique, when deemed necessary, to minimize delays.

Our method treats every function of RTOS as a black box. Thus, its aims are to construct tests for each independent entity according to its specifications and determine whether the input/output behavior of each function meets its specifications. Black-box testing alludes to tests we conduct at each function interface, and demonstrates that all functions are operational (meaning that input is properly accepted, and output, both in the form of actual returned values and side effects, is produced correctly). A point of interest in black-box tests of individual functions is that many RTOS functions perform certain operations not by themselves, but by calling other functions. This stems from the highly modular nature of RTOS, which is necessary to provide added flexibility and support for multiple targets. As a result, we have to enhance the aforementioned generic definition of a function's interface to address this type of function. So, the presence of function calls in black-box individual function tests is justified.

For white-box testing, we selected the branch coverage criterion to handle the size and complexity of RTOS. Ideally, we would use the white-box approach to test both individual functions and modules. However, as modules get more complex, white-box testing becomes impractical because it requires defining all logical paths inside each function, exercising them, and evaluating the results. So the white-box technique is only for the individual function tests of all basic system, Posix, and common low-level functions (except those coded in assembly language).

We perform the test of each individual function or module both positively (testing on normal inputs) and negatively (testing the system's reaction to abnormal inputs). Furthermore, if we can identify input cases not reported in the specifications, we design unspecified test cases.

For unspecified test cases, we should make a distinction between system calls and internal (kernel) functions, as there is a different approach between the design and testing points of view. Although it may be a design decision that RTOS performs no parameter checking for internal functions, it is important for testing to remain consistent and exercise every function using incorrect input, including functions that the user may not directly call. It is safer to check each function independently, without making any assumptions as to whether it is called by another function with correct or incorrect parameters. Bearing this in mind, the unspecified test cases for internal functions do not infer an omission by the design group, but only reflect a design decision. Therefore, we should distinguish these two categories of unspecified test cases.

Table 1. Checklist for sys_create_semaphore function.

Functionality	Description
1	If *count* is illegal, set error_number to BAD_ARGUMENT_1 and return SYSTEM_ERROR.
2	Get the index of the first free entry. If no semaphore is currently available, set error_number to NO_MORE_SEMAPHORES and return SYSTEM_ERROR.
3	Save the index of the next free semaphore.
4	Initialize the semaphore counter with *count*.
5	Mark the semaphore table entry as used by a user semaphore and return its identification number.

Thus, a carefully selected and systematic combination of black- and white-box testing maximizes test coverage while keeping test complexity at an acceptable level.

Examples of individual function testing

We present examples of both black- and white-box individual function tests of the same function (sys_create_semaphore). These examples show that the methods discussed earlier are not alternative ones, but investigate different aspects of the tested function.

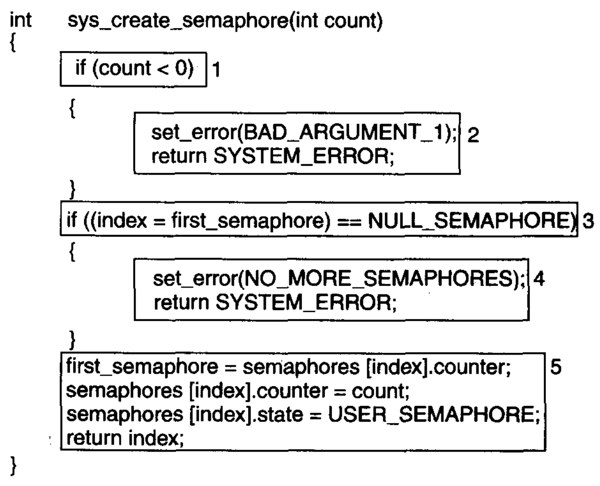
Black-box testing. To test each function, we construct a checklist and test specification table (see examples in Tables 1 and 2). The checklist contains the function operation, iden-

Table 2. Black-box test specification table for sys_create_semaphore function.

