Automatic verification for Assertion Based Verification: How can a SPIRIT IP-XACT extension help?

Sofiene Mejri
STMicroelectronics
Tunis, Tunisia
Telephone +216 7010 5295
Sofiene.Mejri@st.com

Mirella Negro Marcigaglia
STMicroelectronics
Catania, Italy
Telephone +39 095 740 4461
Mirella.negro@st.com

ABSTRACT

In this paper, we describe the application of a new automatic verification flow for Assertion Based Verification. A prototype of this approach is described, starting from a standard specification format, which is adopted by all our Digital IPs, through the generation of a SPIRIT IP-XACT standard with added verification extension, ending with the automatic generation of the set of checkers and coverage items that are finally used for formal or dynamic verification of the RTL implementation. In this way, we implement a "correct by specification" verification environment versus one in which we verify the RTL functional behavior. Our goal is to reduce the time required to generate the verification environment and to make sure to generate the complete set of basic checkers and coverage items required to verify the Digital IP design.

1 INTRODUCTION

1.1 General Concepts

The business requirements of continuously reducing project time while always delivering successful first time silicon products make it more and more attractive to look at promising automatic verification flows, which should give the advantage of reducing scheduling time and guarantee more rigorous and complete verification. However, the major challenge is to develop a flexible and powerful flow that can be used with complex digital IPs, be reusable, and start from an existing textual standard specification format, in order to avoid investing further effort in the specification phase (i.e. to write a SPIRIT IP-XACT view). Formal verification has become increasingly important in the verification flow for Digital IP,, and in some cases it is enough to verify all of the IP's functionality without the addition of any simulation techniques. There are some functionalities that are easier than others to be verified using a formal flow, and it is exactly on such functionalities that we started to work for automatic checkers generation. The generated checkers can be as well used in dynamic verification. A digital IP is generally a component of the SoC, which can be programmed by software by writing a set of internal registers. Via software programming, it is also possible to monitor the IP's status either by polling the status of its internal registers, or by activating an IR (Interrupt Routine) when an interrupt signal is raised by the digital IP. Every digital IP has a common set of configuration and status registers, and the IP functionality depends on the value of its registers. Each register can be reset or accessed in read and/or write mode, and its fields can be set or reset by hardware or by software.

The solution presented in this paper exploits the SPIRIT standard XML format for the definition of the IP's registers, IP-XACT, and provides a new methodology by adding an extension, which contains further information, required for the generation of checkers and coverage items to be applied in both formal and dynamic verification. We start by describing the state-of-the-art of automatic functional verification solutions and summarizing their strengths and weaknesses. We then proceed to detail the implementation of a prototype that has been developed for an automatic verification flow, starting from the required specification format, to the new extension of the SPIRIT IP-XACT Standard and the different generated output. We will also highlight the flow requirements in terms of extensibility and flexibility. Then, we will describe its different use models and its impact on other design phases. A functional verification use case will be described to illustrate this prototype with a simple 8-bit bus interface digital IP. The test case is verified in formal, and we will compare the results and evaluate the added value and impact of this automatic verification approach within the functional verification activity as well as the global design flow. Finally, we'll describe possible future enhancements of this approach and its current limitations.

1.2 Specification Languages

In a typical design environment, a Digital IP behavior is described in a functional specification text document created by the hardware designer. This document contains information about the IP behavior, the registers set, and its managing policy. We will refer in our paper to this document as the Functional Specification Document. There are other important details, like the external interface list of signals, the micro architecture design, timing information, for examples, that, depending on the IP implementation, are normally provided in another IP document that is used to perform functional verification and SoC integration, but is not required for the end user application and customer. We will refer in our paper to this document as the Design Specification Document. Both documents rarely follow semantic rules that could allow automatic processing to extract the contained information. The designers, as well as the verifiers, use the Digital IP specification documents to create their own Digital IP environments in different languages: VHDL, Verilog for the RTL design, e language, System Verilog, PSL assertions, and so on, for the verification environment.

2 STATE OF THE ART

2.1 Specification Language

Some attempts have been made to implement a formal specification language, suitable for automatic processing, in order to automatically generate both implementation and verification environments. However, most of the defined specification languages are referred to software applications and programming, like TUG [7] which is oriented to C code generation, and [16], and not to digital verification.

2.2 SPIRIT Standards

The SPIRIT Consortium [29] defines a set of standards to exchange design information among different users, in order to facilitate automatic processing in the different design phases. The standards defined by the SPIRIT Consortium are IP-XACT, which is an XML schema for the description of design components, and SystemRDL [29], a language for describing registers inside the components. A part of the SPIRIT IP-XACT standard related to registers and interfaces are described in the table below.

Table 1: Subset of SPIRIT IP-XACT Standard

Name	Description
Name	The register's name.
Display name	The register's display name.
Description	The register's description.
Adress offset	The register's address offset.
Size	The register's size.
Access	The register's access mode.
Volatile	Indicates wheather the register's value is volatile or not.
Reset	The registre's hardware reset value.
Value	The registre's hardware reset value.
Mask	The mask of registre's hardware reset value.
Name	The field's name.
Display name	The field's display name.
Description	The field's description.
Bit offset	The field's bit offset.
Bit width	The field's bit width.
Volatile	Indicates wheather the field's value is volatile or not.
Access	The field's access mode.

2.3 Automatic Generation

In the context of hardware design verification, at least two kinds of automatic generation exist. The first one is based on specification; the second is based on design implementation. The former techniques, the most common in the EDA industry, automatically perform the functional verification of a design implementation either to implement verification algorithms [10] or to start from an existing implementation of the design itself [12].

