Hindawi Publishing Corporation
Modelling and Simulation in Engineering
Volume 2014, Article ID 105051, 15 pages
http://dx.doi.org/10.1155/2014/105051

Research Article

Runtime Instrumentation of SystemC/TLM2 Interfaces for Fault Tolerance Requirements Verification in Software Cosimulation

Antonio da Silva,[1] Pablo Parra,[2] Óscar R. Polo,[2] and Sebastián Sánchez[2]

[1] *Department of Telematic and Electronic Engineering, Technical University of Madrid, 28031 Madrid, Spain*
[2] *Computer Engineering Department, University of Alcalá, Alcalá de Henares, 28871 Madrid, Spain*

Correspondence should be addressed to Antonio da Silva; antonio.dasilva@upm.es

Received 23 June 2014; Accepted 7 September 2014; Published 23 September 2014

Academic Editor: Luis Carlos Rabelo

This paper presents the design of a SystemC transaction level modelling wrapping library that can be used for the assertion of system properties, protocol compliance, or fault injection. The library uses C++ virtual table hooks as a dynamic binary instrumentation technique to inline wrappers in the TLM2 transaction path. This technique can be applied after the elaboration phase and needs neither source code modifications nor recompilation of the top level SystemC modules. The proposed technique has been successfully applied to the robustness verification of the on-board boot software of the Instrument Control Unit of the Solar Orbiter's Energetic Particle Detector.

1. Introduction

Embedded software plays an important role in today's complex system-on-chips (SoCs) since it allows convoluted features to be implemented flexibly. However in new developments, hardware capable of executing software is not often available until the later stages of the development cycle. To accelerate the design process and to increase the productivity, system-level design has to accommodate software concerns enabling a seamless codesign of software and hardware. Therefore, it is highly desirable to address software development as early as possible. Virtual platforms are executable models of complete systems that provide software developers with working frameworks time before the real hardware is available. They enable the concurrent development of system-on-chip (SoC) hardware and software, significantly shortening their integration times. From an embedded software perspective, the use of virtual platforms allows the development and verification processes to be started earlier in the design flow so as to detect and correct errors that would otherwise propagate towards the final implementation stages. Moreover, it is easier to access and modify the internal state of the virtual prototypes, so that a comprehensive fault injection campaign and fault tolerance assessment can be carried out. This helps to achieve what every embedded software developer is fundamentally looking for: predictability and robustness [1].

System-level modeling languages are used to start the design process from an abstract level and apply a top-down design methodology through a refinement process [2]. The SystemC transaction level modeling [3] raises the abstraction level of system descriptions, focusing on the exchange of data between components through communication channels or sockets. Because of their advantages, the descriptions of TLM systems can be used for design space exploration, for early architectural performance estimations and to allow an earlier software development commencement by joining the hardware and software design flow together. SystemC/TLM is written in C++, which is a widely known programming language that has been a popular starting point for describing executable hardware/software systems models. Using TLM, system models are quick to write and give an executable version of the specification, which allows a very fast simulation. For system-level design, these languages allow hardware and software components to be described in a single framework. Furthermore, the only development tools needed are regular

C/C++ compilers and debuggers, with which embedded systems designers are already well acquainted. The use of virtual platforms gives developers far more visibility and control over system design in comparison to traditional development methodologies. Any state is within reach and any condition can be triggered. Therefore, virtual platforms have become widely used in avionics and space software development environments before the hardware becomes available. Current research focuses on experimental techniques and tools that allow software robustness verification through fault injection using virtual platforms.

The remainder of the paper is organized as follows: Solar Orbiter's mission characteristics along with the embedded software development challenges and novel contributions are set out in Section 2; relevant related works are detailed in the same section. Section 3 describes the proposed wrapper insertion framework. Section 4 discusses some issues as regards performance and usability using different interposition code insertion techniques. Section 5 describes the real scenario in which an early software robustness evaluation by means of fault injection on a TLM2 virtual platform (Leon2ViP) has been carried out. Finally, Section 6 contains the conclusions.

2. Problem Statement and Related Work

2.1. Solar Orbiter Mission. Solar Orbiter [4] is a planned sun-observing satellite, under development by ESA, and is scheduled to be launched in January 2017 as a baseline. At its closest point, the spacecraft will be closer to the sun than any previous spacecraft, almost one-third of the Earth's distance from the sun. Because of the proximity of the sun, the spacecraft must withstand powerful bursts of atomic particles coming from the solar atmosphere. From an on-board software designer's perspective, it is essential to look out for permanent soft errors resulting from latch-up failures in SDRAM/EEPROM memories. The Space Research Group (SRG) of the University of Alcalá is in charge of the development of the instrument control unit (ICU) for the Energetic Particle Detector (EPD) on-board Solar Orbiter, along with the corresponding boot and application software. For the early development and verification of the ICU's bootloader software, a framework with the ability to run the same binary code that will run on real hardware was needed. It also had to emulate SDRAM and EEPROM permanent errors, a fact that is difficult, if not impossible, on real hardware. The ICU boot software is in the critical path of the project so its verification should be addressed at an early development stage for any test case missed in this process can affect the quality of the overall onboard software. Thus, the robustness requirements call for an exhaustive testing of the boot process and possible corruption of application binaries stored in the EEPROM or stuck-at faults in the SDRAM application deployment areas. Bearing this in mind, from a hardware dependent software point of view, such as the boot software, the major problem of carrying out early development and testing activities is the absence of a hardware platform on which to run it. Other points to keep in mind are the effort and risks associated with bringing up hardware dependent software such as boot loaders. The classic approach for developing this kind of software has been to design the hardware, make a physical prototype, write the code, and then integrate the hardware and software. This methodology is nowadays too slow and calls for an alternative to the traditional software-after-hardware design flow in order to get started with software development and testing before the hardware is ready. To shortcut these issues the SRG has developed Leon2ViP, a LEON2 virtual platform with fault injection capabilities, which has been built around SystemC/TLM2 interfaces given previous experiences with LEON3 systems [5]. For the ICU boot and application software development, Leon2ViP provides faster edit, compile, and debug cycles and, at the same time, a more controllable and observable environment for the verification activities. Furthermore, even if the hardware was already available, this virtual platform offers, not possible otherwise, nonintrusive and exhaustive debug and fault injection capabilities. The overall ICU hardware/software codesign is described in [6].

A key point of the proposed framework is that it enables the work of the design and verification teams to be decoupled. As a design principle, the Leon2ViP TLM2 code that implements system components like Instruction Set Simulators, memory modules, or SpaceWire network interfaces should contain just functional code in order to emulate the behaviour of the component and represent a functional golden model of the hardware. All fault injection codes employed to emulate memory stuck-at faults must be applied in TLM2 transaction interfaces and not embedded in the model's code. This leads to the necessity of intercept TLM2 calls to the memory modules in order to corrupt data read or written from/to memory or peripherals.

The interception library presented in this work is not intended to validate the Leon2ViP virtual platform itself. In fact, the library is part of the virtual platform and has been developed to help in the ICU boot software development and testing of the basic recovery mechanisms in those cases when the nominal boot sequence is not possible. It has been also used in the codesign of the SpaceWire core used for communications from/to the spacecraft.

Although the interception library has been developed specifically for the Leon2ViP virtual platform, it can used as a separate instrument in order to insert wrappers in TLM2 designs. What these wrappers are used for is up to the library user. In Section 5, shows an example of how the wrapping library has been used to insert a fault injection wrapper in order to simulate stuck-at zero faults in memory access. The goal is to verify the correctness, according to the specifications of the software that runs on the virtual platform.

2.2. Paper Contribution. This paper presents the results of attempting to provide a generic framework that provides dynamic binary instrumentation to TLM2 models. The approach is based on the mechanism used by C++ to implement late binding in virtual method calls. Our targeted application is the Leon2ViP virtual platform.

(1) To our knowledge, this is the first library using C++ virtual table hooking as a dynamic binary instrumentation technique in order to intercept TLM2 socket primitives. The basic idea was introduced in [7] and in order to validate the approach a few code snippets specifically tailored for Microsoft VisualC++ compiler were given. This work presents a full development of the idea, intercepting all TLM2 interfaces and taking into account the compiler differences. Moreover, an implementation for the most popular compilers, VisualC++ and GCC, is free released.

(2) The proposed library is able to insert wrappers into the transaction path without modifying the TLM2 model source code description. This makes the technique useful for the validation of third party Intellectual Property (IP) TLM2 cores, which can be distributed as object modules. Thus, no knowledge about the source code or methods names are needed in order to insert wrappers into the transaction communication path.

