EASI2L: A Specification Format for Automated Block Interface Generation and Verification

Chintan Kaur[1] Ravi Narayanaswami[2] Richard Ho[2]

[1] Rensselaer Polytechnic Institute, Troy, New York, 347-571-1176, chintan.kaur@gmail.com
[2] Google Inc., Mountain View, California, {swamiravi, riho}@google.com

Abstract - **Digital designs for FPGAs and ASICs are typically composed of blocks of logic connected with a variety of interface communication paths including handshake control signals, buffered data buses and broadcast values. For interfaces that are not a standard bus protocol, such as AMBA or OCP, the specification of the data signals, control signals and the transfer protocol is often in a natural language, such as English, supplemented with timing waveforms in a microarchitecture specification. This often leads to omissions or misinterpretations on interfaces spanning design blocks written by different designers. In addition, the task of verifying the correct functional behavior of the interface is time consuming and also subject to interpretation of the written microarchitecture specification. Our observation is that the majority of interfaces used in designs are actually variations of a small set of interface types. Each interface type comprises a set of parameterized signals and a control protocol. In this paper, we present the results of our interviews with a selection of designers about the interfaces they use. We then propose a set of interface types, a specification format that can describe real interfaces in terms of the library of interface types and finally, we describe a tool that generates design and verification code from the interface specification. Source code is not released as part of this publication.**

I. INTRODUCTION

System-on-chip (SoC) designs are usually composed of multiple custom designed or IP subsystems. To handle the complexity of design, these subsystems are usually further subdivided into blocks and sub-blocks. In this paper, we will generically refer to blocks, sub-blocks or even deeper hierarchies as simply 'blocks'. The communication between different system components and with the outer world occurs through interfaces. While many subsystems communicate using standard high-level protocol buses such as AMBA and OCP, most smaller and less complex blocks communicate via simpler ad-hoc interfaces. We postulated that although the functionality of blocks differ substantially in different designs, the majority of the ad-hoc communication interfaces could be expressed using a small, finite set of interface primitives. These primitives could be defined and re-used in a way similar to the standard high-level interfaces. We will describe these interface primitives in the next section.

Architectural details about designs are provided in microarchitecture documents which are typically lengthy and tedious to read/write. They can also be easily misinterpreted by different readers. Designers need to carefully study the interface they are communicating to every time to account for small differences in the intended behavioral properties between different designers. Ambiguities in design specifications can lead to confusion for those producing and consuming interfaces and can easily become a source of error and bugs. Moreover, it can take a week or more to understand interface requirements and set up an initial verification environment.

We take advantage of our observation about low-level interfaces being standard and design a specification format for interfaces. This structured information is easy to understand both by human designers and is machine processable. We have also developed a tool called EASI2L, Easy Interface Implementation Tool, to use the specifications and automate block interface generation and verification. By generating interfaces automatically, the error due to discrepancies in understanding between different designers would be minimized and would promote reuse. Also, automating the task of redundant interface code generation would free up engineers' time and they can focus on functional specification and implementation.

To automate interface generation, the first task is to decouple interface specification from the functional microarchitecture specification. For this, an interface specification format is defined in the form of protocol buffer messages. This specification format is discussed in Section III.

The structured information is then used by EASI2L to generate standard interface RTL for low-level point-to-point interfaces and UVM based verification files (UVM Agents - sequencers, drivers and monitors,

example sequence library, example test library, virtual sequencer, assertions, coverage, TLM analysis ports and UVM environment) by leveraging Go language's function maps and text template functionality. Assertions are automatically generated to ensure standard behavior of the interfaces. Coverage points make sure that the design has been tested with all kinds of stimuli that can be sent on the interface. The interface connections between different hierarchies of the design block are also automatically handled. Testbench top file for each design block is generated along with all the tool specific files required for elaboration and simulation of the design and verification environment. The tool is covered in detail in Section IV and examples are provided in Section V.

II. INTERFACE TYPES

Consider a chip/system. The system is divided into subsystems, which contain blocks and the blocks divide into sub-blocks. Interfaces are defined as bundles of signals which operate according to a particular protocol and enable the transfer of data between different hierarchies within the system. For any interface to be specified, a designer typically defines the parameters given in Figure 1.

1. Control
- a. Data Transfer Mode
  - Single Non-Pipelined Transfer
  - Pipelined Transfer
  - Burst Transfer
  - Split Transfer
  - Out-of-Order Transfer
  - Broadcast Transfer
- b. Flow-Control

2. Fields
- a. Address Lines
  - Address Bus Width
  - Separate Read/Write Address lines or Shared
- b. Data Lines
  - Data Bus Width
  - Separate Read/Write Data lines or Shared
- c. Other Sideband/Test/Debug/Security Signals

3. Initiator/Target/PassThrough

4. Clocking
- a. Frequency
- b. Duty Cycle
- c. Clock domain crossing

5. Reset
- a. Synchronous or Not
- b. Clock
- c. Active High/Active Low

Figure 1: List of generic interface parameters defined by the designer while designing any interface between blocks

We investigated interface primitives by interviewing designers from a variety of backgrounds who had experience in a number of different domain areas. We analyzed the natural language descriptions of the protocols they used and classified all the protocols into the following flow-control mechanisms:

1. Static, Cycle Accurate, Timing Based
2. Valid-Ready / Req-Ack
3. Push - Almost Full / Vldx - Rdyx
4. Put Get / Ready-before-Valid
5. Valid-always Ready
6. Credit Flow

Other high level interfaces, such as AXI4 and PCIe were also used. We propose a specification format in the next section such that the library can be extended to incorporate these high level protocols.

