HAL
open science

Simulatorbased testing environment for avionics software: a feasibility study

S Flores, P Le Meur, J F Renneson

To cite this version:

S Flores, P Le Meur, J F Renneson. Simulatorbased testing environment for avionics software: a feasibility study. 4th International Congress ERTS 2008, Jan 2008, toulouse, France. insu-02269750

HAL Id: insu-02269750

https://insu.hal.science/insu-02269750v1

Submitted on 23 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Simulator-based testing environment for avionics software: a feasibility study

S. Flores[1], P. Le Meur[2], J.F. Renneson[2]

1: Atos Origin Integration, Rue Ampère, 31672 Labège
2: Airbus France, 316 Route de Bayonne, 31060 Toulouse

Abstract: Hardware execution targets are widely used for software testing in the avionics industry to ensure that the tests represent the behaviour on the actual aircraft's calculators to a maximum degree.

At the same time, software simulators of the Integrated Modular Avionics structure are used on several system integration benches for calculator validation or as a part of full flight simulators.

We substituted a hardware target with a specific software simulator in order to create a new testing platform that provides application developers with features not found on hardware-based environments. Such features include greater availability of testing platforms, debugging capabilities and an easier and faster testing process. Our goal was to evaluate the feasibility of a full migration to the resulting simulator-based testing environment. We identified several industrial constraints and technical problems to achieve this migration and approached them by developing progressive prototypes of the components of the new testing system and of the interfaces between them. Our project's validation plan included comparing the results of both hardware-based and simulator-based platforms when executing a set of representative avionics tests.

Keywords: Integrated modular avionics, simulation, embedded software testing.

1. Introduction

1.1 Background

Embedded software development involves thorough testing phases. In the avionics industry the means of testing become almost as critical as the tests themselves because the whole development process is subject to certification by air navigation authorities, using the DO-178B standard.

This standard defines a set of criticality levels according to the consequences of a software failure, from level A (catastrophic consequences) to level E (no effects). A typical development cycle of airborne software includes unit and integration testing phases in which hardware targets are used to execute the tests. These hardware targets are the central part of the testing platforms that we currently use, named Target-Based Operating System Simulation (TBOSS). The TBOSS targets simulate an IMA airborne calculator and the operating system on it, and even though some of their components are not the same as those in the aircraft, certification authorities have controlled their representativeness.

Unfortunately, good representation comes at a price. Application developers are often very limited in essential aspects of the testing process such as debugging or even the physical availability of the testing platforms due to the small number of hardware targets. In our industrial environment in particular, we will also have to deal with other problems such as the lifespan of the hardware, the cost of licensing of the tools, and the technological dependence to the providers of the testing platform.

All these reasons made us consider the possibility of developing a new testing platform based on an existing IMA structure software simulator. This simulator called *Software Workshop for IMA Modules (SWIM)* is currently used by another entity in our organisation to validate flight calculators by

integration test benches. It is also part of full flight simulators that are employed for pilot training.

SWIM provides a given airborne application with an emulated behavior of its environment as if it interacted with its real operating system and underlying hardware. In the IMA architecture, the means of interaction with the operating system are defined by the ARINC 653 standard, which is thus implemented by SWIM.

This simulator recreates the real-time behavior of a single IMA processing unit (CPIOM) and of the operating system that controls it (MACS2). This makes SWIM a host structure capable of executing complete IMA applications on a GNU/Linux operating system.

On the other hand, software unit and integration tests are a very controlled execution of a few components of the final application. In this tests several conditions such as time, return codes or initial values are often simulated or forced.

Our reasoning is that a simulator capable of executing complete applications might also be able to run verification tests of each one of their components, even though it was not designed for such a purpose, and even though the tests differ from complete application in many ways.

1.2 Motivation

A testing platform based on SWIM would be completely software-based, which would allow us to make it more available to developers (by simply copying it or launching several instances) than the hardware-based platforms. The whole testing process would become easier and faster as there would not be a need to compile and load configuration sources for each test. In the long term, greater debugging features could be added to the new platform because we would have access to all of its source code; which would also allow us to use it without paying any license fees, or even to adapt it to our particular needs. Finally, creating our own testing platform would give us a better control of the testing process. This becomes increasingly important as we do not want to depend on a single provider of testing platforms.