2.3. Related Work. As regards the use of virtual platforms, SystemC/TLM2 is an interoperability standard for memory-mapped bus modeling and is a key enabler for the development of virtual platforms, serving as a bridge between hardware and embedded software designers, especially for hardware-dependent and communication software development [8, 9]. Beyond the use of TLM2 for processor buses modeling, TLM2 extensions have been proposed in order to model embedded system networks. For example the work [10] proposes an extension of transaction level modeling to perform system/network design-space exploration in networked embedded systems (NESs).

Using SystemC transaction level modeling (TLM) it is possible to model mixed hardware/software systems in order to simulate the software behaviour in the presence of faults in the hardware. For example, works [11, 12] use this methodology for the design and testing of fault tolerant systems implemented on an FPGA platform with different types of diagnostic techniques. The experimental results show the fault coverage and how Single Event Upset (SEU) occurrences cause faulty behaviours in the implemented systems. References [13, 14] use the same approach to verify the software of networked embedded systems long before the final hardware is available.

Coming from industrial environments, the work [15] describes the enhancement of a previous tool that allows an effective transition from the system-level development phase to the software level development phase, throughout a case study based on a hybrid electric vehicle development. Another work [16] presents a system-level codesign and coverification case study. In this work, a processor bus functional model (BFM) is used to combine native software execution with a cycle-accurate interconnect simulator and an HDL simulator.

Fault injection is mandatory in experimental dependability evaluation. Thus, the work of [17] introduces fault injection methods for register transfer level (RTL) system descriptions into SystemC. Related to TLM, the work [18] presents an example of system-level fault injection in untimed functional TLM models based on FIFO channels. Related to fault tolerant TLM2 designs, the work described in [19] proposes a hardening method for inter component communication protocols.

The assertion-based verification (ABV) of SystemC/TLM models is a wide field of research. Assertions capture specifications of the system being designed in an executable form. Then, they act as monitors during the simulation, detecting and reporting errors close to their source as well as establishing coverage information. Several works, such as [20–22], among others, use this approach to perform system-level verification. Other works use the same approach to perform tracking/snooping of the transactions interchanged between TLM modules, for example [23, 24]. Assertions are usually written in specific languages such as Property Specification Language (PSL) and they must be translated to specific assertion code that must be placed in between the transaction's initiators and targets.

For the perspective of this work the most important issue is how the assertion code is introduced into the system model description. A common way to add additional code at specific points of the source code is by using aspect oriented programming (AOP). In AOP one *aspect* is a feature linked to some parts of a program, but which is not related to the program's primary function. It is based on source-to-source translation and allows invoked methods to be wrapped with pre/post condition checkers. From this point of view aspects can be seen as a way of inserting wrappers into the transaction path. Aspect C++ [25] is an aspect-oriented extension of C and C++ languages and is widely used to add aspects to SystemC descriptions [26, 27].

The previous works mainly use AOP for the verification of the hardware model described in SystemC, either RTL or TLM. The closest works to ours are [28, 29]. In these works transactions checkers are inserted in a virtual platform based on TLM2 interfaces. The goal of the checkers is to detect the wrong duration and sequence of the transactions in the TLM2 design.

As said before, the insertion library described in this work was not designed with the aim of verifying hardware TLM2 models but for the verification and monitoring of the software running on a virtual platform built around TLM2 interfaces. However, it can also be used for other purposes as discussed in the next section.

3. TLM2 Wrapper Insertion Library Design

Talking about the instrumentation of a model description there are basically two features to consider, see Figure 1. The former is the knowledge and availability of the model source description in order to know where to instrument. The latter is the binding procedure; this is how the instrumentation code is inserted into the model.

In the case of the interception library proposed in this work, it is important to emphasize that its main goal is to

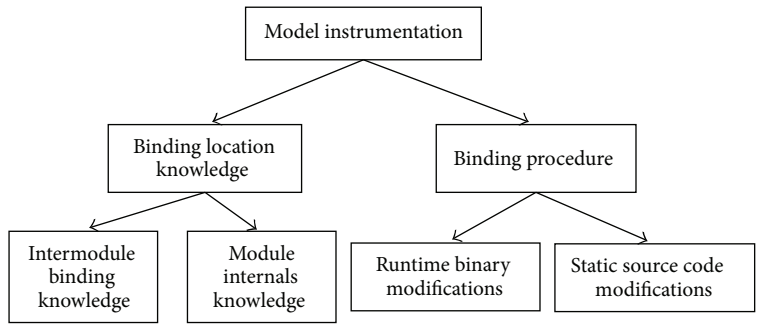


Figure 1: Instrumentation features.

provide basic services in order to intercept TLM2 transactions at runtime. This means that the modules already exist, their TLM2 sockets have been bound, SystemC has ended its elaboration phase, and simulation has begun. Even more, instrumentation should be carried out without access or knowledge of the TLM2 model source code, just the TLM2 socket name of the target is known. The need to use runtime binary instrumentation on the TLM2 socket intermodule bindings stems from this.

Figure 2 shows several ways of intercepting the transactions exchanged between two TLM2 modules. The location of the inserted interception or verification code is stressed in all cases. One transaction initiator, a SPARC Instruction Set Simulator (ISS), is bound to a transaction target and uses the TLM2 nonblocking transport interface `nb_transport_fw/nb_transport_bw` to carry out transactions. These transactions are supposed to be memory read/write operations. The usual way to intercept the transaction path is the use of an interposition module placed in between the modules, (see the first case in Figure 2). In this case the initiator is bound to a module inserted into the transaction path and the interposition module is bound to the target. The second case in Figure 2 shows a C++ aspect placed in the target module in order to intercept forward nonblocking transaction calls. The `nb_transport_fw` call is intercepted by means of an `nb_transport_aspect`, so the preprocessing of the incoming transaction and postprocessing of the results can be carried out. It is important to point out that in all the aforementioned approaches a source code modification is required to perform the transaction path modification. The initiator must be bound to the interceptor module or to the target. All of these bindings are done at compile time and do not change during the execution of the model. Even more, to code those bindings a profound knowledge of the TLM2 model source code is needed. For example, to insert an `aspect` around a method, the name of the method must be known. This is known as static code instrumentation.

The objective of the proposed library is to allow the insertion and removal of interposition code at runtime, as is shown in the third case in Figure 2. No modifications are made to the TLM2 modules and the original socket binding established at compile time is modified at runtime without knowledge of the modules' source code. Only the name of the TLM2 socket is necessary. It is important to emphasize that the library presented in this work just provides the basic services to intercept TLM2 transactions. The specific transaction processing is built using these basic services. Once a transaction path is intercepted by means of a wrapper, it can be used in several testing scenarios such as transaction tracking or snooping, experimental dependability evaluation through fault injection, property assertions, and TLM2 protocol compliance verification. As an example, [30] uses the technique being described in order to track the transactions between TLM2 components. The work [31] uses the same technique to insert a TLM2 protocol compliance checker in a nonblocking transport scenario.

The last case shown in Figure 2 is not an interception scenario. It is shown to describe the relationship of the interception scenario with other testing technologies such as *e* language (IEEE 1647). *e* is a Hardware Verification Language (HVL) mainly tailored to implement verification test benches. In this case an *e* test bench running in the initiator is bound to a TLM2 socket [32, 33]. The test bench provides the stimuli to carry out the verification of the target module.

3.1. TLM2 Interfaces Definition. SystemC is a C++ object-oriented framework for the description and simulation of systems. All of the system's building blocks, from basic signals to the abstract transaction level interfaces, are described using a class hierarchy, where complex classes are defined from the basic ones by means of inheritance. In a class hierarchy, it is common to find classes which define only an interface for its derived ones. No instance of a base class is actually created, only a description of an interface is given. This is done in C++ making the base class abstract, which means that at least one method is declared as pure virtual. When an abstract class is inherited, all pure virtual methods must be implemented, or the inherited class becomes abstract as well. Creating a pure virtual method allows an interface to be described without being forced to provide an implementation. The derived class must provide its own specific version of the virtual method. Several design patterns use polymorphism to invoke different functionality through a unique and standard interface. Polymorphism is a key concept in interfaces definition in C++ software designs.