For example, SystemRDL, the SPIRIT standard developed by Denali Software [28], and used by them to implement a SystemRDL compiler, Blueprint[5], can automatically generate and synchronize register views for specification, implementation, verification, and documentation, starting from a SystemRDL specification view. However, the SystemRDL approach implies that the hardware designer or the system architect writes the IP specification directly in SystemRDL language and not take the design specification as input format.

In the context of formal verification, some verification tools also provide built-in checkers that may include dead code, FSM transaction/ reachablility / deadlock, for example, but not inputs or outputs toggling (IFV, Cadence), which in the context of formal verification has an added value to spot problems (outputs toggling) and to assure that the verification environment is not over-constrained (inputs/output toggling).

Some other automation tools like Enterprise Planner (Cadence), provide traceability from functional and design specs to the verification plan and maintain consistency between specs and the verification plan. However, these kinds of tools do not allow the automatic generation based on the given specification.

In addition to that, even if the available observed solutions may offer general facilities like test bench generation, RTL registers and field definition, alias definition for each register address, for example, they do not offer specific facilities for the ABV, like the generation of configuration constraints ((IP enabled => all configuration register must be stable), and ((IP enabled => reg1.filed1 =x"F").

Based on this, a new generation solution must be made, and we choose to generate a SPIRIT IP-XACT view from the specification documents as a first step to do it.

2.4 Specification Design Document

The inputs for our flow are the specification documents, which are written in a text format. We compared some word-processing software programs and formats. The Word Binary File Format (.doc) is the native Microsoft Office Word format, and its content can be read and edited by some free software programs like OpenOffice WriterOpenOffice Writer and AbiWord [30]. The Rich Text Format (RTF) is used as a standard for data transfer between word processing software, and it allows document formatting, migration forward and backward in versions, and is widely supported by most word processing editors [25]. The Adobe Framemaker format (.fm) can handle complex documents and format XML very easily [1]. The Maker Interchange Format (MIF) allows information exchange by creating a text file that is easy to parse and preserves text, graphics, and formatting information, through the usage of filters [2]. The Adobe Portable Document Format (.pdf) enables users to exchange and view electronic documents independently from the source editor used [3]. Finally, the HTML format allows wide publication capability, supporting hypertext links and embedded applications [18]

The following table illustrates some differences between these different text formats.

- Tagging: whether the parts of the document could be distinctly tagged.

- Readable content: whether the content of the file could be read and edited with basic text editors other than the default editor.
- Suitable for exchange: whether the format allows file exchange without the need for licenses for the application to be able to read the document.
- Free editors: whether there are free editors that allow creating, reading, and editing the document.
- Styles: whether the format supports styles for formatting the text.

Table 2: Different text format comparison

	Readable content	Suitable for exchange	Free editors	Tagging	Styles
DOC	Yes	Yes	Yes	No	Yes
RTF	Yes	Yes	Yes	No	Yes
FM	No	No	No	Yes	Yes
MIF	Yes	No	No	Yes	Yes
PDF	No	Yes	No	No	Yes
HTML	Yes	Yes	Yes	No	Yes

The criteria of choice of inputs are availability of specification format, extraction and processing capability, and facility of use. For extraction capability, at least two alternatives are available: based on styles, by associating a particular font style to a group of data, then using this embedded information to extract the required data. This approach is used in some commercial tools for verification plan item recognition, when the verifier has to use a particular templates with well-defined styles and fill in each verification plan item accordingly. Then the tool will recognize all sections and items based on that.

The extraction based on tags is done by associating to the needed document item an appropriate kind, also called tag, which will be hidden and embedded in the specification document source and never appear in the visible part of the document.

Even if the two approaches allow data extraction, styles present some weakness compared to tagging, such as the need to change the original document styles or impose these styles to the designers. Styles are also not suitable for all types of data presentation. For verification plan extraction, which is not our scope, the style-based approach works well, but for table specification extraction, where all table columns' titles must follow the same style, tagging provides the ability to associate a particular, different tag to each column title, and as a consequence, allows the extraction based on a particular column.

For other criteria of choice, the different specification format has a similar level of facility of use. In our case, we adopted Adobe Framemaker format (FM, MIF) since it is the designer-adopted format. This meets our expectations such as availability, extraction capability, ease of use, and finally, even if the delivered format .fm is not directly suitable for processing; a conversion to .mif format, suitable for processing is immediately available by using 'save as mif.' The former file will be the start point for the automation flow.

2.5 Data Storage

We analyzed our requirements to store data within our flow: we needed a simple but efficient data storage solution, and therefore we compared FMS (File Management System) with DBMS (Data Base Management System) [9, 6]. The result is summarized in the following table.

Table 3: Data storage comparison

	Advantages	Disadvantages
FMS	▪ Simpler to use. ▪ Less expensive. ▪ Fits the needs of small businesses ▪ Popular FMS's are packaged along with the operating systems of personal computers.	▪ Does not support multi-user access. ▪ Limited to smaller databases. ▪ Limited functionalities. ▪ Redundancy and Integrity issues.
DBMS	▪ Greater flexibility. ▪ Greater processing power. ▪ Better data integrity. ▪ Supports simultaneous access. ▪ Provides backups and recovery controls. ▪ Advanced security.	▪ More difficult to learn. ▪ In general more expensive. ▪ In general packaged separately from the OS. ▪ Slower processing speeds. ▪ Require skilled administrators.

For the complexity of our flow we decided that an FMS was the best solution, being simpler yet covering our requirements.