Tested functionalities	Given input	Type	Expected output	Actual output	OK/not OK
1	*Count* = –32768/–1	N	SYSTEM_ERROR (error_number should change to BAD_ARGUMENT_1)	SYSTEM_ERROR (error_number has changed to BAD_ARGUMENT_1)	OK
2	*Count* = 0/32767 (no free semaphore slots)	N	SYSTEM_ERROR (error_number should change to NO_MORE_SEMAPHORES)	SYSTEM_ERROR (error_number has changed to NO_MORE_SEMAPHORES)	OK
4 5	*Count* = 0/32767 (free semaphore slots; first free semaphore id = 5)	P	Semaphore_id (created semaphore's counter should be set to *count*; the semaphore should be marked as user semaphore)	5 (created semaphore's counter has been set to 0/32767; the semaphore has been marked as user semaphore)	OK

tification, arguments, return value, and a table of functionalities (functionality numbers and descriptions). The test specification table lists tested functionalities, given input, type (positive, negative, or unspecified), expected output, real output, OK or not OK.

An actual test, which is represented by a row in the test specification table, proceeds as follows: We examine the function's operation with the given input. Whatever is found in parentheses in the given input column corresponds not to an actual argument of the function, but to either a variable or a condition that affects the functionalities under test. Thus, its definition is necessary for test execution. The equivalence-partitioning approach has identified two equivalence classes for the input argument count, one in which count is greater or equal to zero and one in which it is negative. Since count is an integer (and represented by 2 bytes), the first equivalence class ranges from 0 to 32,767 and the second from −32,768 to −1. In these ranges, we selected the two boundary values (instead of using random ones) to account for the boundary value analysis complement to the equivalence-partitioning approach.



```
int    sys_create_semaphore(int count)
{
        if (count < 0)   1
        {
              set_error(BAD_ARGUMENT_1);   2
              return SYSTEM_ERROR;
        }
        if ((index = first_semaphore) == NULL_SEMAPHORE)   3
        {
              set_error(NO_MORE_SEMAPHORES);   4
              return SYSTEM_ERROR;
        }
        first_semaphore = semaphores [index].counter;   5
        semaphores [index].counter = count;
        semaphores [index].state = USER_SEMAPHORE;
        return index;
}
```

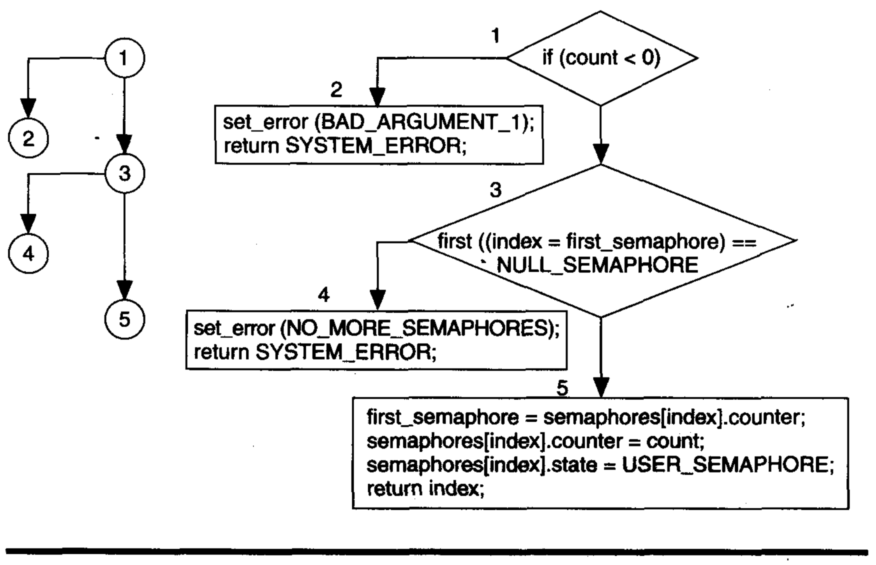
Figure 1. Source code for sys_create_semaphore.

The expected-output column presents the expected results of the function's execution with the given input. These results consist of the function's return values (if any) plus certain actions (enclosed in parentheses) that have taken effect. The table does not present variables appearing inside the function's code because the black-box approach does not consider them. However, a variable altered by side effects, according to the specifications, may appear in parentheses (as it is not an actual return value). The examples in Tables 1 and 2 show that black-box testing cannot check the third functionality "save the index of the next free semaphore," since variables inside the function code are not monitored.

Function sys_create_semaphore creates a semaphore. The argument *count* represents the semaphore counter, and the function returns either the semaphore identification number or SYSTEM_ERROR if the creation fails.

