Security EDA Extension through P1687.1 and 1687 Callbacks

Michele Portolan, Vincent Reynaud, Paolo Maistri, Regis Leveugle, Giorgio Di Natale

Univ Grenoble Alpes, CNRS, Grenoble INP[1], TIMA, 38000 Grenoble, France

{name.surname}@univ-grenoble-alpes.fr

***Abstract*— In recent years, the world of VLSI testing has been living a huge transformation pushed by constraints and requirements coming from a large variety of sources and applications. The traditional need for higher accessibility and controllability led to solutions such as IEEE 1687, while the need for reuse is pushing for innovations like P1687.1. All the while, these same features are raising security concerns for malicious attacks or reverse engineering. Standards usual approach of relying on Domain-Specific Languages to convey information to the EDA tools has difficulty in handling such disparate and often conflicting needs, with the risk of a dangerous proliferation of custom and incompatible solutions. In this paper, we show how the usage of Callbacks, defined in P1687.1, can help solve this issue.**

***Keywords*— Automated Test Environments, Domain-Specific Languages, Callbacks, Reconfigurable Scan Networks, Secure Access, Authentication**

INTRODUCTION

Today's System-on-Chips (SoCs) are subject to extremely different and often contradictory requirements. On the one hand, their usage in critical applications such as automotive pushes for functional safety and high reliability, both in terms of screening for fabrication defaults and in life-time error detection and handling. During the years, several standards for Design for Test (DfT) have been developed to help achieve high testing quality and ease access to embedded resources, the most important and widespread being the derivation of the IEEE 1149.1 (JTAG) [1], such as IEEE 1500 [2] and more recently IEEE 1687 [3] and P1687.1 [4]. These standards propose both common hardware constructs to allow plug-and-play composition of resources developed by different actors and software automation thanks to Domain-Specific Languages. On the other hand, SoCs are increasingly carrying confidential or sensitive information: DfT can easily become an Achilles' heel for malicious attackers trying to tamper with the system. While solutions have been proposed to secure parts of the DfT infrastructure, they often require important modifications to both the hardware and, more importantly, the automation flow: the incorporation of non-standard features requires a huge amount of workaround and/or custom software patches, considerably slowing down widespread adoption and time-to-market.

The problem of Security, from a Standard's point of view, is its evolution speed: Fault Models depend on physical and fabrication characteristics of a given technology. Once they have been specified, they will of course be refined by new research and characterization, but the basis will remain the same and therefore the same standard will be able to treat them with minimal modification. For instance, IEEE 1149.1 [1] did not change much from its first issue in 1991. On the other hand, new Attacks are developed at an alarming speed and they purposefully aim the weakest points of a system. Updating a Standard is a long and fastidious process and it unfeasible to do it for each new threat. It is therefore fundamental to provide easy ways of extending it while maintaining a stable base, so that custom solutions can be added in a timely fashion : ideally, the best one will eventually be included in future Revisions. In our previous works, we showed how the Automated Test Flow itself can be extended thanks to P1687.1 Callbacks [5], and how a dynamic challenge/response authentication mechanism can be included inside an IEEE 1687 flow [4]. In this paper, we will leverage these approaches to propose a fully unified solution that allows plug-and-play deployment of integrated Security features inside a Standard Automated Test Flow.

The paper is organized as follows: first we will provide a State of the Art of the current solutions, both in terms of Security and Automated Flow. Section 2 will introduce our approach, which in Section 3 will be applied to the security field, most notably by leveraging the Encrypted SIB. Lastly, Section 4 will draw conclusions and point out future evolutions.

1 STATE OF THE ART

1.1 Scan Securization methods

The security of the test infrastructure can be addressed according to two main issues: access control and confidentiality. In this section, we summarize the latest state of the art with respect to these two problems.

Access control to the test infrastructure aims at allowing only authorized users to access the internal state of the scan chain. This addresses the major issue of using the test port as a backdoor to the system, without resorting to definitive techniques such as physically removing the connection (e.g., fuse blowing, removing the connector, etc.). Such definitive techniques are useless facing some categories of attackers and are not acceptable when the test port is also used for in-field monitoring or updates. The basic principle of access control is that only users with an authentication token (usually, a secret

[1]Institute of Engineering Univ. Grenoble Alpes

key) can access the scan chain. Such mechanism should have minimal impact on the cost and performance of the test controller, while at the same time being secure and with adjustable granularity. Several different users (or categories of users) might require different access privileges to the system. The opportunity of changing the secret key discourages hardwiring the access condition into the circuit, as in [6]: their solution, named Locking SIB, uses a Boolean condition on the content of additional scan flip-flops, added to the existing scan chain, to open a Segment Insertion Bit (SIB), but it is vulnerable to replay attacks as well. This vulnerability is addressed in solutions based on challenge-response protocols, such as Fine-Grained Access (FGA) [7] or Segment-Set Authorization Keys (SSAK) [8]. These solutions exploit a secret previously shared and generate a session-dependent access key starting from a random value, generated in the circuit, and exchanged through the scan chain.

The above solutions, based on a robust protocol, ensure that the internal state of the circuit is accessible only to authorized users. In these circumstances, hence, an attacker is not able to tamper with the circuit but may still be able to eavesdrop and monitor the exchanges over the scan chain. This can be addressed by encrypting the scan data transiting over an insecure channel, such as the test port. Test vectors are encrypted on the server before sending the values to the test port, and need to be correctly decrypted in the chip before being used. Likely, the test results are encrypted before leaving the sensitive region in the chip, and will be decrypted off-chip by the tester with the proper key. The choice of the most suitable cryptographic primitive depends on the targeted tradeoff between security, flexibility, and cost. Block ciphers are well-known and robust [9], but their fixed block size is not suited to encrypt sequences of varying length, such as Reconfigurable Scan Chains (RSNs)RSNs, and padding is nonetheless required if the total length does not correspond to an integer number of blocks. In this respect, stream ciphers are more flexible [10], as they are able to provide an endless stream to mask the test vectors, independent of the length or structure of the scan network. Stream ciphers produce an encrypted output, bit by bit, starting from a secret key and an Initialization Vector (IV): while the former is usually chosen by the user (or the designer), the latter has to be provided by the circuit in order to avoid potential attacks by a malicious user. Hardwiring the IV into the circuit is a possibility [11], but in this case the stream cipher can be broken by analyzing the stream output. The circuit should hence be able to generate itself a random IV value, which can be shared to the user by extending the Instruction Set of the Test Controller [12]. Security is still preserved, as the user can read such value, but not able to control it.

