A generic IP-XACT generator for configurable Intellectual Property

By

Hemant Kashyap

09MCE007

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

May 2011

A generic IP-XACT generator for configurable Intellectual Property

Major Project

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science and Engineering

By

Hemant Kashyap

09MCE007

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

May 2011

Declaration

This is to certify that

i) The thesis comprises my original work towards the degree of Master of Technology in Computer Science and Engineering at Nirma University and has not been submitted elsewhere for a degree.

ii) Due acknowledgement has been made in the text to all other material used.

Hemant kashyap

Certificate

This is to certify that the Major Project entitled " A generic IP-XACT generator for configurable Intellectual Property" submitted by Hemant Kashyap (09MCE007), towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Mr. Mukesh Chopra
Guide,
Manager FDDF,
Technology Research and Development,
ST Microelectronics, Greater Noida

Dr. S.N. Pradhan
Internal Guide and Professor,
Department Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr K Kotecha
Director,
Institute of Technology,
Nirma University, Ahmedabad,

Prof. D. J. Patel
Professor and Head,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Abstract

As chip design became more and more complex and concept of reusable hardware design (Intellectual property) evolved. IP-XACT IEEE P 1685 standard was developed for integrating and reusing Intellectual property within tool flows. IP-XACT IEEE P1685 is a xml based standard for capturing metadata of a design (HDL code).It only captures the metadata that is relevant to integration not the underlying functionality of the design. Assembly tools which are based on IP-XACT read these metadata to integrate IP's. Hardware definition languages can contain various design configurations in a single design based on macro definition .These macros values define the actual the configuration of design that is actually synthesized.

IP-XACT currently has this limitation that it can only capture one hardware configuration at a time. So for a configurable design designer has to manually create different metadata for different configuration manually. In this report we present a methodology to extend the standard to support configurable Intellectual Property.

Proposed methodology is such that it can be integrated into the existing assembly tools without making additional changes to the tool. The proposed methodology is assembly tool independent, which is acheaved by making communication between assembly tool and configuring mechanism standardized by using TGI API's and SOAP protocol. Methodology provides a mechanism to append configurability information into the metadata and a mechanism to generate a particular configuration.

This report presents an implementation for the methodology and a deployment model. Integration and use model with different assembly tool is presented.

Acknowledgements

I am deeply indebted to my thesis supervisor Mr Mukesh Chopra for his constant guidance and motivation. He has devoted significant amount of his valuable time to plan and discuss the thesis work. Without his experience and insights, it would have been very difficult to do quality work.

I would also like to extend my gratitude to my Internal Guide Prof. S.N Pradhan for his guidance and fruitful suggestions.

Here I would also like to thank my teammates at ST microelectronics espeically Bhawna chopra and Sparsh Arun for their support and help.

Last, but not the least, no words are enough to acknowledge constant support and sacrifices of my family members because of whom I am able to complete the degree program successfully.

- Hemant Kashyap
09MCE007

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This work aims at providing a methodology that extends IEEE P1685 standard [5] for IP-XACT so that configurable Intellectual Property which is a design source code can be handled within design flows. It has been a limitation of the standard that it cannot handle designs which are configurable (contains macros).Configurable designs contain multiple design configuration in a single source. The work in this thesis presents a methodology that is design tool independent and standardized so that it can be used in any IP-XACT based assembly tool flow.

1.1 IP-XACT based assembly

IP-XACT or IEEE P1685 is a standard for capturing meta data related to a HDL code that can then be used for integration in a given tool flow. IP-XACT based assembly is based first capturing the metadata related to design block into xml [11] description then using that description for integration into IP-XACT based design.

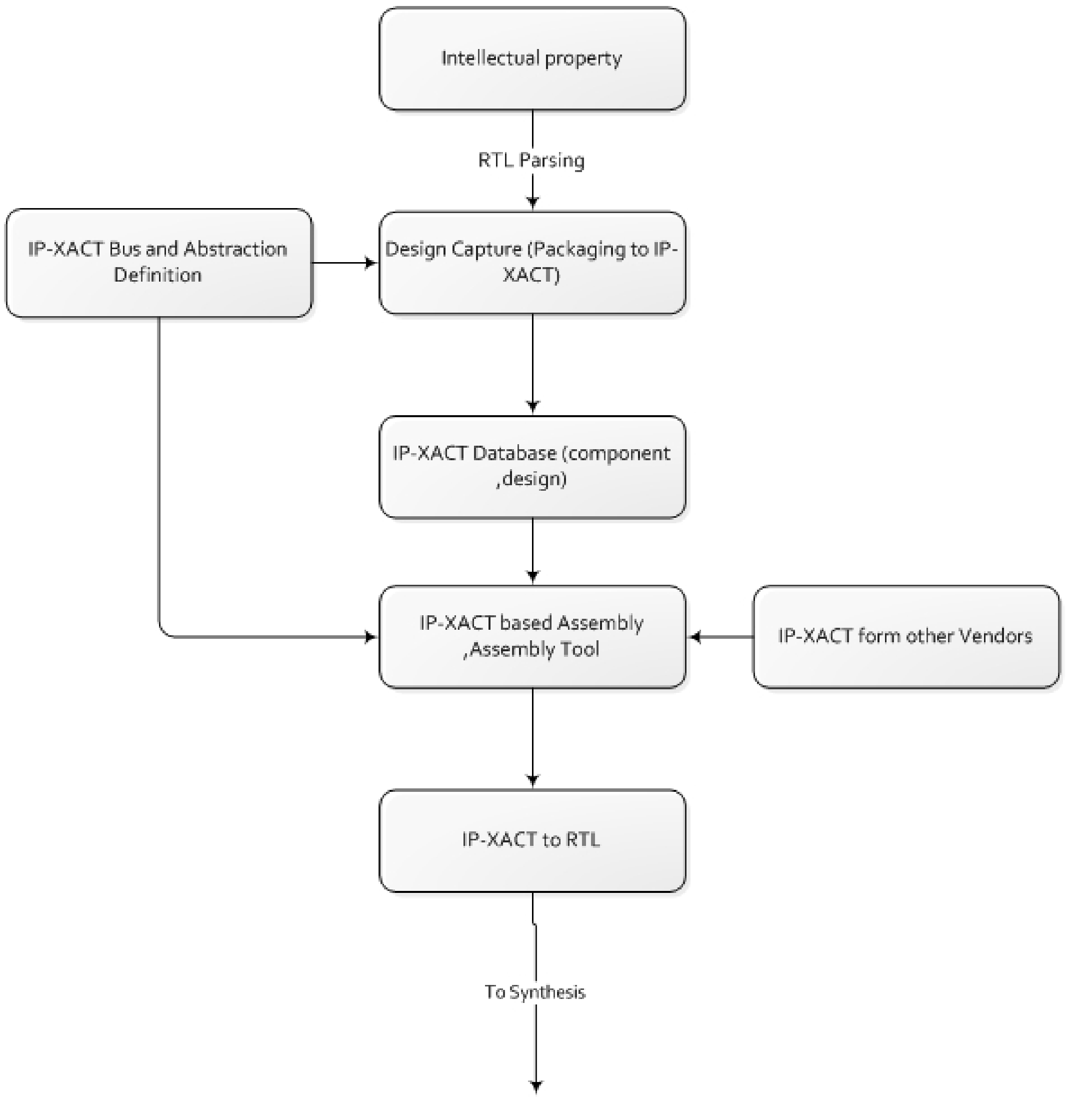

Intellectual property
RTL Parsing
IP-XACT Bus and Abstraction Definition
Design Capture (Packaging to IP-XACT)
IP-XACT Database (component ,design)
IP-XACT based Assembly ,Assembly Tool
IP-XACT form other Vendors
IP-XACT to RTL
To Synthesis

Figure 1.1: IP-XACT Assembly flow

Figure 1.1 shows the IP-XACT based assembly flow.Through metadata it becomes easy to integrate, reuse Intellectual property into tool flows .Tools working on integration of design block are independent of the actual design code written. IP-XACT also allows for creating entities like bus interfaces that allow connections during integration to be specified in a group thus reducing the total number of connections.So IP-XACT acctuly standardize the representation of information of a design which is relevent to integration .

1.2 Motivation

IP-XACT based packaging and Assembly is getting more and more attention from the designers. The standard uses xml based metadata capturing which can then be used for integration purpose with tool flows hence reducing designer time. Also as the standard defines proper syntax and semantic for metadata capture errors during assembly (connecting different modules) are also reduced.