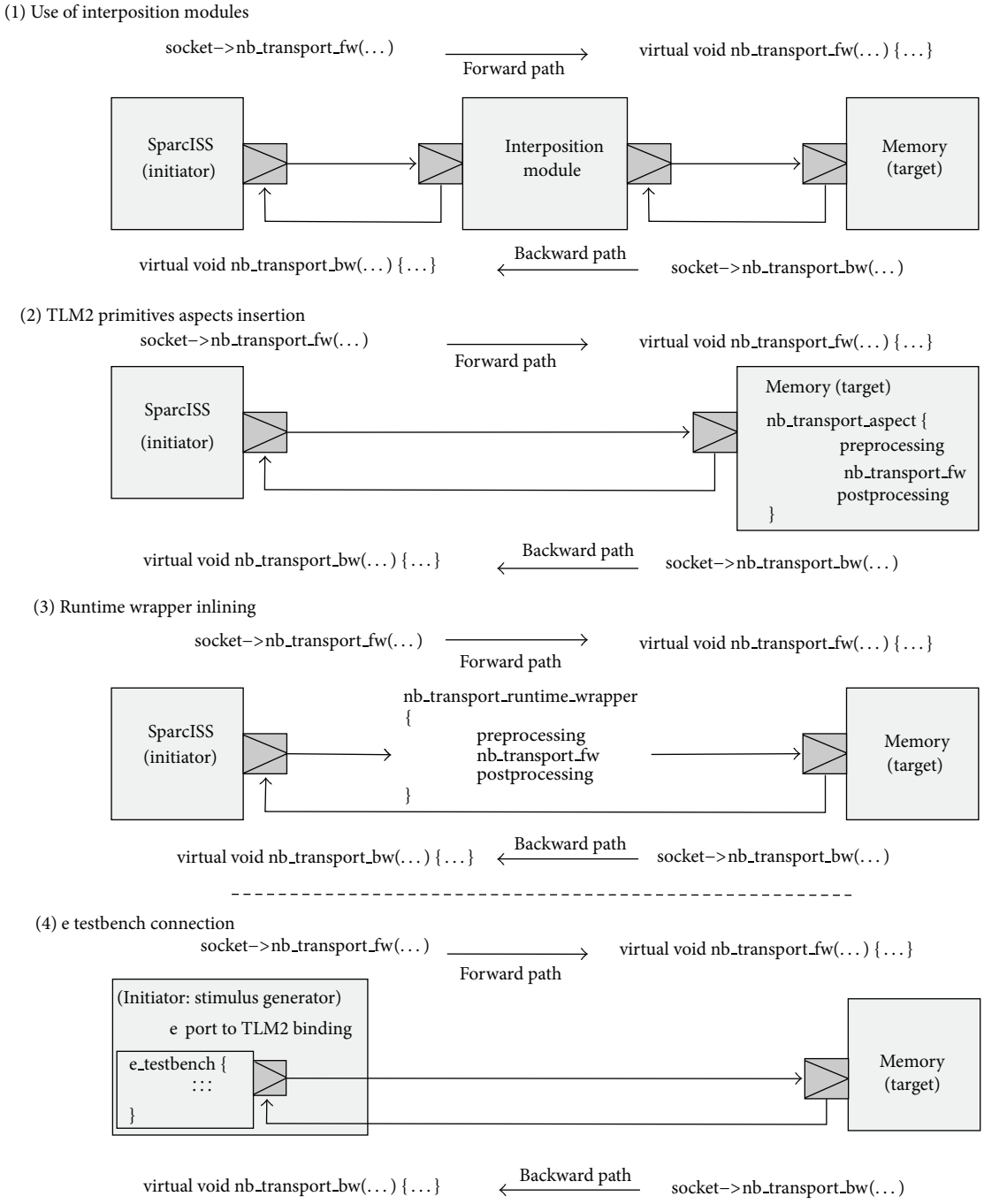


Figure 2: Interposition techniques and testbench languages.

The basic hierarchy of the TLM2 `tlm_fw_transport_if` interface implemented by a target socket is shown in Figure 3. This interface inherits the descriptions of four transport methods:

(i) `tlm_fw_nonblocking_transport_if`

(ii) `tlm_blocking_transport_if`

(iii) `tlm_fw_direct_mem_if`

(iv) `tlm_transport_dbg_if`.

All interfaces are abstract classes and use pure virtual methods to define the supported calls, their incoming parameters and their return values. As an example, the code of the `tlm_fw_nonblocking_transport_if` interface is also shown in Figure 3; it defines only one pure virtual

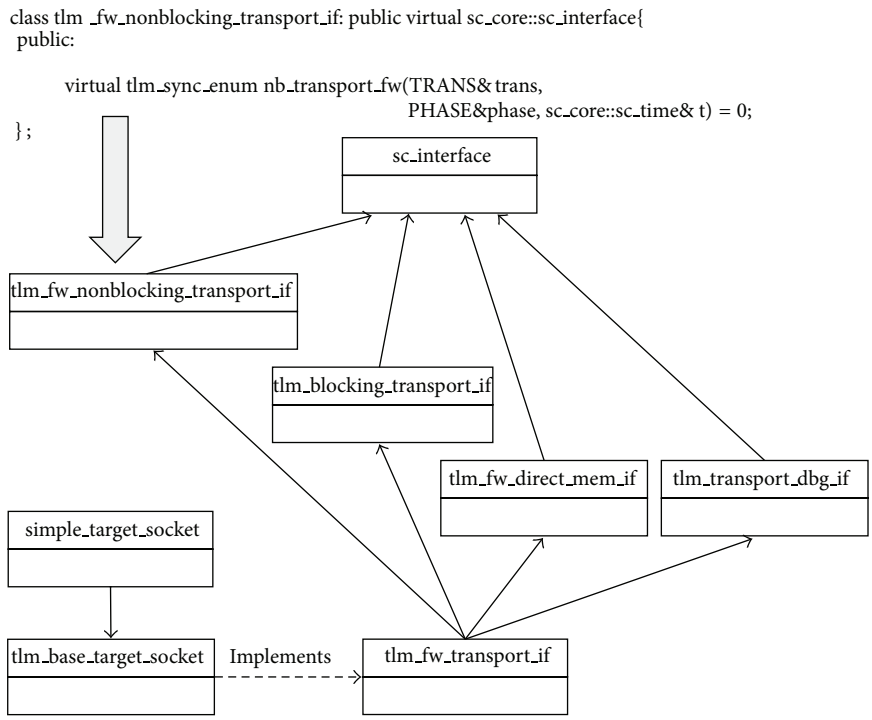


Figure 3: SystemC/TLM2 C++ forward interfaces hierarchy.

method called `nb_transport_fw`. Every instance of a target socket that implements this interface must provide its own implementation of the `nb_transport_fw` method.

3.2. C++ Polymorphism Implementation. Polymorphism is, with inheritance, one of the essential features of an object-oriented programming language like C++. It provides separation of interface definition from the particular implementation of that interface, decoupling "what" from "how". The virtual methods allow one type to express its distinction from other similar types, as long as they are both derived from the same base type. The distinction is expressed through a different implementation of the methods' behaviour. These methods must be called through base class pointers.

Connecting a function call to a function body is called *binding*. When binding is performed at the time of compilation, it is called early binding. On the other hand, late binding means the binding occurs at runtime, depending on the class of the object. Late binding is also called dynamic binding or runtime binding. When a language implements late binding, there must be some mechanism to determine the class of the object at runtime and call the appropriate method. In the case of a compiled language, the compiler still does not know the actual object class, but it inserts code that finds out how the invocation has to be resolved and finally calls the right method. Late binding only occurs with virtual methods, and only when the call is made through base class pointers.

Figure 4 describes the "big picture" of a virtual call. Each time a class containing virtual methods is created or derived from a class that contains virtual methods, the compiler creates a unique virtual method address table (VTABLE) for that class. In this table it places the addresses of all the methods that have been declared virtual in this class or in the base class. It is by using this table that the addresses of the invoked methods are obtained at runtime. Note that virtual tables are class specific and that there is only one virtual table for each class regardless of the number of object instances.

It is possible to modify the VTABLE and insert the address of a wrapper method to carry out the necessary processing of the transaction parameters and monitor the behaviour of the system. What is more is that the VTABLE can be duplicated so that the change only affects a particular class instance and not all of them. This is a great improvement on AOP programming, for it applies to the class definition regardless of the number of instances of the class. When wrapping is no longer needed, the VTABLE can be restored to its original value. It is important to note that the particular layout of the VTABLE is not explicitly defined in the C++ Application Binary Interface (ABI) so some particularities are C++ compiler dependent. However, all C++ compilers use similar approaches. Common C++ compilers have undocumented compilation switches that export the class layout. For Microsoft VisualC++, `-d1reportAllClassLayout` can be used. For GCC's g++ compiler, the switch is `-fdump-class-hierarchy`. The following tools and works are useful to find out the internal structure of C++ programs [34–36].