In general, the encryption process has been proposed on the full test vector: once inside the SoC, considered as a trusted region, the data is decrypted and used, and responses are re-encrypted before reaching the scan-out port. If several blocks (IPs) are in the same scan chain, each of them can intercept the decrypted flow. In order to allow testing both secure and insecure IPs on the same scan path, cryptographic primitives should be placed at the interface of each protected segment: by doing so, insecure IPs would not be able to read the protected parts of the test vector. A first solution has been proposed in [13], where an Encryption SIB (eSIB) extends the Secure SIB (SSIB) [7], [8] to include encryption capabilities at negligible cost, but with an additional dynamic constraint. The cost is very low due to the reuse of authentication circuitry. In global approaches, the generation of the key stream is straightforward, as the encryption process is synchronous with the test vector; if confidentiality is used at IP-level, then the encrypted stream has to be properly aligned at the input and the output of the protected instrument. Moreover, the process is made even more complex if the structure of the scan chain is reconfigurable, which requires the tester to have a dynamic model representative of the internal state of the system. The approach and tool presented in this paper provide an efficient solution to this problem.

1.2 Automated Test Flow and Domain-Specific Languages

The term « Automated Test Flow » resumes all the steps that allow to generate and execute test operations on a given System Under Test. The principle, depicted in Figure 1, is pretty simple: a Test Generation Tool (TGT) is given a set of files containing information about the Design and its Design-for-Test (DfT) features, combines them with a set of Targets (typically, the Fault Models) and obtains a set of output files: these might be both Pattern files and other types of DfT files. This process can be iterated several times with different TGTs, and the final set of Output files is sent to the Execution backend to be applied to the Design Under Test.

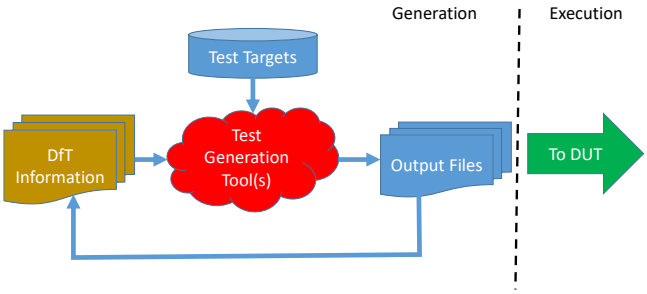


Figure 1 Traditional Test Generation Flow

As an example, this could be a typical sequence:

1) Input: Synthethised Verilog Netlist
   Targets: ATPG Insertion
   Output: Verilog with Inserted Scan Chains
2) Input: Verilog with Inserted Scan Chains
   Targets: ATPG Generation
   Output: STIL pattern file
3) Input: Verilog with Inserted Scan Chains + STIL pattern file
   Targets: Scan Compression
   Output: Verilog with Test Compression + Compressed STIL pattern file

4) Input: Verilog with Test Compression + Compressed STIL pattern file

   Targets: JTAG Wrapping

   Output: BSDL + SVF pattern file

The exact sequence depends of course on the DfT/Design strategies, the EDA Toolchain and the Execution backend, but there is a common basis: information exchange through files. For this, the Test Flow provides a multitude of Domain-Specific Languages (DSL), i.e. languages able to provide information about a particular step. While each EDA provider has his own set of DSLs, the necessity of simplification and inter-operability quickly pushed for a development of a common set of DSLs. Each standard has at least one: 1149.1 [1] has BSDL, 1500 CTL[2], 1687 ICL and PDL [2], etc.

While Hardware Description Languages like Verilog and VHDL focus on the hardware itself ("What is the Design") with synthesis in mind, Test DSLs rather focus on describing which DfT features described in a given Standard are implemented ("What is inside the design?") and TGT tools aim at using them. This is possible because Test Standards usually prescribe a set of DfT constructs in their Hardware parts, so the DSL does not need to explain their functioning.

For instance, 1149.1 dictates that a compliant system must have a standardized TAP controller connected to one Instruction Register (IR) and one or more Test Data Registers (TDR), as depicted in Figure 2. It also specifies the expected behavior of the System when a set of Instructions is loaded in the IR.

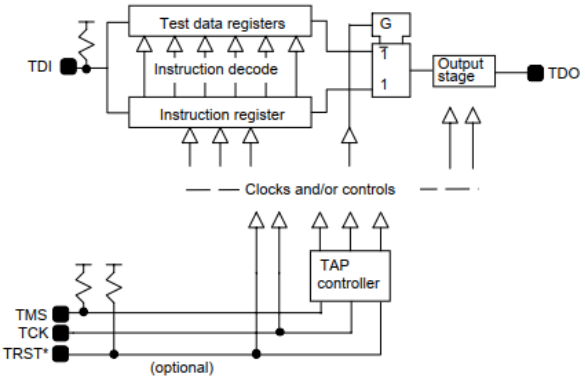


Figure 2 Schematic of an 1149.1 System, from [1]

As a result, the Boundary Scan Description Language (BSDL), simply has to enumerate the existing IRs and TDRs and their decoding, without needing to detail either their connections, or the Finite State Machine implemented inside the TAP or the effect of the Instructions, as in the following snippet:

```
    attribute INSTRUCTION_LENGTH of Top: entity is 4
    attribute INSTRUCTION_OPCODE of Top: entity is
"EXTEST(0000),"&
    "SAMPLE(0001),"&
    "INTEST(0010),"&
     "ijtag_en(0001),"&
    "IDCODE(00100),"&
    "HIGHZ(1110),"&
    "CLAMP(1111),"&
    "PROBE(0000),"&
    "BYPASS(1111)"
```

This allows DSL files to be simple to generate and parse, effectively streamlining the EDA flow.