As IP-XACT is widely used it was always required to enhance the standard to meet the designers requirement .One such requirement is to handle configurable Intellectual property designs and a solution to generate a particular configuration for the intellectual property. Designers use different assembly tools from different vendors so the solution must be standard for all different tools.

1.3 Scope of Thesis

Objective of this thesis is to provide a methodology

a. To handle configurable Intellectual Properties in IP-XACT based assembly flows.

b. Which is assembly tool Independent.

c. Is IEEE P1685 standard compliant .

Work of this thesis aims at overcoming a limitation of IP-XACT standard and to enhance the standard to provide designer with a complete IP-XACT based assembly flow.

1.4 Thesis Organization

This section describes as to how thesis is organized as

Chapter 1, *Introduction* This chapter provides an background of IP-XACT asssembly flow .It also provides motivation and scope of this thesis.

Chapter 2, *Literature Survey* Chapter provides study of IP-XACT standard and related terminology.It also discuss configurability in HDL.

Chapter 3, *Problem Definition* Elaborates problem definition and details about existing methadology for handling configurability in intellectual property.

Chapter 4, *Proposed Methadology* This chapter details about the proposed methadology and use model.It also discuss about the new syntax for configurability as well as generation methodology.

Chapter 5, *Implemantation* Here details of implementation of above proposed methadology is presented.Tools used in implementation are also discussed .Finally chapter provides results and outputs of the use model with different assembly tools.

Chapter 6, *Conclusion and future scope* Finally this chapter presents conclusion and future scope for the proposed solution.

Chapter 2

Literature Survey

2.1 Design flow and HDL

In electronics, a hardware description language or HDL is any language from a class of computer languages, specification languages, or modeling languages for formal description and design of electronic circuits, and most-commonly, digital logic. It can describe the circuit's operation, its design and organization, and tests to verify its operation by means of simulation.[9]

HDL provides text based description of the functional and temporal behavior of the hardware circuit. Synthesizer is a tool which converts the given HDL code to netlist (which is interconnection of generic hardware primitives).

HDL is analogues to a software programming language but is very different. A HDL has ability to handle many parallel processes executing simultaneously such as flip-flop and adders. It can also represent timing delay information which is not possible in a software programming language. Software programs are compiled for a given microprocessor architecture where as HDL code is synthesized.HDL cannot be used for software programming similarly software languages cannot be used for modeling hardware.

However in terms of syntax they may appear same. Both types of languages can support conditional compilation (in terms of HDL it is synthesis).Both contains code blocks and many other syntax like if else etc.

In recent times as electronic systems grow increasingly complex, and reconfigurable systems become increasingly mainstream, there is growing desire in the industry for a single language that can perform some tasks of both hardware design and software programming. SystemC is an example of such-embedded system hardware can be modeled as non-detailed architectural blocks (blackboxes with modeled signal inputs and output drivers). The target application is written in C/C++, and natively compiled for the host-development system (as opposed to targeting the embedded CPU, which requires host-simulation of the embedded CPU). The high level of abstraction of SystemC models is well suited to early architecture exploration, as architectural modifications can be easily evaluated with little concern for signal-level implementation issues.

2.1.1 HDL Languages

Many HDL languages have been designed for digital design few of them are listed in table I

Language	Description
VHDL	Widely used language
Verilog	Widely used language
Handle-C	It is a rich subset of C, with non-standard extensions to control hardware instantiation with an emphasis on parallelism.
SystemC	A standardized class of C++ libraries for high-level behavioral and transaction modeling of digital hardware at a high level of abstraction, i.e. system-level
SystemVerilog	A superset of Verilog, with enhancements to address system-level design and verification

Table I: HDL Languages

From the above given HDL languages Verilog and VHDL are most widely used in the industry other languages like SystemC are currently evolving.

2.2 HDL based assembly and Integration

Generally in HDL blocks are written which has a defined periphery (Ports).These blocks are known as leaf level blocks if they do not contain other blocks .These leaf level blocks are then assembled together to form a complete design. Leaf level blocks may contain primitive operations for the design. Two types of design methodology are used [9].

2.2.1 TopDown

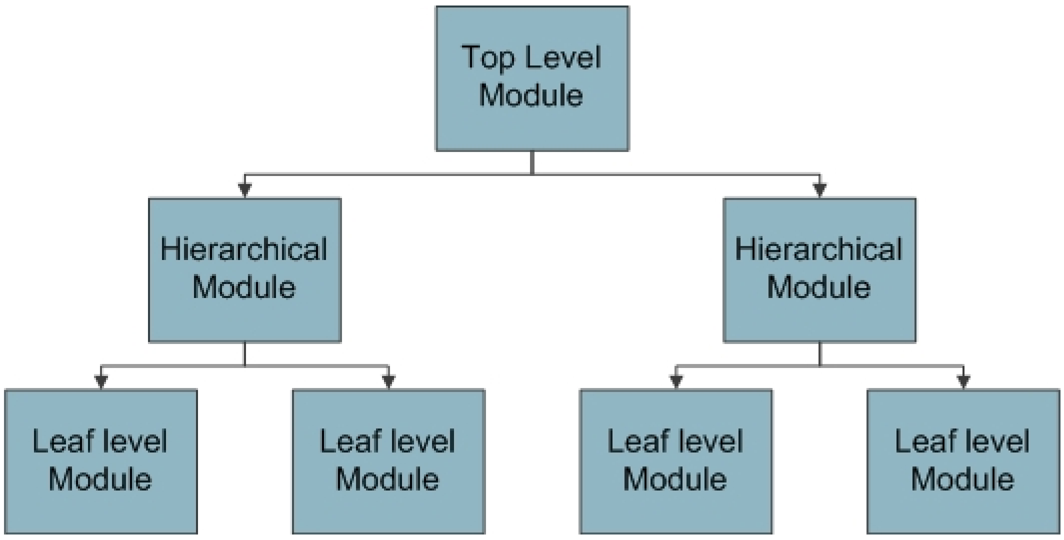


Figure 2.1: Top down Design

Here top level module specification are made first and then top module is broken into low level modules based on functionality then in this way at last leaf level blocks are designed.

2.2.2 Bottom up

This is reverse of Top Down approach. Other mixed strategy is also used where top level specifications are made and by design architects. Whereas logic designers break top level module in small blocks of functionality .Meanwhile circuit designers design

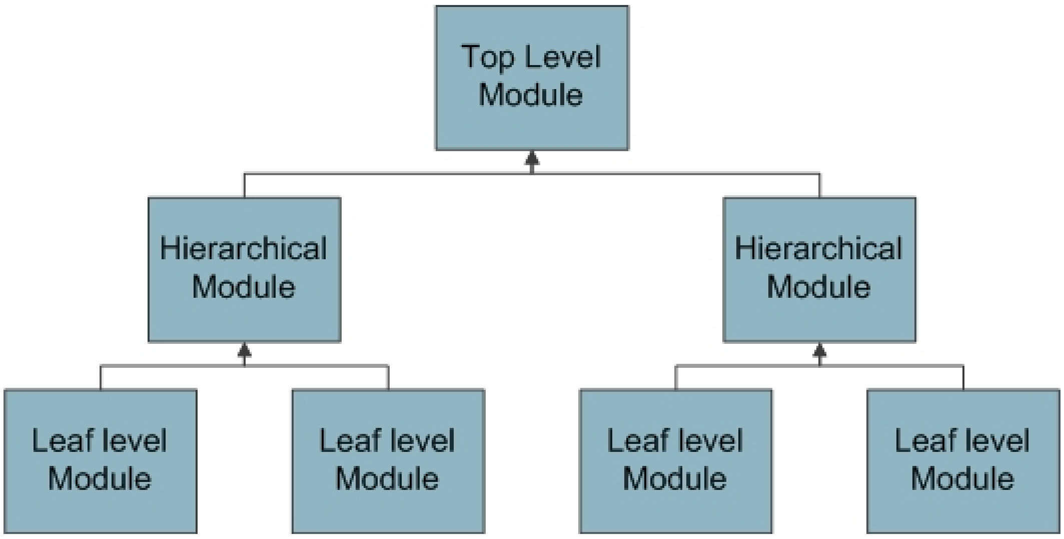


Figure 2.2: Bottom up design

leaf level modules and lower hierarchical blocks they both the flows meet at a certain point.