3 USUAL FLOW

3.1 Description

The common verification flow both for formal and dynamic verification starts with the specification documents (functional behavior, IP micro-architecture, list of input/output signals) and consists of development of a set of checkers that verify the correctness of IP behavior during the formal proof or the simulation process. The checkers are hand written, in a verification language that depends on the adopted verification flow (PSL or SV assertions, e or SV checkers, for examples). In coverage driven verification, coverage is used to measure the quality and completeness of such an approach.

3.2 Limitations

This typical approach requires a long development time; the checkers development and debugging could take an average of four to 16 man/weeks, for a full verification of medium complexity Digital IP, and two man/weeks for the basic checks only. In addition, there is no guarantee of completeness with respect to the specification.

4 AUTOMATION FLOW

4.1 Principle

The automation flow is based on the automatic extraction of the design description starting from a given specification document. This automation flow, is composed by three separate steps: the preparation, in which the design specification is prepared for the automation flow, followed by the generation step, in which all output and data are generated, and finally by the use model and application step in which the generated items are used in a functional verification context. Pre-processing checks and post-processing checks are made to insure the correctness of each of the steps.

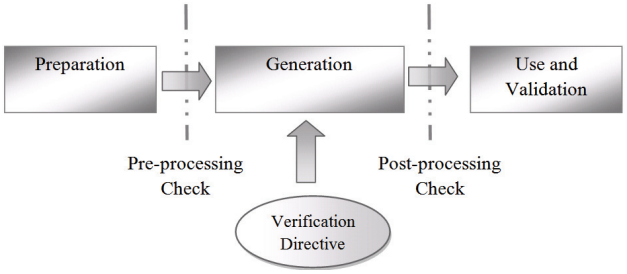

Figure 1: Verification Directive

In each step of the automation flow, we considered key aspects such as the extensibility, maintainability, and flexibility.

4.2 First Step: Preparation

The proposed flow is able to extract only the information that is available in a table format, using tagging techniques. A unique category name will be dedicated to each type of table, and when using the flow, each table in the specification document will be associated to one of these categories. This operation is called tagging.

Dealing with standard specification conforms to supported templates. When the given specification document is compatible with one of the supported templates by the automation tool, only table tagging will be required. For example, the table below describes the list of tags used for one supported template.

Table 4: List of Tags

Tag name	To use with
t1_f2sRegMapTable	Registers' map table
t1_f2sFieldsMapTable_*RegisterName*	Fields' map table
t1_f2sFieldsDescTable_*RegisterName*	Fields' description table

Each table inside the specification must be tagged using one of the previously described tags to allow the automatic extraction of the data.

4.2.1 Dealing with a new template document

To allow maximum flexibility, the automation flow supports two kinds of use model in terms of input. The first one is used when the specification document is following a particular template. The specification template is defined by the IP/SoC design team, and all IP specification will use the same template. This template will also be used as a reference during the implementation of the automation flow, and any further modification of this template will require a modification of some module of the automation tool.

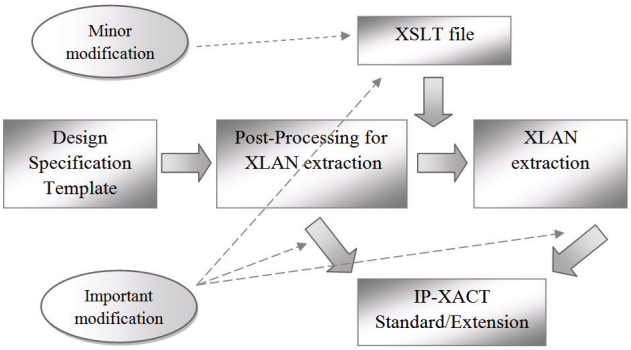

Figure 2: Multiple specification support

It is always possible to extend the flow; according to the impact of the changes made in the default template. In case of a minor change like adding a new table for input/output description or defining a new register size (i.e.: 16 bits, those supported by default are 32 and 8 bits), we need only to update the XSLT file to match the new template. We could also have major changes in the specification template - for example, a table that contains complex and composed data in the same table field (i.e.: in the same table field ,the name[size] and the hardware reset value co-exist). In such a case, in addition to the required change in the XSLT file, additional processing must be done before the generation of the SPIRIT Std and SPIRIT IP-XACT extension view.

Very often, design description can be provided by a different organization that did not follow the required format for the automation tool; the second part of this section (Dealing with non-conforming specification) provides the solution for this situation.

4.2.2 Dealing with non-conforming specification

A non-conforming specification is a specification document that is not following any template supported by the automation tool. The non-conformity could be of two types:

- The required information exists, but it is not in the expected format.
- The required information is missed and may exist in another document.

To insure the flexibility of the automation flow, another use model is supported, to be used when the given IP specification did not follow any template. The verifier can use an intermediate document called VerifSpec, delivered with the automation tool, in which the verifier himself will fill, in the different ready to use tables, all the required information by hand. This document will be the starting point for the automation process instead of the original specification document.

4.3 Pre-processing Checks

To add more robustness in the automation flow, in addition to the recommended visual pre-processing checks that insure the compliancy of the given specification document to the golden specification template, another automatic check is done after the table tagging.

