Building Portable Stimulus Into Your IP-XACT Flow

Petri Karppa
petri.karppa@nokia.com
Nokia Networks
Tampere, Finland

Lauri Matilainen
lauri.matilainen@nokia.com
Nokia Networks
Tampere, Finland

Matthew Balance
matt_ballance@mentor.com
Mentor, A Siemens Business
Wilsonville, OR

***Abstract*- The upcoming Portable Stimulus standard from Accellera provides an abstract format for declaratively specifying verification intent for a given design-under-test (DUT), enabling multiple implementations of that intent to be gen-erated for the different verification platforms (simulation, emulation, prototypes, etc.) used throughout a typical verification flow. At Nokia, we use IP-XACT (IEEE Std 1685-2009) to characterize the IP blocks of our System-on-Chip (SoC) designs. This characterization captures the designers' knowledge of a given block using machine-readable meta-data that can be used by an SoC assembly tool to build the actual design. We have found that each of these standards can increase our productivity by enabling reuse from the IP block to SoC level. This paper will examine ways to achieve even better results by combining them in our verification flow.**

I. Introduction and Motivation

Composing SoC designs by hand continues to be an error-prone process. Stitching RTL together by hand requires lots of boilerplate code to be hand-created, resulting in errors being introduced such as incorrectly-connected interfaces or incorrectly-configured IPs. Both in-house and commercial tools have been created to help address the SoC-assembly challenge. Many leverage the IP-XACT standard [1] for characterizing the interfaces, memory maps, and files of IP blocks, and describe the assembly of those IPs into a hierarchical design. Automating creation of the RTL netlist from an IP-XACT-based description saves development time and reduces IP-connection mistakes.

Creating SoC-level tests is also a challenging and time-consuming process. The bare-metal software environments used to test SoCs are typically limited to hand-coded directed tests, limiting the number of test scenarios that can be created and increasing the risk of missing corner cases not envisioned by the test creator. Accellera's emerging Portable Stimulus Specification (PSS) standard language [2] enables use-case scenarios to be easily and productively designed, and efficiently targeted to SoC environments using automation.

Given the wealth of design knowledge captured within an IP-XACT-based description of designs and IPs, it seems sensible to find opportunities to leverage that information to make the SoC-level test creation process even more productive.

II. IP-XACT introduction and NOKIA use model

IEEE 1685 is an XML meta-data standard to describe intellectual property (IP) characteristics in standardized XML format [1]. The standard is developed by Accelera working group and there are two IEEE generations available (2009 and 2014). Since, at NOKIA we are mainly using 2009 standard, in this paper all examples are based on 2009 standard and all references to the standard are done using commonly known abbreviation IP-XACT.

In addition to IP component description, IP-XACT also offers mechanisms to integrate IP-XACT component within design and configure component instantiations using designConfigurations. The goal of the IP-XACT is to standardize IP integration, configuration and generation flows between different EDA vendors and IP/SoC teams and furthermore automate different parts of design flows using higher design abstraction level.

A. Nokia use model

Fig. 1. shows a high-level view of the Nokia SoC Design flow. Key drivers for the flow development has been design automation, IP reuse and higher design abstraction.