White-box testing. In white-box testing, the only difference in the structure of the test specification table is in the first column, now called tested paths because the focus is on exercising control-flow paths instead of specific functionalities. In addition, the contents of such a table differ because we must observe all variables inside the function's code and the table must present their values. From the function's source code (Figure 1) we construct the corresponding flow graph (Figure 2). Next, we select the test paths according to the branch coverage criterion. In this simple example, the selected paths are actually all possible ones. We monitored the execution of the function on the test cases using the ARM6 debugger to determine the exact path traversed.

We used the McCabe metric v (cyclomatic measure) to determine RTOS complexity and consequently the complexity of testing.[9] The metric's form is $v(F) = d + 1$, where F is the program's flow graph and d is the number of binary decision nodes in F. This measure is an upper bound on the



1
2
3
4
5
1
if (count < 0)
2
set_error (BAD_ARGUMENT_1);
return SYSTEM_ERROR;
3
first ((index = first_semaphore) ==
NULL_SEMAPHORE
4
set_error (NO_MORE_SEMAPHORES);
return SYSTEM_ERROR;
5
first_semaphore = semaphores[index].counter;
semaphores[index].counter = count;
semaphores[index].state = USER_SEMAPHORE;
return index;

Figure 2. Flow graph for sys_create_semaphore.

Table 3. White-box test specification table for sys_create_semaphore function.

Tested paths	Given input	Type	Expected output	Actual output	OK/not OK
1-2	*Count* = –32768/–1	N	SYSTEM_ERROR (*error_pointer ← BAD_ARGUMENT_1)	SYSTEM_ERROR (*error_pointer = BAD_ARGUMENT_1)	OK
1-3-4-5	*Count* = 1/32767 (no free semaphore slots)	N	SYSTEM_ERROR (*error_pointer ← NO_MORE_SEMAPHORES)	SYSTEM_ERROR (*error_pointer = NO_MORE_SEMAPHORES)	OK
1-3-5	*Count* = 1/32767 (free semaphore slots; first free semaphore id = 5)	P	Semaphore_id (semaphores[5].counter ← 1/32767) (first_semaphore ← 6) (semaphores [5].state ← USER_SEMAPHORE)	5 (semaphores [5].counter = 1/32767) (first_semaphore = 6) (semaphores [5].state = USER_SEMAPHORE)	OK

number of paths (that is, white-box test cases) and satisfies the branch coverage criterion. The cyclomatic complexity metric of the following example, according to the above definition, is $v(F) = 3$. So, the maximum number of paths to examine is 3.

Table 3 demonstrates that white-box testing provides different results for variables inside each function, such as the variable first_semaphore, which corresponds to the third functionality of the Table 1 checklist.

We perform module and integration testing in an analogous way using the black-box technique. Although the tables are larger, the test table structure remains the same, since each module consists of several functions. We consider functions to form a level when called by other functions. An RTOS characteristic that affects module and integration testing is that even the largest modules are not more than four levels deep, while most of them are only two to three levels deep. In fact, functions that call other functions form all of the modules. As a result, the module test of the function at the top of a module may appear, at first, quite similar to the function's individual test. As we discussed earlier, this is not true.