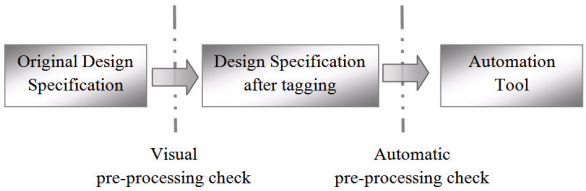

Figure 3: Pre-processing checks

The automatic pre-preprocessing focuses on the table tags as well as the table content. The checks related to the table tags are the following:

- *Search for Register Map Table*: this step will just look for a table tagged "RegMapTable." If not found, the validation process will be stopped because it's mandatory to have this table - otherwise neither the validation nor the generation process can take place.
- *Checking the Register Map Table:* once we verified the existence of the table in the document, we perform some checks on the content of this table. We begin by checking that the address offset of each register is specified and given in hexadecimal format. Then we move to the registers' name column, where we should find the names and the expression "Reset value." Finally, we check the rest of the columns of the table that contain the fields' names of each register and the corresponding reset value.
- *Registers counting*: this step will use a temporary generated MIX file to determine the number of registers used in the design and described in the specification.
- *Fields Map Tables counting*: the aim of this check is to be sure that each register has its associated table describing its fields.
- *Fields Description Tables counting*: the aim of this check is to be sure that each register has its own table containing the description of its fields.
- *Retrieve Registers names*: this step will output a list of the all the registers' names, which will be parsed to make sure that they don't contain any unsupported notation; the list will also be re-used for the next validation steps.
- *Look for the fields map and description tables of each register*: to make sure that all the tables are described in the specification and are correctly tagged.
- *Generate the validation report*: in this step, we are going to collect information from the previous checks to generate the final report on the specification document and to report whether the analyzed input format could be used to automatically generate the desired XML SPIRIT file.

The content validation focuses on the table content as explained below:

- Check the fields' names conformity between the Register Map Table and the corresponding Fields map Table. It's mandatory to have the same field names in the two tables.
- Check the completeness of the mandatory information for the SPIRIT IP-XACT.

4.3.1 When the IP-XACT view is provided

When the IPs' SPIRIT view is already available and provided, the standard SPIRIT view will be used as input for the tool. In the functional specification document, only the tables describing the DUV general information, history etc…. will be tagged. Implementation details and additional required information like RTL signals path are retrieved in the design specification document as well as in verification directive.

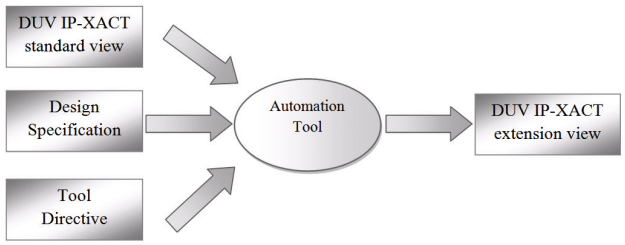

Figure 4: Standard IP-XACT support

The verification tool requires additional information that might not be present in the design specification document, such as the main clock/reset signal, test/scan mode signal, or bus interface type (AHB, APB, ST7...). These verification directives and all the missing ones maybe provided via command line or GUI.
At this point, we collected all the required information for the generation phase.

The collected data will be stored in two particular XML formats: the SPIRIT IP-XACT *(detailed in section 2.2)* and the SPIRIT IP-XACT extension *(detailed in section 4.4.1),* which embeds the verification directive and DUV reference as well. Starting from this point, any data generation will be based only on the mentioned SPIRIT IP-XACT extension view, expected to cover all needed data and directive in the functional verification process.

4.4 Second Step: Generation

4.4.1 SPIRIT IP-XACT extension generation

What we define as the SPIRIT IP-XACT extension is a simple extension of the SPIRIT Standard, which includes the additional requirements for functional verification. For example, during the verification process, when we use a gray box approach, the SPIRIT IP-XACT does not provide the information about the RTL path for a given register, nor about the software reset value of it. So, in this paper, we propose an extension of the SPIRIT IP-XACT view to complete all the verification process requirements. A part of the required extension is described in the table below:

Table 5: Subset of SPIRIT IP-XACT Extension

Name	Description
RTLDesingName	Vhdl:Entity(architecture) Verilog: Component name
GenericsName	Vhdl: The generics name Verilog:The parameter name
GenericsDefaut	The generic signal's verification value
GenericsValue	The generic signal's default value
GenericsImpact	The affected field, register or signal by the generics (i.e. filed x reserved)
GenericsDescription	The description of the generic
RegRTLType	The register'sRTLtype (i.e. std_logic_vector)
RegRTLPath	The register's RTL location
RegSwReset	The register's software reset value
FldAccessMode	The field's access category
FldRTLPath	The field's RTL location
FldRTLType	The field's RTLtype (i.e. std_logic_vector)
OutHwReset	The output's hardware reset value
OutSwReset	The output 's software reset value
Timing	The output 's synchronisation
EdgeActiveLevel	The output 's active level.
Type	The output 's type (i.e. std_logic_vector)

By adding generics value in the new SPIRIT IP-EXACT extension, we allow the automation tool to generate generics dependent VE,checkers and coverage according to the "GenericsImpact" of the "GenericsValue." For example, if the access mode of a particular register field is generics dependent, the generated checkers/coverage will take this in to account based on the SPIRIT IP-XACT item "GenericsImpact."

The SPIRIT IP-XACT standard, defines five access types:

- read-write
- read-only
- write-only
- read-once
- write-once

These five access modes describe the frequently used mode, but not all possible accesses. For example, based on this access type, we are not able to describe a register field with write protection enabled only when a certain condition like flag setting or input toggling. This kind limitation is a killer for an automation process and prevents us from covering complex behavior.

As a consequence, we started to define an exhaustive set of access types. To assure the completeness of the access modes in SPIRIT IP-XACT extension, we adopted the following approach. We first defined the six basic access modes, described in the following table.