1.3 Constraints

There are nonetheless some very important constraints to take into account. The first and most important is that the simulator-based testing platform must be backwards-compatible with the previous one. This is an essential requirement to a full scale migration because there are several thousands of existing tests that support the reliability of different avionics applications currently in service, and modifying them would imply an enormous effort. This means that we must adapt the new simulator-based testing platform to the existing tests and not the contrary. The execution of these tests should have the same results in both hardware and simulator-based platforms and should be generated from the same sources.

The testing interface should remain the same to the application developer, so that the eventual migration is transparent to them.

We also want to keep modifications to SWIM to a strict minimum, as we wish to be able to evolve with the original simulator and take advantage of eventual new features.

1.4 Feasibility and problems

To all this constraints we add the fact that SWIM was not originally made to fulfill our testing needs.(mainly unit testing and integration testing for software verification).

Due to the complexity of the current hardware-based platform and to the strong constraints we face, we decided to evaluate the feasibility of a total migration to a simulator-based platform by creating a prototype of this platform. This prototype helped us:

- to identify the potential problems of the migration
- to anticipate to these problems and to take into account their solutions from early stages of the creation of the new platform
- to foresee any blocking conditions and eventually to decide about the feasibility of a full migration to the new platform.

The first part of our project was to identify the differences between the execution environments in SWIM and in the hardware targets. We present them

here as a means to expose the problems that made us question the feasibility of a full scale migration.

Main purpose: the TBOSS targets were designed to **test** a single component of an application with representativeness being the most important feature. The ability of the application developer to simulate API calls is also important as it allows him to easily test different scenarios, for example forcing the execution of a certain branch of code.

SWIM on the other hand was made to be the first **simulation platform** of complete IMA applications (as opposed to just components), providing them with all necessary operating system access services and behavior.

Executable Format and configuration: the TBOSS targets use the executable and linking format (ELF) for the PPC 755 processor. A single executable must be built outside the target and then loaded to it. The executable is constituted of a binary image of the system software (MACS2 OS and other drivers), a binary image of the application software (the test), a boot loader that copies everything in the right place on memory, and a binary image of the configuration table (configuration parameters for the MACS2 OS, drivers, RAM and communication means). Of course; we need to recompile the sources of most of this components for each test.

SWIM in contrast is an executable itself. Applications are loaded at run-time in the form of ELF shared objects (dynamically-linked libraries). These shared objects just contain the application as the system software is provided (or rather emulated) by SWIM. All configuration in SWIM is done by regular files. One of the XML configuration files is particularly important because it describes the period used by the OS simulation's scheduler, the shared objects to be loaded and the communication channels to be used.

Communication Channels: An important term regarding communications in IMA applications is *APEX port*. These ports are the main internal communication mechanisms between IMA partitions, processes and external devices. If we must exchange information with an executing IMA application, we must connect with the APEX ports in some way. TBOSS and SWIM take different approaches to this.

TBOSS targets use a set of tools to upload the executable into memory. Aside form this, all communication in and out the target is made by a special interface that lets us implement control programs by including a C header file and a static library. We will cover this in more detail later. We do not have access to the source code of any of these tools.

SWIM does not impose any means of communication : it just provides a communication interface. SWIM users can implement the communication stack the way they want, and compile it as a dynamic linked library. In fact, the current users of SWIM generate the code of this stack automatically from files describing the external interfaces of the hosted avionics applications.

Execution sequence: TBOSS targets execute applications 'step-by-step' in a basis of a period called *minor frame* (MIF). Roughly, this duration represents the minimal configurable time duration of the OS scheduler. As we will explain in further detail later, the execution of each MIF is conditioned to the reception of a specific command from the user.

SWIM on the other hand executes applications continuously during a predefined (but configurable) number of MIFs. The execution will continue without stopping until this number of MIFs is reached.

Representativeness: the TBOSS targets ensure that the compiled code will behave as in the airborne calculators because it uses the real MACS2 OS, a very similar processor and a very similar memory scheme; as opposed to SWIM that executes on Intel x86 processors and just emulates the behaviour of the MACS2 OS on a GNU/Linux System.

2. Methods

In this section we present the development of our prototype as well as the problems we encountered, the solutions we implemented and the validation strategy.

For simplicity, we will call our prototype and the existing hardware-based platform 'the SWIM platform' and 'the TBOSS platform' respectively.