Unfortunately, this comes with a major limitation: true to their name, DSLs can only describe what it in their Domain, i.e. what is in the original scope of the Standard. This puts a serious strain on evolution. To keep on the 1149.1 example, BSDL is only able to describe daisy-chain or star topologies, so when designers started to devise more creative connections they were forced to make custom modifications to both BSDL and EDA tools. A famous example is the BSCAN2 Scan Linker [15]: it is pretty much a "TAP of TAPs", that can connect up to 8 TAPs to a single JTAG port, and whose selection follows the same principles of a normal TAP. Regardless of its apparent simplicity, to the authors' best knowledge there is no Standard support yet for this component, and each EDA company needs to implement custom code to use it. This is because the "intent" of BSCAN2 is impossible to express not only in BSDL, but in all existing DSLs. Even 1687's Instrument Connection Language (ICL) cannot be used, because its application domain is behind a TAP, not before.

Up to now the solution has been to exploit new standards to add the desired features in new DSLs, but this cat-and-mouse game is reaching its limits because of fast evolution pace of DfT solutions. For instance, the upcoming P2654 and P1687.1 Standard Working groups [4] realized that the variability of Test Interfaces at the system level is so huge that it is impossible to propose a DSL able to support all possible solutions, and are moving toward solutions involving Callbacks, whose principles are described in the following sub-sections.

1.3 System Verilog PI: Callbacks for Simulation

In Computer Science, the problem of supporting evolutions in algorithms without a significant impact on source code base is a classical and well-known issue. It is in fact one of the main strengths of Object-Oriented programming: Templates and Software Design Patterns [16] can be used to make the source code modular and make it easy to write and add new modules. Anyway, the addition of a new element (class) requires the recompilation of the whole software.

The concept of Callback takes the process a step further: a placeholder is put into the executable code, and at run-time it can be resolved by "calling" a piece of external software, which will process the input data and give "back" the result, as depicted in Figure 3. It is what is called "load linking", to distinguish it from compilation-time linking. Callbacks are pretty straightforward when both the Main and the Library have been compiled from the same Programming Language, but can become pretty tricky when this is not the case. The key for the successful implementation of a Callback scheme is

therefore a clear specification of the input and output data formats.

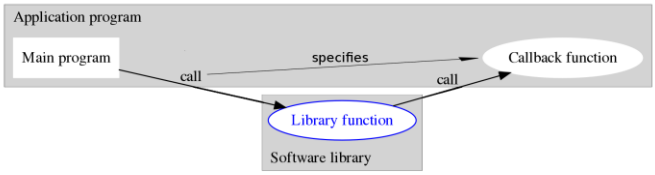


Figure 3 Callback Scheme, from Wikipedia

A famous example in the EDA world is the System Verilog Programming Interface [17], that allows RTL (Register Transfer Level) simulators to execute external code for testbench purposes. Its simplest expression, the Direct Programming Interface (DPI), is composed of two layers:

- A SystemVerilog (SV) layer, that defines the data types and functions calls from the Simulator point of view. Functions can either be "imported" (external functions executed in the simulator) or "exported" (SV functions which can be called from the external code). This takes the form of `import` and `export` pragmas to be used in the SV testbench file.
- A DPI Foreign Language Layer, that defines the Application Programming Interface (API) for a given language to specify argument passing and data type conversion. This takes the form of a normative `svdpi.h` header that must be provided by all simulators

In the standard, only a C layer is given, with the possibility for users to add their own. This is a choice both of simplicity to avoid over-cluttering the standard document and efficiency: most languages provide interfaces to C, which can be used to access the DPI layer. The final setup is depicted in Figure 4: the User source code (left-hand side of the picture) is compiled and linked against SV DPI libraries (not depicted) to obtain an Object Code (in the middle), that is then loaded at run-time by the Simulator into the final SV application (right-hand side of the picture).

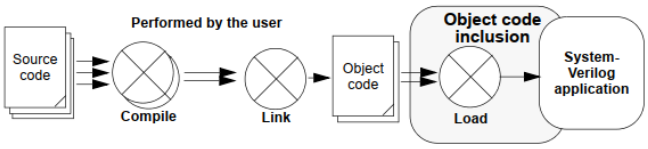


Figure 4 Inclusion of object code into a SystemVerilog application, from [17]

The two layers allow easy symbolic referencing, as depicted in Figure 5 (based on a code example from [17]): on the right-hand side, the SystemVerilog layer defines an import and an export point thanks to the related pragmas. On the left-hand side, the C DPI Layer does the same: the import of the svdpi.h layer ensures the usage of compatible types and references.

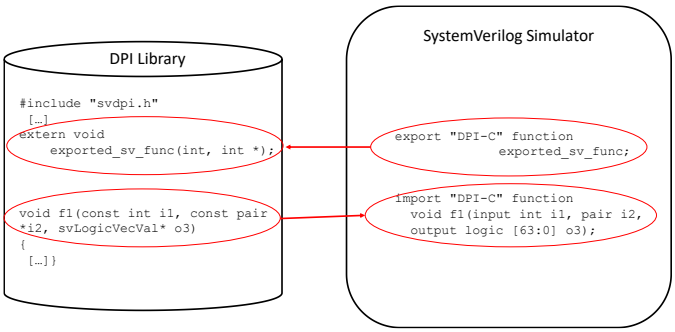


Figure 5 Symbolic referencing in SV DPI

While the DPI object Library final itself is not directly inter-operable, all EDA Tools provide examples and compilation Makefiles and the Standard mandates specific command-line options for the SV compilers, making porting between simulators trivial.

Of particular interest in this context is one evolution of the PI concept: the Verification Programming Interface (VPI) [17]. While applying the DPI principle to allow bidirectional communication between the Simulator and the External Code, VPI also proposes a series of data constructs and functions that allow the user to directly interact with the Simulation Model by accessing a common abstraction (the "Model Diagram") and extracting and modifying internal values from individual RTL elements.