Table 6: Basic register access mode

Basic access	Access mode description
WNA	Write Not Allowed. Any write access is ignored
WANP	Write Allowed and Not Protected. All write access must succeed.
WAP	Write Allowed but Protected Write when protection enabled are ignored. Write when protection disabled must succeed.
RNA	Similar to WNA , but for read access
RANP	Similar to WNA , but for read access
RAP	Similar to WNA , but for read access

Then we cross these basic modes: in each loop, we must select one write access type from the three write access modes, and select one read access type from the three read access modes. The mathematical interpretation of this cross helps to accurately identify the number of possible access modes for register access, which is: $C_3^1 \times C_3^1 = 9$ access modes. The SPIRIT IP-XACT supports only three of them as described in the table below.

Table 7: IP-XACT vs IP-XACT Extension

Basic access		Correspondence with IP-XACT extension	Correspondence with IP-XACT
Read	Write		
RNA	WNA	Reserved	Not supported
RNA	WANP	Write only	Write-only
RNA	WAP	Write only protected	Not supported
RANP	WNA	Read only	Read-only
RANP	WANP	Read Write	Read-Write
RANP	WAP	Read / Write protected	Not supported
RAP	WNA	Read only protected	Not supported
RAP	WANP	Read protected / Write	Not supported
RAP	WAP	Read protected / Write protected	Not supported

Finally, SPIRIT IP-XACT defines two access modes, read-once and write-once, with no direct equivalent in the extension, since they are a subset of the more general definition in the IP-XACT extension. For example, write-one access mode in IP-XACT is included in Read/Write-protected access mode in the IP-XACT extension.

4.4.2 Functional checkers generation

We classify functional checkers in two categories: basic and specific checks. Basic checks will be verified using the automation flow and will cover the following:

Hardware reset checks: generated based on registers or on register filed, according to the granularity required by the verifier. The same kind of checks are done for output signals.

Software reset checks: generated for all registers, fields, and outputs impacted by the software reset of the design.

Read/Write access: generated in black box and gray box approach, based on registers and register field granularity.

Interrupt generation: generated in double directions to check the interrupt setting when the flag is set and enabled, and to check that the interrupt enable condition exists when interrupt arises.

DMA request: generated in double directions to check the DMA request setting when the flag is set and enabled, and to check that the DMA enable condition exists when DMA request arises.

Specific checks will cover all complex features of the Design Under Verification (DUV) and will be defined and embedded in the verification environment directly by the verifier himself.
All checkers are generated for both gray box and black box approach. Black box checkers are the recommended ones, being implementation independent, but since they are more complicated, sometimes having the possibility to debug using the gray box checkers is a valuable aid. The white box approach will be used by the verifier only when required, and it will not be considered during the automatic generation process.

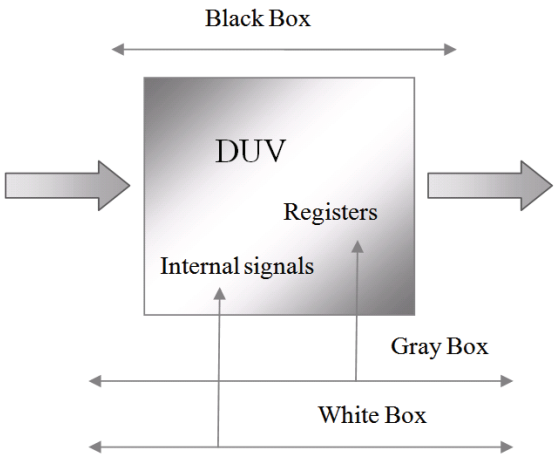

Figure 5: Different approaches used

Checks related to Hardware/Software reset or read/write access are also duplicated for better flexibility. Parts of them are generated based on a field's granularity: each field has a dedicated checker. Duplication is also used at register level: ideally a dedicated checker will be assigned to each register. The register fields can be accessed in different modes, such as read/write, read only or write only, reserved, and so forth, as specified in the SPIRIT IP-XACT standard. In our SPIRIT extension, new access modes have been added, such as for write protection. This additional information will allow us to generate 2 to 4 checkers more for each field. During the debug phase, the granularity of checkers based on fields is more precise since it highlights problems related to the single field, but when the VE and DUV are quite stable, and for performance reasons, it's recommended to run only the checkers on registers.

4.4.3 Basic Functional Coverage

The basic functional coverage is implemented for input, output signals, and registers. For each of them, four dedicated coverage items are generated to check coverage of rising edge, falling edge, low and high values.
In a formal verification context, input, output and register basic coverage prevents an over-constrained or dirty verification environment, as well as eventually spotting functional bugs. For example, a potential functional bug may be discovered when all issues related to the VE are addressed (i.e. clean and not over-constrained), but the formal proof for a cover item related to an output interrupt rising edge signal never occurs, causing that item to fail.

In a dynamic verification context, basic functional cover items will provide an additional coverage metric to measure simple signal toggling or register fields' values coverage.
From the basic coverage items generation, the verifier is able to build complex functional coverage items, combining sequences of events.

4.4.4 Functional Constraints Generation

In the context of ABV, an assertion-based approach is used to force the design into a particular configuration. This is done based on constraints on registers and fields. Four types of functional constraints are generated:

Stability constraints type1: when the IP is enabled, the configuration register config_reg_i must be stable.
Stability constraints type2: when the IP is enabled, the configuration field config_reg_i_field_j must be stable
Configuration constraints type1: when the IP is enabled, the configuration register config_reg_i equals a particular value.
Configuration constraints type2: when the IP is enabled, the configuration field config_reg_i_field_j is equal to a particular value. For example, if a register field has two bits length, four constraints will be generated to cover the different possible values.

The abovementioned constraints will be used during the assertion-based verification process to quickly configure the design just by enabling or disabling the given constraints, which dramatically reduces the verification time.

4.4.5 Test Bench Generation

The test bench generation is based on three components. The first one is the RTL top level file in which the DUV and VIPs are instantiated. The second one is the definition file in which all registers and fields are declared. The third one is the mapping file, in which all registers and fields are mapped to the corresponding signals in RTL.