2.2.3 HDL Reuse

As design became more and more complex and concept of SOC (System on chip evolved) where a chip can contain a microcontroller, memory and timers etc. This makes design and interconnection between components very complex. Also designers started reusing the components in chips like microcontroller or memory. This made design process assembly and integration of given modules or IP's (Intellectual Property).

2.3 IP-XACT standard

IEEE P1685 standard for IP-XACT an extensible Markup Language (XML) schema1 for meta-data documenting intellectual property (IP) used in the development, implementation, and verification of electronic systems and an application programming interface (API) to provide tool access to the meta-data.This schema provides a standard method to document IP that is compatible with automated integration techniques. The API provides a standard method for linking tools into a system development framework, enabling a more flexible, optimized development environment. Tools compliant with this standard will be able to interpret, configure, integrate, and manipulate IP blocks that comply with the IP meta-data description. The standard is independent of any specific design processes. It does not cover those behavioral characteristics of the IP that are not relevant to integration.

2.3.1 IP-XACT Schema

IEEE P1685 defines xml schema to capture HDL metadata .Xml uses xml 1.0 standard and use of xpath [12] expression is allowed for value dependency expression. Along with the standard xpath function other functions are defined like spirit: number, spirit: translate etc ,however standard does not define implementation of these functions.

2.3.2 IP-XACT Objects and Interection

In IP-XACT standard following objects [5] are defined as

a. A bus definition description defines the type attributes of a bus like if it is addressable ,maximum slaves that can connect etc.

b. An abstraction definition description defines the representation attributes of a bus,like ports and direction in master slave .

c. A component description defines an IP or interconnect structure ex a microcontroller.Please refer Appendix A for more information on IP-XACT component.

d. A design description defines the configuration of and interconnection between components like between microcontroller and memory.

e. An abstractor description defines an adaptor between interfaces of two different abstractions.

f. A generator chain description defines the grouping and ordering of generators.

g. A design configuration description defines additional configuration information for a generator chain or design description.

IP-XACT objects can interact as shown in the figure 2.3

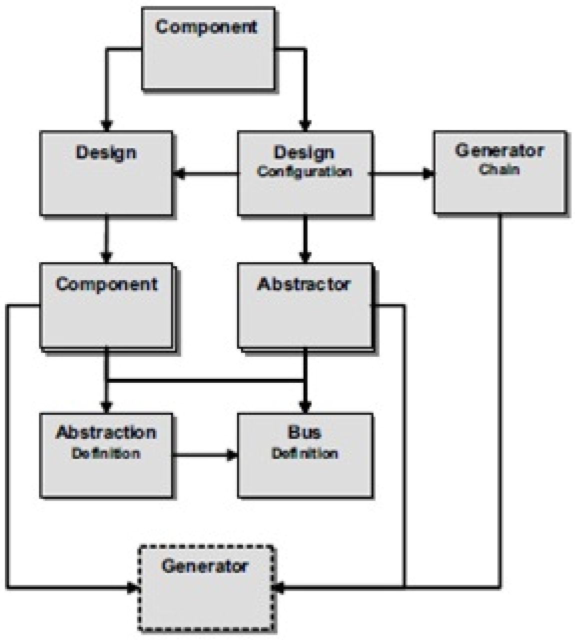


Figure 2.3: IP-XACT object Interection

Every IP-XACT object is identified using four fields VLNV (Vendor ,library ,name version) ,these values are mentioned in every object

2.3.3 Generators and corresponding API's specification.

Generators are executable objects (e.g., scripts or binary programs) that may be integrated within a DE (referred to as internal) or provided separately as an executable (referred to as external). Generators may be provided as part of an IP package (e.g., for configurable IP, such as a bus-matrix generator) or as a way of wrapping point tools for interaction with a DE (e.g., an external design netlister, external design checker). An internal generator may perform a wide variety of tasks and may access IP-XACT compliant meta-data by any method a DE supports. IP-XACT does not describe these protocols. An external generator (often referred to as a TGI generator) is an executable program or script invoked from within a DE to query or configure design descriptions and their related component and abstractor descriptions. External generators can use the TGI to access their IP-XACT meta-data descriptions (as currently loaded into the DE) and perform the various operations associated with those descriptions. In addition, external generators shall only operate upon IP-XACT compliant meta-data through the defined TGI. Generators can be referenced from a component, abstractor, or generator chain description. Generators can also be grouped and ordered in generator chain descriptions and those chain descriptions contained inside other chain descriptions. This sequencing of generators is critical for providing script-based support for SoC flow creation. Description of implementation and usage of Generator is not defined in the standard.

For more informatation please refer Appendix B.

2.3.4 TGI API

The tight generator interface (TGI) is the method a generator uses to efficiently access a design or component description in a DE-independent and generator-language-independent manner. Therefore, a generator running on two different DEs produces the same results. TGI method signature is defined by the standard but it is to the Design Environment to implement the API's and use model.
Ex getComponentVLNV gets the component object VLNV currently in context.

For more information please refer Appendix B.

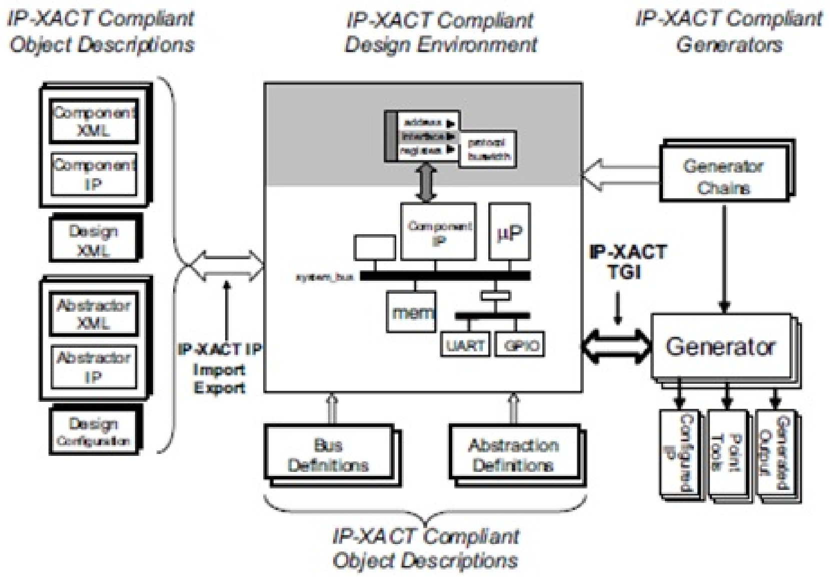


Figure 2.4: IP-XACT Design Enviornment

2.3.5 Design Enviornment

A DE enables the designer to work with IP-XACT design IP through a coordinated front-end and IP design database. These tools create and manage the top-level meta-description of system design and may provide two basic types of services: design capture, which is the expression of design configuration by the IP provider and design intent by the IP user; and design build, which is the creation of a design (or design model) to those intentions.

As part of design capture, a system design tool shall recognize the structure and configuration options of imported IP. In the case of structure, this implies both the structure of the design [e.g., how specific pin-outs refer to lines in the hardware description language (HDL) code] as well as the structure of the IP package (e.g., where design descriptions and related generators are provided in the packaged IP data-structure). In the case of configuration, this is the set of options for handling the imported IP (e.g., setting the base address and offset, bus width) that may be expressed as configurable parameters in the IP-XACT meta-data. As part of design build, generators may be provided internally by a system design tool to achieve the required IP integration or configuration, or provided externally (e.g., by an IP provider) and launched by the system design tool as appropriate.

A Design Environment is called P1685 compliant if it supports import Export of IP's in IEEE P1685 format. Also it supports TGI API's for interfacing with external generators.

2.4 Configurable HDL Design

As discussed earlier a HDL language like a software programming language can contain conditional compilation statements [9].

For ex in C language we have compiler directives "#ifdef" and "#endIf" for conditional compilation based on some macro value defined as "#define". Ex

```
#ifdef MACRO
Controlled text
#endif
```

Here "controlled text" is only compiled when MACRO is defined else not .These compiler directives decide the actual structure of code which will be compiled after preprocessing of macros.

Similarly in HDL languages these conditional compilation statements represent which provides a way for conditional synthesizing. Here this report will take example of Verilog and try and understand kind of conditional compilation statements.

