Testing a Communications Driver for the IEEE 1394 High Speed Serial Bus Standard

C. Koutsikas[1], D. Staikos[1], K. Bitsakos[1], N. Papaspyrou[1], N. Malevris[2]

[1] National Technical University of Athens, Greece
{koutsi,dstaikos,kbits,nickie}@softlab.ece.ntua.gr

[2] Athens University of Economics and Business, Greece
ngm@aueb.gr

Abstract

Software testing is a difficult and tedious task. For a network driver, software testing is still more laborious. The main reason for this is that a communications driver does not act as a stand-alone application used by an end user. A communications driver acts as a part of the operating system kernel and is used by other operating system components and/or applications through a general purpose *Application Programming Interface* (API). This paper discusses the experience that has been gained by testing a communications driver that provided a general purpose API to both kernel and user mode components. The driver was implemented on the Windows NT 4.0 and Windows 98 operating systems, and its underlying medium was an IEEE 1394 network.

1. Introduction

Software testing is a hard and laborious task. Its goal is to ensure the proper functionality of their software by locating and removing errors, thus "ensuring" its correctness. This is the most important factor in developing high quality programs. Many methodologies have been developed in order to make the testing process easier and more effective. Towards the same direction many tools have been constructed to automate and support the software testing process. No matter how many attempts have been made to solve the testing problem, in practice it is still one of the most painstaking phases in a software product's life cycle. The main reason is that each product is different, and therefore, different criteria may be required to determine a product's correctness and quality.

The current paper discusses the experience that has been gained by testing a communications driver that provided a general purpose API to both kernel and user mode components. The driver was implemented on the Windows NT 4.0 and Windows 98 operating systems, and its underlying medium was an IEEE 1394 network. The communications driver loads as part of the operating system kernel. It is used by other operating system components and/or applications, usually through a general purpose *Application Programming Interface* (API). The whole product consists of three subsystems: the functions of the 1394 Class Driver (kernel-mode interface), the functions of the user-mode support interface (user-to-kernel translation layer) and the functions of the user-mode DLL (FireAPI). Directly testing the kernel interface is a time-consuming task, since it requires the development of test cases in

kernel mode, and kernel-mode programming is not something that the average test professional has in his tool chest.

This forced the software testers to seek alternative ways of testing the 1394 Class Driver. In most cases the user-mode API layer simply acts as the gateway to the respective kernel-mode function. More than 95% of the kernel mode functionality is exposed to the user-mode layer. This consideration led the software testers to the decision to concentrate their testing on the user-mode API, which allows for considerably easier construction of "testing drivers".

Actually, the correctness of the functions of the kernel-mode API reflects on the correctness of the functions of the user-mode API layer. It is impossible for a user mode function to operate correctly if the corresponding kernel function does not.

The responsibility of unit testing the functions of the kernel-mode components was left to their developers.

In this paper the following definitions will be used:

- *Device driver:* A low level module (that is running with kernel mode privileges) acting as a medium between the operating system and a hardware device.
- *Process*: A running application that consists of a private virtual address space, code, data, and other operating system resources, such as files, pipes, and synchronization objects that are visible to the process. A process also contains one or more threads that run in the context of the process.
- *Thread*: A unit of execution that belongs to a process. A thread can execute any part of an application's code, including code that is currently being executed by another thread. All threads in a process share the process' virtual address space, global variables, and other operating-system resources.
- *Node*: A node typically is a 1394 network card. More than two such cards may reside on the same computer. However, a node may be any device capable of communicating over 1394, typically consumer electronics like digital cameras, printers, scanners, etc.
- *Data structure*: A block of memory that contains data in a specific format. E.g. a linked-list is a data structure. So is an integer or a more complex structure consisting of a character, a string and a pointer.
- *Priority level:* A scheduler-dependent feature. It determines things like how much CPU time a thread has before it gets pre-empted, whether it can be pre-empted by other threads or interrupts, etc.

The driver loads at boot time and exposes a set of functions to user mode applications and device drivers. Although testing the functions of the user-mode API layer is a much easier task than testing the 1394 Class Driver, the application of existing techniques for black and glass box testing [1,2,3,4] was difficult, because of the product's special characteristics:

- A general purpose API allows "arbitrary" usage, rather than the much more predetermined usage required by drivers that only act as components of the operating system (for example an NDIS driver).
- It allows both isochronous and asynchronous network communication.
- Its functions are fully reentrant.
- Processes share common data structures.
- Per-process data structures are shared among all threads of a process.
- Protocol operation is timing sensitive.