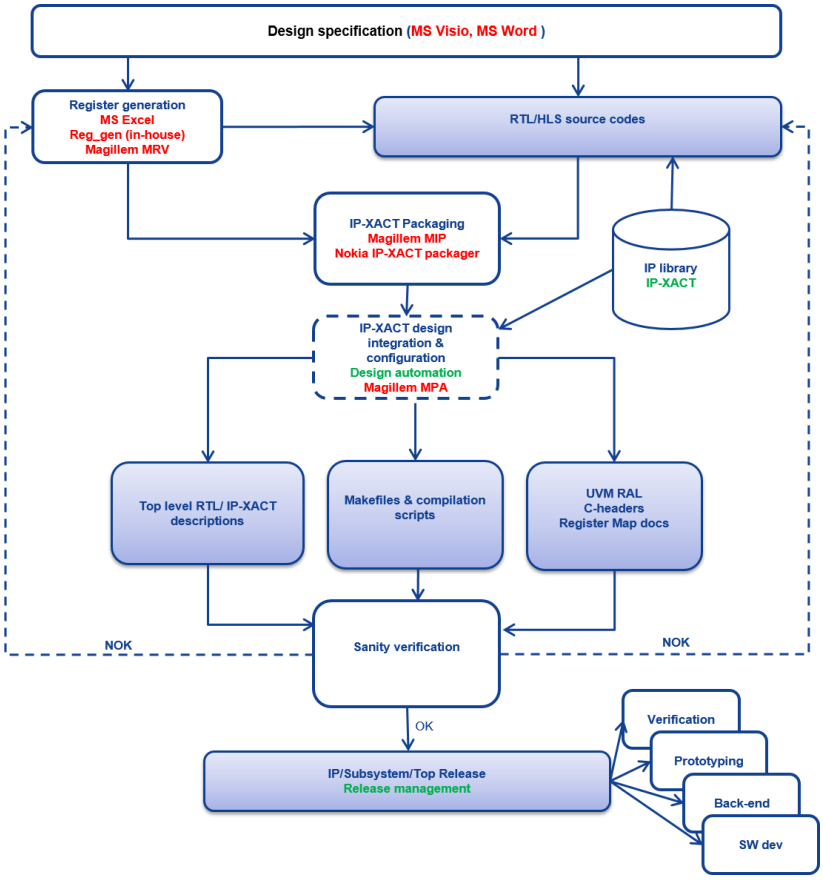

Figure 1 - High-level view of the Nokia SoC Design Flow

At Nokia we have selected IP-XACT meta-data to encapsulate all our design, IP and configuration content in standardized manner. The flow is based on commercial design services tool set [3] plus in-house tooling and flow development.

As always, the flow execution starts from the design specification work. To help RTL or HLS designers work, we have implemented in-house register generator tool (reg_gen), which generates VHDL register bank implementation from IP-XACT memoryMap description. MS Excel template is used as user interface to define registers, register fields and memories for the register bank.

When a VHDL register bank is generated and integrated within structural IP design, the module can be packaged as a leaf IP-XACT component. Our IP-packaging flow uses a commercial packaging solution to automate the IP-XACT component creation. In addition to the commercial solution we have created our own script templates and flow wrappers to help IP developers task.

The dotted line box in the middle of Figure 1 describes the integration phase where new IP-XACT components (hierarchichal and non-hierarchical) are configured and integrated together using a graphical user interface. This flow phase can be omitted for leaf modules.

The generation phase begins from the hierarchy manipulation task, where design hierarchies are re-grouped into hard-macro boundaries based on ASIC layout. This is the so-called “Virtual Hierarchy” feature of the assembly tool that we use, where we re-organize instances and manipulate design hierarchies without losing the existing connections. Hierarchy manipulation is also used when we need to create prototyping or emulation friendly top releases. E.g. All ASIC technology specific components are removed from prototyping releases. When new IP-XACT hierarchies are generated, we can run the structural RTL netlisting phase for all hierarchical IP-XACT designs.

The next step is to utilize memoryMap and interconnect component address information in generation to create system wide UVM RAL files, register map documentation and C code headers for verification & SW development. Since we have stored all RTL file references into IP-XACT meta-data, we are able to automate the Makefile and compilation script generator for all needed tools. Now, sanity verification has all required inputs from the previous

flow phase to run sanity checks and eventually we can make release serving all of design team customers. E.g verification, prototyping, back-end and SW development

Table I below lists the information which is stored into IP-XACT during our packaging flow. Each IP has a unique VLNV (Vendor, Library, Name, Version) value. This identifier needs to be unique throughout the organization, since all IP-XACT references are based on VLNV values. RTL files, including the autogenerated VHDL register bank, are added to the XML fileSet as file references. At Nokia, we define at least 4 different views/FileSet for each component. All standard interfaces are mapped into busInterfaces (e.g. AXI4 slave) to automate integration work and to enable design wide memory map calculation. As described in flow Fig. 1. IP-XACT memoryMap information can be obtained from excel description into actual module meta-data.

TABLE 1
IP-XACT ELEMENT USED AT NOKIA

IP-XACT Element	**Description**	**Example**
VLNV	Vendor, Library, Name, Version identification tuple	Nokia.com, common, axi_clock_bridge, 1.0
View/FileSet	Views (incl fileSet ref.) for each phase of the ASIC design process	Simulation, emulation, synthesis, dummy
Ports/BusInterfaces	IP entity ports mapped into busInterfaces	AXI4, AHB, APB etc.
modelParameters	VHDL generics and Verilog parameters	DATA_WIDTH_G
memoryMaps	SW accessible configuration registers and memories	DATA_RATE_CTRL_REG R/W (0x10)
Bridges/BaseAddressess	Slave to master interface mapping within interconnection components. Includes addressing information	AXI4 XBAR slave to master bridges + base address = 0x1000 000

III. ANATOMY OF A PORTABLE STIMULUS DESCRIPTION

Accellera's Portable Stimulus Specification (PSS) language enables productive description of test scenarios in such a way that processing tools can retarget the scenarios to various verification environments – including simulation, emulation, and prototype. Beyond the environment-portability benefits evoked by the standard's title, the structure of a PSS description provides some unique benefits when approaching the challenge of creating subsystem and SoC-level tests. The following sections describe the key elements of a PSS description.