We implement only *non-pipelined* data transfer type for this paper. The other data transfer types such as *pipelined*, *burst* and *broadcast mode* are also commonly used and could be implemented in the future. In *pipeline mode*, multiple requests can be pipelined up to a certain pipeline depth. For *burst*, with single request, multiple data items can be requested. In both cases, data is sent after every constant interval of processing time as in *non-pipelined mode*. In *broadcast* mode, data is broadcasted to multiple destinations at the same time.

Next, we discuss each of the flow-control mechanisms in detail.

A. Static, Timing Accurate

In this handshaking protocol (Fig. 2a), the slave is always ready to provide the data whenever master requests for it. The slave processes the request in finite, pre-defined time which can be different for different designs. There is no actual handshake and dead data field is sent for bubbles in data.

B. Valid - Ready/Request - Acknowledge

In *Valid-Ready* (Vld-Rdy) handshake protocol (Fig. 2b) when the block has data to send, valid is asserted and it waits for the ready signal. When both valid and ready are high on same clock cycle, data is considered to be sent. Ready can also be asserted without waiting for data to be available. In this case, sometimes designers call it *Request-Acknowledge* (Req-Ack) protocol.

C. Push - Almost Full/Vldx - Rdyx

These are also similar to the Vld-Rdy protocol with a minor difference that ready is deasserted ‘x’ cycles before the FIFO is full to accommodate for path delay between the blocks.

D. Put-Get/Ready-before-Valid

In this flow control protocol (Fig. 2c), get signal should be asserted before the put is asserted. That is, the block must be ready to accept the data before the other block ries to send the data.

There are bubble cycles in Ready-before-Valid handshake since the data is not transferred continuously. After every data transfer, valid is deasserted and wait for ready to be asserted. But the slow data transfer doesn’t matter when only a few important packets are being sent over time and we don’t want to keep sender blocked with valid signal asserted. In such cases, it is helpful to know in advance whether the receiver is ready or not.

E. Valid-always Ready

This is also a variant of Vld-Rdy with ready always true. That means the receiver is always ready. There is no need to have separate ready signal. The timing diagram of this protocol is shown in Fig 2d.

F. Credit Flow

In this flow control protocol (Fig. 2e), a credit counter is maintained at the sender. Over time, the receiver sends credits to the sender according to the availability of buffer space to receive data. The sender needs to have at least one credit available to send data to the receiver. At every data transfer, the current credit balance is decreased by one.