- External events affect the driver's operation (dynamic topology changes, network traffic, hardware failures, etc).
- The Class Driver optimizes the 1394 protocol's operating parameters depending on the physical topology of the network.
- High performance is a crucial aspect (400 Mbps data rate). This requires maximizing concurrency and minimum interdependencies, which leads to complex program code.

Additionally, most product specifications are related to more than one function, making the integration and system testing a complicated process. Considering this situation, the software testing team decided to apply the ANSI/IEEE Standard [5,6] as a framework for testing the product.

According to this standard the test case construction is based on the characteristics of a unit or a group of units, where a unit can be a function or a group of functions. These directives assist the testers to generate and document many test cases for the test requirements generated by the product's special characteristics. Heuristics were developed to derive test cases from the test requirements. Tools were used to support the testing process, either for documentation or for observing the expected or unexpected (memory leaking) results. Additionally 'test drivers' were constructed to facilitate the test procedure by automating batch test execution.

This testing approach became a guidebook for the designation of many test cases. Approximately 250 bugs were detected by testing the functions of the user-mode API layer, the majority of which were attributed to the Class Driver. However, in some cases the problems were caused by shortcomings in the user-mode code. The final result was the construction of a product that appears to be highly reliable, judging from the relatively small number of bugs (40) that have been found by the customers during its usage for over a year.

The organization of the paper is as follows. The basic definitions related with a communications driver are introduced in the next section. Section 2 introduces the basic principles of the IEEE 1394-1995 standard. Section 3 presents the special characteristics of the API library. Section 4 describes the framework used in testing and section 5 presents the problems encountered therein. In section 6, the results of testing the communications driver are discussed. Section 7 designates benefits and learning objectives and finally the concluding remarks are outlined in section 8.

2. The IEEE 1394 High Speed Serial Bus Standard

IEEE 1394 is a high-speed serial bus interface standard, which combines the multimedia consumer, the computing and the industry area. At the end of the 80's Apple developed the high-speed data transmission system and trademarked it as "FireWire". In 1995 the IEEE (Institute of Electrical and Electronic Engineers) standardized the high performance serial bus [7].

Today many electronic devices, including computers, printers, digital cameras, camcorders, monitors, VCRs, scanners, HDDs and other storage media, can be interconnected on the same local network and communicate using this technology. IEEE 1394 is a *universal interconnect* solution that has been designed to bring together a variety of peripherals and consumer devices on the same network. The current implementations support data rates of 100, 200 and 400Mbps, and the

800Mbps standard are in draft stage. Additionally, all future speed improvements will be made backward compatible with the currently available data rates. This allows a scalable topology capable of supporting multiple data rates. For example, the IEEE 1394-1995 architecture supports 100, 200 and 400 Mbits/sec nodes in the same network.

The key-enabling feature of this architecture is the ability to support guaranteed-bandwidth for data transmission with very low overhead. This capability is extremely important for applications in which accurately-timed transfers are needed, such as digital audio and video applications. This feature is one of the reasons why this network is well suited as a multimedia interface for both the computer and the consumer environments. In addition to the isochronous transfer capability there is the asynchronous operation mode. In this mode a packet of data is sent from one node to another and the receiving node generates a physical layer acknowledgement (i.e. the packet was physically correct – the bits are valid) and a transaction layer acknowledgement (i.e. the packet was logically correct – the packet contents are reasonable).

Isochronous transfer is used for data that need guaranteed bandwidth (or precise timing) but it cannot guarantee delivery because isochronous packets don't get any acknowledgements. When the sender wants guaranteed delivery of data, then asynchronous transfers should be used.

Another very important feature of this protocol is that 'Live' cable insertions and removals can be performed without damaging the network (hot-plugging). The protocol actually specifies how to reconfigure the network on-the-fly whenever a node is added or removed from it. Physical addresses are automatically reassigned when the topology is changed (the bus is self-configurable). IEEE 1394 is based on a "memory-mapped" architecture in which all resources are viewed as memory locations. This makes data transactions more efficient and simplifies software driver designs.