1.4 P1687.1 and P2654: Callbacks for test

The P1687.1 and P2654 Working Groups have both been investigating the issue of Access to the Test infrastructure, even if from different points of view: as an extension of chip-level DfT for the former, and as a system-level Test Access Management for the latter. Both came to the same conclusions: current DSLs are insufficient for the task, and the high variability of solutions makes it close to impossible to define a "one-solution-fits-all" new language. Even though neither standard is yet complete, the general consensus is to move to a callback-based solution [4], where each Interface implements one or more "Transformations" on the stream of data and commands. The WGs are converging over the definition of the exchange format for these transformations as a derivation of the Relocatable Vector Format introduced in [5], but no decision has been made yet on the exact form the Callbacks method will look like.

2 EDA Extensions Through Callbacks

In this section, we show how by embracing the Callback model and applying it extensively to the Test Flow it is possible to obtain a complete and flexible support for any arbitrary DfT solution. The solution has been implemented and validated on the MAST tool [18], which is capable of supporting both 1687 and P1687.1.

The starting point is an extension of the abstraction of the System Under Test as a set of configurable resources presented in [18] and it is depicted in Figure 6.

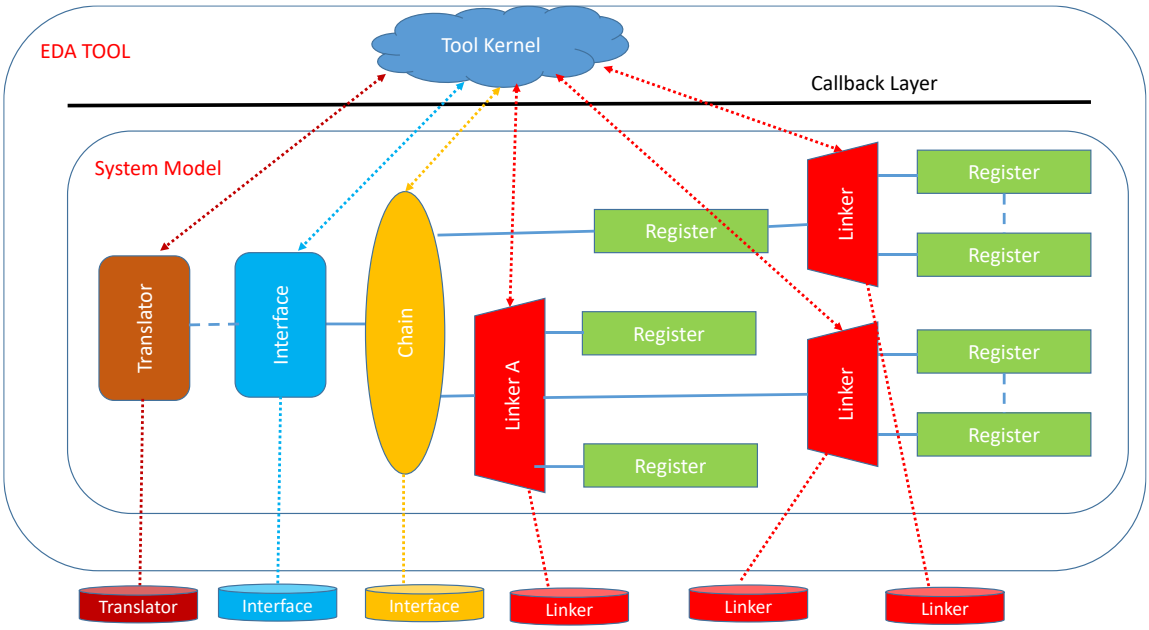


Figure 6 SUT Functional Abstraction

In the upper half of Figure 6, the Tool Kernel interacts with the System Model, obtained from the DSL files, to perform its operation. Traditionally, this interaction is completely Tool-*specific*, and it is the reason why any non-standard-compliant solution needs custom code modifications.

This modeling must be seen as the first step towards standardization: it is the basis of the DSL that will actually be used by P1687.1 tools.

In this paper, we propose to introduce a Callback Layer, on the model of SystemVerilog DPI, allowing users to provide custom code (in the bottom half of the Figure) to extend the support of the EDA tool to any custom elements. This is done in two steps: first, a unified Functional Abstraction Model is provided to represent the system, and then a standardized set of callbacks is associated with the relevant nodes. This can be seen as a novel application of Model Diagram VPI concept to the domain of testing.

2.1 Functional Abstraction Model

We define a "Functional Abstraction Model" as the Minimal Information Set needed by the Tool Kernel to access the SUT, configure its topology and generate Operations on the interface. For this, we need 5 types of nodes:

The first two nodes are passive, i.e. they are the target of read/write operations but do not modify the state of the system. For this reason, they do not have any callback associated with them.

- A REGISTER (in green): it is the base element of any topology. It can be, for instance, a 1687 Scan register connected to an Instrument. It is the final destination of all write operations and the source of all read operations.
- A CHAIN (in yellow): it represents a set of registers connected in sequence (daisy-chain) and that are therefore always accessed together.

The other nodes are active, i.e., they can directly modify the state of the SUT either by changing its topology or by issuing operations. They each have a standardized set of callbacks associated with them, which will be detailed in the next section.

- A LINKER (in red): it is a hierarchy-enabling element, which allows to select one or more of its child nodes by changing its configuration. It can be, for instance, a 1687 ScanMux.
- An INTERFACE (in blue): while the previous three nodes are used for retargeting, an Interface role is to translate the chain-level vectors into one or more operations. It could be, for instance, an 1149.1 TAP. In P1687.1 terms, it is often called a DPIC (Device Port Interface Controller) and its role is to generate a flow of RVF packets [5].
- A TRANSLATOR (in brown): its role is to translate operations from one Interface to another. An example could be an I2C-to-JTAG converter.

2.2 Standardized Callback Sets

As introduced earlier, each of the 3 active nodes have a set of callbacks associated with them for interaction with the Tool Kernel. The set depends on their functionality in the System Model

2.2.1 Linkers: the Path Selector

A topology-enabling element has the role to modify the active path of the circuit depending on its internal status. For instance, a 1687 ScanMux will be connected to another

element depending on the value of the register identified by the "SelectedBy" statement and the truth table associated with it. Our Linker abstraction depicted in Figure 7, is an extension of this behavior.