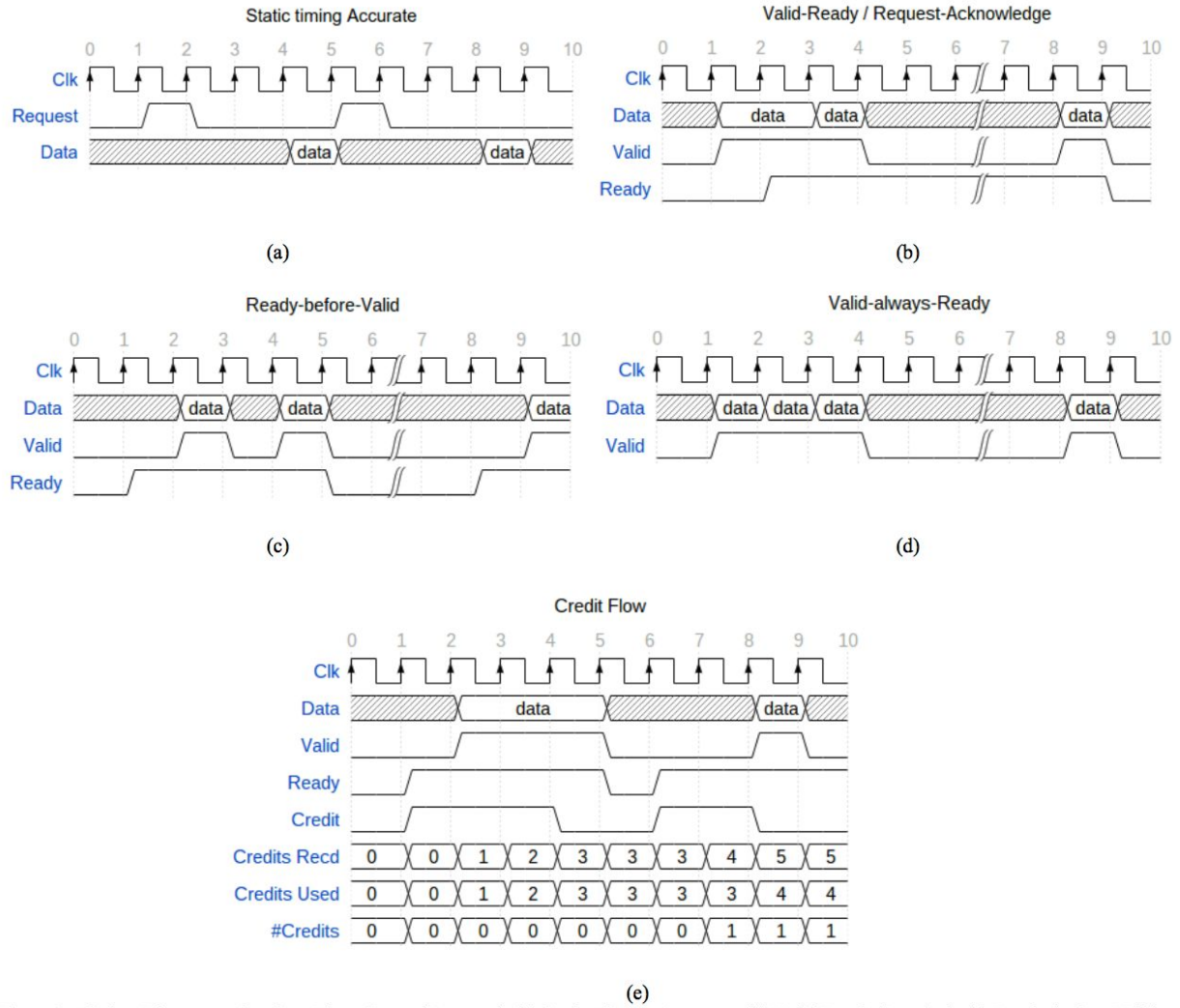

Figure 2: Timing Diagrams of various Flow Control Protocols (a) Static Timing Accurate, (b) Valid-Ready/Req-Ack, (c) Ready-before-Valid, (d) Valid-always Ready, (e) Credit Flow

Of the aforementioned flow control protocols, *valid-ready* and *vldx-rdyx* are most widely used in designs, closely followed by *credit flow*. We chose to start by implementing *credit flow* because it is relatively complicated compared to other protocols in the list.

In our interviews, many designers echoed the need of having standardized naming conventions for interfaces. For example, the same valid-ready type interface is called request-acknowledge by some which causes confusion. Also, protocols with the same name sometimes have different interpretations by different designers. It was also suggested to have a methodology that gives users a template to easily add new interfaces to the tool library. Handling connectivity between different layers of design was seen as a boring and error-prone task which had potential to be automated. Designers were very interested in having automatic addition of retiming blocks. They also suggested to optionally generate interfaces with flops on the outputs to simplify timing analysis at the top level. We have tried to incorporate many of these inputs while designing the specification format for the interfaces and developing the tool.

III. Specification Format For Interface Types

Inputs from the designers are used to design an interface specification format to decouple interface specification from the functional specification in the microarchitecture documentation (Fig. 3). We chose protocol buffers [1] (protobuf) messages to define these specifications. Protobufs are helpful because they are language and platform independent and easily extensible for serializing structured data. Once the structure of the data is defined, automatically generated code can be used to read and write the structured data using different languages. As would be seen in the next section, we use Go language for taking in specification inputs and generating desired SystemVerilog files.

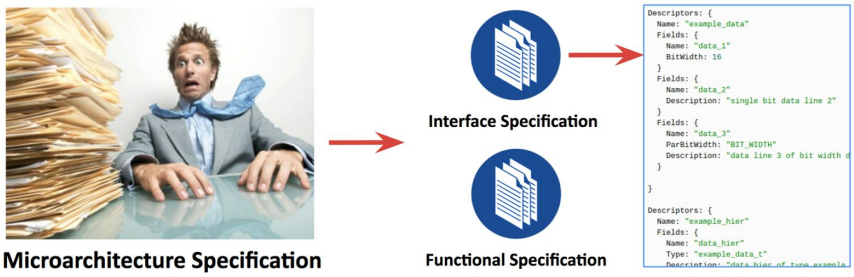

Figure 3: Decoupled interface and functional specification

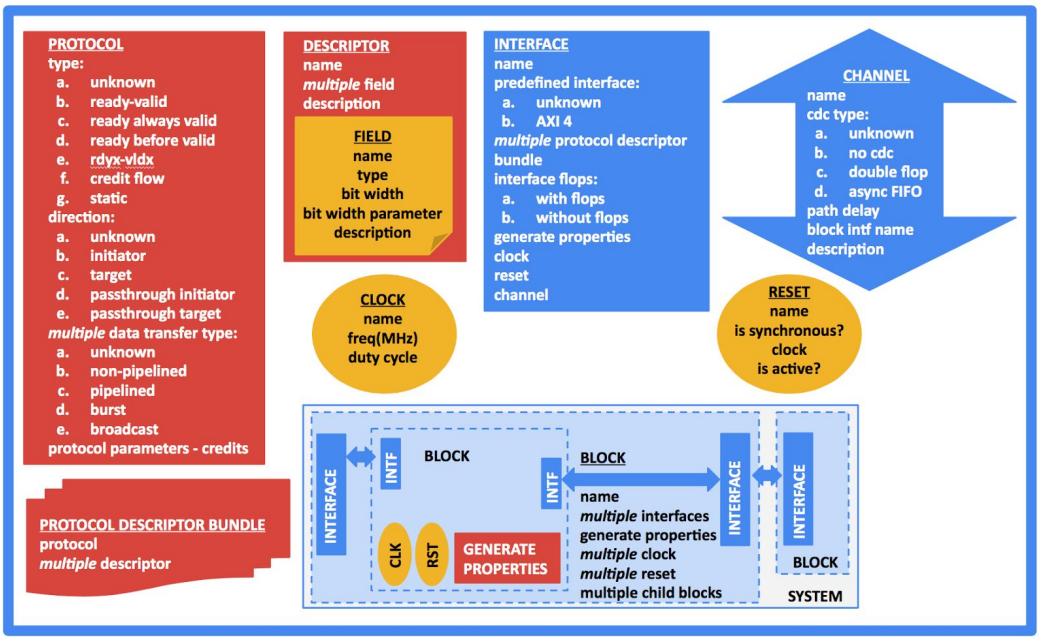

Figure 4: System level block diagram showing protocol buffer message types and hierarchy