2.1 Imposed Architecture from TBOSS

As stated in section 1.3, a major requirement for a new simulator-based testing platform is its capacity to execute existing tests without modification. This imposed a certain architecture to the SWIM platform as we needed to determine if we could reuse some of the components of the TBOSS platform. An overview of the architecture of the latter is shown in figure 1.

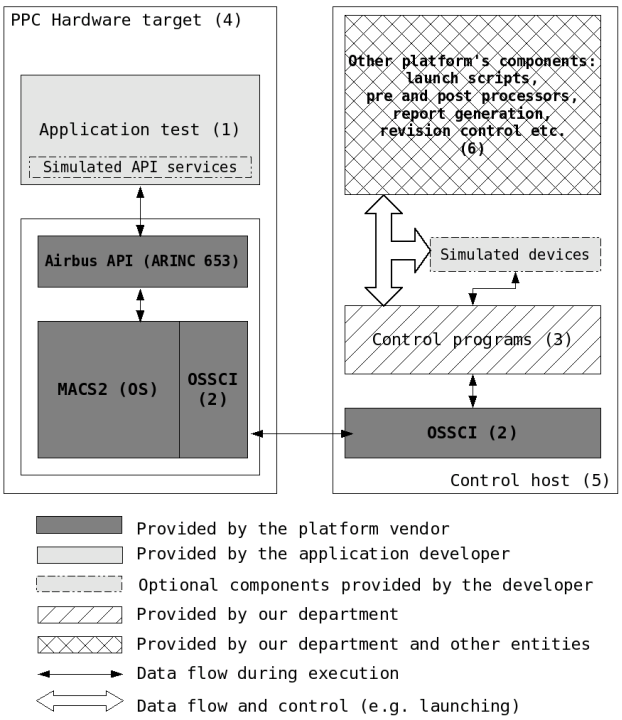

Figure 1. An overview of the TBOSS platform

In this architecture, the tests (1) are executed in a target which has no means of communication with the external world but a control interface (2) called 'Operating System Simulation Control Interface' (OSSCI). A set of programs (3) running on Solaris workstations control the test exclusively through this interface. All the information exchanged between the execution of the test and the developer is also transmitted by the OSSCI.

We can see this as a communication layer between the execution target (4) and the control host (5). Its implementation remains completely inaccessible to us, as the only source code provided is a C header file that establishes the signatures of eleven functions (OSSCI commands) which must be called by control programs in order to interface with the test.

The platform also includes several other tools (6) such as launch scripts, configuration management and revision control facilities, pre-processors, post-processors and report generators. These tools constitute an interface to the developers and thus should remain unchanged. Some of these tools are provided by the developers, some others by our department, and some others by third parties. We do not have access to the source code of all of them.

2.2 Architecture of the SWIM Platform

The creation of the SWIM platform consisted of three main tasks:

- the implementation of a new OSSCI communication layer using a client-server scheme.
- the implementation of the corresponding modifications on the original SWIM so that it would respond to the commands of the new OSSCI.
- The integration of the new and ancient components.

Of course, this tasks were not made sequentially but rather in parallel. We present the fundamental aspects of each one and then we present some interesting problems that we faced.

<u>The new OSSCI.</u> It consisted in two static libraries, one to be linked to SWIM ('the server') and one to be linked to the control programs ('the clients'). We used an object-oriented approach to design this libraries.

On the client side, we chose the simplest of the existing control programs to use within our prototype, simply linking it against our new client library and thus taking advantage of the hidden implementation of the OSSCI.

We reimplemented some of the functions that allow a control program to interface with the test. Some others where not implemented because they were not immediately necessary for our feasibility study (as no control program used them). The details of the implementation of these functions are out of the scope of this document, but we would like to briefly highlight some of them:

- **OSS_Init**: initializes the hardware target and the OS. In our implementation it gets the current SWIM's APEX port configuration and establishes TCP/IP socket connections accordingly.
- **OSS_Run**: Asks for the execution of the test during a specified period of time. As we will explain more in detail later in this document, the fact that SWIM lacks this functionality made the reimplementation of this OSSCI command critical.
- **OSS_End**: Ends the execution of the test. In our implementation it closes all connections and frees temporary resources.
- **OSS_Message_Input**: asks for the transport of predefined messages from the control host to an APEX port in the application. We internally use TCP sockets to send these messages.
- **OSS_Message_Output**: asks for the transport of predefined messages from an APEX port in the application to the control host. We internally use our TCP sockets to send these messages.