3.3. Binary Layout and Compiler Dependencies. When a call to a virtual function is made through a base class pointer (i.e., a late binding call), the compiler quietly inserts code to fetch the VTABLE pointer (VPTR) and look up the requested

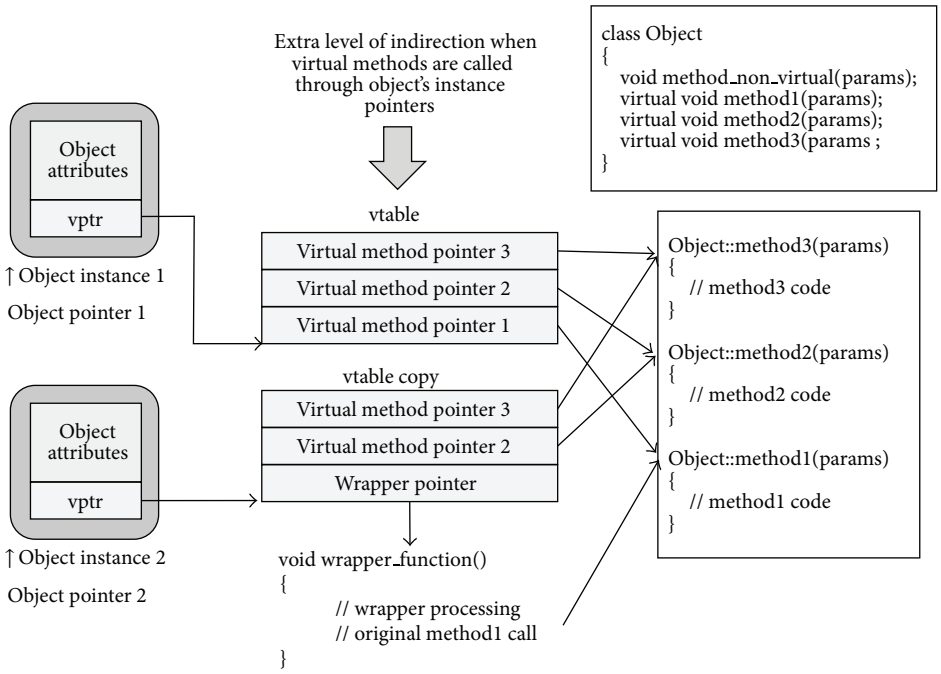


Figure 4: C++ virtual table basic structure.

method address in the VTABLE, thus calling it and causing late binding to take place. All of this VTABLE setting up for each class, initializing the VPTR and inserting the code for the virtual function call, happens automatically. Figure 5 shows the binary layout of `atlm_fw_transport_if` object and how a `b_transport` virtual method invocation using two different object pointers is resolved from the point of view of Microsoft VC++ and GCC 4.1 compilers. This situation is more complex since two key object-oriented features, such as polymorphic calls and multiple inheritance, are used at the same time. For TLM2 interfaces the arrangement of both compilers is as follows.

(i) Microsoft VisualC++ (VC) distributes the information of each inherited class by placing them one next to another. If an inherited class has virtual methods, a pointer to its particular VTABLE is included. Since each interface has only one virtual method, there are four VTABLEs with only one entry pointing to the particular implementation of each method. When a call is made through a derived class pointer, `tlm_fw_transport_if *p_fw` in Figure 5, the pointer is adjusted to point to its specific VPTR and class data. This pointer modification is known as *pointer fix-up*. Finally, since each interface has only one method, the first entry of the current VTABLE (i.e., entry 0) is used to call the right method.

(ii) The GCC 4.1 compiler places information of all classes at the beginning of the object. Depending on the kind of pointer used to call the method, different sets of VTABLEs are used. If a call is issued through the object's pointer, again `tlm_fw_transport_if *p_fw` in Figure 5, the derived class' VTABLE is used. On the other hand, if the call is made by means of the specific class pointer, for example `tlm_fw_transport_if *p_b`, the destination address is taken from its particular VTABLE. The latter case leads to the need for a pointer fix-up before the method code begins in order to allow access to the class' data. This pointer adjustment code is known as `thunk code`.

In short, VC makes a pointer adjustment before VTABLE lookup while GCC does it afterwards.

3.4. Wrapper Library Design. As the TLM2 interface methods are all "virtual," every call is made through the VTABLE of the object instance. Therefore it is possible to modify the VTABLE to insert some kind of function wrapper without modifying the original source code as described in Figure 4. The class diagram shown in Figure 6 details the relationship between all the elements that model the wrapper infrastructure. For simplicity, not all the attributes and methods are shown. The base class of the wrapper library is `TLM2_Wrapper`. This class provides the interface to handle all TLM2 interfaces. Since all of them receive the same processing, only the `b_transport` interface is explained on the understanding that the others work in the same way.

Here it is important to understand the "is a" and "has a" relationship between classes. `TLM2_Wrapper` is the base class of the hierarchy and provides a basic pass-through wrapping service. This basic behaviour can be particularized by the use of inheritance, redefining just one method of the base class. For example, the `Fault_Injection_Wrapper` in Figure 6 provides its own version of the `b_transport_processing` method. This class "is a" specialized `TLM2_Wrapper`.

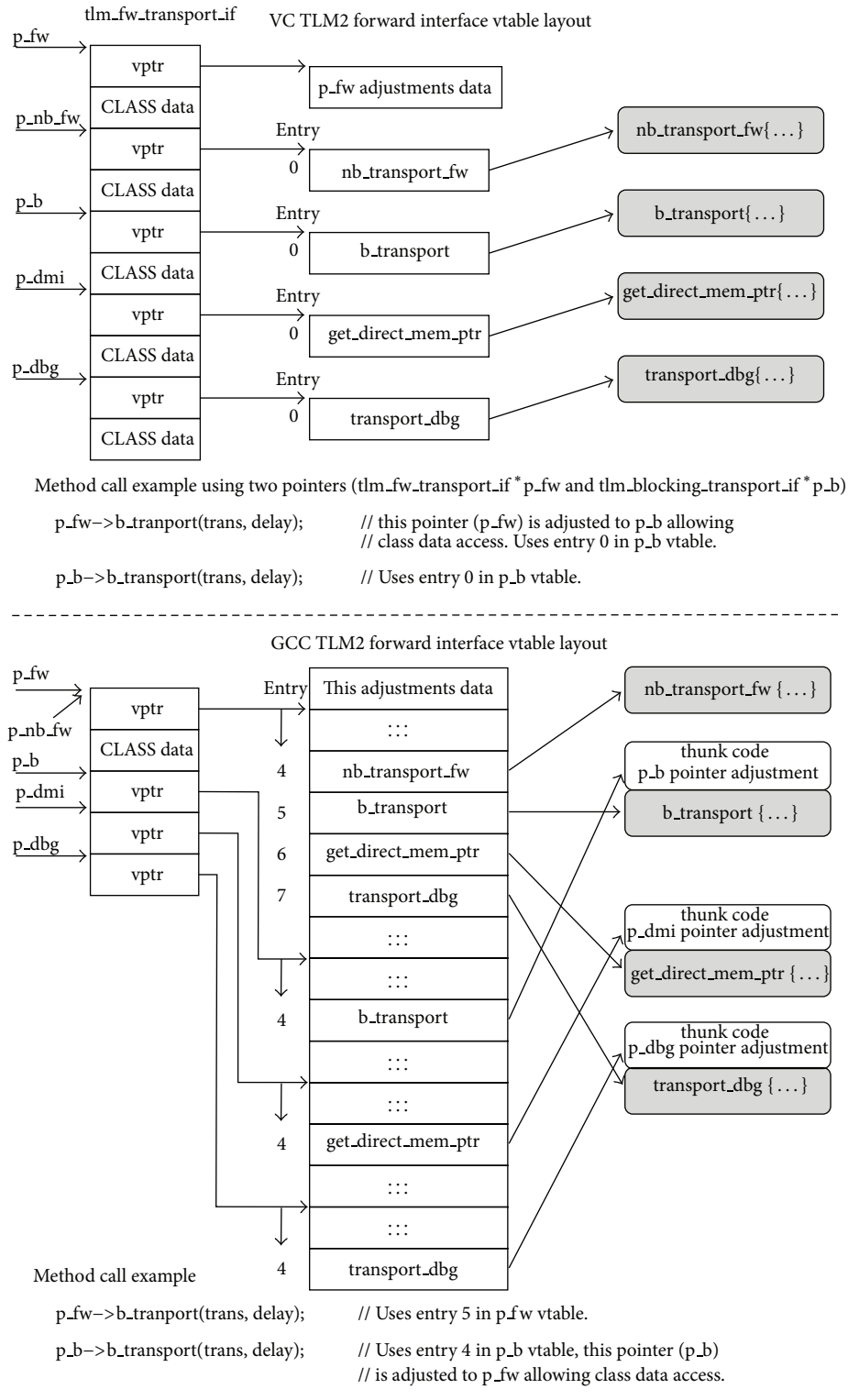


Figure 5: Microsoft VC versus GCC TLM2 forward interface binary layout.