We show the protocol buffers message types and hierarchy in Fig. 4 and the message members are described in Fig. 5. As the message name suggests, clock and reset define all the clocks and resets in the system. Field message is used to define each signal. Type of signal (logic, wire or some predefined structure), signal bit width or parameterized bit width can be provided. Multiple fields can be grouped under a descriptor. Descriptors are bundled with protocol in a protocol-descriptor bundle. When bundled, each descriptor is generated with separate flow control signals of type protocol. Protocol specification includes type of protocol (ready-valid, credit flow, etc.), direction of protocol (target, initiator, etc.), type of data transfer and protocol parameters (if any). An interface is a group of these protocol descriptor bundles along with the associated clock, reset and channel. Channel provides name of the block and interface which the interface it is instantiated under would connect to. One can also optionally define clock domain crossing method and path delay encountered in the channel. We can also have predefined interfaces. When these are used, default values for fields, descriptors and protocols associated with predefined interface are taken for generation of design and verification environment. Interfaces can be generated optionally with or without flops. Generate properties message can be used to select the properties to be generated for the interfaces or blocks.

```
//=============================================================
// Message Clock
// name            : Required field to identify clock uniquely
// freq_mhz        : Frequency of Clock in MHz
// duty_cycle      : Percentage of period for which clock is High
//=============================================================

//=============================================================
// Message Reset
// name            : Required field to identify reset uniquely
// is_sync         : Is reset synchronous with the clock?
// clk             : Clock Name reset is synchronous to
// is_active       : Is reset active high (true) or active low (false)
//=============================================================

//=============================================================
// Message Field
// name            : Required field to identify signal uniquely
// type            : Type of field, default is logic
// bit_width       : Signal BitWidth
// par_bit_width   : Signal BitWidth defined using parameter
// description     : Signal Description
//=============================================================

//=============================================================
// Message Descriptor
// name            : Required field to identify descriptor uniquely
// fields          : Group multiple signals under single descriptor
// description     : Descriptor Description
//=============================================================

//=============================================================
// Message ProtocolParameters
// credits         : Number of credits for the interface
//=============================================================

//=============================================================
// Message ProtocolSpec
// protocol_type        : Type of protocol
// protocol_direction   : Direction of protocol
// data_transfer_type   : Type of Data Transfer
// protocol_params      : Protocol specific parameters
//=============================================================

//=============================================================
// Message ProtDescBundle
// protocol        : Flow control protocol
// descriptors     : Multiple descriptors,
//                   each generated with separate flow control signals of type Protocol
//=============================================================

//=============================================================
// Message GenerateProperties
// sv_imp          : SystemVerilog Implementation,
//                   For now only SV_STRUCT is available.
//                   SV_INTERFACE can be added in future.
// rtl             : Set to generate RTL
// assert          : Set to generate Assertions
// cover           : Set to generate Coverage
// uvm             : Set to generate UVM Verification Environment
// doc             : Set to generate documentation
//
// effect of options rtl, uvm, doc are visible at block level and
// assert and cover at interface level
//=============================================================
```

```
//=============================================================
// Message Interface
// name                 : Required field to identify interface uniquely
// pre_defined_interface: Use a predefined interface
//                        Can manually override default values by defining
//                        data and address bus widths using descriptors,
//                        Protocol direction, etc.
//                        If no values are given, default settings are used
// prot_desc_bundles    : Define Protocol descriptor bundles
//                        Automatically defined if pre-defined interface used
// interface_flops      : Generate interfaces with/without flops
// gen_prop             : Properties to be generated for the interface
// clk                  : Provide Clk used by the interface,
//                        No separate signals generated
// rst                  : Provide Rst used by the interface,
//                        No separate signals generated
// chnl                 : Provide channels for block connections,
//                        Defined with child_blocks under BlockSpec
//=============================================================

//=============================================================
// Message Channel
// name            : Required field to identify channel uniquely
// cdc_type        : Clock Domain Crossing handling type
// path_delay      : Path Delay encountered in channel
// blk_intf_name   : Interface connected via this channel,
//                   follow syntax <BlockName_InterfaceName>
// description     : Channel Description
//=============================================================

//=============================================================
// Message BlockSpec
// name            : Required field to identify block uniquely
// interfaces      : Define all the interfaces used in the block
// gen_prop        : Properties to be generated for the block
// clk             : Provide Clk Names to generate ports
// rst             : Provide Rst Names to generate ports
// child_blocks    : Name all child blocks and define channel for each interface
//=============================================================


// Each of the following message types requires separate specification file

//=============================================================
// Message System
// blocks          : Define all blocks in the system
//                   For field Clk and Rst, just provide Name
//                   For field interfaces, provide Name and
//                   define interface.Clk Name and interface.Rst Name
// channels        : Channel connects the blocks through respective interfaces
//                   Only Channel Name is needed.
//=============================================================
//=============================================================
// Message Lists for better user experience. Avoids repetition.
// ClockList       : Define all the clocks in the system
// ResetList       : Define all the resets in the system,
//                   For field Clk, just provide Name
// DescriptorList  : Define all the descriptors and fields
// InterfaceList   : Define all the interfaces,
//                   Clock and Reset for the interface need
//                   to be given with block definition.
//                   For field descriptor, just provide Name
// ChannelList     : Define all the channels in the system
//=============================================================
```

Figure 5: Protocol buffer messages for interface specification. Message name is followed by message members and their description.