Verilog supports macros as "'define ¡name¿ ¡value¿" and then these defined values can be used in any the HDL program.

Conditional compilation can be achieved as by using using following Syntax in verilog.

The 'ifdef, 'else, 'elsif, and 'endif compiler directives work together in the following manner:

a. When an ‘ifdef is encountered, the ifdef text macro identifier is tested to see if it is defined as a text macro name using ‘define within the Verilog HDL source description.

b. If the ifdef text macro identifier is defined, the ifdef group of lines is compiled as part of the description and if there are ‘else or ‘elsif compiler directives, these compiler directives and corresponding groups of lines are ignored.

c. If the ifdef text macro identifier has not been defined, the ifdef group of lines is ignored.

    (1) If there is an ‘elsif compiler directive, the elsif text macro identifier is tested to see if it is defined as a text macro name using ‘define within the Verilog HDL source description.

    (2) If the elsif text macro identifier is defined, the elsif group of lines is compiled as part of the description and if there are other ‘elsif or ‘else compiler directives, the other ‘elsif or ‘else directives and corresponding groups of lines are ignored.

    (3) If the first elsif text macro identifier has not been defined, the first elsif group of lines is ignored.

    (4) If there are multiple ‘elsif compiler directives, they are evaluated like the first ‘elsif compiler directive in the order they are written in the Verilog HDL source description.

    (5) If there is an ‘else compiler directive, the else group of lines is compiled as part of the description.

And other set of syntax are
The ‘ifndef, ‘else, ‘elsif, and ‘endif compiler directives work together in the following manner:

a. When an ‘ifndef is encountered, the ifndef text macro identifier is tested to see if it is defined as a text macro name using ‘define within the Verilog HDL source description.

b. If the ifndef text macro identifier is not defined, the ifndef group of lines is compiled as part of the description and if there are ‘else or ‘elsif compiler directives, these compiler directives and corresponding groups of lines are ignored.

These compiler Directives are useful when

a. Selecting different representations of a module such as behavioral, structural, or switch level.

b. Choosing different timing or structural information

c. Selecting different stimulus for a given run

d. Selecting if some port, register is defined.

For example in verilog [9]

The example below shows a simple usage of an ‘ifdef directive for conditional compilation. If the identifier behavioral is defined, a continuous net assignment will be compiled in; otherwise, an and gate will be instantiated.

```
module and_op (a, b, c);
output a;
input b, c;
‘ifdef behavioral
wire a = b & c;
‘else
and a1 (a,b,c);
‘endif
endmodule
```

Using these directives different configuration of a single HDL code can be defined

2.5 Configurability in IP-XACT

IP-XACT currently has this limitation that it cannot express the macro related configurability in the IP-XACT object and hence it can only capture a single configuration of HDL code. It means that for a particular configuration user needs to manually create metadata description related to each configuration required and then use it in assembly flow.

Chapter 3

Problem Definition

This chapter presents the problem definition and existing methadology

3.1 Problem Definition

IP-XACT currently has this limitation that it cannot capture metadata related to a design which is configurable (contains many configurations based on macro definitions).These designs do occur frequently as they are designed to be reused with different configurations. So a methodology is required which is IP-XACT IEEEP 1685 compliant and is assembly tool independent. As IEEE P1685 standard specifies the manipulation or generation of metadata (component object) must be independent of the design environment. It means that generation of a particular configuration of metadata cannot be generated from design environment. The methodology must provide a solution that for a configurable design a particular design configuration metadata description is generated at the time of IP-XACT based integration.

3.2 Existing Methodology

Existing methodology is to manually create various metadata descriptions possible for a design component and use them in integration. This is very time consuming and error prone.

Chapter 4

Proposed Methadology

This chapter describes the prescribed methodology and use model.

4.1 Methodology

The Proposed methodology comprises of two parts.

a. Creating a generic metadata for configurable IP's which contains all possible configuration information of the design. Then adding configurability information into the metadata description which can be controlled by parameters to generate configuration specific metadata.

b. A configuring mechanism that provides configured component whenever design environment reads a component from an IP-XACT database

.

As shown in the figure when metadata (IP-XACT component object is created) at that time a generic IP-XACT component is created by reading all possible configurations into a single metadata (component).Then configurability information is added to the metadata description process is described later. This configurability information is in terms of new XML defined tags that captures configurability information.

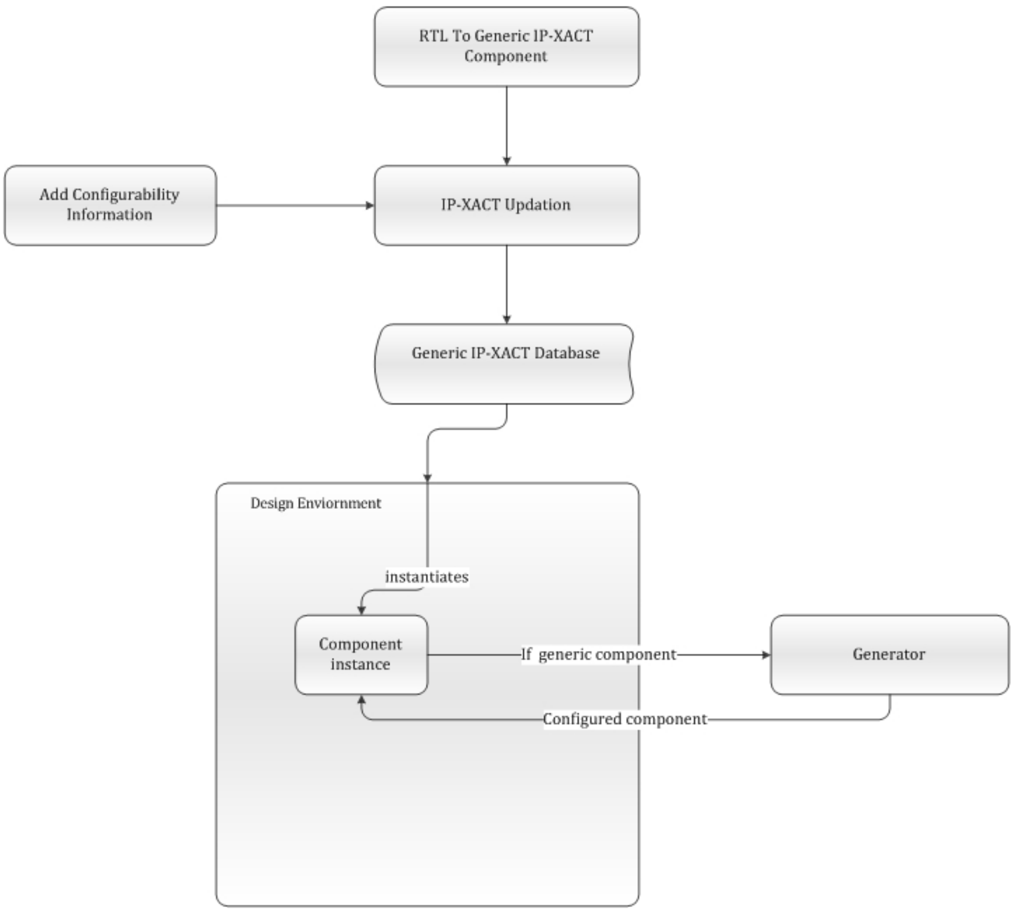


Figure 4.1: Proposed Methodology Flow

Once a generic metadata is created and configurability information is added then when Design environment instantiates a component a generator program is invoked which returns a configured component to the design environment. So a certain design configuration is only instantiated.

4.1.1 Creating a generic Component

An IP-XACT component is the central placeholder for the objects meta-data. Components are used to Describe cores (processors, co-processors, DSPs, etc.), peripherals (memories, DMA controllers, timers, UART, etc.), and buses (simple buses, multi-layer buses, cross bars, network on chip, etc.).It captures the data that is relevant for integration purpose like ports and interfaces at periphery.

A component may contain information related to

a. Ports in design

b. Bus interfaces

c. Register information

d. Address Blocks

e. RTL Parameters

Configurability required in a component for a configurable design is [8].