On the server side, the library executes its method *serve* to receive and treat the internal OSSCI messages that arrive from the client.

Again, their implementation is out of the scope of this document but we highlight some of them:

- **prepare_connections:** called on reception of a message sent by OSS_Init, it sends the client the APEX port configuration of the current test and prepares all connections.
- **destroy:** called on reception of a message sent by OSS_End, it destroys all connections and frees temporary resources.
- **send_next:** called on reception of a message sent by OSS_Message_Output, it sends the next ready message to the specified port.
- **receive_next:** called on reception of a message sent by OSS_Message_Input, it receives the next message waiting on the specified port.
- **serve:** this method treats the received messages from the server and dispatch the corresponding actions, such as calling the methods above or continuing the execution of the test. We deliberately put this method at the end because we will talk more about it in the next section.

Modifications on SWIM: as we previously stated, we wanted to keep SWIM free of modifications. But because SWIM was not originally designed to fulfill our testing needs, some modifications were necessary. We tried to minimize their impact by isolating them on a specific compilation option.

The main change to the SWIM behavior was the inclusion of the OSSCI server's *serve* method call at the very beginning of the execution of each MIF.

This method call is blocking, effectively preventing SWIM to continue the simulation of the test. Once in the method, the OSSCI server will wait for events and dispatch actions according to the messages from the OSSCI client. The method will not return until a special message sent by OSS_Run is received. This process can be seen in figure 2.

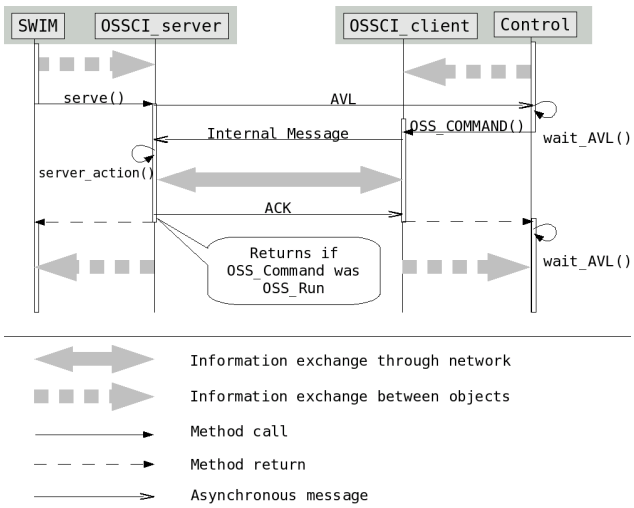

Figure 2. Simplified sequence diagram of a test

This modification to the SWIM behavior totally affects the real-time execution of the software. This is true because the execution of the applications (the tests) become synchronous to the arbitrary call of the OSS_Run function by the control programs. Nevertheless, this does not represent an obstacle to our purposes because the whole simulation is stopped upon the execution of the *serve* method, even the simulated flow of time. We achieve this by calling the *serve* method before any internal SWIM

timer is initialized or incremented, so this suspension will not affect the effective execution time of the MIF.

Integration of the components: as we explain in figure 3, the SWIM platform consisted in our modified version of SWIM in a GNU/Linux system (1) controlled by a Solaris host (2) through TCP sockets. The simplest of the control programs (3) was linked against our new OSSCI client library (4) which allows communication with the OSSCI server (5). The server itself has been incorporated to SWIM, that loads the shared object of the test (6) at runtime.

It is important to mention that although we did not include all of the other components (7) of the ancient platform in our prototype, the interfaces to them were not changed.