Block specification message defines the interfaces used by the block to communicate with other blocks or the outer world. Associated clocks, resets and child blocks are also described. At the system level, all the blocks in the complete system are listed. All the clocks, resets, descriptors, interfaces and channels are collectively listed in their own clock list, reset list, descriptor list, interface list and channel list respectively. These lists along with the system message specification file are used by the EASI2L described in the next section for generation of the required files.

IV. EASI2L

Now that our specification format for interfaces is designed, we need to read this structured data. Go language [2] is used with protocol buffers for taking input and generating the SystemVerilog design files and verification environment. Go provides efficient text template library and function maps, which are very useful for our application.

We name our interface generation and verification system as EASI2L, which is short for Easy Interface Implementation Tool. Four kinds of files used - protocol buffer files, go files, specification files and template files. On building protocol buffers, code library to read the protobuf input is automatically generated. Functions from this library are used in our Go code to read the interface specifications. Templates are then used to generate files based on specifications.

Two Go files are maintained, one is the library file (autointf.go) which contains all the functions and the other (autointf_generate.go) is front runner and is used for interaction with the user. Fig. 6 lists all the available functions in the library and Fig. 7 describes algorithm in autointf_generate. When the code is run, three optional inputs can be provided: output, specification and template directory paths. The specification directory should have system_spec.txt, channel_list.txt, interface_list.txt, descriptor_list.txt, clock_list.txt and reset_list.txt files to populate the "System" protocol buffer message properly.

autointf.go

function *ReadSpec*
ReadSpec function reads given specification files and fills SysSpec structure of type protobuf System

function *WriteSpec*
WriteSpec function reads given protobuf System structure and writes out specification file

function *IspecFuncMap*
IspecFuncMap function provides mapping from names to template functions

function *Generate*
Generates all the required files
- Create Required Directory Structure
- Generate descriptor Structs file
- Generate Irun components
- Generate Interface files
- Generate Block files
- Generate Channel files

function *GenerateDirs*
GenerateDirs creates required directory structure in output directory

function *GenerateIrun*
GenerateIrun creates components required for compilation and simulation - filelist, tcl file, irun command file

function *GenerateIntfFromTemplate*
GenerateIntfFromTemplate clubs redundant task of template reading, execution and writing to file and provides it as function. It operates on protobuf Interface message.

function *GenerateIntf*
GenerateIntf generates interface components for the given protobuf using templates at provided path. It uses GenerateIntfFromTemplate.
- Generate Interface UVM files for each interface only when UVM option true
- Generate Assertions and Coverage files only when respective options are true

function *GenerateBlockFromTemplate*
GenerateBlockFromTemplate clubs redundant task of template reading, execution and writing to file and provides it as function. It operates on BlockSpec message.

function *GenerateBlockUVM*
GenerateBlockUVM generates block components for the given protobuf using templates at provided path. It uses GenerateBlockFromTemplate.

function *GenerateBlockRTL*
GenerateBlockRTL generates block module and channel components for the given protobuf using templates at provided path. It uses GenerateBlockFromTemplate.

function *GenerateDesc*
GenerateDesc generates system verilog file for the descriptor according to the template file provided GenerateDescStruct uses the descriptor list file as input, while GenerateDesc uses populated System protobuf message as input.

function *GenerateDescStruct*
GenerateDescStruct function takes as input a descriptor file name, template filename and output path. It generates descriptor structs for use in design and verification environment and saves it in a "struct_defs_pkg.sv" text file.

Figure 6: List of available functions in EASI2L library

autointf_generate.go

```
Get flags and directory paths from user
Initialize file path variables
if onlyGenStructs:
        Generate and save descriptor structures on file
else:
        call function ReadSpec to read specification files
        call function Generate to generate required files
        call function WriteSpec to write combined system specifications
exit
```

Figure 7: Algorithm for autointf_generate

There is an optional fourth flag "onlyGenStructs" which gives user flexibility to generate the descriptor structs independently. If this flag is used, only descriptor_list.txt specification file needs to be available in specification directory and the remaining specification files are optional. Fig. 8 shows an example descriptor specification file, template file for generating descriptors and the output SystemVerilog descriptor structs.

```
Descriptors: {
    Name: "example_data"
    Fields: {
        Name: "data_1"
        BitWidth: 16
    }
    Fields: {
        Name: "data_2"
        Description: "single bit data line 2"
    }
    Fields: {
        Name: "data_3"
        ParBitWidth: "BIT_WIDTH"
        Description: "data line 3 of bit width defined by parameter BIT_WIDTH"
    }
}

Descriptors: {
    Name: "example_hier"
    Fields: {
        Name: "data_hier"
        Type: "example_data_t"
        Description: "data_hier of type example_data_t"
    }
    Description: "example of hierarchical data struct"
}
```

(a)

```
typedef struct packed {
    logic [15:0] data_1;
    logic data_2; // single bit data line 2
    logic [BIT_WIDTH-1:0] data_3; // data line 3 of bit width defined by parameter BIT_WIDTH
} example_data_t;

/* example of hierarchical data struct */
typedef struct packed {
    example_data_t data_hier; // data_hier of type example_data_t
} example_hier_t;
```

(c)

```
{{if .Description}}  /* {{.Description}} */{{end}}
 typedef struct packed {
{{range .Fields }}{{if .Type}}    {{.Type }} {{else}}    logic {{end}}{{if .BitWidth}}[{{GetWidth .BitWidth}}:0]
{{end}}{{if .ParBitWidth}}[{{.ParBitWidth}}-1:0] {{end}}{{.Name}};{{if .Description}} //
{{.Description}}{{end}}
{{end}}
 } {{.Name}}_t;
```

(b)

Figure 8: (a) Sample descriptor specification file, (b) Descriptor template file, (c) Output SystemVerilog descriptor structs based on specifications

Similar to Fig. 8b, there are template files for the rest of the UVM based verification environment that includes UVM Agents - sequencers, drivers and monitors, example sequence library, example test library, virtual sequencer, assertions, coverage, TLM analysis ports and UVM environment. Assertions and coverage points are also included in templates. Testbench top and files needed for elaboration and simulation also have their templates.