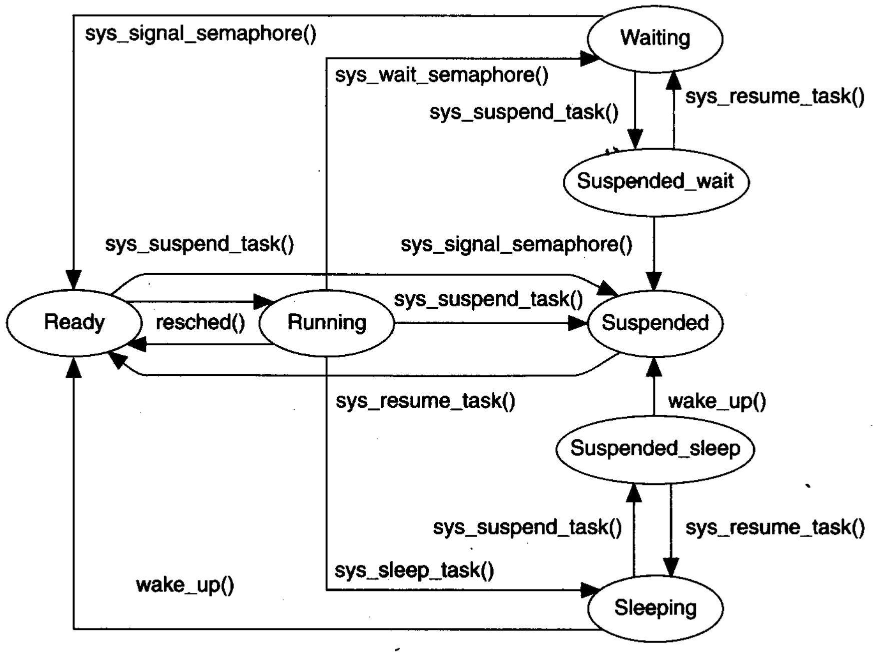
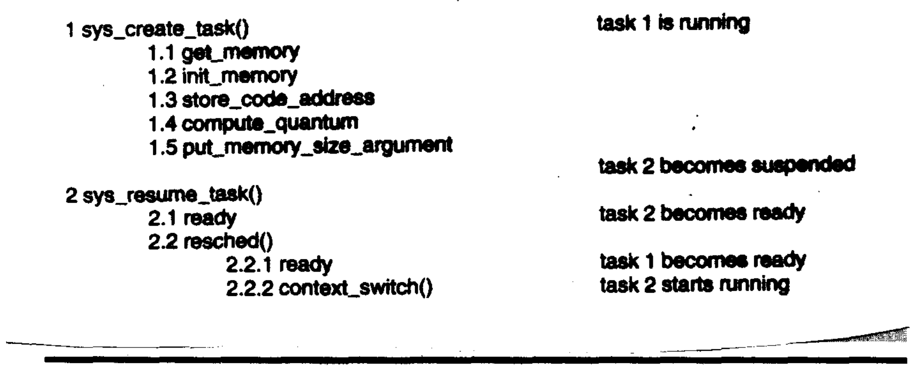


Figure 3. Task state diagram.

Behavioral testing example

In a typical class of events, a new task with a higher priority (task 2) may preempt an earlier one (task 1). It does not matter what the specific tasks are or their priorities, as long as task 2's priority is greater than task 1's. This is because all these events belong to the same class.

Figure 3 shows the possible states of a task in RTOS, as well as the system calls used for the transitions between states. We describe the expected behavior of RTOS as a consequence of this event in Figure 4 (next page). We must introduce the event to test whether RTOS reacts as expected and



```
1 sys_create_task()                              task 1 is running
        1.1 get_memory
        1.2 init_memory
        1.3 store_code_address
        1.4 compute_quantum
        1.5 put_memory_size_argument
                                                 task 2 becomes suspended
2 sys_resume_task()
        2.1 ready                                task 2 becomes ready
        2.2 resched()
                2.2.1 ready                      task 1 becomes ready
                2.2.2 context_switch()           task 2 starts running
```

Figure 4. Execution path.

Table 4. Results of CLEAR RTOS software tests.

Tests	Basic system	Posix support	Low-level	Total
All functions				
OK	60	24	17	101
Not OK	10	21	2	33
Total	70	45	19	134
Black-box functionalities				
OK	209	85	18	312
Not OK	6 (1P, 5N)	48 (9P, 39N)	2 (2P)	56 (12P, 44N)
Total	215	133	20	368
Black-box test cases				
OK	227 (94P, 133N)	86 (58P, 28N)	18 (18P)	331 (170P, 161N)
Not OK	6 (1P, 5N)	52 (12P, 40N)	2 (2P)	60 (15P, 45N)
Total	233	138	20	391
White-box functionalities				
OK	382	158	17	557
Not OK	16 (11P, 5N)	10 (9P, 1N)	1 (1P)	27 (21P, 6N)
Total	398	168	18	584
White-box test cases				
OK	243 (126P, 117N)	115 (76P, 39N)	12 (12P)	370 (126P, 50N)
Not OK	16 (11P, 5N)	12 (11P, 1N)	1 (1P)	29 (23P, 6N)
Total	259	127	13	399

then use the ARM6 debugger to monitor the actual behavior of RTOS, as in white-box testing.

Results

The test result assessment provides evidence of the software's quality and reliability. Of course, we should bear in mind that all errors are not of the same importance. The following tables and graphs present the cumulative results of testing for the total number of functions, functionalities, and paths tested. Of the functions we tested, 18 percent were low-level functions, 30 percent Posix support, and 52 percent basic system. Table 4 presents the results of RTOS testing.

Figure 5a shows the testing results for the number of tested functions. We considered a function not OK even when one of its functionalities or paths is not performed correctly.

Figure 5b presents the results of testing black-box functionalities. The functionality types (positive or negative) are in parentheses.