Finally, there is a number of other features that make this protocol unique. All nodes are practically equal and every node that is "capable" can be the "root" node and/or the "Cycle Master" node of the bus. There is no need for a "master" unit as is the case with USB for example. Each node can be powered through the bus, which means that small devices (i.e. a desktop camera) need no external power supply. There can be up to 63 devices on a 1394 bus, and up to 1023 different buses interconnected with 1394 bridges.

3. The 1394 Class Driver

As previously mentioned, the Class Driver (Fig.1) loads as part of the operating system kernel and is used by other operating system components and/or applications, usually through a general purpose *Application Programming Interface* (API). It exposes around 80 functions divided to categories according to their functionality. There are functions for hardware initialization, input/output processing (receive/transmit), implementing the transaction layer logic, controlling the isochronous transmission and monitoring network events.

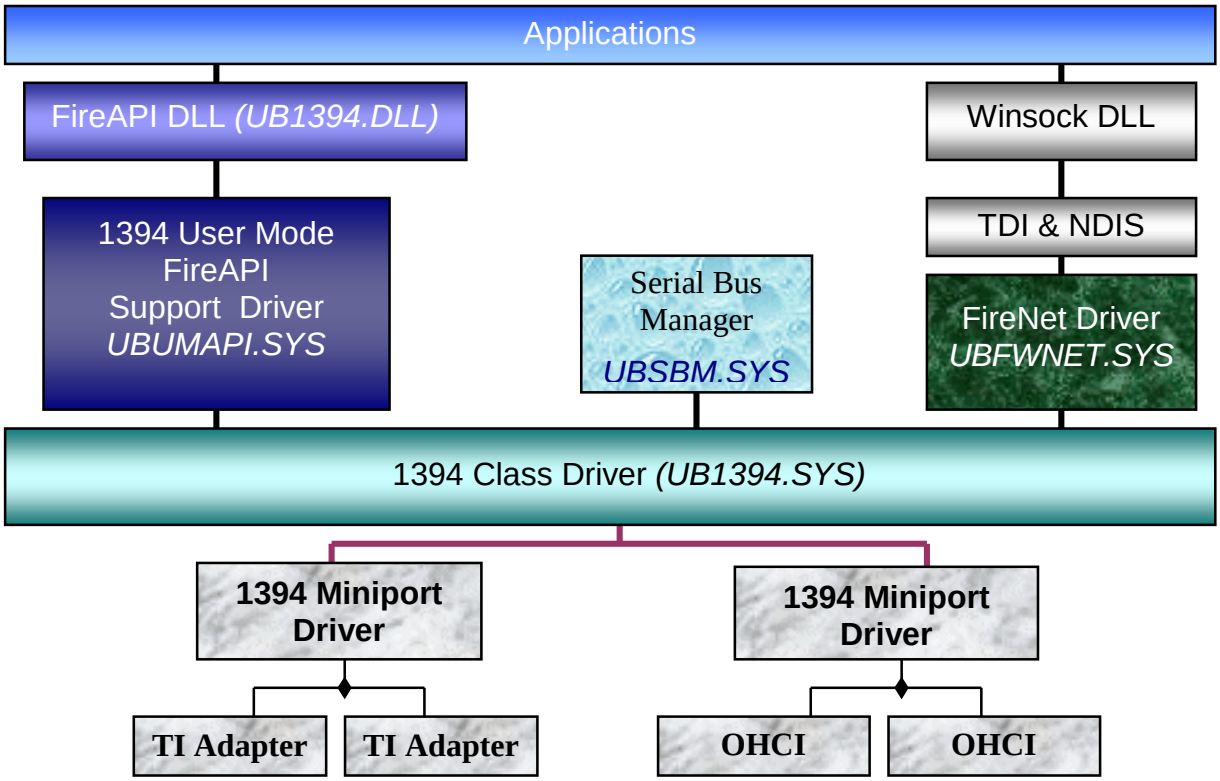

Fig. 1 Architecture of the IEEE 1394 network driver

In an earlier section we listed some of the product's special characteristics. These characteristics lead to serious difficulties in the testing process:

- Many processes can use the user-mode API layer concurrently. The functions affect a set of data structures that are 'shared' among all processes use the API. This means that the results that a process will get from calling the functions may depend on what other processes using the API are doing, or what earlier runs of the same process have performed.
- The driver implements an asynchronous network protocol, which in most cases implies that the entities involved in a single logical operation (e.g. an asynchronous transaction) may reside on different nodes. This means that the code for a single test case is distributed across two or more computers. Moreover the network topology is an important factor that has to be taken into consideration.
- Being implemented as a kernel driver, the API can be considered as part of the operating system. This means that the software runs at the highest privilege level and as a result any problems/bugs are either fatal (operating system crash) or persistent and cumulative (out of memory conditions that make the system useless).
- The functions are fully reentrant. This means that access to the 'shared' data structures must be fully synchronized by the driver's code. However, it is extremely difficult to design program logic that will fully test shared resource access and correct synchronization.
- The driver implements several asynchronous functions, with quite a complicated interface. For example a function may transmit multiple request-packets to different nodes, each of which has to asynchronously

transmit a response-packet. These response-packets may arrive at different times and in different order depending on the state of the requests' receivers. Testing such a case requires the deployment of multiple receivers (part of the test suite) and the coordination of these receivers so that (a) the responses they send to the request transmitter are returned in the same, reversed and mixed order, (b) one or some or all nodes fail to send one or more or all responses, etc.

- The operation of the protocol is inherently timing sensitive. This means that operations that appear to function correctly may break down in situations where considerable delays are introduced, for example due to increased system workload or network traffic.
- The Class Driver optimizes 1394 protocol's operating parameters depending on the physical topology of the network on which the tests are being run. This means that code that appears to work with one set of physical parameters, may fail with another configuration.
- High performance is a crucial aspect of this software. Testing must ensure that unexpected drops in performance do not occur while the system evolves. Maintenance of test measurement data is required if this goal is to be achieved.

4. Testing framework

Considering the characteristics described in the previous section, it is obvious that the traditional software testing techniques cannot be applied (as they are) in testing the API software. Considering this situation, the software testing team decided to apply the ANSI/IEEE Standard [5,6] as a framework for testing the product. According to this standard the test case construction is based on the characteristics of a unit or a group of units. Using these directives the testers planned, generated and documented the test cases and the test procedures for the unit testing as well as for the integration and system testing. The test cases based on the test requirements (generated by the special characteristics of the driver) were constructed by using heuristics that were developed for this purpose. 'Test drivers' were constructed to facilitate the unit test procedure by automating batch test execution.

An overview of the testing life cycle is shown in Fig.2.

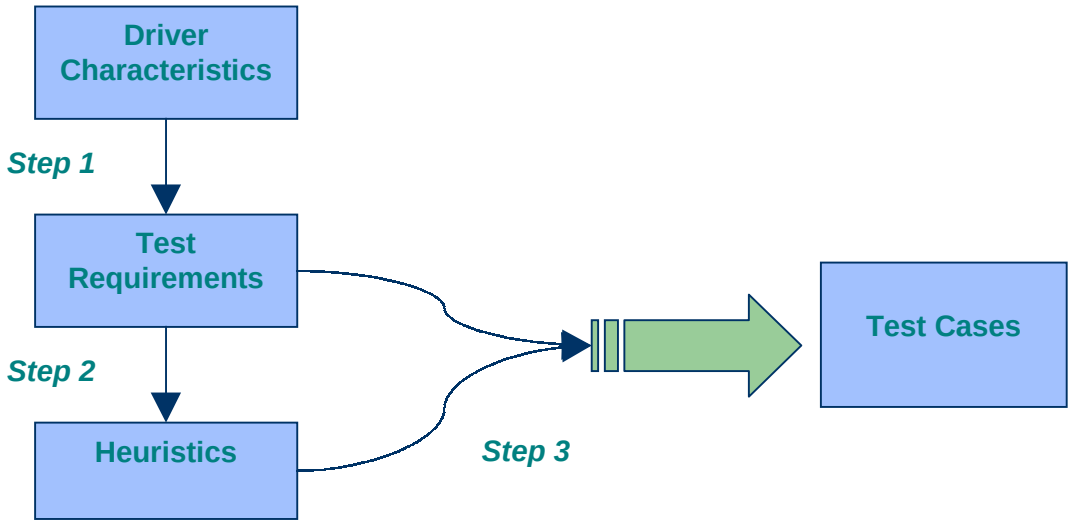

Fig. 2 The testing life cycle

In unit testing, testers generated test cases for each group of test requirements related to a function, and then documented them in a separate Test Design Document [5]. Heuristics were developed to cover the test requirements related to the type of each function parameter. The goal of these test requirements was to cover the bounds of the input data, as well as to uncover invalid test cases. Suggestively, we present some of the heuristics we considered mandatory for the unit testing:

- If the parameter is a pointer to a memory area, then it must take the following values:
  - Null value
  - Pointer to read-only memory area
  - Pointer to read/write memory area
  - Pointer to memory area that has no access rights
  - Pointer to memory area that is partly accessible
  - Pointer to memory area that is inaccessible
  - Pointer to memory area that is less/greater/equal than the needed area
  - Pointer to memory area that is used by another process
- If the parameter is an array, then it must take the following values
  - The same directives as if the parameter points to a memory area
- If the parameter is an integer, then it must take the following values
  - The maximum/maximum plus 1 integer
  - The minimum/minimum minus 1 integer
  - The zero value
- If the parameter is an unsigned integer, then it must take the following values
  - The maximum/maximum plus 1 integer
  - The zero value
  - The –1 value or a negative value
- If the parameter is a handle, then it must take the following values
  - The NULL handle
  - A handle of a different data structure
  - The right type of handle
  - A random value
- If the parameter is a data structure, then we test each field of the data structure following the above directives.

Integration and system testing was more difficult. We had to gather all the testing requirements that included the specifications of groups of functions. To achieve this goal, we tried to find the test requirements for the combinations of two or more functions that had related functionality. We documented this result in separate Test Design Documents, one for each combination. Heuristics were developed to cover the test requirements related to the special characteristics of the driver. The following points outline the most important heuristics:

- Repetitive execution of self-contained tests by a single thread. A self-contained test is a test that should -theoretically- leave the system unaffected after it is completed.
- Open-Close Tests. This is a test primarily for kernel memory leaks. For any set of functions that allocate and free a resource (open/close a handle), the functions are called in a tight loop while the amount of system resources is monitored. The functions should not fail, regardless of the number of calls.

- Many threads of the same process execute any self-contained test concurrently.
- Many different processes execute any self-contained test concurrently.
- Execution of tests while the system is under great CPU/disk stress.
- Execution of tests by low-priority threads.
- In networked tests, the following combinations of speed for the source and destination PCs should be tested:
  `{{slow, fast} source} x {{slow, fast} destination}`
- Networked tests should always be tested through loopback as well.
- All networked tests should be executed over a busy network.
- Several key network topologies are identified and tested against (flat chain, full tree, etc).
- Abnormal termination tests. A process that runs a test is abnormally terminated and restarted repeatedly, so that proper cleanup on the part of the Class Driver can be verified.

5. Difficulties encountered during testing

Though we had a testing framework that guided tester's actions, we encountered problems during testing. Some of these problems were:

Problem 1

Due to the layered structure of the driver, when a problem was detected, sometimes it was difficult to find out which module was responsible. That problem is likely to happen on a layered model driver. In order to prevent such issues the developers had embedded lots of debug and diagnostic messages on every layer of the driver from the very beginning of the development process. These messages made the execution paths "traceable", a fact that considerably helped the testers in locating the source of a defect.

Problem 2

Due to the network nature of the driver, some tests were based on the client/server model. In that case it's difficult to decide if a problem occurred on the client, on the server or on both the client and the server. A very useful tool that helped locating the problem was an IEEE 1394 bus analyser. This is a tool that captures all the packets transmitted through the 1394 local bus, stores them in memory and presents them in an easy readable format. We used the CATC 1394 bus analyser.

The main difficulty of using such a tool is that when a log is collected from the bus analyser it usually consists of several thousands of packets which require manual inspection, which is obviously quite a time consuming process.

An interesting approach that our testers came up with, in order to facilitate the manual analysis of the lengthy logs that were produced by the bus analyser, is the use of 'synchronization' packets. These are 1394 packets with human readable contents that could be used to get the analyser logs in-synch with our test-program logs. This simple idea proved to be a valuable assistant in our testing efforts and we recommend its use to other test professionals in related fields.