Required in	Type
Ports	Presence/Number of ports/Naming of ports
Bus interfaces	Presence/Number/physical port names in port maps
Registers	Presence/
Instance Naming	Updating
VLNV	Updating

Table I: Configurability Requirement

For a configurable RTL design first a generic metadata is created which contains all possible configurations (taking into account all the possible port instantiations)

To express this configurability new syntax were added to the XML to express this configurability.It was seen that two types of configurability was required.

a. Presence absence of an entity : this requires if an entity is present or not in a given component or not.like a port should be present or not etc.

b. Looping for an entity Looping means no of instances of a port so if a design requires that a array of ports to be created with a change in name.

This configurability information can be added into the IP-XACT component vendor extension field in IP-XACT component elements these fields are ignored by standard IP-XACT.

4.1.2 Configurability Syntax

The Syntax for configurability is explained here VE should be associated with "ConfigIp" prefix namespace and are written as <configIp:if> . <configIp:presence> </configIp:presence> is the top most element containing other elements .All configuration elements are written inside this tag.

a. if - Governs existence of an entity in context based on xpath expression.(Context is described later)
   <configip:presence>
   <configIp:if expr="xpath" />
   </configIp: presence>
   Single If governs the presence of port .If expr is evaluated to true port exits otherwise false .Here entity in context is port itself as no tag is preceding the if condition.

b. loop - Governs number of existence of entity in context. In case of nested statements, it would be passed sequentially. Like if configIp:if is written after configIp:loop then it expresses for(each loop iteration) if(expr)
   <configip:presence>
   <configIp:loop start="xpath" end="xpath" incr="xpath" loopvariable="i" >
   targ_$i_ack
   </configIp:loop>
   </configIp: presence>

   Single loop governs how many instance of a given port exists i.e. (end-start) with port names governed by the pattern specified as loop value .For example in this case if start=0 end=5 and incr=1 then 5 new ports will be added as targ_0_ack, targ_1_ack, targ_2_ack, targ_3_ack, targ_4_ack.

Expressions used are based on Xpath expressions which contains variables defined in parameter and model paramterer in component.
For example lets take example of a port configurability a port Defined in one of the configuration in verilog as

```
‘ifdef macro
input ack[10:0];
‘endif
```

IP-XACT component it will become

```
<spirit:port>
<spirit:name>ack</spirit:name>
<spirit:wire>
<direction>in</spirit:direction>
<spirit:vector>
<spirit:left>10</spirit:left>
<spirit:right>0</spirit:right>
</spirit:vector>
</spirit:wire>
<spirit:vendorExtension>
<config:presence>
<config:if expr=”id(’macro’)”/>
<config;presence>
</spirit:vendorExtension>
</spirit:port>
```

Here macro becomes a component parameter which is boolean type and its presence or absence governs if this port is Present in component or not.

4.1.3 Generation mechanism

Once a generic description is created and configurability information is added then the configuration mechanism is required when design environment instantiates a particular component. As shown in figure when design environment instantiates a component object at that time it invokes generator program.

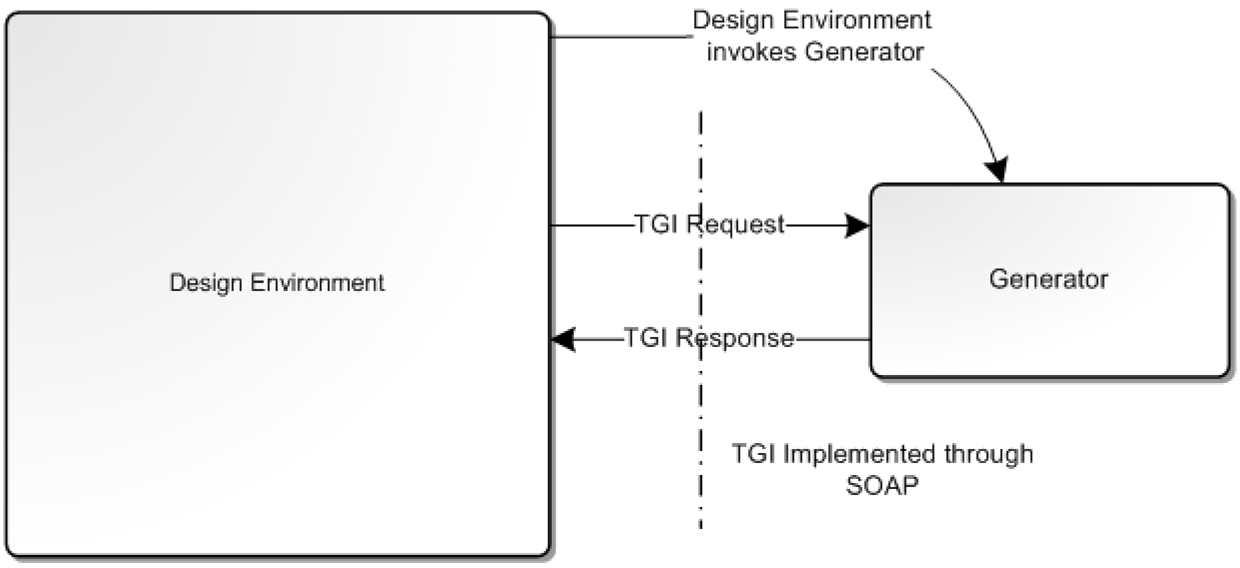

Design Environment invokes Generator
Design Environment
TGI Request
TGI Response
Generator
TGI Implemented through SOAP

Figure 4.2: Generation Mechanism

Location of generator is specified into the component itself in the generator section along with required parameters as shown. Communication between Design environment and Generator is through TGI API (Tight Generator Interface) which can be implemented using SOAP.

Design enviornment in itself hosts a SOAP server which implements TGI commands .When Design Enviornment wants to interect with generator it first starts the server embedded in it then starts the generator and calls the

4.1.4 Deploying the generator

Generator can be deployed in one of the two ways :

a. File System Generator can be deployed as on a filesystem and can be accessed using file system path file:path_to_executable (e.g., file:/usr/jdoe/bin/mygen.pl or file:../bin/ mygen) defines the path for invoking the generator on the machine from which the DE was invoked.

b. Web Server Generator can be hosted on a web server and can be invoked as http://web_address:port_number (e.g., http://www.acme.com/generator:1500) defines the URL of a generator implemented as a Web-based server.

Chapter 5

Implementation

This chapter presents implementation of the above prescribed methodology.

5.1 Tools

In this section we discuss about the tools and technologies that were studied to implement the methodology.

5.1.1 IP-XACT to Object oriented mapping

IP-XACT is a xml based standard so to work with ip-xact it was required to have a mechanism for read and manipulate IP-XACT documents programmatically hence an API was required. These API will help to access and manipulate IP-XACT documents programmatically.

One approach is to use DOM (Document object model)[11] to parse the IP-XACT documents and then apply then create IP-XACT documents as a simple xml file and then validate it with IP-XACT schema. However there are a few problems with this approach.

a. To create or manipulate a valid IP-XACT programmer must know all the rules/constraints and entities to add or manipulate for ex if he want to add an IP-XACT element he should know all the details and schema rules.

b. Code to add or manipulate the IP-XACT must be written from scratch .As IP-XACT is quite a big standard this process is time consuming and error prone.

Another way is to map the IP-XACT schema to Object oriented structure[13].As XML in itself is Object oriented it can be mapped to an object oriented programming language.

XML Beans

XMLbeans [3] is a tool that allows you to map XML to object oriented paradigm and allows for generation of a java library for accessing and manipulating the given XML. This maps XML tags and elements to Java features and constructs and provides API as "set" and "get". While a major use of XMLBeans is to access your XML instance data with strongly typed Java classes there are also API's that allow you access to the full XML infoset (XMLBeans keeps XML Infoset fidelity) as well as to allow you to reflect into the XML schema itself through an XML Schema Object model.