Figure 5c presents the outcome of black-box test cases. These results are not the same as in Figure 5b, as we test certain black-box functionalities in more than one test case.

Figure 5d presents the results of testing white-box functionalities. These functionalities differ from black-box ones in that they consider the internal operation of each function and thus are more detailed and greater in number.

Finally, Figure 5e shows the results of executing white-box test cases, in which each test case corresponds to a different path. The test cases must exercise at least all the paths according to the selected coverage criterion (branch coverage), and in simple cases we selected all possible paths, according to the most-complete path coverage criterion.

In individual function testing, the average value of the McCabe metric we have encountered is about 5, while its maximum value is 10. These values reveal only part of the effort devoted to testing activities, and we must consider the total size of the tested code (in this example, 720 Kbytes of C source code).

In general, discovering few errors does not imply inadequate testing, but just that CLEAR RTOS runtime is very stable. This is especially true of its basic system profile, because we can identify most of the discovered errors as specification rather than implementation error.

Nor does this fact imply that the Posix code is of lower quality. We tested early versions of Posix functions, and certain argument checks, although specified, had not been implemented.[6] The low percentage of erroneous white-box test cases (Figure 5e) supports the quality of Posix code. White-box test cases are based on coded operations rather than on those that are specified but unimplemented. In addition, the code structure was not final because there is the option of omitting certain functions to save space.

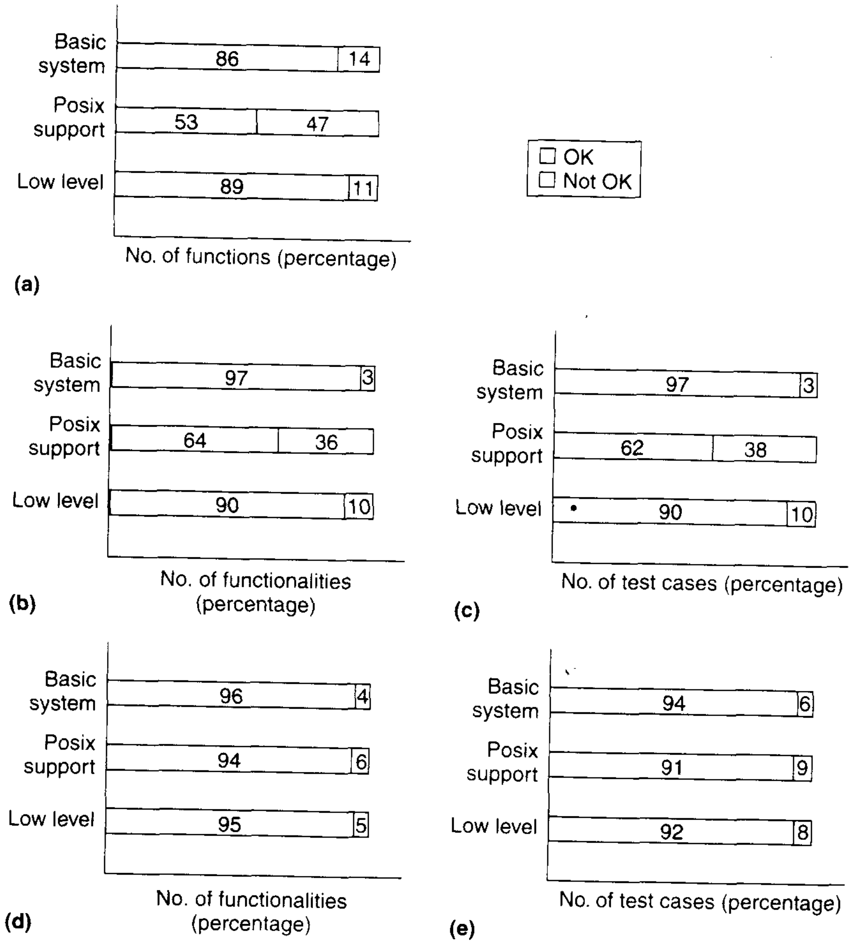


Figure 5. Graphical representation of results in Table 4: all functions (a), black-box functionalities (b), black-box test cases (c), white-box functionalities (d), and white-box test cases (e).