On the RTL side, block design top file is created for each of the blocks and sub-blocks using templates. These files contain input-output signal ports and any necessary protocol handling logic such as sending credits in a slave credit flow interface and maintaining the credit count in master credit flow interface. They also contain logic to connect the interfaces according to provided specifications (Fig. 9). Fig. 10 shows the list of template files used by the tool.

Declare module “block_top”. This is a wrapper module and instantiates child blocks besides handling signal connections.

module <block name>_top (
input wire <interface name>_<descriptor name>_<signal type>,
output logic <interface name>_<descriptor name>_<signal type>,
...
);

For each child block, create dummy signals **when interface is of type Initiator or Passthrough Initiator**. These signals will be later used to connect to signals in containing block and child blocks.

logic <child block name>_<interface name>_<descriptor name>_<signal type>;
...

For signals in current block, create dummy signals and make appropriate connections.

if ProtocolDirection equals “Passthrough Target”:
logic *<block name>_<interface name>_<descriptor name>_<signal type>;*
then depending upon underlying signal direction:
assign <block name>_<interface name>_<descriptor name>_<signal type> = <interface name>_<descriptor name>_<signal type>; ***or***
assign <interface name>_<descriptor name>_<signal type> = <block name>_<interface name>_<descriptor name>_<signal type>;
...

if ProtocolDirection equals “Passthrough Initiator”, then depending upon underlying signal direction:
assign <message channel, blk_intf_name>_<interface name>_<descriptor name>_<signal type> = <interface name>_<descriptor name>_<signal type>; ***or***
assign <interface name>_<descriptor name>_<signal type> = <message channel, blk_intf_name>_<interface name>_<descriptor name>_<signal type>;
...

Instantiate module “block”. This module contains relevant RTL logic for data exchange over interface. Module exists only when at least one of the interface is active. When block has all interfaces of passthrough type, it only acts as a wrapper to make relevant connections between child blocks.

<block name> <block name>_inst (
if ProtocolDirection equals “Initiator” or “Target”:
.<interface name>_<descriptor name>_<signal type> (<interface name>_<descriptor name>_<signal type>),
...
);

For each child block, instantiate child block top. When dealing with slave interface, i.e. of type Target or Passthrough Target, member “blk_intf_name” of message “channel” provides the master block-interface-name it connects to.

<child block name>_top <child block name>_top_inst (
if ProtocolDirection equals “Initiator” or “Passthrough Initiator”:
.<interface name>_<descriptor name>_<signal type> (<child block name>_<interface name>_<descriptor name>_<signal type>),
...
if ProtocolDirection equals “Target” or “Passthrough Target”:
.<interface name>_<descriptor name>_<signal type> (<message channel, blk_intf_name>_<descriptor name>_<signal type>),
...
);

Figure 9: Algorithm for creating signal connections between interfaces automatically

rtl *(RTL directory)*

block_top.sv.template	- contains module instantiation and signal connections
block.sv.template	- contains basic interface protocol logic

sim *(UVM based Verification templates directory)*

intf_agent *(Agent directory, UVM based templates for interfaces)*

- intf_pkg.sv.template
- intf.sv.template
- intf_config.sv.template
- intf_seqitem.sv.template
- intf_master_driver.sv.template
- intf_slave_drive.sv.template
- intf_assertions.sv.template
- intf_monitor.sv.template
- intf_sequencer.sv.template
- intf_cov_collector.sv.template
- intf_agent.sv.template

- block_pkg.sv.template
- block_virt_sequencer.sv.template
- block_env.sv.template
- block_base_seq.sv.template
- block_base_test.sv.template
- block_seq_lib.sv.template
- block_test_lib.sv.template

block_tb.sv.template	- block testbench top file
descriptor_struct.sv.template	- SystemVerilog descriptor structs
block_tb.tcl.template	- tcl template file used with Cadence's Irun flow
block.f.template	- file list template file used with Cadence's Irun flow
irun.sh.template	- Cadence's Irun flow for elaboration & simulation

Figure 10: Template files used by EASI2L

Once the tool runs successfully, elaboration and simulation can be done using a standard simulator. All the required files are generated along with an example command to run and visualize waveforms for a sample testcase. Output of EASI2L is pictorially shown in Fig. 11. We discuss our complete interface implementation system in the next section using an example.