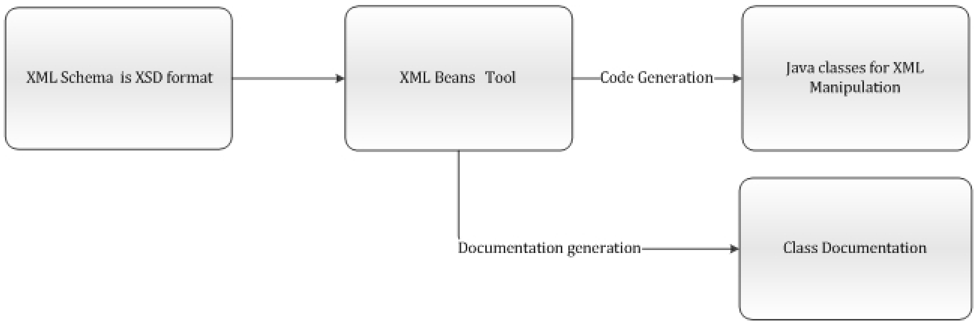


Figure 5.1: XML Beans

It takes input XML schema and generates corresponding Object oriented mapping. Major features of XMLBeans are

a. Full XML Schema support. XMLBeans fully supports XML Schema and the corresponding java classes provide constructs for all of the major functionality of XML Schema. This is critical since often times you do not have control over the features of XML Schema that you need to work with in Java. Also, XML Schema oriented applications can take full advantage of the power of XML Schema and not have to restrict themselvs to a subset.

b. Full XML Infoset fidelity. When unmarshalling an XML instance the full XML infoset is kept and is available to the developer. This is critical because because of the subset of XML that is not easily represented in java. For example, order of the elements or comments might be needed in a particular application.

API that it provides is:

a. XmlObject The java classes that are generated from an XML Schema are all derived from XmlObject. These provide strongly typed getters and setters for each of the elements within the defined XML. Complex types are in turn XmlObjects. For example getCustomer might return a CustomerType (which is an XmlObject). Simple types turn into simple getters and setters with the correct java type. For example getName might return a String.

b. XmlCursor From any XmlObject you can get an XmlCursor. This provides efficient, low level access to the XML Infoset. A cursor represents a position in the XML instance. You can move the cursor around the XML instance at any level of granularity you need from individual characters to Tokens.

c. SchemaType XMLBeans provides a full XML Schema object model that you can use to reflect on the underlying schema meta information. For example, you might want to generate a sample XML instance for an XML schema or perhaps find the enumerations for an element so that you can display them.

5.1.2 Axis tool

SOAP Protocol

SOAP [10], originally defined as Simple Object Access Protocol, is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks. It relies on Extensible Markup Language (XML) for its message format, and usually relies on other Application Layer protocols, most notably Remote Procedure Call (RPC) and Hypertext Transfer Protocol (HTTP), for message negotiation and transmission. SOAP can form the foundation layer of a web services protocol stack, providing a basic messaging framework upon which web services can be built. This XML based protocol consists of three parts: an envelope, which defines what is in the message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing procedure calls and responses.

SOAP and WSDL

One of the major usage of SOAP protocol is to implement networked service .These networked service requests are a way to request XML-related functionality from a remote machine over a network such as the Internet. SOPA is used to enable these network services provides XML based data exchange. In this service model there is one client which also implements SOAP client and queries the server.The server implements the server side of the service and is usually hosted on an HTTP server. The queries are RPC based when SOAP is using HTTP as underlying protocol. The server provides the list of service which are methods or API that client can call on the service through a WSDL file (Web Service Description File).So by describing a WSDL file we can generate the given client and server side code automatically with given tools.

Axis [1] is a SOAP implementation generator which generates client and server side of SOAP implementation in java given a wsdl file as shown in the figure. This was used to implement the client side of TGI API implementation.

If we compare Axis with other SOAP implementation we find that it has lowest latency delay as compared to others as shown in the table for character string message [15].

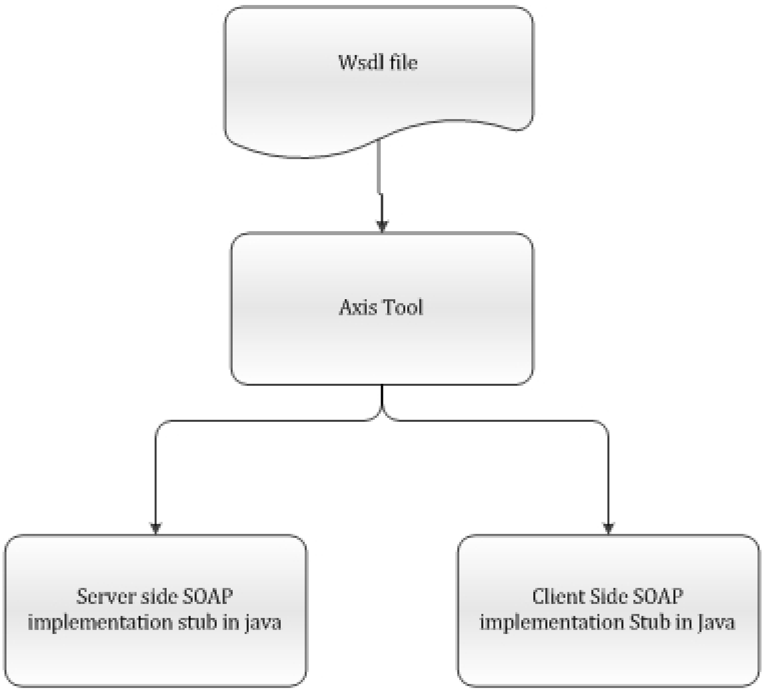

Wsdl file
Axis Tool
Server side SOAP implementation stub in java
Client Side SOAP implementation Stub in Java

Figure 5.2: Axis tool

5.1.3 JACL and TCLBlend

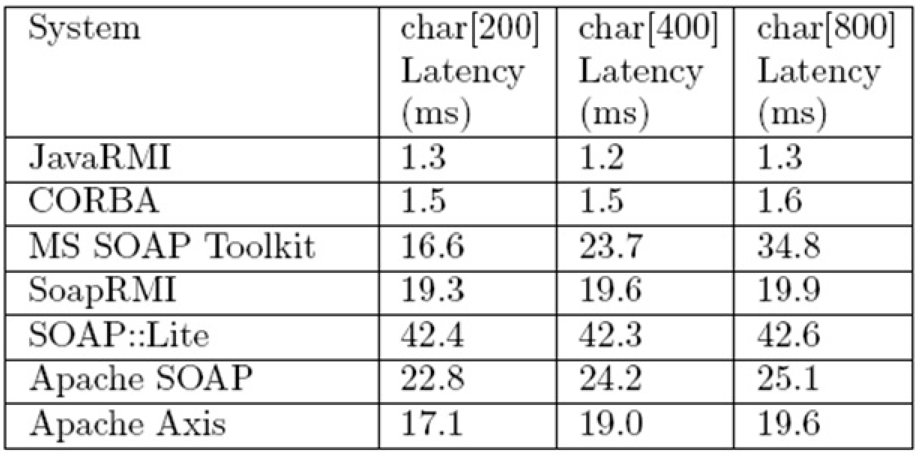
One of these tools is used to create user interface which is a TCL based shell.

JACL

Jacl [6] is an implementation of a Tcl interpreter written entirely in Java. As a result, a Java program can easily embed a Jacl interpreter in an existing Java program. This ability to add a scripting language to a Java application is analogous to the ability to add scripting to a C based application provided by the C implementation of Tcl. Jacl allows Tcl scripts to call Tcl commands written in Java. Jacl also allows Tcl scripts to use Java classes, objects and methods directly.

The benefits of using Jacl are: no need for a separate interpreter process, pure Java implementation, and the ability to call Java methods from Tcl scripts.

The drawbacks are that Jacl cannot make use of a Tcl extension written in C, and Jacl does not implement all the features provided by the C based of Tcl. Jacl



System	char[200] Latency (ms)	char[400] Latency (ms)	char[800] Latency (ms)
JavaRMI	1.3	1.2	1.3
CORBA	1.5	1.5	1.6
MS SOAP Toolkit	16.6	23.7	34.8
SoapRMI	19.3	19.6	19.9
SOAP::Lite	42.4	42.3	42.6
Apache SOAP	22.8	24.2	25.1
Apache Axis	17.1	19.0	19.6

Figure 5.3: Axis tool comparision

currently does not implement "interp", "socket", "fileevent", and some less important utility procedures that are available in the C based of Tcl.