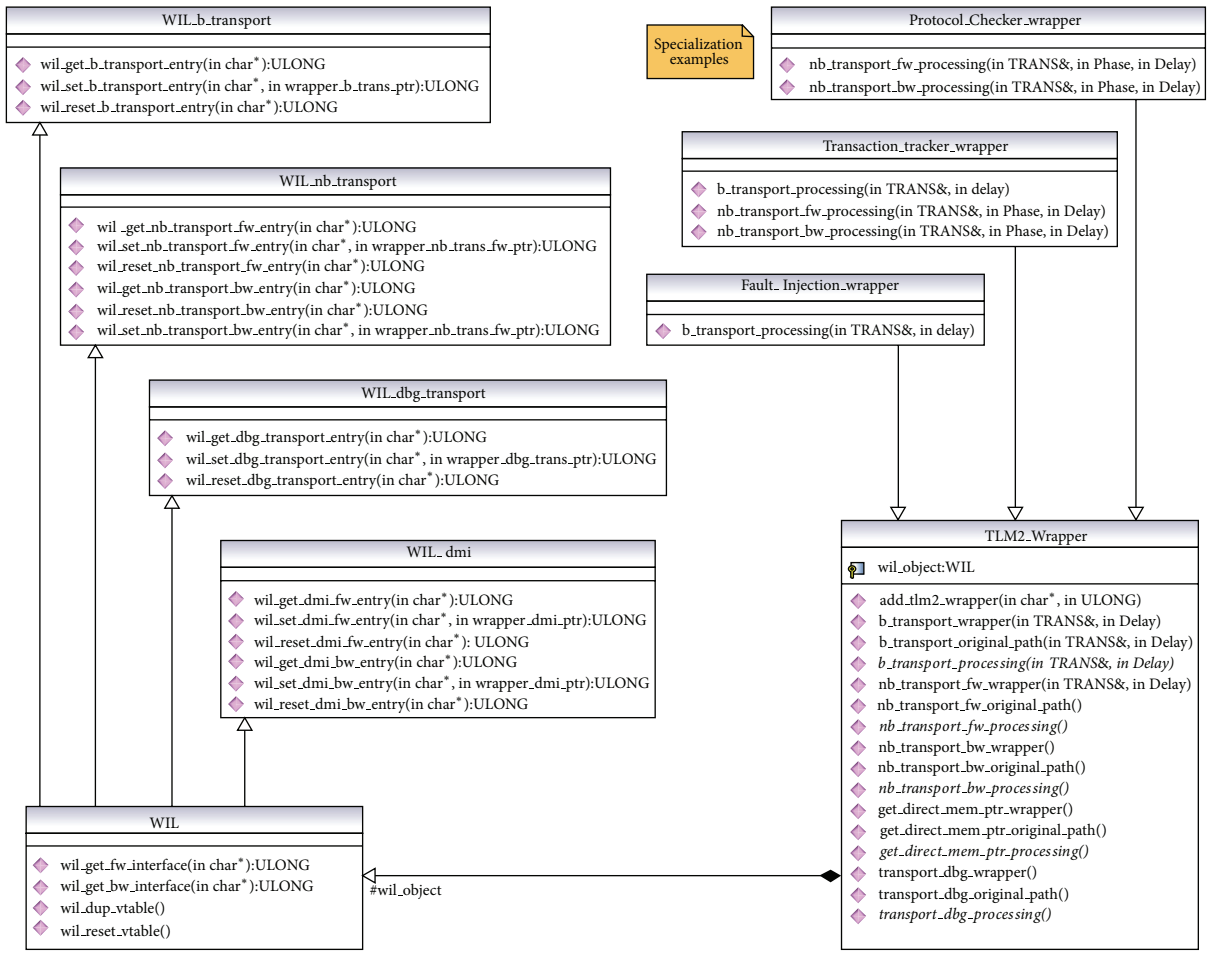


Figure 6: TLM2 wrapper class diagram.

On the other hand, the `TLM2_Wrapper` uses the low level services provided by the `WIL` class to modify and restore the VTABLE of the wrapped interface. This is a "has a" relationship. "WIL" stands for wrapper interception library and provides the methods to get, set, and reset the specific VTABLE entries of each TLM2 interface, taking into consideration the compiler binary dependencies. The declared interface for the `b_transport` handling is made up of the following methods.

(i) `b_transport_wrapper`: the start address of this method is inserted into the corresponding VTABLE entry of the intercepted target socket. It just redirects the call to `b_transport_processing`.

(ii) `b_transport_processing` (virtual): this method implements the desired behaviour of the wrapper. The default implementation in the base class just calls `b_transport_original_path` resulting in a pass-through mode. As it is declared virtual, this method is intended to be redefined in the derived class for specialized behaviour.

(iii) `b_transport_original_path`: this method is used to call the original entry of the modified VTABLE in order to maintain the overall transaction path.

The `WIL` class has four methods.

(i) `wil_get_fw_interface` is for internal use. This method gets the TLM2 forward interface pointer given the hierarchical SystemC name of the socket.

(ii) `wil_get_bw_interface` is for internal use. This method gets the TLM2 backward interface pointer given the hierarchical SystemC name of the socket.

(iii) `wil_dup_vtable` duplicates the VTABLE of an object, given its forward and backward TLM2 interface pointers.

(iv) `wil_reset_vtable` restores the original VTABLE of an object, given its forward and backward TLM2 interface pointers.

The specific handling methods for each interface are inherited from the `WIL_b_transport`, `WIL_nb_transport_fw`, `WIL_debug_transport`, and `WIL_dmi` classes. The handling methods for `WIL_b_transport` are

(i) `wil_get_b_transport` which gets the associated VTABLE entry value, given the hierarchical tical SystemC name of the socket.

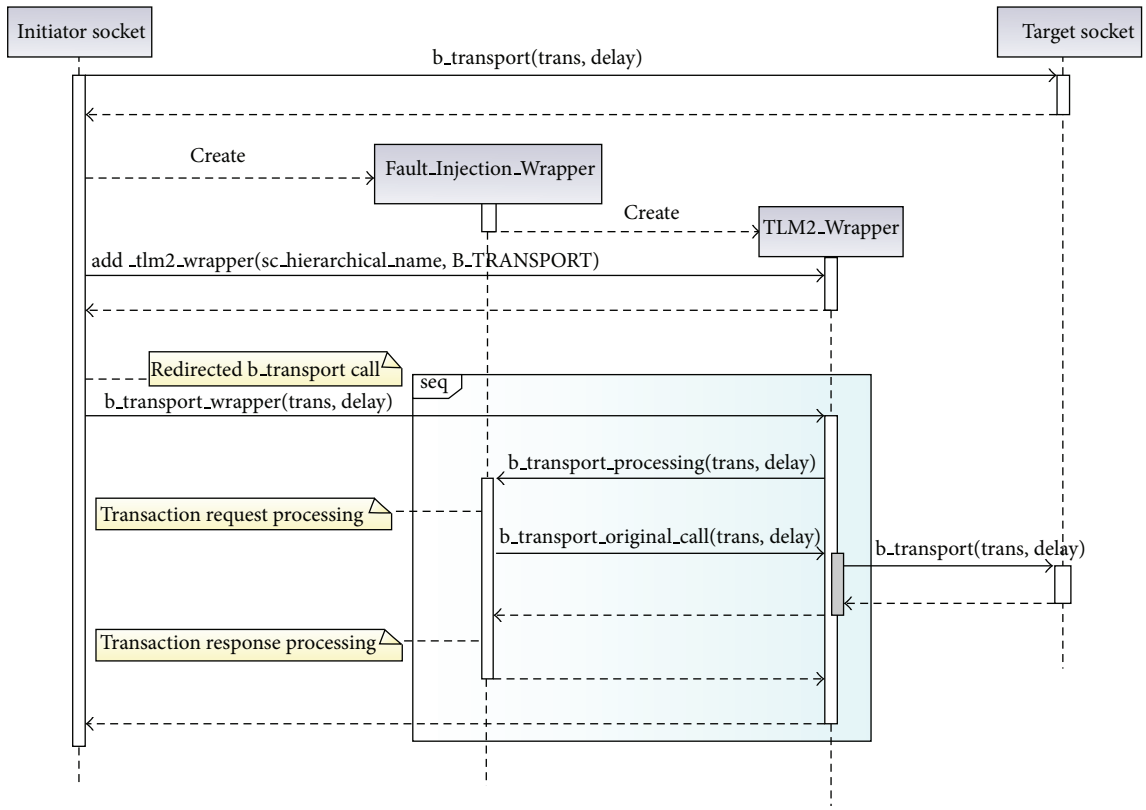


Figure 7: TLM2 wrapper sequence diagram.