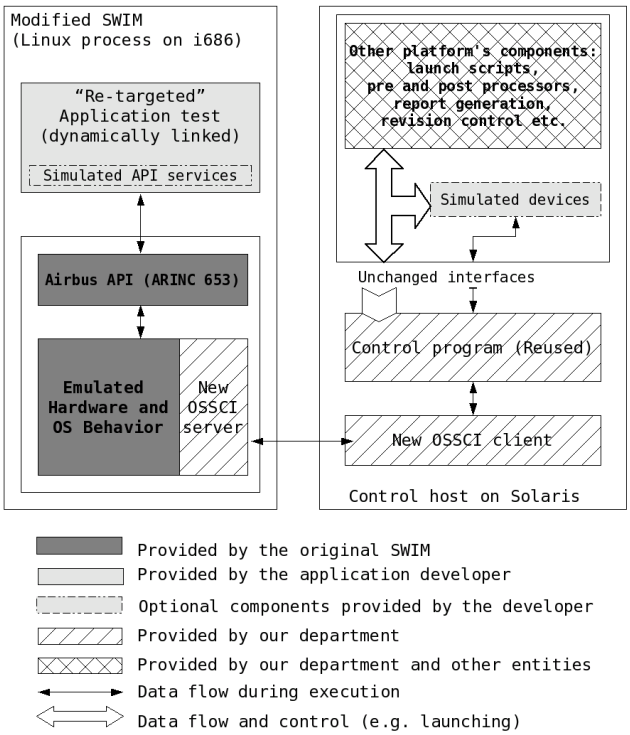

Figure 3. An overview of the SWIM platform

2.3 Problems encountered and solutions

Until now, we have explained the architecture and the implementation of the SWIM platform according to the constraints we had. The next step was to test our prototype through validation tests. We will explain our validation strategy in further detail later, but first, we present some of the problems that we encountered during the testing phase and the steps we took to solve them.

Differences between the executable formats in the TBOSS targets and SWIM: As we explained in section 1.4, SWIM and TBOSS take different executable formats. Our first problem to was to determine which sources to compile to generate the shared objects needed by SWIM. The build process itself was also very important because some tests made assumptions about the testing platform. For example, some tests could rely on the TBOSS hardware to clean certain areas of memory. Needless to say, these areas are not the same under the SWIM platform. The most interesting problem that we had, concerned the memory segment in which variables initialized to zero were stored.

The default policy of the GNU C compiler (which we used to generate the shared objects of our tests) is to put variables that are initialized to zero into the BSS segment (the BSS segment is the memory zone in a process that contains the uninitialized global data structures). This can save space in the resulting code, but can cause problems with programs that explicitly rely on variables going to the data section. This was the case in some of our tests, and the result was that global variables explicitly initialized to zero had in fact different, random values.

The solutions to all the problems related to different executable formats between SWIM and TBOSS was to carefully identify the necessary compiler options to generate machine code that would behave in the same way in both platforms.

Endianness: The control host used in a TBOSS platform runs on Solaris/Sparc platforms. Besides, one of the control programs was specifically designed to be run on Solaris systems (it uses specific Solaris' shared memory mechanisms to communicate with simulated devices). Because of this, we decided to keep all of the control programs on that platform instead of porting them to GNU/Linux (which is the platform where SWIM runs). This brought some problems concerning the endianness of the data sent through the network: Solaris/SPARC platforms are big-endian while Linux/ia32 platforms are little-endian. The solution to

this was simply to include proper treatment of the integers transmitted.

Symbol resolution and overwriting: As we explained in section 1.4, TBOSS purpose is to be used in verification tests. The developer may want to force the execution of a certain branch within the tested code. To do this, developers use *code stubs*. These stubs are actually redefinitions of the functions called by the software undergoing tests. They exist to simulate the existence of the rest of an application for the specific part of software being tested. For example, if we are testing a module *m* and we know that *m* calls a function *f* (defined outside this module), we may want to simulate *f* so that we can force its return code to decide about something:

```
...
// we know this code exists in the module m
// which we are testing:
rc = f(param1, param2, param3)
if (rc == NO_ERROR) {
    // do something
}
else {
    // do something else
}
...
```

The developer just rewrites a very simple definition of *f* that returns the desired value instead of incorporating the code of the real *f* function into the executable.

This is in fact very simple, but it becomes much more complicated when the developer needs to redefine a system call, like a service of the AIRBUS API (the ARINC 653 interface). Such service would then be defined twice in the same executable: once by the basic software (the OS) and then another time by the application's test.

In practice, some of the tools of the TBOSS platform pre-process the sources to eliminate all duplicated symbols and obtain the desired effect (that of the developers' redefinition of the service).

But we could not use the same tools for the SWIM platform because the execution environment is completely different. Only the interface between the components is the same.

Every time that a test redefined a service of the ARINC 653 interface, we faced the problem of duplicated symbols in the execution of our tests in the SWIM platform. Because SWIM and its applications are compiled separately, no compiler warning was ever raised.