A. Actions

An action is a fundamental unit of behavior in the PSS language. Actions can contain random and non-random fields and constraints on those fields. Actions specify scheduling relationships (if any) to other actions in terms of data input and output 'ports', and resources that are required.

```
	buffer mem_seg_b {
		rand bit[31:0]			addr;
		rand bit[31:0]			size;
	}

	resource dma_channel_r { }

	action dma_mem2mem_xfer_a {
		input mem_seg_b			src;
		output mem_seg_b	dst;
		lock dma_channel_r	channel;

		constraint size_match {
			src.size == dst.size;
		}

		// Target implementation left unspecified
	}
```

Figure 2 – PSS Action

Figure 2 shows an example PSS action that performs a memory-to-memory DMA transfer. The transfer requires that some other action run first to supply the source of the data (the src field). It produces an output describing the destination of the data. The transfer requires exclusive access to a DMA channel resource, which is noted with the lock field of dma_channel_r type.

Note that no implementation has been specified for the dma_mem2mem_xfer_a action. Action implementation can be very target-specific, so for flexibility the implementation will be supplied once the target environment is known.

B. Components

A significant amount of subsystem and SoC-level test creation is focused on validation, which emphasizes ensuring that the design adequately satisfies the purpose for which it has been created. Design resources, such as the number of DMA channels, are a key determinant in both the traffic that can run and the overall performance of the design. PSS components encapsulate actions along with the resources to which they have access.

```
component dma_c {
	buffer mem_seg_b {
		rand bit[31:0]			addr;
		rand bit[31:0]			size;
	}

	resource dma_channel_r { }

	pool dma_channel_r	channels[16];

	action dma_mem2mem_xfer_a {
		// . . .
	}
}
```

Figure 3 – PSS Component

Figure 3 shows the dma_c component that contains the dma_mem2mem_xfer_a action previously shown. Note that the dma_c component groups the action with a pool of dma_channel_r resources. The pool contains 16 dma_channel_r

objects, each representing one DMA channel. The pool of channel resources constraints how many DMA transfers a scenario can run in parallel, as well as assigning each transfer one of the available channels.

C. Component Tree

A full system scenario, of course, will involve multiple IPs. Representing the available resources and IP-specific actions is done by creating a component tree for the system that contains instances of PSS components available in the system.

```
component sys_c {
	cpu_c		core_cluster_0;
	cpu_c		core_cluster_1;

	dma_c		dma0;
	dma_c		dma1;

}
```

Figure 4 - System Component Tree

Figure 4 shows an example system component tree. In this system scenario, we have two instances of a CPU core cluster and two DMA engines. It isn't necessary to describe all resources that are actually present in the system. Only the resources related to the scenario being described need to be captured in the component tree.

D. Test-Realization Specification

Different verification environments have very different mechanisms for test realization. In a UVM-based IP- or subsystem-level environment, design functionality is typically exercised by running UVM sequences. At SoC level, that same design functionality is exercised by calling C functions. In some cases, users may create a single common API usable across IP, subsystem, and SoC environments. In other cases, however, it makes more sense to adapt the test intent to the available mechanisms for exercising the design.

PSS provides type extension and overriding mechanisms that make it easy to layer environment specifics into an abstract description.

```
package dma_c_pkg {

	import void dmac_start_xfer(
		bit[31:0]		channel,
		bit[31:0]		src_addr,
		bit[31:0]		dst_addr,
		bit[31:0]		size
	);

	extend action dma_c::dma_mem2mem_xfer_a {
		exec body {
			dmac_start_xfer(
				channel.instance_id,
				src.addr,
				dst.addr,
				src.size
			);
		}
	}
}
```

Figure 5 - DMA Transfer C Realization

Figure 5 shows how type extension can be used to specify an SoC-appropriate realization for the DMA transfer action. The import statement declares the prototype for an external function implemented in C that the portable stimulus test can call. The extend statement inserts a call to the dmac_start_xfer function, with appropriate values from the portable stimulus description, into the dma_mem2mem_xfer_a action. Note that both the imported function and the extension to dma_c::dma_mem2mem_xfer_a action are encapsulated in a package. This makes it easy to manage the realization mechanisms for multiple environments in the same codebase.