Problem 3

The hardware proved to be unreliable several times, especially when new chip models were introduced or adapters were received from new hardware vendors. Erroneous behaviour due to hardware faults can be quite confusing, since such errors

are usually intermittent, occurring at random times. It is quite difficult to diagnose such situations because a test may fail after several thousands of repetitions, and then never fail again for days. The bus analysers have very limited memory (64MB) compared to the available bandwidth (it takes 4 seconds of moderate traffic to completely fill the buffer, and 64MB means tens of thousands of packets) so it is rather difficult to be able to capture and analyse such situations.

Moreover, unexpected hardware behaviour can easily cause the software to crash in completely unpredictable ways, leaving the developers without a clue as to what went wrong. It is often the case that the system is left in an unusable state, which means that it is impossible to peek at the adapter's registers and get some information about the adapter's state at the time of the crash.

Even worse, as experience has shown us, intermittent errors in most cases imply a software defect rather than a hardware problem, while at the same time driver developers demonstrate a psychological tendency to "blame it on the hardware" whenever they are unable to determine what went wrong.

This causes a self-fulfilling prophecy situation. The developer does not believe the software has a bug, so he performs an analysis that validates his assumptions that the software behaves correctly, rather than trying to determine how the software could misbehave.

It is very difficult for humans to fight against such a tendency, even when they know its existence, because they immediately relate any new problem to actual hardware problems that had been discovered in the past and had caused similarly weird symptoms.

The problem that the testers had to face in this case is whether they should include code in their tests to validate (to a certain point of course) that the hardware behaved correctly, or they should assume that the hardware will properly behave at all times (which obviously will have a considerable effect on the way the test code is written).

After several efforts, the testers concluded that it would greatly facilitate their work if they developed a separate test-suite which would test the hardware operations separately, so that they could get a high degree of confidence that the adapters were reliable. Having such a tool would then allow the testers to develop their code with the assumption that it will run on hardware that is operating as expected.

This test suite was written to test the fundamental hardware operations plus some specific hardware errors that had been discovered in the past. Writing a software module that tests the hardware itself is an interesting issue in itself since it is often a recursive problem: How can a software module know if a packet was transmitted correctly without being informed by the hardware? One solution is to install a second adapter on the same PC (or use another PC), which will receive the packet and validate its contents. However, what happens if the packet contents are corrupted? Does this demonstrate a transmission or a reception error? We could then add another receiver on the same bus, usually a different PC, which would allow us to conclude whether we are facing a transmission problem (both receivers get bad data) or a reception problem (one of the receivers gets good data but the other gets bad data).

And how do the 2 PCs get to talk to each other so that they can verify that the test is running OK? Obviously they have to be connected through an independent network (e.g. an Ethernet network), which they can consider to be reliable.

By now it should be clear that such a test is far from trivial to implement correctly. Interestingly enough, the main problem that the testers had to face was reluctance from upper management, which could not get truly convinced that developing a 'hardware-tester' software suite would be valuable enough to justify a couple of weeks dedicated to this task.

6. Testing results

The testing team started the testing process half a year after the implementation of the kernel-mode API layer. The whole testing process (design and implementation) lasted one year. The number of the user-mode API layer, as previously mentioned, was 80 functions. The heuristic approach generated approximately 5000 test cases and resulted in the detection of 250 errors in both the user-mode and kernel mode layers. Considering that the number of lines of code for both layers is 50,000, the testing team succeeded to detect five bugs per each thousand lines of code. The testing team believes that this ratio is a combined result of many factors: a) the nature of the software, b) the experience of the developers, c) the effectiveness of the testing approach and d) the given testing time (one year). The product was used by customer's applications for at least one year. The number of defects that have been reported by the customers is only forty, that means 0,8 bugs per each thousand lines of code. The defects were corrected and updated versions were given back to the customers.

The above results are reported in Table 1. Fig. 3 shows the number of faults detected during the development and maintenance.