In addition to that, some templates are generated, making the manual checker's implementation easier, facilitating verification re-use, clear coding style and quality of the verification environment. For example, in the context of formal verification, one proposed approach is to dedicate four kinds of checks for each output and register field; these four checks will guarantee complete coverage of the signal behavior.

Check1: *when the output setting condition occurs, is the output set?* It's a direction-1 test. To distinguish this kind of test, the naming convention adopted is "SCSET" which means: Sufficient Condition to SET.
Check2: *when this output is set, is the output setting condition was happen?* It's a direction-2 test. To distinguish this kind of test, the naming convention adopted is "NCSET" who means: Necessary Condition to SET.
Check 3 and 4 are similar to the 1 and 2 and dedicated for the resetting condition:
Check3: *when the output resetting condition occurs, is the output reset?*
Check4: *when this output is resetting, is the output resetting condition was happen?*

A template file will be dedicated for output and another one for registers. For example, a PSL VHDL flavor template, dedicated for registers, will follow the structure below:

```
----------
-- reg_x
----------
  -- reg_x_field_y
    --SCSET
    property ..._SCSET_regx_fieldy is always( );
      assert ..._SCSET_regx_fieldy;

    --NCSET
    property ..._NCSET_regx_fieldy is always();
      assert ..._NCSET_regx_fieldy;

    --SCRESET
   property ..._SCRESET_regx_fieldy is always();
      assert ..._SCRESET_regx_fieldy;

    --NCRESET
   property ..._NCRESET_regx_fieldy is always();
      assert ..._NCRESET_regx_fieldy;

  -- regx_field_z
     ……
----------
-- reg_y
----------
   …..
```

In this way, in addition to the basic checks fully generated, the automation flow will provide not only an advanced starting point for the verifier, but also better reuse and quality by adopting a unique coding style.
Header files are also supported and dynamically generated from the SPIRIT IP-XACT extension, and contain the design reference, the verifier reference, and date of generation. All this information is already stored in the SPIRIT IP-XACT extension.

4.5 Multiple Verification Language Support

To be able to easily support multiple verification language, we generate the checker and coverage in an intermediate format, and then a specific language-dependent checker/coverage item will be generated, as explained in the figure below

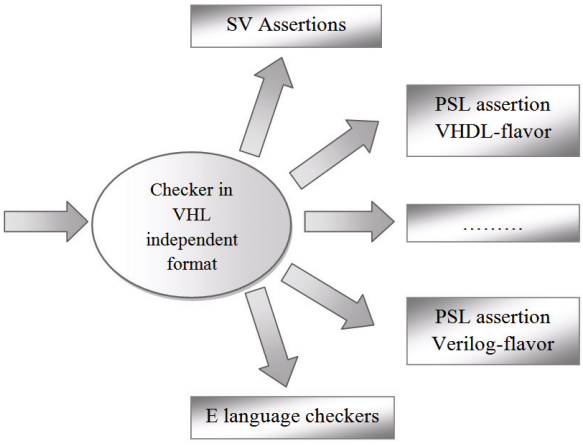

Figure 6: Pre-processing checks

All generated files and items will follow a common and rigorous coding style and naming conventions with headers structure, facilitating readability and re-use of the VE.

4.6 Post-processing Checks

Once the generation process is completed, some post-processing checks are done to make sure that the generation process was successful. The first part of these checks secure the correctness of the generated SPIRIT IP-XACT and SPIRIT IP-XACT extension views by verifying that all the mandatory SPIRIT IP-XACT information is present (i.e.: registers, input, output). In addition to that, the SPIRIT IP-XACT extension checks cover the existence of the mandatory information for the verification flow like master reset signal, main clock signal, and bus system,
The second kind of post-processing checks focuses on the generated checkers and coverage items. We verify the number of generated checkers/coverage items for each kind of test. Each checker/coverage item expression is analyzed to ensure the existence of the required mandatory information (i.e.: abort condition, main clock.). Each generated file is then analyzed to guarantee the correctness of headers, verification language general requirements and syntax (i.e.: entity, architecture, component existence).

4.7 Third Step: Use Model and Application

For this new flow, we propose two main use models: IP verification use mode and general use model.

4.7.1 Description of verification use model

In the IP verification use model, by using all generated data for verification purposes, we insure the completeness of the checkers and coverage items with respect to the information provided in the specification documents, and we provide an easy and fast flow to reiterate specification and consequently verification changes that might occur in the project cycle, for example ,due to bug fixes.
In addition to this, having a standard generated coding style will help verification reuse and the application of common verification strategy within the verification team.

4.7.2 Description of the general use model

The SPIRIT IP-XACT view, and the new SPIRIT IP-XACT extension one, contain valuable data for further automation processes. In some cases, like automatic generation of C libraries for application validation, the SPIRIT IP-XACT view is enough. In other cases, like FPGA prototyping, SoC integration, and SoC verification, the new SPIRIT IP-XACT extension view is required. So the new proposal for SPIRIT verification extension increases the return of investment (ROI) on Digital IP projects, because the generated SPIRIT View, standard and verification one, can be reused in different phases of the product development flow.

5 APPLICATIONS

5.1 Design Under Verification and Context

The first block to be considered is a simple control block, used in a System-on-Chip (SoC) Touch Sense Control application. The design hierarchy summary is described in the table below.

Table 8: Design Summary

Registers	Fields	Inputs	Outputs	Address	Data bus
21	145	23	36	8	8

The goal is the full verification of this DUV, the basic checks will be generated by the proposed automation flow, and the specific checks will be hand written by the verifier.