Besides, in a GNU/Linux system, the policy of the dynamic loader (the program that loads the shared object into memory) is simply not to load a symbol that has been already defined in memory, hence giving precedence to the first symbol and silently ignoring the second. The result was that many of the existing tests failed because they were calling the actual system service instead of the developer's stub. This problem is illustrated in figure 4.

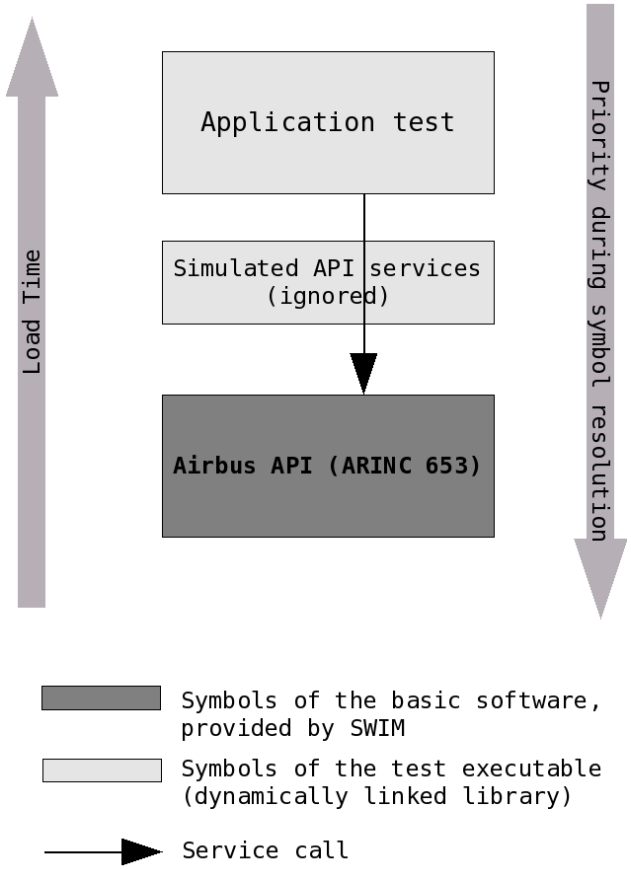

Figure 4. Default symbol resolution in GNU/Linux

To overcome this obstacle we took advantage of the same dynamic loader policy. We slightly modified SWIM and its build process to encapsulate all the AIRBUS API services in a dynamic library. Then we forced the SWIM's main binary to load the shared object of the application before that of the system API, as seen on figure 5.

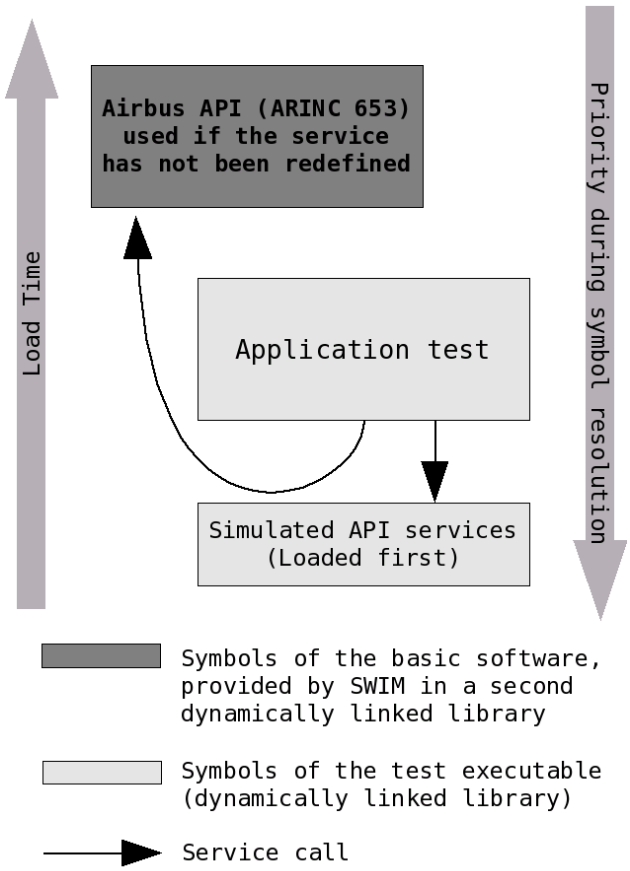

Figure 5. Symbol resolution in the SWIM platform

2.4 Validation Strategy

To validate our prototype, we took a *subset* (a module) of the Air Traffic Control software used on the A380 (ATC, a level C IMA application). The reliability of this subset is verified by several hundreds of unit tests, organized in 40 different groups. From each one of these groups, we generated a test executable in the form of a dynamic linked library that was then executed on the SWIM platform. Finally, we compared the results of the execution with those obtained with TBOSS targets.