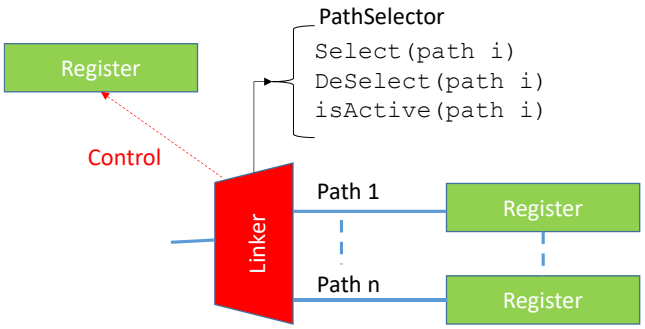


Figure 7 Linker Abstract Model

The Linker commands one ore mode Paths (in this example, connected to a Register each), numbered from 1 to n. The Control Register is saved as a reference, while instead of using a Truth Table like in ICL, we propose a set of 3 standardized callbacks, called a "Path Selector".

- *Select(path i)*: changes the value of the Control Register so that path "i" is selected. When i=0, the Linker is considered to be closed, like for a SIB;
- *DeSelect(path i)*: changes the value of the Control Register so that path "i" is no longer selected. If no path is selected anymore, it is equivalent to Select(0);
- *isActive (path i)*: returns 1 if path i is already selected by Control. It allows the Tool Kernel to query the state of the Linker without needing internal knowledge of it.

2.2.2 Interfaces: the Access Protocol

The role of an Interface, or DPIC in P1687.1 terms, is to provide an interface between two domains: on one side there are one or more retargeting domains, i.e. a series of chains for which it is possible to define input and output vectors, and a Transaction domain, where operations are performed on the Interface itself to deliver the afore-mentioned vectors. The most famous example is the 1149.1 TAP, depicted in Figure 8: a Retargeter can compute vectors to its right-hand side border, which can be applied on the left-hand side by issuing SIR or SDR Operation to deliver them either to the Instruction or Data branch respectively. Please note that the selection of the target Data Register is done inside the retargeting domain thanks to the Linker node defined previously: the TAP itself has no direct mean of choosing it.

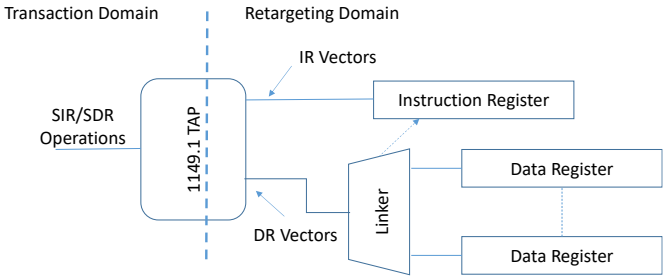


Figure 8 Functional Representation of an 1149.1 TAP

We therefore propose as Model an extension of this behavior, depicted in Figure 9: an Interface node can have one or more Endpoints, each one connected to a retargeting domain. Vectors can be delivered to endpoint 'i' by calling the related callback with its cardinal number as input parameter.

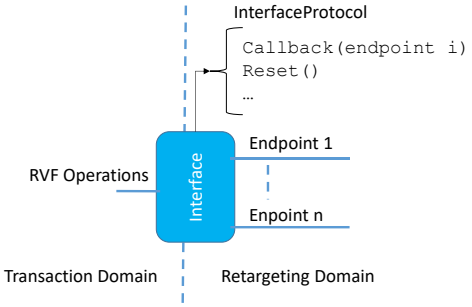


Figure 9 Interface Abstract Model

A Callback will have as effect to generate one or more Operations in the Transaction domain, each one represented by a RVF packet [5]. The synchronization of this RVF stream with the retargeting domain is left to the Tool. An example of this process can be found in [4].

2.2.3 Translators: the Translator Protocol

In the context of a complete system, being it a board with discrete components or an integrated System-on-Chip, it is often not possible to directly access the first Interface. This can be because of the need to reduce the pin count and therefore share the same resources [15] or because the system requires a completely different access infrastructure [4]. In these cases, the internal Interface is therefore connected with one or more adapters that effectively transform the Operations issued (i.e., the stream of RVF packets) into a different stream. For instance, it could be possible to operate a JTAG interface through an I2C adapter. In all cases, the procedure is always the same: the RVF operations need to be transformed, either by changing their internal data or by altering the stream itself. We therefore propose the model of Figure 10 : conceptually, it is almost the same as an Interface, with the only important difference that the Translator already works on RVF operations, and is not on the boundaries of a Retargeting domain.

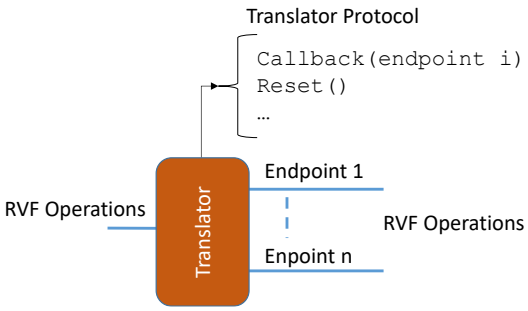


Figure 10 Translator Abstract Model

2.3 Optional Callbacks and Nodes

Up to now, we only considered vector-related Operations. In reality, both Interfaces and Translators will have also other types of operations, like Reset or configuration (e.g., setting clock frequency). These operations will of course need to be part of the two protocols, but at this moment in time no clear consensus has been reached in the Working Groups about the nature and number of them. Anyway, another advantage of the Callback method is that it is extremely easy to add new methods that will simply be ignored by older Tools, therefore guaranteeing backward compatibility. Similarly, it is possible to define generic callback-bearing nodes for special behaviors, which will simply be ignored by Tools not supporting them.

These solutions would be of course non-standard compliant, but they would still be supported by a P1687.1 Tool with minimal modification. This extensibility capability can be extremely useful to adapt the Flow to sudden changes in the application environment (e.g., the discovery of a new security threat) without waiting for a full Standard Revision, which could take years. We will present an example of this usage in Section 3.