5.1.1 Original document description:

The original document contains 30 pages and 6 kinds of tables. The automation tool focuses only on information presented in these tables, and from them it extracts all the needed data. The first table contains general information like the design name, the revision number, and the author, for example.

Table 9: General Description

Filename	Design name	Design rev	Chrono.	Author	Division	Release / Date	Prev. release	Where-used
…	…	…	…	…	…	…	…	…

The history table describes all the specification document revision/comments.

Table 10: History

Date	Revision	Main changes
….	…	…

Each one of the 21 registers of the design is represented by two kinds of tables. The first table describes the field mapping of the register and the access mode of each one. The second one contains the detailed description of these fields.

Table 11: Register Description

7	6	5	4	3	2	1	0
Fld7	Fld6	Fld5	Fld4	Fld3	Fld2	Fld1	Fld0
rw	ronly	res.	rw	rw	read/ wprot	ronly	rw

Table 12: Field Description

Field	Description
Bits 7:2	…
Bit 1	…
Bit0	…

Other valuable information is present in the register map table, which contains the address offset and the reset values of registers.

Table 13: Register Mapping

	Name	7	6	5	4	3	2	1	0
Address Offset	Reg. name reset value	Fldx 0h	Reserved 0h		fld x	..	..	..	..
04h	Reg_name	rw	res.		rw	rw	rw	r	r
..	..	..	..		..	..	..	..	..

Finally the specification document contains the interface table described below.

Table 14: Design Interface

Signal	Type	Directive	Source / destination	edge active level	Timing	HW /SW reset	Description
…	…	…	.....	…	…	…	…

5.2 Preparation Step

The first step consists of tagging the tables, the register map table and all the fields map and fields description, which must be tagged as follows:

Register map table: This table must have the tag *"t1_f2sRegMapTable"*.
Fields map table: these tables describe a map of the fields of each register and the access mode. These tables must have the tag *"t1_f2sFieldsMapTable_ REGSITER_NAME";* the tag must contain the name of the corresponding register.
Field description table: These tables describe each field of the registers and made of two columns: the range and the corresponding description. These tables must have the tag *"t1_f2sFieldsDescTable_ REGISTER_NAME"*; as for the fields map tables, the tag must contain the name of the corresponding register.
Interface table: Contains input and output signals, this table must be tagged with "t1_f2sInterfaceTable".

This step has taken about 15 minutes to be achieved.

5.3 Pre-processing Step

The automatic pre-processing checks took around five minutes. The following results were obtained:

Register Map Table content: Some reserved fields are missing their corresponding reset values.
Table format: The first fields map table is not conforming to the mapping of the register map table (the SWIx fields have been assembled in one group).
Tags: the fields map tables and the fields description tables of each register are not tagged.

After addressing the previous issues, the second pre-processing check returns a positive result. Finally, this step has taken about 15 minutes to be achieved.

5.4 Generation Step

5.4.1 Checkers and coverage

Checkers are generated in both black box and gray box approach as well as based on register and field description. The list is described in following table.

Table 15: Generated Assertions and Coverage

	Files	Number of properties GB/BB
HW reset	hwrst_output.psl	255/NA
	hwrst_reg_wb.psl	63/21
	hwrst_field_wb.psl	435/145
SW reset	swrst_output.psl	NA /NA
	swrst_reg_wb.psl	NA /NA
	swrst_field_wb.psl	NA /NA
Read/ Write	write_read_reg.psl	NA/36
	write_read_field.psl	NA/145
Read	read_reg_wb.psl	36/NA
	read_field_wb.psl	145/NA
Write	write_reg_wb.psl	36/NA
	write_field_wb.psl	145/NA
Cover	cover_inputs.psl	1080/NA
	cover_outputs.psl	916/NA
	cover_registers.psl	672/NA
	cover_fields.psl	672/NA
Total number of properties		4802

The automation tool generates different kinds of assertions and cover properties. When it is possible, two kinds of assertions are generated: the first one uses only interface signals (input/output), we referenced it with BB in the previous table, and the second one is based on both interface signals and the registers value that we referenced with WB in the previous table. We use both kinds, but during the debugging phase, especially for read and write access, we start to secure the correctness of the read and the write access separately first, and then we run the combined assertions using only the interface signals by checking the sequence *read -> write -> read* .

5.4.2 Example of generated assertions

The automation tool generates the assertions/checkers and coverage items in different languages; in our verification environment example we generated PSL assertions in Verilog flavor.

The first example is a PSL cover used in formal verification to check the capability of the interrupt line INT to rise, as well as to confirm that the verification environment is not over-constrained.

```
..._cover_output_INT_rise : cover {rose(INT) };
```

Another example is an assertion to check that the field SWI5 of the register SWIR took the correct hardware reset value.

```
.._nareset_rise_field_TS_SWIR_SWI5_eq_0: assert always
({[*1];rose(nareset)}|->{TS_SWIR_SWI5==1'b0});
```

The last assertion example checks that the field EN of the register CR took the right value after a write access. In bold font we clarify the manual update that must be done for a subset of registers requiring an additional delay.

```
.._write_field_TS_CE_EN_eq_DBI2: assert always ({ nsel ==
1'b0 && TS_SWIR_ADD == 1'b1 && rw == 1'b0 }  |=>
{[*1]; prev ( dbi[2],2) == TS_CR_EN})
abort ( nareset == 1'b0 )@(posedge clk);
```

5.4.3 Post-processing step and application

Automatic post-processing checks took around five minutes, and no issue is reported. Then we started the functional verification of the DUV using the CADENCE IFV formal verification tool. We started to evaluate the correctness of the verification environment structure and constraints by looking at cover items results. All fails were expected and due to targeted configuration or disabled features.