3. Results

After solving the problems that we exposed, we were able reproduce the results of the TBOSS platform in our prototype for each one of the tests: all of the unit tests executed in the exact same way in both platforms, starting from the exact same sources, and giving us the same results, byte per byte.

We created a prototype of a software simulator-based testing platform. The prototype uses some of the components of the hardware based platform while some others were completely reimplemented.

We also produced a set of new, modifiable, object-oriented OSSCI libraries that implement an interface between SWIM and existing control programs.

Finally, we obtained a modified instance of the SWIM simulator that is paced by the OSSCI and that executes IMA applications in a MIF basis, without interfering with the results because the real-time behavior is not compromised.

4. Discussion

4.1 Interpretation of the results

The identical results of the validation tests in both platforms and the minimal modification of control programs, suggest that a total, backwards-compatible migration to a simulator-based testing platform is possible in the short term.

Although we have not formally evaluated if our prototype is a better testing platform than the one of TBOSS (see section 1.2 “motivation”), we can already tell some differences : The tests finish faster in our platform because we do not need to recompile the basic software for each test, or load the executables from an external host into the target's memory.

The minimal modification of the control program, a central part of both of the testing platforms, shows that it is also possible to reuse most of the existing components.

The modifications made to the control program of the SWIM platform concerns only the OSSCI. None of the interfaces with the rest of the platform were affected. We therefore believe that other control programs can be modified in the same way and that none of the other components of the existing platform must be modified to achieve full backwards compatibility with an eventual simulator-based platform.

Our new OSSCI libraries are not only an interface between SWIM and control programs but also an interface between existing tests and any other execution platform. The fact that they are object-oriented will allow us use them easily with other execution simulators; in fact work is already being done on this subject. We also want to stress the

open nature of these libraries. While the implementation of the OSSCI on the TBOSS platform is not accessible, we can freely use and modify these libraries according to our needs. We even use some *Free Software* internally, although we are not planning to release these tools.

4.2 Future work

Our results allow us to think of the implementation of a SWIM-based testing platform that could be widely used by developers during the test phases of their avionics software. The representativeness of such a platform could be questioned, but in any case the platform could be used to detect problems prior to execution in more representative environments.
Much of the work done for our prototype can be directly exploited in a production-level SWIM-based testing platform.
We will continue to explore the use of simulated environments to test avionics software, specifically the use of full hardware *virtualization* to achieve total representativeness.

5. Conclusion

This paper was a technical report of our experience developing a prototype of a testing platform for avionics software based on the SWIM IMA structure simulator. We used this prototype to identify problems that could prevent us from creating a backwards compatible, simulator-based testing platform and to migrate to it. We identified several conditions that should be taken into account in the event of such a migration. Our results suggest that this migration is possible in the short term.

6. Acknowledgements

We would like to thank Laurent Gillet, Jean-Baptiste Jouve and Philippe Seraud for their technical and moral support during the development of this project.

7. References

[1] Free Software Foundation: "*The Free software definition*", http://www.fsf.org/licensing/essays/free-sw.html

8. Glossary

IMA: Integrated Modular Avionics. An architecture for sharing computing resources in a real-time airborne network.

CPIOM: Core Processing and Input/Output Module – A single calculator in the IMA structure.

SWIM: Software Workshop for IMA Modules – a software simulator of the IMA Structure

TBOSS: Target-Based Operating System Simulation – a set of proprietary tools based on a hardware target to simulate the execution of IMA software with assured representativeness.

OSSCI: Operating System Simulation Control Interface – a part of the TBOSS platform that enables external, bidirectional communication with the outer world.

MIF: Minor Frame: The minimal time unit in which application partitions can be configured in an IMA application.

ARINC 653: Is a standard developed by Aeronautical Radio, Incorporated (ARINC) that defines the interface between an IMA application and the underlying operating system.

Airbus API: an implementation of the ARINC 653 standard that adds some extensions specifically for Airbus aircraft.

Free Software: software that can be used, studied, distributed and modified under certain legal restrictions stated in [1].