TESTING RTOS is an ongoing activity, and we will present a more complete evaluation soon. Until now, experience from the application of our testing methodology confirms some general assertions:

- An independent group should perform testing since it is not easy for the development group to perform objective and highly error-revealing tests.
- A method should use both black- and white-box techniques, at least in the unit testing phase, since they reveal different kinds of errors.
- We should give special emphasis to selecting the appropriate coverage criterion, which assures an adequate confidence level while limiting testing effort to an acceptable level.
- Following an incremental bottom-up integration strategy is best, since the development phase is also bottom-up and proceeds in parallel with testing.
- A test strategy also requires behavioral testing to examine a real-time operating system's reaction to events from as many event classes as possible.

Finally, following a systematic testing approach not only ensures a predefined reliability level, but also permits the best possible organization of work from the project management point of view. This is particularly important for large projects in which several partners rely on each other's results and time constraints are strict.

Acknowledgments

This project received 50 percent of its funds from the Commission of the European Union.

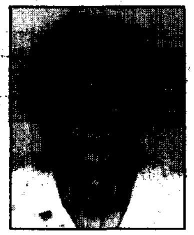
References

1. R.S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, 1992.
2. R.S. Freedman, "Testability of Software Components," *IEEE Trans. Software Eng.*, Vol. 17, No. 6, June 1991, pp. 553-564.

3. G.J. Myers, *The Art of Software Testing*, John Wiley & Sons, New York, 1979.
4. I. Sommerville, *Software Engineering*, Addison-Wesley, Reading, Mass., 1992.
5. J.A. Wise, V.D. Hopkin, and P. Stager, eds., "Verification and Validation of Complex Systems: Human Factor Issues," NATO ASI Series F, Vol. 110, Springer-Verlag, Berlin, 1993.
6. C. Farris and P. Petit, "Final Specification for the Real Time Runtime Support for Deeply Embedded Applications," Esprit Project 8906-OMI/CLEAR, Tech. Report 4.1.2-01, Etnoteam S.p.A., Milan, Italy, Dec. 30, 1994.
7. V.C. Gerogiannis and M.A. Tsoukarellas, "SAT-A Schedulability Analysis Tool for Real-Time Applications," *Proc. Seventh Euromicro Workshop on Real-Time Systems*, IEEE Computer Society Press, Los Alamitos, Calif, 1995, pp. 155-159.
8. R.A. DeMillo et al., *Software Testing and Evaluation*, Benjamin/Cummings Publishing Company, Reading, Mass., 1987.
9. J.P. Myers, Jr., "The Complexity of Software Testing," *Software Engineering J.*, Jan. 1992, pp. 13-24.
10. V.C. Gerogiannis, K.D. Economides, and M.A. Tsoukarellas, "Runtime Validation Report," Esprit Project 8906-OMI/CLEAR, Tech. Report 6.1.1-01, Advanced Informatics Ltd., Patras, Greece, June 29, 1995.
11. R.L. Glass, "Real-Time: The Lost World of Software Debugging and Testing," *Comm. ACM*, Vol. 23, No. 5, May 1980, pp. 264-271.
12. *IEEE Std 1003.1b-1993, Standards for Information Technology—Portable Operating System Interface (Posix)—Part 1: Application Program Interface (API) [C Language]—Amendment: Realtime Extensions*, IEEE, Piscataway, N.J., 1993.

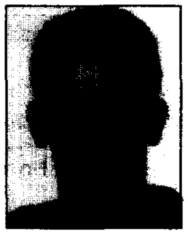
Manthos A. Tsoukarellas is a professor of informatics at the Patras and Mesologi Technological Education Institute in Greece. He is also founder, president, and executive director of Advanced Informatics Ltd. His research interests include the monitoring and performance evaluation of real-time systems.

Tsoukarellas served as an evaluator of the Esprit proposals for the European Commission and has been involved in several Esprit projects.



Vasilis C. Gerogiannis is a PhD student at the University of Patras and also works for Advanced Informatics Ltd. Specification mechanism for real-time systems, real-time scheduling, and software testing are his special interests.

Gerogiannis holds an MS in computer engineering and informatics from the University of Patras.



Kostis D. Economides is currently with Advanced Informatics Ltd., where he is responsible for the Esprit III OMI/CLEAR project's testing activities. His current research interests include software testing, broadband networks, and networking interconnectivity.

Economides graduated from the University of Patras, with a BS in computer engineering and informatics.

Direct questions concerning this article to Manthos A. Tsoukarellas, Advanced Informatics Ltd., 35 Gounari Ave., 26221 Patras, Greece; manthos@advinfo.pat.forthnet.gr.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 165 Medium 166 High 167