Hardware reset assertions showed an issue related to the hardware reset value of one register; the reset of this register was made in two steps, but the specification was only describing the stable one, so a modification in the specification was done to highlight this point.

For read and write access, an important subset of the PSL assertions was initially failing, and after some debugging, we discovered that for some registers, a one cycle delay was required to get them updated after a write access. So after updating their corresponding set of assertions by adding a one cycle delay in each assertion expression, all write accesses were successfully passing except one of them that failed even with the added cycle of delay. The investigation showed that after a write access, the update of this register value is variable and dependent on additional conditions that were not documented. Read access was successful.

This quick generation of ready-to-use assertions reduced the time to build the verification environment and to write the set of basic checks and coverage from about two weeks (traditional flow) to around two hours (this new flow), and the verifier could quickly debug the spotted issues and reiterate the generation and verification.

6 CONCLUSIONS

This work has shown the added value in the Digital IP design phase, both in terms of reducing project time quality improvement. Future development will focus on the generation of other flows, starting with the SPIRIT extension view, such as C code generation for application validation and customer documentation. In addition to this, we will also embed the capability to automatically generate the SystemRDL description in our flow from the specification documents.

The results presented in this paper are opening challenging investigations to a wide deployment of the automatic verification impacting different design life cycle. This approach, based on SPIRIT, also increases the synergy between the different teams involved in an SoC development process, leading to improved productivity and reduced design project time, while improving quality and reuse of the design verification environment.

7 ACKNOWLEDGMENTS

Special thanks to all people that contributed to the implementation of this work, especially Oussama Chelbi, Baha Bennour, Mohammed Beji, Amira Hasnaoui, and Farid Timoumi.

8 REFERENCES

[1] Abode FrameMaker7.1. Database Publishing

[2] Adobe FrameMaker7.0. MIF Reference Online Manual

[3] Adobe Portable Document Format Version 1.7

[4] Benefits of Rich Test Format (RTF). Desktop Publishing. Presentations & Word Processing

[5] Blueprint Compiler: https://www.denali.com/en/products/blueprint.jsp

[6] Callari, F. File Management Systems

[7] Chiang, C.C.. 2006. TUG: An Executable Specification Language, Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science 2006

[8] Common Format and MIME Type for Comma-Separated Values (CSV) Files RFC 4180.

[9] Corey, D.E. DBMS vs. File Management System

[10] Curzon, P., Tahar, S.. 2001. Automating the Verification of Parameterized Hardware using a Hybrid Tool, 13th International Conference on Microelectronics.

[11] Dahan, A., Geist, D., Gluhovsky, L., Pidan, D., Shapir, G., Wolfstahl, Y., Benalycherif, L., Kamdem, R., Lahib Y.. 2005. Combining System Level Modeling with Assertion Based Verification. Proceedings of the 6th International Symposium on Quality of Electronic Design (ISQED 2005), pages 310-315.

[12] De Loore, B.J.S. and Kostelijk, A.P. . Automatic Verification of Library-based IC Designs

[13] Ecker, W., Esen, V., Steininger, T., Velten, M., Hull, M.. 2006. Specification Language for Transaction Level Assertion. In Proceedings of the 2006 IEEE International High Level Design Validation and Test Workshop (HLDVT'06), pages 77-84.

[14] Eisner, C., Fisman, D.. 2006. A pratical Introduction to PSL. SPringer, New York.

[15] Extensible Markup Language (XML) 1.1 (Second Edition) W3C Recommendation 16 August 2006.

[16] Fertalj, K., Kalpić, Vedran Mornar, D.. 2002. Source Code Generator Based on a Proprietary Specification Language, Proceedings of the 35th Hawaii International Conference on System Sciences.

[17] Hsiung, P.A. and Cheng, S.Y.. 2003. Automating Formal Modular Verification of Asynchronous Real-Time Embedded Systems, Proceedings of the 16th International Conference on VLSI Design (VLSI'03).

[18] HTML 4.01 Specification W3C Recommendation 24 December 1999

[19] IP-XACT 1.5 specification, see http://www.SPIRITconsortium.org/

[20] Jacobi, C., Weber, K., Paruthi, V., and Baumgartner, J. 2005 Automatic Formal Verification of Fused-Multiply-Add FPUs. DATE 2005.

[21] Karayiannis, T. , Mades, J ., Schneider, T. , Windisch, A. , Ecker, W.. Using XML for Representation and Visualization of Elaborated VHDL-AMS Models

[22] Knäblein J., Sahm, H. Automated formal method verifies highly-configurable HW/SW interface. http://www.scdsource.com/article.php?id=332

[23] Nan, T. Fundamentals of Information Systems

[24] Oliveira, M., Hu, A.. 2002. High-Level Specification and Automatic Generation of IP Interface Monitors. In Proceedings of the 39th Design Automation Conference (39th DAC), pages 129-134.

[25] Rich Text Format (RTF) Specification, version 1.9.1, http://www.microsoft.com

[26] Rogin, F., Klotz, T., Fey, G., Drechsler, R. and Rülke, S.. 2009. Advanced verification by automatic property generation,IET Comput. Digit. Tech. -- July 2009 -- Volume 3, Issue 4, p.338–353

[27] SystemRDL 1.0 specification, see http://www.SPIRITconsortium.org/

[28] SystemRDL 1.0: https://www.denali.com/en/products/systemrdl_about.jsp

[29] The SPIRIT Consortium official website http://www.SPIRITconsortium.org/home

[30] Word Binary File Format (.doc) Structure Specification, http://www.microsoft.com