(ii) `wil_set_b_transport` which sets a specific VTABLE entry with a new value. This new value is supposed to be a wrapper method address.

(iii) `wil_reset_b_transport` which restores the original value of a VTABLE entry.

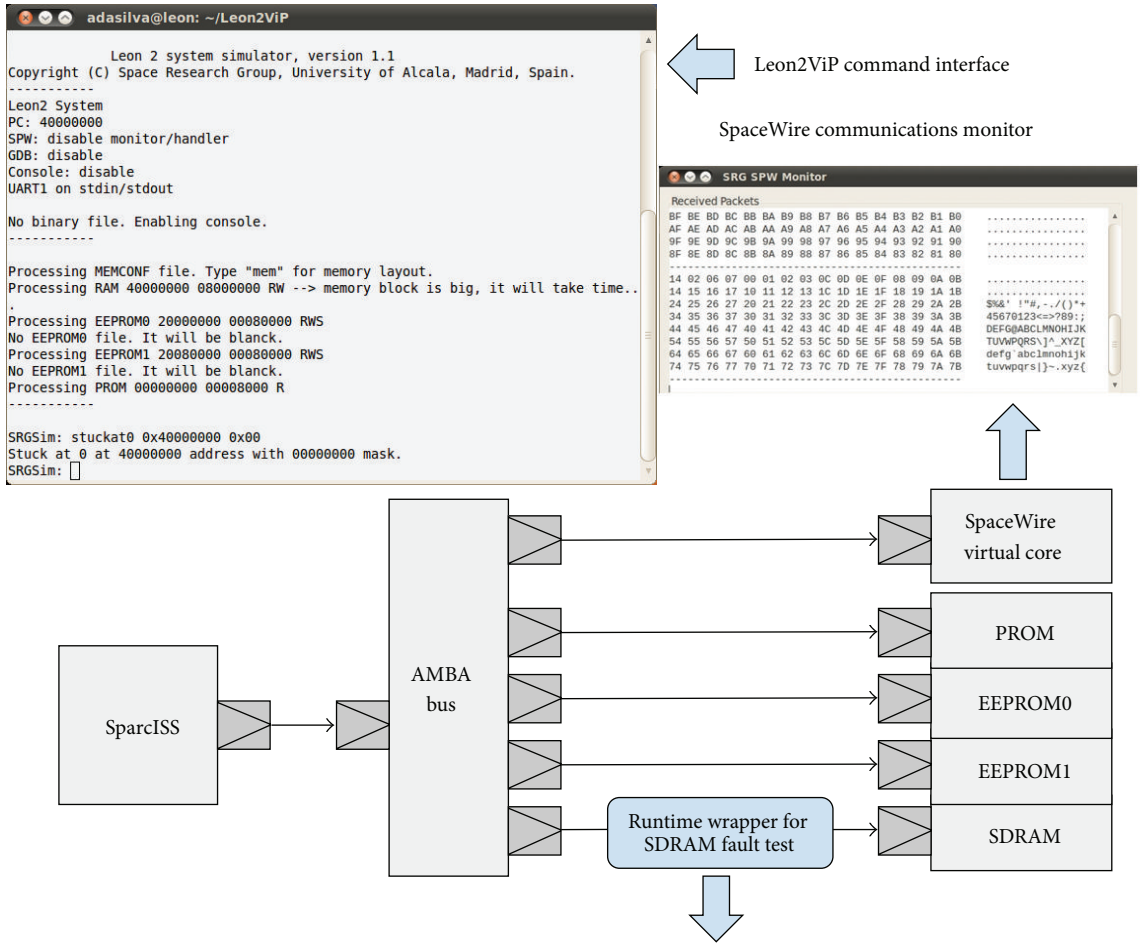
(iv) `wil_get_mapped_b_transport_entry` which is for internal use and gets the original entry mapped onto the actual wrapper. This allows the original values to be restored when it is desired.

Figure 7 shows the method's call sequence in a TLM2 `b_transport` intercepted environment. The main actor is an initiator socket bound to a target socket. The first sequence shows the nominal sequence when the initiator issues a `b_transport` call. As expected, the corresponding `b_transport` method is executed in the target. Next, in order to intercept these kinds of calls and track the information interchanged between both modules, a `Fault_Injection_Wrapper` is created by means of the C++ `new` operator. After the creation of the wrapper, the `add_tlm2_wrapper` method is invoked to insert the wrapper into the transaction path. To do this, the hierarchical SystemC name of the initiator socket is passed as a parameter. The other parameter is a constant value indicating the specific interface to be intercepted, `B_TRANSPORT` in this example. Once the wrapper is inserted, the same call issued by the initiator in the first sequence, is now redirected to the `b_transport_wrapper` method. As said, the default behaviour of this method is to call `b_transport_processing` but since it is declared virtual and redefined in the derived class, the `Fault_Injection_Wrapper` version of this method is called. `b_transport_processing` can now carry out a preprocessing of the incoming transaction parameters before calling the `b_transport_original_path` method that maintains the original transaction path. After the transaction is processed by the target socket, the call flow returns again to `b_transport_processing`, allowing some kind of postprocessing of the returned values. Finally, the return of this method leads to the completion of the original `b_transport` transaction call.

3.5. Usage Example. The following code snippets show an example of how to use the library. First, the declarations of an initiator and a target, both with a TLM2 socket attribute, are as follows (see Algorithm 1).

Given the previous declarations, the object instances are obtained through the `new` operator. Finally, both sockets are bound through the bind method of the initiator socket. From this moment, every TLM2 method call through the initiator socket brings about the execution of the corresponding response method in the target module (see Algorithm 2).

If it is necessary to intercept the call to the `b_transport` method of the target module, it is enough to define a `tlm2_wrapper` class with the particular processing method. In the example just `b_transport_processing` method is defined. This method can carry out some pre- and/or postprocessing




```
void tlm2_parser_wrapper::b_transport_processing(transaction_type& trans, sc_core::sc_time& delay)
{
    // cout << "---B_TRANSPORT Wrapping---" << endl;

   // if WRITE command and address match, apply MASK on data that is going to be written

   // Call target original b_transport on SDRAM
   b_transport_original_path_call( trans, delay);

   // if READ command and address match, apply MASK on data read
}
```

Figure 8: SoLO bootloader fault injection framework.

```
struct Initiator: sc_module
{
tlm_utils::simple_initiator_socket<Initiator> socket;
::::
struct Target: sc_module
{
tlm_utils::simple_target_socket<Target> socket;
::::
```

Algorithm 1

```
p_initiator = new Initiator("Initiator");
p_target    = new Target   ("Target");
p_initiator->socket.bind( p_target->socket );
::::
```

Algorithm 2

```
void tlm2_wrapper::b_transport_processing( transaction_type& trans,
                                           sc_core::sc_time& delay )
{
//    pre_processing( trans, delay );
cout << "---B_TRANSPORT Pre_wrapper---" << endl;
b_transport_original_path_call( trans, delay );
//    post_processing( trans, delay );
 cout << "---B_TRANSPORT Post_wrapper---" << endl;
}
```

Algorithm 3

```
tlm2_wrapper *p_wrapper = new tlm2_wrapper( "top.Initiator.socket_0" );
::::
p_wrapper->add_tlm2_wrapper( B_TRANSPORT );     // Wrapper inlining for
                                                // B_TRANSPORT interface
::::
top.p_initiator->socket->b_transport( *trans, delay ); // Call doesn't change
::::
p_wrapper->remove_tlm2_wrapper( B_TRANSPORT ); // Wrapper removing
```

Algorithm 4

of the incoming call parameters, as well as calling the original target method as is shown in Algorithm 3.