2.4 Simplified ICL Tree: Callbacks-based DSL

As previously stated, the abstraction presented in this Section will ultimately be used as the basis of the P1687.1 DSL, which could be a revision of ICL or a completely new language. For experimentation, we developed a "Simplified ICL Tree" (SIT) language that is in fact a direct representation of this model, and that has already been used for instance in [4]. Without going into the details of a language that is in fact just a sandbox, we will present and comment some examples of its usage in the rest of the paper.

3 Security Extensions

The domain of Security is an ideal candidate for our Abstraction: the solutions are often based on complex algorithms and precise sequences, which are difficult to express in traditional DSL because their "intent" is purposefully obfuscated and hidden. For this reason, while hardware solutions are relatively widespread, their support by the standard EDA flow is almost non-existent, and they all rely on either heavy pre- and post-processing steps or custom modules inside EDA tools. We will show here how thanks to our abstraction it is possible to have a P1687.1-capable tool handle a completely custom solution, by using our MAST tool [18] as an example platform. All the following examples were validated against RTL simulations.

3.1 Dynamic Authentication through Callbacks

The problem of Authentication-based access like [7], is that the handling of keys is not part of the IEEE 1687 standard, and must therefore be added by the user through custom pre and/or post-processing of the vectors. The solution proposed in [8], and depicted in Figure 11 is radically different: the authentication is part of the configuration algorithm itself thanks to an "SSAK Protocol" that is added to the Tool Kernel.

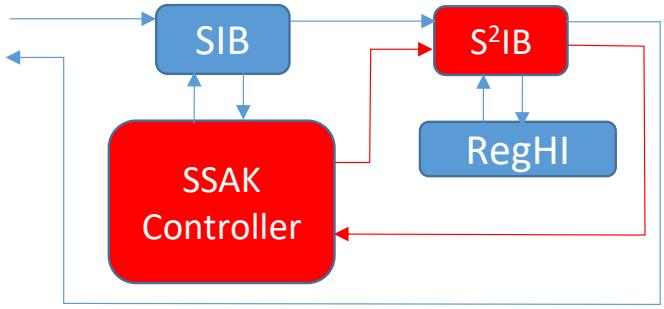


Figure 11: Fully Automated Authentication [8]

The SSAK protocol is actually divided in two parts: the SSAK Controller that is responsible for the challenge/response itself, and one or more S²IB muxes associated to critical elements (RegHI) , identified by a cardinal number, that can be opened only when the authentication has been successful [7]. This type of complex inter-dependency is extremely difficult, if not close to impossible, to express in traditional DSL.

In our modeling, we followed exactly the SSAK protocol: we defined two PathSelectors, one responsible for the Controller, and the other for the S²IB. For the Controller, Select/Deselect will trigger an Authentication sequence. The S²IB PathSelector will query the status of the Controller and either trigger a Challenge/Response or directly open the Mux. The SIT representation of the system of Figure 11is a direct mirror of this scheme, as shown below.

```
1.  SIB SSAK_SIB POST HIGH
2.  (
3.  LINKER SSAK_Controller SSAK SSAK_CONTROL_REG
        1 "0x72c4358f5a8a07af3d0f7d560a872a2b 13"
4.      (
5.        REGISTER SSAK_CONTROL_REG 128 )
6.       )
7.  REGISTER S2IB_1_ctrl 1
8.  LINKER S2IB_1
        S2IB  SSAK_Controller,S2IB_1_ctrl  1 "1"
9.     (
10.     REGISTER regHI 12
11.    )
12. )
```

Line 3 defines the "SSAK_Controller" Path Selector, notably providing the SSAK key and the maximal number of supported S²IBs. Line 8 instantiates a S²IB, providing both the

link to the SSAK_Controller to which it depends on and its cardinal position in the secure chain.

The MAST tool exploits the two identifiers in line 3 and 8 to look for the right callbacks following the same principles of SystemVerilog DPI introduced in Section 1.3.

3.2 Scan Encryption Through Callbacks

The other typical solution to provide security is to encrypt the stream of data being exchanged over the TAP. Even though conceptually simple, this solution is quite complex to implement in terms of software: the coding/encoding operation depends on the length of the scan chains and therefore requires some important pre and post processing steps to correctly adapt the bitstream [19]. In P1687.1 terms, on the other hand, the solution is quite simple: a stream cypher is in fact a Translator node whose callback modifies the data content of each RVF packet. The setup is depicted in Figure 12, using the Trivium stream cypher.

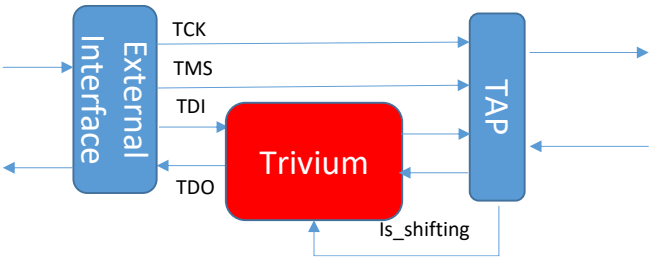


Figure 12 Trivium Stream Cypher

Its SIT description is straightforward, as in the following snippet:

```
1.  TRANSLATOR top Simulation
2.   (
3.   TRANSLATOR Secure Trivium
    "0F62B5085BAE0154A7FA 288FF65DC42B92F960C7"
4.   (
5.   JTAG_TAP […]
6.   […]
7.  )
```

The Protocol is identified by the symbol “Trivium” and it is initialized by providing the Secret Key and the Initialization Vector. The Tool will simply have to call the Transformation Callback to provide security: the integration with the standard flow is complete.

3.3 Encryption SIB: custom callbacks

The Encryption SIB (eSIB), introduced in [13], is probably the first attempt to combine Authentication and IP-level Encryption. It only requires two more gates with respect to a S²IB, as illustrated in Figure 13.