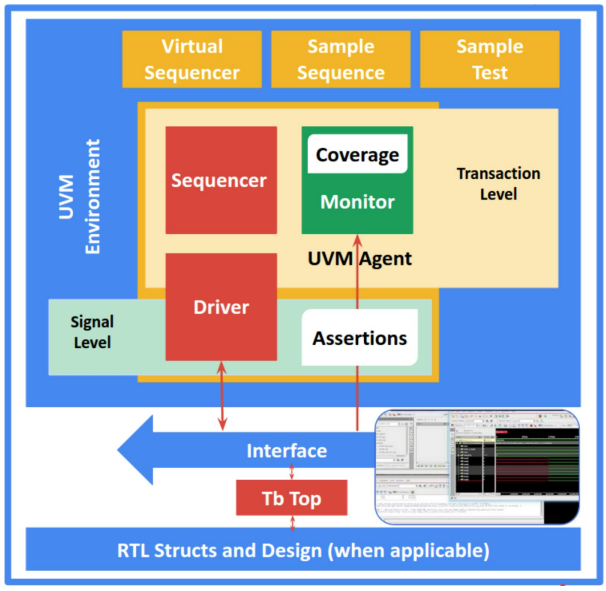

Figure 11: Output of EASI2L

V. EXAMPLES

In this section we demonstrate the functionality of our specification format and EASI2L using an example. We consider 5 blocks in our example system - *block1, block3, block31, block32* and *block33*. *Block31, block32* and *block33* are child blocks of *block3*. Pictorially, the hierarchy is shown in Fig. 12.

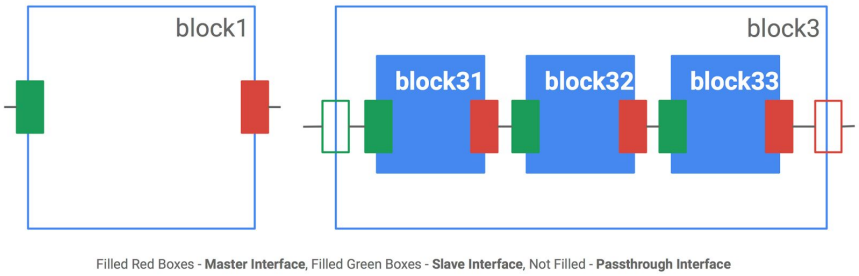

Figure 12: Example System

We first start by describing all the clocks used in the complete system. Next, we define all the resets in the system. After this, we list all the descriptors. This is followed by the interface list, channel list and system specification.

As shown in Fig. 12, *block1*, *block31*, *block32*, and *block33* contain one master and one slave interface each. *Block3* doesn't contain any active interfaces and consists only of passthrough master and passthrough slave interfaces.

The interfaces are defined in the interface list. Only names of the descriptors are enough under protocol descriptor bundles message. For generate properties messages, only cover and assert are valid options at the interface level. Other options hold value only at the block level. For coverage files and assertions, at least UVM generation should be selected at the block level. Clock, reset and channel will be added to the interface when we are defining the blocks.

Next, we need to fill in the system specification. This file contains all our blocks. We list the name of the blocks, all the clocks and resets associated with the block (only the names are sufficient), all the child blocks (if any, again only names of the child blocks are sufficient). The interfaces which are part of each block are mentioned. Here, we provide the clock and reset information that we skipped in the interface list file. Channel names are also given for slave and passthrough master interfaces. Detailed channel definitions are provided in the channel list file. Blk_intf_name in the channel definitions specify the block and interface to which the current interface is getting connected. It should follow the exact (block name)_(interface name) format. The properties that need to be generated can be specified using generate properties message. SystemVerilog structs are used in RTL logic and SystemVerilog interfaces are used on the verification side. This was a choice we made to avoid some issues previously encountered using the interface construct in RTL design code.

This completes our system specification part. Next, we will see how these files are used by the tool to generate respective files.

The Go script, `autointf.go`, provides a library of functions to generate the complete design and verification environment. First of all, the specification files are read and compiled into one complete System protobuf using ReadSpec function.

Then depending upon which properties are to be generated, directories are created by calling GenerateDirs function. Each block gets its own directory under the output directory. RTL design is under rtl/ sub-directory and verification environment under dv/ sub-directory. The dv directory also contains files to be used with standard simulator. Sub-directory sv/ under dv/ contains the top level testbench file, block level UVM environment, base sequence and base test case. Sequence library and test library are available under tests/ directory in dv/. The agent i.e. interface environment, along with assertions and coverage collector exists under agent/ directory in dv/sv/ directory. Separate driver, monitor and sequencer are generated for each descriptor under an interface. The credits and data can be generated back to back or with delays. This is randomly chosen in the example testcase.

On the RTL side, the *block.sv* file contains the protocol handler logic in case of master or slave interface. The credits are always assumed to be free in slave interface and valid data is always available in case of master interface. The designer would update this into his own control logic. If there are only passthrough interfaces, block.sv file is empty. The block_top.sv module instantiates the block module along with any child modules. It acts as a wrapper file and contains logic that makes the necessary connections between different blocks. Interfaces can optionally be generated with flops. This helps in timing closure at the floor plan boundary.

In the end, the complete system specification protobuf is written to SYS_Spec.txt file under output directory for reference. It can be slightly difficult to write system specification files for the first time. But once the flow is

understood, the tool greatly reduces the time needed to specify interfaces and is able to generate ready to run, complex environments in seconds.

VI. Related Work

A flow to automatically synthesize bus-based architectures is described in [3], [4] and [5]. In [3], architectural parameters are first extracted from design decisions by the designer or automatic tools. These parameters are used to customize architecture templates for processors, memory modules and communication networks. A microarchitecture is generated using processor and protocol library form macroarchitecture. The communication protocol (e.g. FIFO, handshake, etc.), parameters (e.g. FIFO size), protocol-specific functions (e.g. fifo-available, fifo-write, etc.) are given in macro-architecture. Communication is represented at cycle accurate level at the microarchitecture level.