IV. Portable Stimulus Test-Creation Flow

When a portable stimulus description is created by hand for a given SoC, the user will typically go through the following steps.

First, the user must identify the IPs, such as DMA, accelerator, and CPU, that will be part of the test scenario, and gather the relevant PSS descriptions for each. A component tree for the system must be created with appropriate component instances corresponding to the IP instances present in the system being tested.

Often a family of SoC variants will have a library of reusable test scenarios that can be customized and applied across all variants. For example, a generic memory-map test action can be created and used across the entire SoC family simply by customizing it with the memory map for the specific SoC. A generic cache-coherence test action can be reused quite widely simply by customizing it with the cache parameters used in a given system. The verification engineer will need to identify the relevant design attributes required by the test scenario library and appropriately customize the test actions.

The user will need to compose SoC-specific test scenarios, using IP-specific primitive actions as well as actions from the test-action library. In many cases, these test scenarios will need to leverage design knowledge, such as the memory map or memory subsystem hierarchy.

Finally, the user must identify and select appropriate test-realization code for the target environment. This means going back to the IP-specific components and identifying the appropriate PSS packages that specify the platform-specific procedural-interface methods and type extensions.

At this point, the user can begin creating tests from the test scenario and platform-appropriate test realization.

V. Portable Stimulus Integration into Hierarchical IP-XACT Flow

In order to be able to use Portable Stimulus in a hierarchical IP-XACT flow the PSS content of each component need to be included into the same IP-XACT package together with the design content. The same basic elements that are used to describe design intent in Table 1 can be used to describe the PSS content as well. The needed additional elements to include the PSS material on top of design content are highlighted in Figure 6.

Figure 6 - Associating PSS Content with IP-XACT

As can be seen in Figure 6, there is one *fileSet* & *view* pair. The new PSS *fileSet* includes all the PSS sources of the IP component as well as the environment (UVM, C) specific implementations of the leaf actions. The PSS *view* refers to the PSS *fileSet*, and contains the characteristics of the PSS component. It includes the declarations of the *components* and root *actions*. Each *component* and *action* declaration specifies the full type name of the component or action, as well as any parameter definitions.

IP-XACT is designed to be extensible, enabling new meta-data to be added that may or may not be comprehended by existing tools. The IP-XACT *vendorExtensions* is used to introduce PSS-specific meta-data into the IP-XACT structure. Tooling can be implemented to create all these PSS specific structures automatically at the packaging phase. The packaging tool can read in user specified PSS file list and create the PSS *fileSet*. In addition, the tool can parse through the PSS files and create the PSS *view* and the declarations of the components and actions automatically. This approach is similar to how the design data is packaged.

A. Applications for the Hierachical PSS Description

Since PSS components and actions are specified with each IP's IP-XACT meta-data, and design hierarchy is described using IP-XACT design and designConfigurations, we have all the needed information to automate a hierarchical PSS flow.

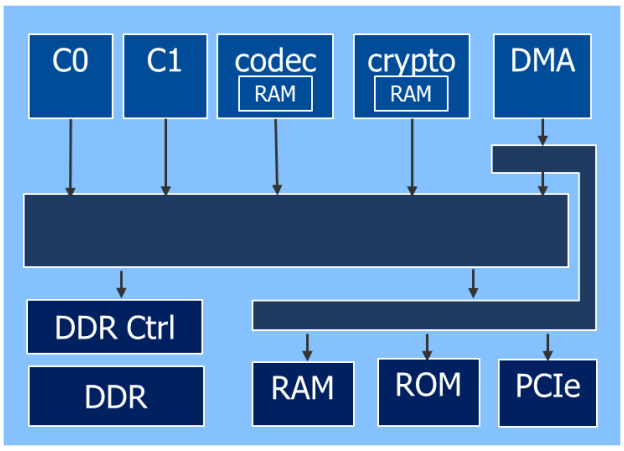

Figure 7 - Example System

If the IP-XACT component descriptions for the system shown in Figure 7 have PSS meta-data, the assembly tool needs only read in the hierarchical IP-XACT DB to learn the available PSS content. This content can be used to derive several PSS structures automatically from the design. Because each IP-XACT component contains meta-data about the associated PSS *component*, an IP-XACT assembly tool can automatically derive the PSS component tree from the IP-XACT design description.

```
component simple_soc_c {

      cpu_c              C0;
      cpu_c              C1;

      codec_c            codec;
      crypto_c           crypto;

      dma_c              DMA;
}
```

Figure 8 - Automatically-Derived Component Tree

Figure 8 shows the component tree derived from the IP-XACT description of the system shown in Figure 7. In this simple system, the description is quite short. In a realistic system, however, there will be many times more IP blocks and thus a much more complex component tree. Using IP-XACT to automatically keep the PSS component tree in-sync with the design as it changes can save significant time and effort.