Lines of code tested:	50,000
Time of implementation:	1,5 years
Complexity:	80 API functions – 1500 functions
Testing time:	1 year
Number of test cases	~5000 test cases
Number of errors detected:	250 bugs (5 per 1000 lines of code)
Uptime:	1 year
Errors found by customers:	40 bugs (0,8 per 1000 lines of code)

Table 1. Testing results

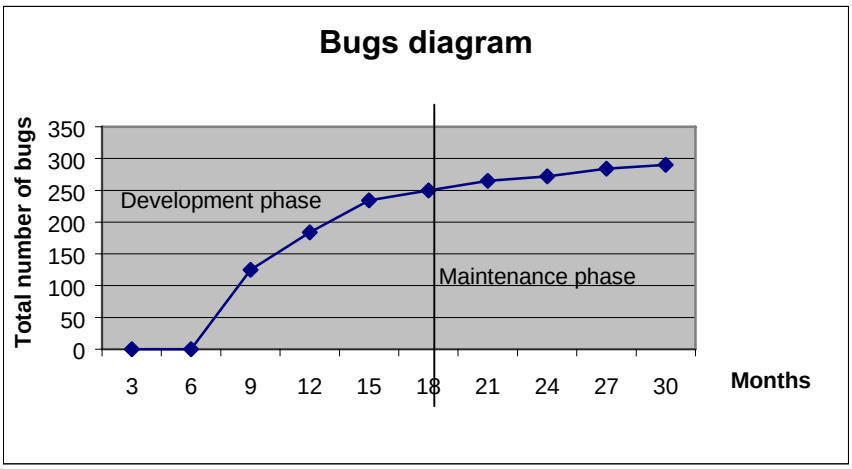

Fig. 3. Number of faults detected

7. Benefits and learning objectives

One of the main goals of this paper is the hope that our approach will help professional testers avoid many of the difficulties in testing similar products. For this

reason the following points outline the main benefits and learning objectives gained by the whole testing process:

- The presence of a user mode DLL greatly facilitates the testing process of kernel modules. It is very difficult for a tester to construct 'testing drivers' at the kernel mode in order to test the kernel-mode functions. Specifying the test requirements themselves is a difficult task. This fact is a result of the special characteristics of a driver. We believe that the notification of the test requirements that we constructed can facilitate the testing community to face similar situations. Traditional testing techniques do not suffice, especially during the integration and system testing phases. Setting up some kinds of test cases can be a very difficult task. The construction of test 'drivers' facilitates the test procedure by automating batch test execution especially in unit testing.

8. Concluding remarks

Testing a software product is a difficult, but necessary task. It is impossible to perform an exhaustive test of a program. Even if the application of a software testing methodology promises good results, the time pressure for a short release of the program is the biggest obstacle. It is almost impossible to achieve an adequately reliable product, unless the testing team is experienced, has a good sense of the program specifications and cooperates closely with the development team. The automated testing tools move towards reducing the testing time and cost. But there are not yet tools able to automate the whole testing process and to facilitate the testing of any kind of software product. Unfortunately, each product has distinct special features that obstruct the use of automated tools or even of a software testing method.

We believe that test professionals should overcome any difficulties by using a combination of techniques, heuristics and tools suitable to the nature of each software product. We also believe that testing specific software needs the creation of test cases designed to cover the special characteristics of such a product. This can be generalized to include similar software products giving rise to new testing methodologies.

Adopting our previously described considerations in testing the IEEE 1394 communications driver we managed to create a heuristic approach that could easily be applied to any similar product. We hope that this approach may assist other test professionals to gain time and effort in testing similar complicated software products.

9. References

[1] Brian Marick, "The Craft of Computer SoftwareTesting", Prentice Hall Series in Innovative Technology, 1995

[2] Cem Kaner, Jack Falk and Hung Quoc Nguyen, "Testing Computer Software", International Thomson Computer Press, 1993

[3] William Perry, "Effective Methods for Software Testing", Wiley-QED Publication , 1995

[4] Glenford Myers, "The Art of Software Testing", Wiley-Interscience, 1979

[5] "ANSI/IEEE Standard for Software Test Documentation", ANSI/IEEE Std 829-1983

[6] "ANSI/IEEE Standard for Software Unit Testing", ANSI/IEEE Std 1008-1987

[7] "P1394 Standard for a High Performance Serial Bus", IEEE Std 1394, 1995

[8] “P1394a Draft Standart for a High Performance Serial Bus (Supplement)”, IEEE, 1997-1998
[9] Don Anderson, “Firewire System Architecture: IEEE 1394a”, Addison-Wesley, 1998
[10] http://standards.ieee.org, Various drafts and Standards for the Network and its applications
[11] http://www.1394ta.org, 1394 Trade Association URL