In [4], a bus architecture synthesis approach is proposed to automate generation of cost effective standard bus-based communication architecture for SoCs while satisfying performance constraints and detecting bus cycle time violations. The approach synthesizes appropriate bus topology automatically and gives values for bus architecture parameters such as arbitration priority orderings, data bus widths, bus clock speeds and DMA burst sizes. These parameters are provided manually by the system architect in our approach.

Similar to [3], paper [5] describes a flow to generate custom bus systems using pre-designed reusable hardware modules. A module library, containing all modules supported for use inside a BAN, bus subsystem or bus system, and wire library, containing different wires to connect modules, are used.

Paper [6] very nicely describes the need for an automated interface generation. Even a simple block such as FIFO can have subtle but complex differences in intended behavioral properties between designs by different designers which can lead to confusions and error. A focussed verification effort at the IP block level ensures correctness only under proper use. The authors describe rule-based interface generation as a solution.

Paper [7] describes reusable register design and verification methodology. Registers are first described in high level language, like SystemRDL which is used by tools to generate RTL, documentation, VMM based register verification environment, and the Register Abstraction Layer-RAL. Such flow enables design and verification teams to work more efficiently with consistent and synchronized view of chip design.

For our approach, we used template based libraries similar to ones described in [3] and [5] to generate standard interface RTL for low-level point to point interfaces. Assertions are automatically generated to ensure standard behavior of the interfaces. This saves a lot of time spent on adding and checking assertions manually. High level verification environment around interfaces is automatically generated as done for automated Register design and verification in [7].

VII. Conclusions and Future Work

A new tool, called EASI2L, was designed and developed for easy interface implementation. Although the RTL design of the interfaces that are targeted with this tool takes only a day or two, the major benefit comes from standardization of the interfaces used across teams. A high level verification environment around interfaces is generated. There are numerous benefits (Fig. 13). Using this reusable methodology, IP designers and verification engineers don't have to write code for their interface protocols from scratch every time. The implementation is quick and consistent. It is easier to understand the standard interfaces used in other designs, and verification procedure around the interface is not repeated. The functional part of the IP can be reused more easily, as it is independent from the interface through which it communicates with other blocks.

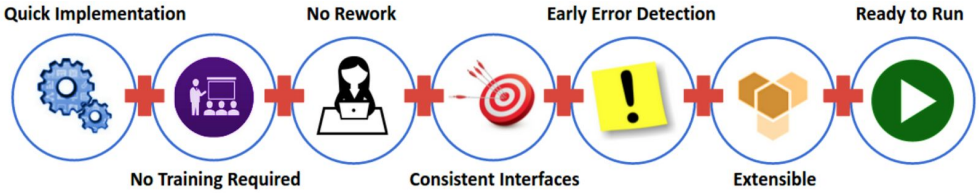

Figure 13: Benefits of EASI2L

Currently, the tool only supports credit flow type protocol but other protocols can be added on similar lines easily. Also, it would be useful to support pre-defined complex interfaces like AXI. The idea is that pre-defined interfaces are made up of underlying interface primitives. Once a user mentions a pre-defined interface, the tool

would use defined protocol descriptor values instead of looking for them in the specification files. Support for multiple data transfer types - Pipelines, Burst, Broadcast - might be useful when complex interfaces need to be generated using the tool. Automatic addition of bridge between the master and slave of two different protocols can also be supported in future versions. For example, when there is rdy_vld at one side and credit_flow at another, rdy_vdy to credit_flow block or vice versa can be instantiated in between. This can be regarded as another child block. Since we already have information about the clocks at the two ends, support for path delay and clock domain crossing under channels can be added. We can also support generation of code for dual reset using the available reset information. Another important yet easy to implement step would be to generate concise and complete documentation automatically. Our proposed interface specification format already supports all the aforementioned additions. We only need to update the EASI2L. Because the framework is ready, adding implementation templates for the required features should be a relatively easy task.

REFERENCES

[1] Protocol Buffers, http://code.google.com/apis/protocolbuffers/ (Date Last Accessed, January 08, 2016)

[2] Go language, https://golang.org/ref/spec (Date Last Accessed, January 08, 2016)

[3] Lyonnard, D.; Sungjoo Yoo; Baghdadi, A.; Jerraya, A.A., "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip," Design Automation Conference, 2001. Proceedings, vol., no., pp.518,523, 2001, doi: 10.1109/DAC.2001.156194

[4] Kyeong Keol Ryu; Mooney, V.J., "Automated bus generation for multiprocessor SoC design," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol.23, no.11, pp.1531,1549, Nov. 2004, doi: 10.1109/TCAD.2004.835119

[5] Pasricha, S.; Dutt, N.; Bozorgzadeh, E.; Ben-Romdhane, M., "Floorplan-aware automated synthesis of bus-based communication architectures," Design Automation Conference, 2005. Proceedings. 42nd , vol., no., pp.565,570, 13-17 June 2005, doi: 10.1109/DAC.2005.193874

[6] Rishiyur S. Nikhil, "Eliminating Verification Using Automated Formal Interface Contracts," (2006), http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.127.851&rep=rep1&type=pdf (Date Last Accessed, January 27, 2016)

[7] Ballori Banerjee; Subashini Rajan; Silpa Naidu, "Automated approach to Register Design and Verification of complex SOC," Design and Verification - Conference and Exhibition (DVCon), 2011