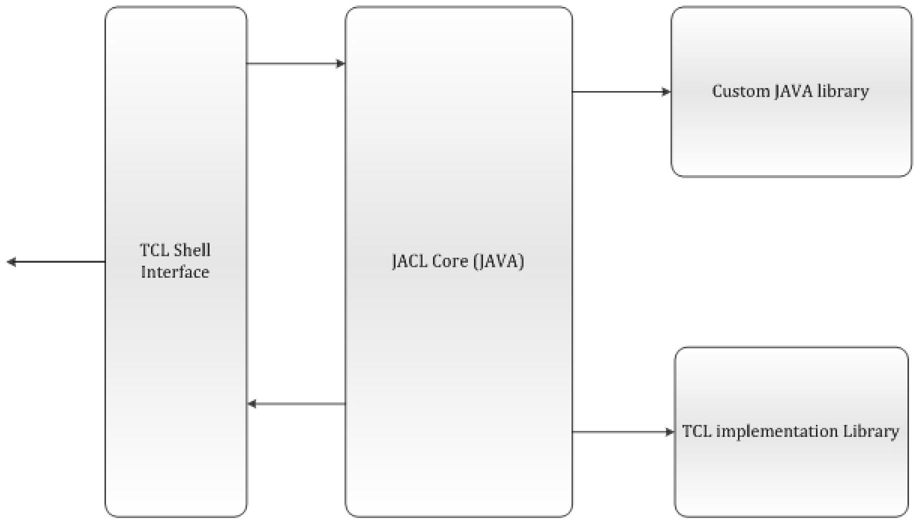


Figure 5.4: JACL Shell

As shown in the figure JACL has a TCL shell interface to the user and an underlying java core which processes the TCL commands and maintains shell session. Shell can be extended by adding new java classes and methods and then invoking the commands on shell. Java interface from the shell is through the java API provided by JACL ex java::new ¡class¿ creates a new object of the class on shell.

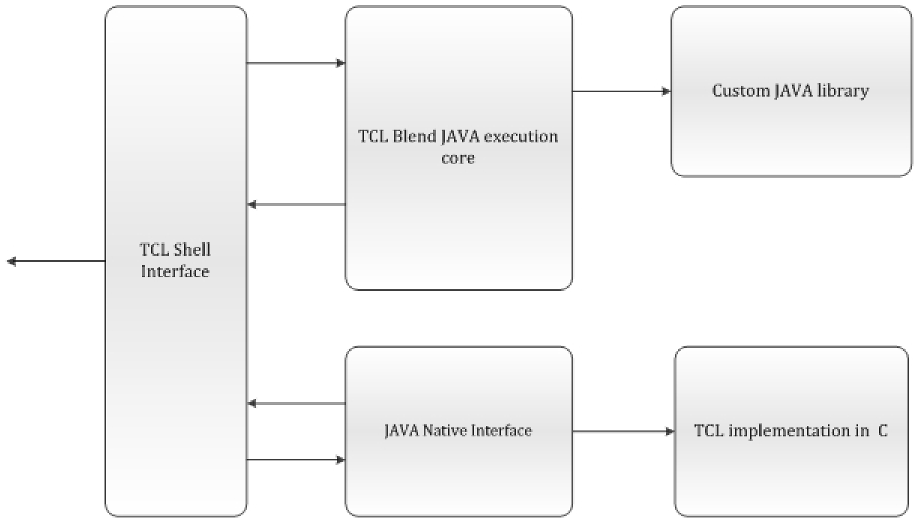


Figure 5.5: TCL Blend

TCLBlend

TclBlend [2] combines some of the benefits from both Jacl and the Feature package to provide a best of both worlds approach. Using TclBlend, a Java program can embed a C based Tcl interpreter like the Feather package. TclBlend also allows Tcl scripts to use Java classes, objects, and methods like Jacl. Both TclBlend and Jacl provide the ability to extend the Tcl interpreter with new Tcl commands implemented in Java code.

Using this feature is like using the C language to write new Tcl command for a C based Tcl interpreter. In addition, TclBlend and Jacl use the Java reflection feature (java.lang.reflect package) to allocate Java objects and invoke Java methods at run-time. This allows Tcl scripts to create, access Java objects, and execute Java methods directly without first creating Tcl commands. The difference between TclBlend and Jacl is that TclBlend uses a C based Tcl interpreter to execute Tcl scripts while Jacl uses a Java based Tcl interpreter to execute Tcl scripts.

The benefits of using TclBlend are: no need for a separate interpreter process, the ability to call Java methods from Tcl scripts, and the ability to use a C based Tcl interpreter which can use Tcl extensions written in C. The C based Tcl interpreter is much faster than the Java based Tcl implementation.

The drawback is that the Java program is not a pure Java implementation, which requires the programmer to distribute several operating system dependent shared library files in addition to the Java classes. Feature comparision TCL,jacl and TCLBlend.

Feature comparision TCL,jacl and TCLBlend

	Embedded TCL interp	Pure java	TCL 8 feature	Java access	C access
Standard TCL	yes	No	Yes	No	Yes
JACL	No	Yes	No	Yes	No
TCLBlend	Yes	No	Yes	Yes	Yes

Table I: JACL and TCLBlend comparision

Performance Evaluation

To evaluate the performance of the given three shells we use three different categories of commands.

a. String manipulation. Sting toupper,trim etc.

b. Arithmetic expression. expr.

c. Input/output. file read/write

If we analyze the results we can see that although JACL is worst performer for all type of commands if performs worst in the case of input output operations. This is to do with the fact that Java IO is slower as compared to C based IO as in case of standard TCL.

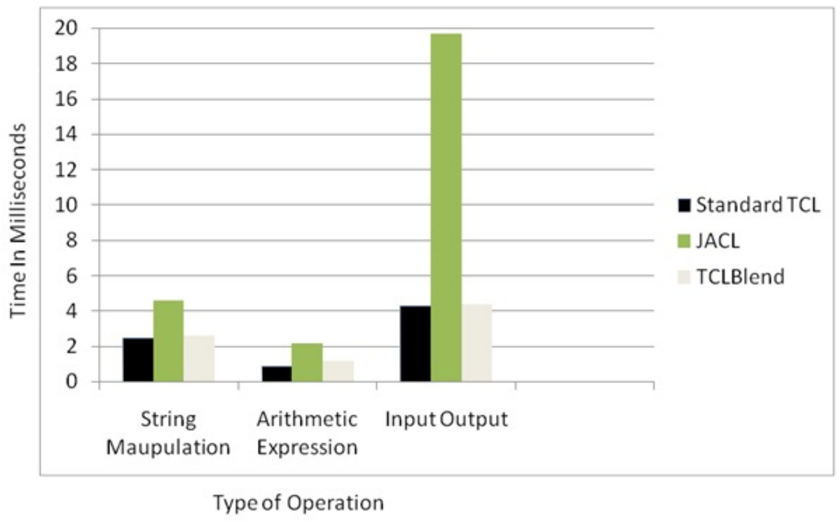


Figure 5.6: JACL and TCLBlend performence

5.2 Generator Design

In this section we discuss the design and implementation of the generator program.

Architecture

Generator program architecture comprises of a parsing library and communication mechanism .It has four main parts as shown in the figure.

a. Soap Client: - This client implements SOAP client which is responsible for TGI related communication with the design environment. SOAP communication is described in later sections.

b. Parsing Library: - This library actually implements the IP-XACT xml schema to object oriented mapping this is achieved through the XMLBeans tool to convert the IP-XACT schema to object oriented mapping.

   Library is available as a jar file which can then be used for parsing and manipulating IP-XACT objects.XML Beans can also generate Documentation (in javadoc format) for the generated API's. These API include classes for creating and manipulating the IP-XACT objects.

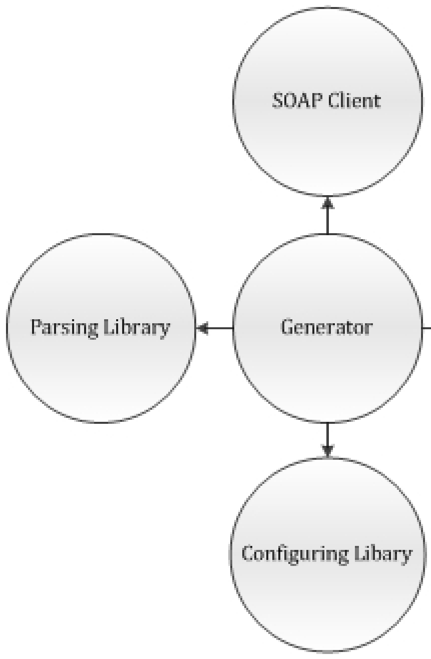


Figure 5.7: Generator Architecture

c. Configuring library: - This library contains the implementation of the configurability algorithm. Algorithm is implemented as java classes. This library uses Parsing library for manipulation of IP-XACT component.

d. Generator Core :- It is the main control for the generator program, it is the one which initializes the generator program , it includes

   (1) Initializing the SOAP client for communication with the design environment.