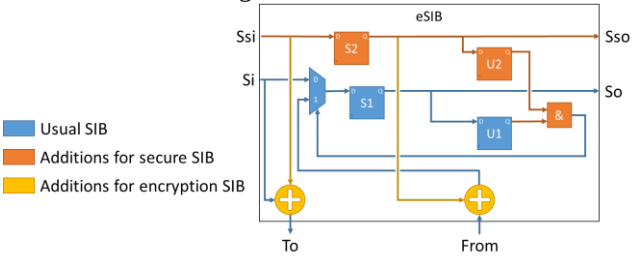


Figure 13 Scheme of an Encryption SIB

The XOR gates are controlled by the encryption stream provided by the Trivium coprocessor (or another encryption processor) used for the global securisation scheme. The overhead is therefore negligible, but in case of malicious IP inserted in the circuit, the flow of data is protected as illustrated in Figure 14.

A Challenge/Response protocol, as introduced in Section 3.1, is used to initialize the Encryption module positioned behind the S²IB. As previously explained, this local encryption requires a dynamic adaptation of the authorization streams, for each IP, depending on the scan chain configuration and the user rights. Since the chain configuration can be changed at any time for e.g., better coverage of a given IP, it is not possible to define a standard access configuration for each IP, while scan chain lengths are modified.

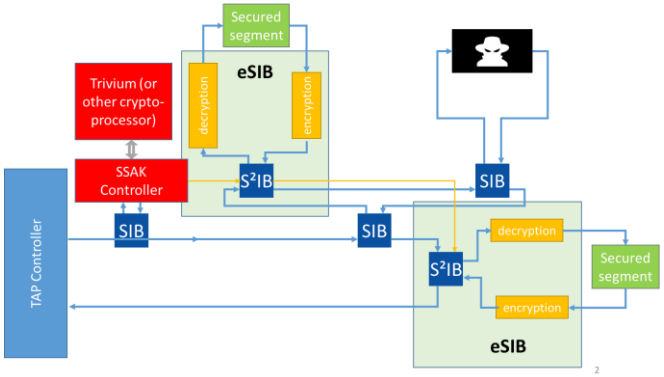


Figure 14 Impact of eSIBs in a global SoC

From the software point of view, there are two challenges:

- The handling of the Trivium Streamer, which needs to receive the SSAK Challenge as the Initialization Value, and which needs to be used by all the eSIBs as the source of the encryption/Decryption KeyStream;
- The phase alignment of the streamer flow, in the middle of the chain.

The second point is of particular importance: traditionally, streamers are put at the root of the Scan Chain as in Figure 12. In this position, they are perfectly symmetrical: all bits going into the SUT will be decrypted once with the corresponding keystream bit passing through the TDI port of the streamer, and all bits coming from the SUT will be encrypted once before exiting the TDO port. This is not true anymore when the streamer acts in the middle of the chain: the symmetry is broken, and each bit will be treated differently depending on its position relative to the streaming module. Figure 15 provides a graphical representation of the analytical equations given in [13]. In relation to Figure 15-a), the “Protected” bits are treated as usual: data coming “fromSUT” and going “toSUT” is encrypted/decrypted when passing through the XOR gates at the input/output if the Streamer ‘(in this case, the eSIB). The only notable difference is the synchronization with respect to the Keystream, which is generated for the whole scan chain and not only for the Protected Segment, which will need only a subset of it. In the “fromSUT”

direction, data is scanned out just after the Capture stage, so the Mask starts at the first bit of Keystream. On the other hand, in the "toSUT" direction, the bits positioned After the Protected section will be scanned first, and the Mask will start after them. This is depicted in Figure 15-b) and Figure 15-c) respectively, with the Keystream depicted as the grey "XOR-MASK", and the Mask as the purple XOR-MASK-PRO.

Figure 15-b) and Figure 15-c) also allow for an easy understanding of the impact of the asymmetric positon of the Streamer. In the "fromSUT" direction, data positioned "After" the streamer is of course not impacted by it, but the data positioned "Before" the streamer will actually have to pass through both XORs of the streamer, being effectively encrypted but at different stages of the Keystream. This is expressed in Figure 15-b) by the MASK_IN_BEFORE and MASK_OUT_BEFORE which depict the segments of the Keystream that need to be used to decrypt the data once received. The same principle is applied in the "toSUT" direction, as depicted in Figure 15-c).

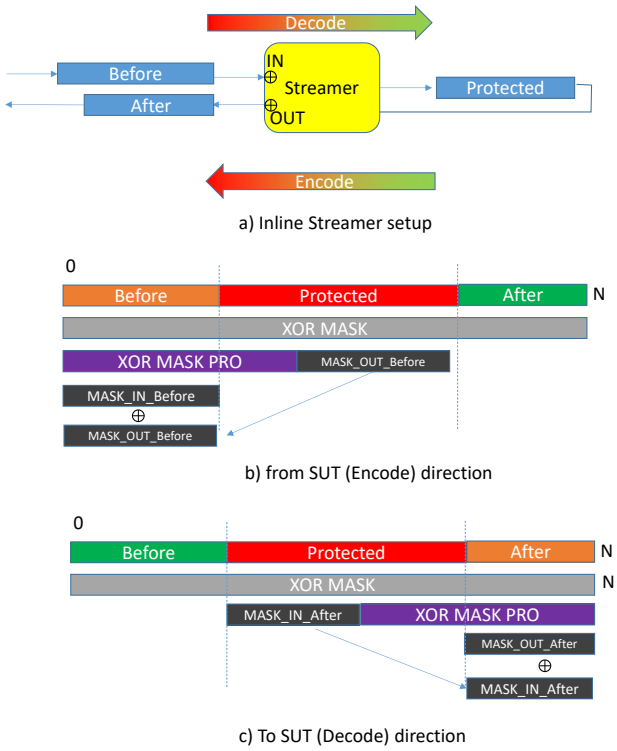


Figure 15 Asymmetric Masking for an Encryption SIB

This masking can therefore be resolved by knowing the length of the scan the length of the three Before, Protected and After sections. This behavior is outside of the scopes of both 1687 and P1687.1: even though the topology itself is quite simple, it is not directly describable in ICL. On the other hand, the Callback approach of Section 2 can be extended by defining two new modules:

- An Optional Callback "get_challenge()" inside the SSAK Controller, that returns the Challenge Value
- A "Streamer" node, similar to a P1687.1 Translator node put in the middle of the scan chain, but with different Callbacks

The Optional Callback does not need any particular modification: as explained previously and in reference to the solution of Section 3.1, it can be simply added to the PathSelector Callback wrapper for the SSAK Linker: it will be ignored by Tools not supporting this feature.