The IP-XACT toolchain can also derive a series of automatic tests from the PSS *action* meta-data. The *action* meta-data describes the top-level actions provided by each component. Auto-created tests are likely to be very simple, but can still be helpful for early testing.

```
component dma_smoke_test_c extends simple_soc_c {

	action dma_test_a {
		repeat (10) {
			do dma_c::dma_mem2mem_xfer_a;
		}
	}
}
```

Figure 9 - Simple DMA Smoke Test

Figure 9 shows a simple smoke test that runs the dma_mem2mem_xfer_a action ten times. Similar focused smoke tests can be created for each IP in the system.

```
component dma_smoke_test_c extends simple_soc_c {

	action dma_test_a {
		repeat (10) {
			schedule {
				do dma_c::dma_mem2mem_xfer_a;
				do dma_c::dma_mem2mem_xfer_a;
				do crypto_c::encrypt_a;
				do crypto_c::encrypt_a;
				do crypto_c::decrypt_a;
				do crypto_c::decrypt_a;
				do codec_c::encode_a;
				do codec_c::encode_a;
				do codec_c::decode_a;
				do codec_c::decode_a;
			}
		}
	}
}
```

Figure 10 - Full-System Smoke Test

More-complex scenarios can also be automatically created. Figure 10 shows a full-system smoke test composed of the top-level actions from all components in the system. Note that the actions are run using the *schedule* keyword in PSS. This allows the PSS test-creation tool to run the actions in series and/or parallel according to the available resources in the system.

True SoC-specific use cases will still need to be created by the verification engineer. However, an IP-XACT description of the design can help with these tests as well. For example, the IP-XACT tool is aware of design content such as *memoryMap*s. This information could be utilized in the PSS editing tool to aid users in creating hierarchical test scenarios and actions. Firstly, the PSS editor could provide auto-completion option for the lower level action references because it is aware of the available lower level components and the available PSS actions for each of them. In addition, the validity of the parameter values in the action references can be checked automatically.

Similarly, because the memory map of the design is known by the tool it can validate the register/memory references.

Just as it currently does for other source files, the IP-XACT descriptions can be utilized to create the compilation script for the PSS code. The compile-script generator can select the correct leaf pss_action implementations for the specified target environment (UVM or C) based on the *fileType* definition in the PSS *fileSet*.

VI. Portable Stimulus Test-Creation with IP-XACT

The information captured in the IP-XACT description, both at the IP and SoC level, can make the PSS test-creation much more productive. The IP-XACT meta-data present with each IP captures the PSS component that represents the IP, as well as view-specific test realization descriptions. Because the hierarchical IP-XACT design description captures all IP instances, an IP-XACT processing tool can easily collect the set of PSS components that are relevant for a given design.

An IP-XACT processing tool can easily generate the PSS component tree based on its knowledge of which IP-XACT component instances have PSS content. An IP-XACT IP and design description also contains a wealth of information about the design specifics, including memory map and memory subsystem structure. An IP-XACT processing tool can easily generate a design-specific memory map for use in customizing a test library or for use by the verification engineer in creating SoC-specific test scenarios. An IP-XACT processing tool can even extract the appropriate test realization PSS description based on the select design view.

The verification engineer is still responsible for composing SoC-specific test scenarios leveraging actions from the action library and IP-specific primitive actions. However, an enormous amount of infrastructure has been derived from the information captured in the IP-XACT meta-data at both the IP and system levels.

Using IP-XACT to assemble designs already provides clear productivity benefits. Adding PSS meta-data to the IP-XACT descriptions enables even more of the test environment to be automatically derived from the design description, boosting verification productivity as well.

References

[1] Accelera IP-XACT working group, http://www.accellera.org/activities/working-groups/ip-xact.

[2] Accellera Portable Stimulus Early Adopter Specification, http://accellera.org/news/press-releases/244-accellera-portable-stimulus-early-adopter-specification-now-available-for-public-review

[3] Magillem . http://www.magillem.com/