Finally, the wrapper class is instantiated through the C++ `new` operator. The constructor receives the hierarchical name of the initiator module as a parameter as defined by SystemC. By means of the `add_tlm2_wrapper` method, the wrapper is inserted into the transaction path. In this example only the `B_TRANSPORT` interface is intercepted (see Algorithm 4).

4. Performance and Usability Issues

The interception library has been coded in a C++ object-oriented style. This makes the specialization of the wrappers very easy through inheritance. The library has less than 2.000 lines of code as shown in Table 1 along with the amount of code that is compiler dependent.

The source code of TLM2 Wrapper Library V1.0 can be found at https://dl.dropbox.com/u/19987939/TLM2Wrapper.zip. The usage example distributed in the main module defines just two TLM2 modules and binds them in the usual way. After that a pass-through TLM2 wrapper is instantiated and the wrapper is inlined in the original transaction path. After invoking all TLM2 primitives, wrapper is removed and the main program ends. Table 2 summarizes the code coverage of the different modules. Data has been obtained through GCOV, the GNU source code coverage analysis tool. As shown in Figure 6 `wil_b_transport` and `wil_dbg_transport` have three methods plus class constructor, `wil_nb_transport` and `wil_dmi` have six methods plus class constructor and `wil` class inherits the twenty-two methods of the previous classes plus its own class constructor. As seen in Table 2 all basic operations for inserting/removing wrappers in TLM2 primitives are all exercised.

Table 3 summarizes the results of a series of performance tests carried out to estimate the overhead associated with the insertion of the wrapper into the transaction path using different kinds of techniques. Some usability considerations

Table 1: Interception library compiler dependant code.

Total lines	Visual C++ specific code lines	GCC specific code lines
1895	109	44

Table 2: Interception libary distribution example coverage.

Module	Lines executed	Branches	Calls
Main	77.9% of 86	80.7% of 52	56.2 of 105
tlm2_wrapper	73.4% of 263	69.7% of 172	60.56% of 142
wil	95.52% of 67	100% of 50	100% of 23
wil_b_transport	89.5% of 19	100% of 12	100% of 4
wil_nb_transport	67.6% of 37	100% of 20	100% of 7
wil_dmi	67.6% of 37	100% of 20	100% of 7
wil_dbg_transport	89.5% of 19	100% of 12	100% of 4

Table 3: Performance and usability issues.

Insertion technique	Single call time overhead	TOP model source code modification	Insertion time	Removing time
Interposition module	30%	Yes	Compile time	Not allowed
VTABLE hooking	20%	No	Run time	Run time

are also included. All tests were carried out using a 1,66 GHz generic laptop with 1 Gbyte RAM memory using Microsoft Windows 7 Professional Service Pack 1. The inserted wrapper implements a pass-through functionality and just calls the target method. No modifications to the input parameters or return values are made. For each initiator socket transaction, the processing time is measured using the time-stamp counter RDTSC [37], present in Intel processors. A series of 1.000 calls to `b_transport` target method were made using the following two configurations:

(i) use of an interposition module between initiator and target;

(ii) use of virtual table hooking to interpose a method wrapper in the transaction path using the C++ library presented in this paper.

Given a TLM2 model, the introduction of any interposition code always introduces some time penalties. Even more if extensibility is design concern. The results show that virtual table hooking used by the interception library introduces a time overhead close to but lower than that obtained with an interposition module. Other important point to consider is that with this technique, only the specific desired call is intercepted while the other calls of the TLM2 interface run in the usual way. Using interposition implies that all methods of the socket interface are intercepted and thus delayed. In the case of the virtual table hooking wrapper, it can be inserted and removed at runtime. Other improvement of the virtual hooking method is that it can be applied to a single object instance instead of a class. This concern is exploited in the library usage example described in Section 5.

5. Inline Wrapper Library Fault Injection Use Case

This section presents a real testing scenario of the ICU's boot software using the Leon2ViP virtual platform and the TLM2 wrapper insertion library. Since the internal architecture of the Leon2ViP virtual platform is not the main focus of this paper, Figure 8 just shows only the main components, stressing the location of the fault injection wrappers:

(i) LEON2 ISS: SPARC V8 untimed Instruction Set Simulator with blocking TLM2 transaction interfaces.

(ii) Memory: PROM, EEPROM, and SDRAM blocks. The memory layout is highly configurable through an external configuration file and the current contents can be read from an external ordinary binary file generated by the compiler toolchain.

(iii) Bus: this module interconnects all the TLM2 components of the virtual platform.

(iv) SpaceWire: Virtual SpaceWire IP core for spacecraft communications. This interface can be mapped onto a SpaceWire monitor or an external SpaceWire hardware based on a Star-Dundee USB SpaceWire brick [38].

For controlling the software execution, Leon2ViP offers a command-line interface with which the user can issue commands for loading memory binaries, for inspecting the internal state of the processor and the memory blocks and for defining breakpoints/watchpoints, among many others. SDRAM and PROM areas are usually filled with external program binary files using a `load` command. Leon2ViP also emulates EEPROM areas; the memory contents of these areas are filled at the system start-up with the contents of a file which has the same name as the memory block. The file should have the same size as the memory block and must contain its binary image. When the execution ends, the contents of the memory blocks are written back to the files. This emulates the behaviour of an EEPROM when data updates are carried out on it using the software data protection (SDP) schema implemented in commercial EEPROM modules.

One of the strictest requirements of the flight software is that the application binaries should be updatable in flight. This characteristic allows the system to recover from application bugs discovered after launch itself. There are two versions

of the ICU application software. EEPROM0 module contains the *Baseline* version and EEPROM1 module contains an *Updatable* version. In accordance with the ICUSW robustness requirements [39], the boot software takes control of the ICU automatically after a reset or a power-on and is in charge of deploying the main application software. The boot software verifies the integrity of the two versions of the application software stored in the EEPROM modules, as well as the integrity of the memory on which they are deployed. Depending on the result of this checking, it will deploy one of the two versions of the application software. If both application software images are damaged, the boot software will wait and remain active until the images have been repaired by means of service telecommands, sent from Earth, that allow memory patch, dump, and check operations. The SDRAM will be tested in order to find possible failures that may condition the system's stack allocation and application binary deployment resulting from stuck-at faults. The stack allocation takes into account that only 32 kibibytes of not damaged SDRAM are required for running the boot software. Again, when neither of the software application versions can be deployed, the ICU boot software must wait for spacecraft telecommands that will load a new version of the application software that avoids the damaged SDRAM areas.

In Figure 8 the user issues a `stuckat0` command at the 0x40000000 address. This means that every read/write transaction on the selected address must be tied to a logical zero. To carry out this memory access corruption a wrapper is inserted into the SDRAM memory path. As long as the wrapper is inserted the fault remains active. The fault injection wrapper developed using the interception library just redefine the `b_transport_processing` method in order to apply the mask to the data read/written in the transaction and is inserted just into the transaction path of the memory module under fault simulation. For example, in order to test the boot behaviour when SDRAM has stuck-at faults the fault injection wrapper is inserted in the SDRAM path. This leaves the PROM and EEPROM transaction paths unchanged; thus fetch operations on PROM boot code and read operations on EEPROM modules are not delayed. A complete description of the first results on testing ICU boot can be found in [40].

6. Conclusions and Future Work

In this paper, we have presented a library for runtime wrapper insertion in TLM2 models. The results presented here are encouraging and show that it is possible to insert transaction wrappers in an easy way with a minimal time overhead and with a great improvement on previous approaches such as interception modules. The proposed technique has been successfully applied on a real early software development and robustness verification scenario. In addition, the technique adopted here is applicable in third-party software component validation, without having access to the component source code. It has the disadvantage of using some not-well-documented and compiler-dependent features. The library design has been made by thinking about its usability and easy customization. This feature greatly reduces deployment times for real-world testing scenarios. The runtime nature of the wrapper inlining has an improved effect overall system simulation since wrappers can be installed on the fly if needed or when some special runtime situation in the system is reached. In the same way, the wrappers can be removed when they are no longer needed. In addition, since the insertion/removal of wrappers is time-consuming and error-prone, automating this task also ensures that the testing process does not compromise the correctness of the final system. A C style version of the library is under development in order to reduce the time penalty introduced in the current version for a single call.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work has been supported by Spanish Ministerio de Economía y Competitividad under the Grants AYA2011-29727-C02-02 and AYA2012-39810-C02-02.