As for the Streamer, it just needs four Callbacks:

- *CurrentMask()*, *NewMask(MaskBits)* and *ApplyMask(PlainText, Mask)*, which can be used by the Tool to implement the masking following the equation of [13] and Figure 12
- *ResetProtocol(InitializationVector)*, that the Tool can use to synchronize its own Cypher with the one inside the SUT.

The last step is the position of the Encrypted SIB inside the scan chain. This is easily achieved by instantiating the Streamer node in SIT:

```
1.  REGISTER Before
2.  SIB SSAK_SIB POST HIGH
3.  (
4.  LINKER SSAK_Controller SSAK SSAK_CONTROL_REG
        1 "0x72c4358f5a8a07af3d0f7d560a872a2b 13"
5.      (
6.        REGISTER SSAK_CONTROL_REG 128 )
7.       )
8.  REGISTER S2IB_1_ctrl 1
9.  LINKER S2IB_1
        S2IB  SSAK_Controller,S2IB_1_ctrl  1 "1"
10.   (
11.    STREAMER Online Trivium  SSAK_Controller
           "0F62B5085BAE0154A7FA"
12.      REGISTER Protected
13.   )
14.  )
15. REGISTER After
```

The "Streamer", as well as the "Encrypted SIB" are not standard features, so a purely P1687.1 Tool won't of course be able to support it, but thanks to the Callback abstraction it can be added with limited effort while preserving complete compatibility with standard features, something that is not possible with legacy approaches as in [19].

4 Conclusions

In this paper, we proposed a complete Abstraction Model for DfT based on the Callback paradigm, leveraging the successful experience of EDA solutions such as System Verilog. We then demonstrated its flexibility by applying it to the problem of Scan Security, providing a fully standardized and coherent flow where legacy solutions relied mostly on ad-hoc workarounds.

Acknowledgments

This work has been partly funded by the French Government under the framework of the PENTA HADES ("Hierarchy-Aware and secure embedded test infrastructure for Dependability and performance Enhancement of integrated Systems") European project.

REFERENCES

[1] IEEE Std 1149.1-2001, “IEEE Standard Test Access Port and Boundary-Scan Architecture”, IEEE, USA, 2001.

[2] IEEE std 1500, “Standard for Embedded Core Test”, ”, IEEE, USA, 2005

[3] IEEE Std 1687-2014, “IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device”, IEEE, USA, 2014

[4] M. Laisne , A. Crouch, M. Portolan,; M. Keim, H.M. Von Staudt , M. Abdalwahab, B. Van Treuren, J. Rearick,, “Modeling Novel Non-JTAG IEEE 1687-Like Architectures”, 2020 International Test Conference (ITC20), November 2020, Washington DC, US

[5] M. Portolan, “The Automated Test Flow, the Present and the Future”, IEEE Transactions on Computer-Aided Design (TCAD), DOI: 10.1109/TCAD.2019.2961328, December 2019

[6] Jennifer Dworak, Al Crouch, John Potter, Adam Zygmontowicz, Micah Thornton, «Don't Forget to Lock your SIB: Hiding Instruments using P1687», IEEE International Test Conference, 2013.

[7] B. Rafal, K. Michael A and H.-J. Wunderlich, “Fine-Grained Access Management in Reconfigurable Scan Networks,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 34, pp. 934-947, 2015.

[8] M. Merandat,, V. Reynaud, E. Valea, J. Quevremont, N. Valette, P. Maistri, R. Leveugle, M.-L. Flottes, S. Dupuis, B. Rouzeyre, G. Di Natale, “A Comprehensive Approach to a Trusted Test Infrastructure,” in Internation Verification and Security Whrokshop, Rhodes 2019.

[9] M. Da Silva, M. Flottes, G. Di Natale and B. Rouzeyre, "Preventing Scan Attacks on Secure Circuits Through Scan Chain Encryption," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 3, pp. 538-550, March 2019, doi: 10.1109/TCAD.2018.2818722.

[10] S. Kan, J. Dworak and J. G. Dunham, "Echeloned IJTAG data protection," 2016 IEEE Asian Hardware-Oriented Security and Trust (AsianHOST), Yilan, 2016, pp. 1-6.

[11] K. Rosenfeld and R. Karri, "Attacks and Defenses for JTAG," in IEEE Design & Test of Computers, vol. 27, no. 1, pp. 36-47, Jan.-Feb. 2010.

[12] E. Valea, M. da Silva, M.-L. Flottes, G. Di Natale, B. Rouzeyre. Encryption-Based Secure JTAG. DDECS: Design and Diagnostics of Electronic Circuits Systems, Apr 2019, Cluj-Napoca, Romania. DOI: f10.1109/DDECS.2019.8724654.

[13] P. Maistri, V. Reynaud, M. Portolan, R. Leveugle. "Secure Test with RSNs: Seamless Authenticated Extended Confidentiality," Proceedings of the 19TH IEEE Interregional NEWCAS Conference, to appear.

[14] Portolan M., Reynaud V., Maistri P., Leveugle R., “Dynamic Authentication-Based Secure Access to Test Infrastructure”, 2020 European Test Symposium (ETS 2020), Tallin, ESTONIA, 2020

[15] Latttice Semiconductors, “Using Multiple Boundary ScanPort Linker (BSCAN2)”, Application note AN8081

[16] Gamma E., Helm R., Johnson R., Vlissides J., “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley, 1995 ISBN 978-0-201-63361-0.

[17] IEEE std 1800-2012, “SystemVerilog -Unified Hardware Design, Specification, and Verification Language”, IEEE, USA, 2012.

[18] M. Portolan, "A Novel Test Generation and Application Flow for Functional Access to IEEE 1687 instruments", Proc European Test Symp. (ETS), pp. 1-6, 2016.

[19] Thiemann et al , “On Integrating Lightweight Encryption in Reconfigurable Scan Networks,” , Proc European Test Symp. (ETS 2019)