References

[1] T. A. Henzinger, "Two challenges in embedded systems design: predictability and robustness," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 366, no. 1881, pp. 3727–3736, 2008.

[2] L. Rabelo, S. Sala-Diakanda, J. Pastrana et al., "Simulation modeling of space missions using the high level architecture," *Modelling and Simulation in Engineering*, vol. 2013, Article ID 967483, 12 pages, 2013.

[3] OSCI, Open SystemC Initiative, SystemC, 2013.

[4] ESA, "Solar orbiter. Exploring the Sun-heliosphere definition study report," Tech. Rep. I-CA2301, European Space Agency, Paris, France, 2011.

[5] A. da Silva and S. Sánchez, "LEON3 ViP: a virtual platform with fault injection capabilities," in *Proceedings of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD '10)*, pp. 813–816, Lille, France, September 2010.

[6] S. Sánchez, M. Prieto, Ó. R. Polo et al., "HW/SW co-design of the instrument control unit for the energetic particle detector on-board solar orbiter," *Advances in Space Research*, vol. 52, no. 6, pp. 989–1007, 2013.

[7] A. da Silva and S. Sánchez, "On the use of dynamic binary instrumentation to perform faults injection in transaction level models," in *Proceedings of the 4th International Conference on Dependability of Computer Systems (DepCos-RELCOMEX '09)*, pp. 237–244, July 2009.

[8] F. Randimbivololona, A. Brahmi, P. le Meur, T. Marie, and R. Beseme, "Final integration test of avionic software in full virtual platform," in *Proceedings of the Embedded Real Time Software and Systems (ERTS2 '14)*, Toulouse, France, February 2014.

[9] Y. Wang, L. Wang, and Z. Zheng, "Application of virtual prototype technology to simulation test for airborne software system," in *Advances in Electronic Engineering, Communication and Management Vol. 2*, vol. 140 of *Lecture Notes in Electrical Engineering*, pp. 653–658, Springer, Berlin, Germany, 2012.

[10] N. Bombieri, F. Fummi, and D. Quaglia, "System/network design-space exploration based on TLM for networked embedded systems," *Transactions on Embedded Computing Systems*, vol. 9, no. 4, article 37, 2010.

[11] S. Cuenca-Asensi, A. Martínez-Álvarez, F. Restrepo-Calle, F. R. Palomo, H. Guzmán-Miranda, and M. A. Aguirre, "Soft core based embedded systems in critical aerospace applications," *Journal of Systems Architecture*, vol. 57, no. 10, pp. 886–895, 2011.

[12] M. Straka, J. Kastil, Z. Kotasek, and L. Miculka, "Fault tolerant system design and SEU injection based testing," *Microprocessors and Microsystems*, vol. 37, no. 2, pp. 155–173, 2013.

[13] W. Lu and M. Radetzki, "Concurrent and comparative fault simulation in SystemC and its application in robustness evaluation," *Microprocessors and Microsystems*, vol. 37, no. 2, pp. 115–128, 2013.

[14] C. Ziemke, T. Kuwahara, and I. Kossev, "An integrated development framework for rapid development of platform-independent and reusable satellite on-board software," *Acta Astronautica*, vol. 69, no. 7-8, pp. 583–594, 2011.

[15] R. Mader, G. Griessnig, E. Armengaud et al., "A bridge from system to software development for safety-critical automotive embedded systems," in *Proceedings of the 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA '12)*, pp. 75–79, September 2012.

[16] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "A case of system-level hardware/software co-design and co-verification of a commodity multi-processor system with custom hardware," in *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '12)*, pp. 513–520, ACM, New York, NY, USA, October 2012.

[17] S. Misera, H. T. Vierhaus, and A. Sieber, "Simulated fault injections and their acceleration in SystemC," *Microprocessors and Microsystems*, vol. 32, no. 5-6, pp. 270–278, 2008.

[18] K. J. Chang and Y. Y. Chen, "System-level fault injection in SystemC design platform," in *Proceedings of the 8th International Symposium on Advanced Intelligent Systems*, pp. 354–359, 2007.

[19] A. Ebnenasir, R. Hajisheykhi, and S. S. Kulkarni, "Facilitating the design of fault tolerance in transaction level SystemC programs," *Theoretical Computer Science*, vol. 496, pp. 50–68, 2013.

[20] W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull, "Interactive presentation: implementation of a transaction level assertion framework in SystemC," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '07)*, pp. 894–899, EDA Consortium, San Jose, Calif, USA, 2007.

[21] A. Habibi and S. Tahar, "Design and verification of SystemC transaction-level models," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 1, pp. 57–68, 2006.

[22] B. Niemann and C. Haubelt, "Assertion-based verification of transaction level models," in *Proceedings of the In ITG/GI/GMM Workshop*, vol. volume 9, pp. 232–236, 2006.

[23] A. A. Ghofrani, S. Abolma'Ali, Z. N. Haghi, and Z. Navabi, "A TLM2.0 assertion library with centralized monitoring approach," in *Proceedings of the IEEE East-West Design and Test Symposium (EWDTS '10)*, pp. 402–406, East-West, September 2010.

[24] L. Pierre and L. Ferro, "A tractable and fast method for monitoring systemC TLM specifications," *IEEE Transactions on Computers*, vol. 57, no. 10, pp. 1346–1356, 2008.

[25] R. Tartler, D. Lohmann, F. Scheler, and O. Spinczyk, "AspectC++: an integrated approach for static and dynamic adaptation of system software," *Knowledge-Based Systems*, vol. 23, no. 7, pp. 704–720, 2010.

[26] D. Tabakov and M. Y. Vardi, "Automatic aspectization of SystemC," in *Proceedings of the Workshop on Modularity in Systems Software (MISS '12)*, pp. 9–14, New York, NY, USA, March 2012.

[27] Y. Endoh, "ASystemC: an AOP extension for hardware description language," in *Proceedings of the 10th International Conference on Aspect-Oriented Software Development Companion (AOSD '11)*, pp. 19–28, ACM, New York, NY, USA, March 2011.

[28] M. Kallel, Y. Lahbib, R. Tourki, and A. Baganne, "Aspect-based ABV for SystemC transaction level models," in *Proceedings of the 21th International Conference on Microelectronics (ICM '09)*, pp. 304–307, December 2009.

[29] M. Kallel, Y. Lahbib, R. Tourki, and A. Baganne, "Verification of systemc transaction level models using an aspect-oriented and generic approach," in *Proceedings of the 5th Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS '10)*, pp. 1–6, March 2010.

[30] A. da Silva and S. Sánchez, "Transactions sequence tracking by means of dynamic binary instrumentation of TLM models," in *Proceedings of the 12th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD '09)*, pp. 723–728, August 2009.

[31] A. da Silva and S. Sanchez, "A grammar based testing framework for T LM2.0 protocol compliance verification," in *Proceedings of 5th International Conference on Dependability of Computer Systems DepCoS, Monographs of System Dependability (RELCOMEX '10)*, Technical Approach to Dependability, pp. 121–132, Oficyna Wydawnicza Politechniki Wrocławskiej, Wroclaw, Poland, 2010.

[32] B. Bailey, F. Balarin, M. McNamara, G. Mosenson, M. Stellfox, and Y. Watanabe, *TLM-Driven Design and Verification Methodology*, Lulu Enterprises Inc., 2010.

[33] H. Froehlich, "Interface Additions to the e Language for Effective Communication with SystemC TLM 2.0 Models," Cadence, 2012.

[34] A. Fokin, K. Troshina, and A. Chernov, "Reconstruction of class hierarchies for decompilation of C++ programs," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR '10)*, pp. 240–243, March 2010.

[35] N. A. Kraft, B. A. Malloy, and J. F. Power, "A tool chain for reverse engineering C++ applications," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 3–13, 2007.

[36] I. Skochinsky, "Practical C++ decompilation," in *Proceedings of the Recon 2011*, Montreal, Canada, 2011.

[37] RDTSC, "Using the rdtsc instruction for performance monitoring," 2009.

[38] STAR Dundee, "Star Dundee Brick," 2013.

[39] O. R. Polo and P. Parra, "SOLO EPD flight software requirements," Tech. Rep., Space Research Group, University of Alcalá, 2012.

[40] A. da Silva, S. Sánchez, Ó. R. Polo, and P. Parra, "Injecting faults to succeed. Verification of the boot software on-board solar orbiter's energetic particle detector," *Acta Astronautica*, vol. 95, no. 1, pp. 198–209, 2014.