   (2) Applying the configuration algorithm on the component.

   (3) Maintaining the log files and performing input output.

5.2.1 TGI API implementation and SOAP

To have the methodology design tool independent it was required that the communication between design environment and generator program be standardized .For this purpose TGI (Tight generator Interface) API's were used as specified in IEEE P1685 standard.

Following are few of the TGI Commands that were used

TGI API	Description
‘ init	Initialize communication between design environment and generator.
end	End communication
getGeneratorContextComponentInstanceID	ID for the component instance associated with the currently invoked generator.
getComponentInstanceComponentID	ID for the component associated with given instance (crossing from design to component file).
getComponentInstanceXML	Return the component XML in text format. Schema version is DE dependent.

Table II: TGI API implementation

As shown in the execution scenario the TGI API's are implemented using the SOAP protocol to make the communication between design environment and generator standardized.

SOAP Client is created with the help of Axis tool which reads a wsdl file to generate client and server stub which can then be modified.TGI commands are provided as a wsdl file and Axis generates its client and server side stubs.

As shown in the figure TGI API as a wsdl are given then Axis will convert that into client and server side implementation. It is to be noted that it is not required to have both client and server side implementation of Axis are to be used only client can do with a compliant server implementation.

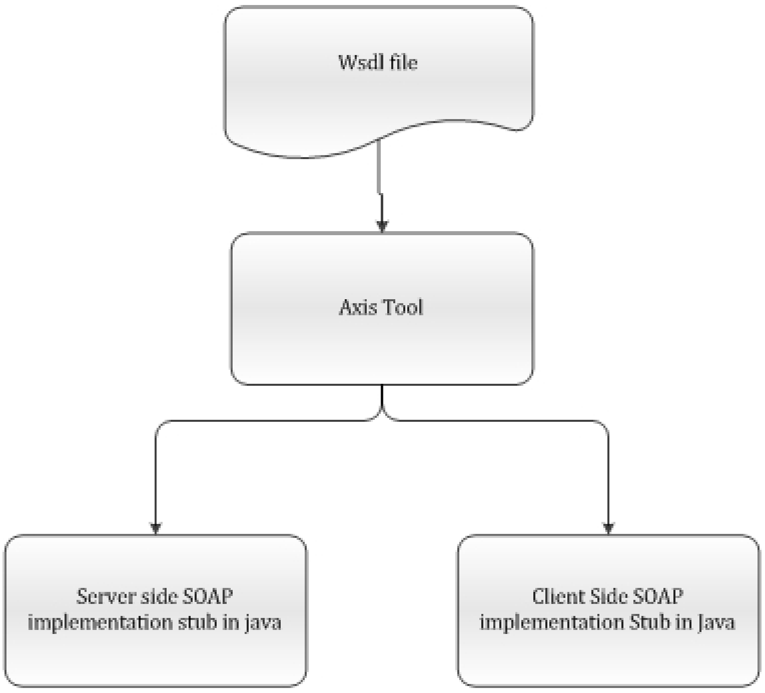

Wsdl file
Axis Tool
Server side SOAP implementation stub in java
Client Side SOAP implementation Stub in Java

Figure 5.8: Axis tool code generation

5.2.2 Design Environment Generator Communication

As shown in the sequence diagram the generator and design environment communication starts with design environment starting the web server and then generator starts the SOAP client which then communicates with design environment.

As shown in the figure the TGI communication starts with init command and ends with end TGI command.

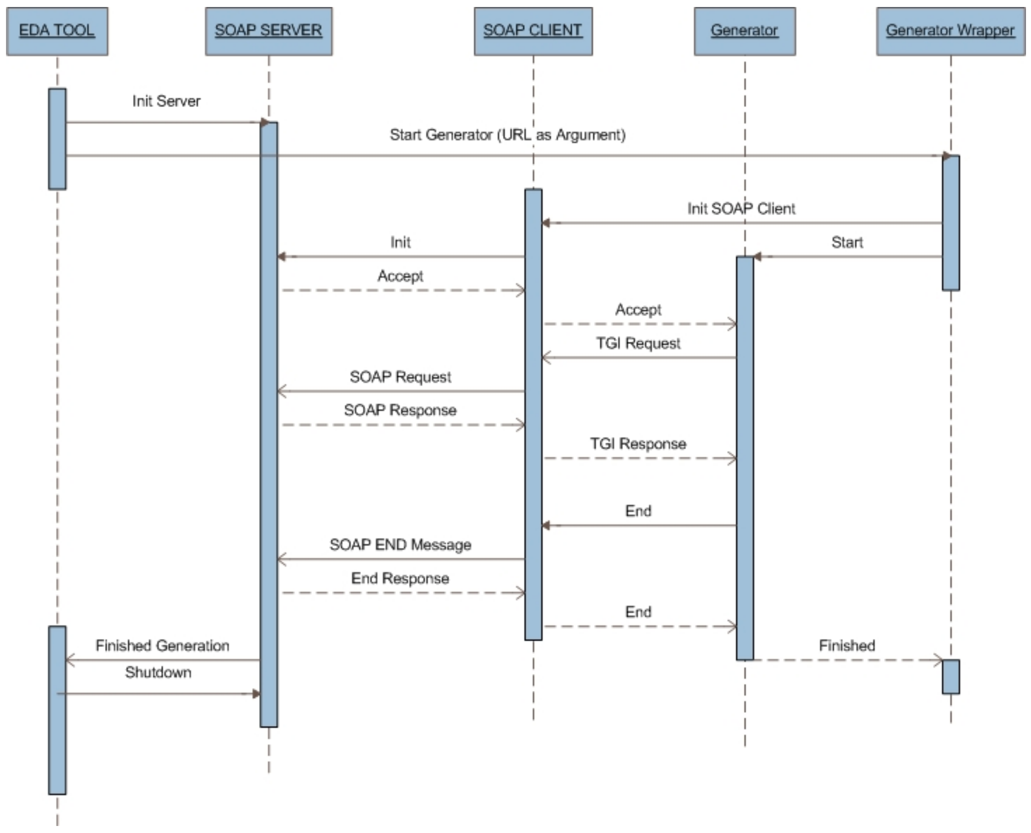


Figure 5.9: Design Enviornment Generator Interection

5.2.3 Configuring Algorithm

The algorithm that configures the given component works as The Configuring algorithm works as follows

a. Connect to the Design Environment server initialize TGI Communication.

b. Read the xml component from the design environment.

c. Use argument list to resolve parameters in component .two types of parameters are resolved based on embedded XPATH expression.

    (1) Component parameters

    (2) Model Parameters

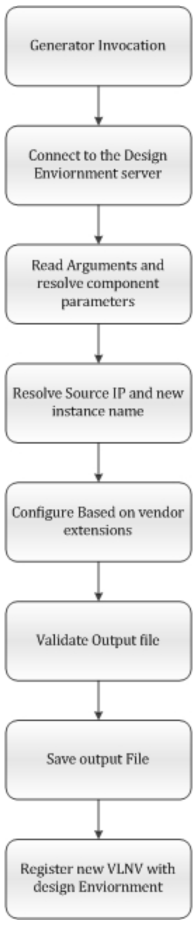

Generator Invocation
Connect to the Design Enviornment server
Read Arguments and resolve component parameters
Resolve Source IP and new instance name
Configure Based on vendor extensions
Validate Output file
Save output File
Register new VLNV with design Enviornment

Figure 5.10: Generation Algorithm

d. Resolve the source of the component .This is done to check that the component in context is a already configured component. Also resolve the configured component name.

e. Configure the component .Configuration is done in following sequence

    (1) Ports

    (2) Bus Interfaces

    (3) Registers

f. Validate the component to be IP-XACT compliant.

g. Register the new component with design environment.

5.2.4 Design Environment updates

Along with the implementation of the methodology it was required to update the existing in-house IP-XACT assembly tool IP-XACT assembler. Updates were to provide API support in the tool to allow user to add configurability information in the IP-XACT component.

An TCL based user interface was created with the help of TCLBlend where tool core which is written in java was plugged into TCLBlend .Then new library was created to add the configurability information in a IP-XACT component. These API are available as commands as TCL based commands. Example.

```
addNamespace "config" "www.st.com"
addConfig "port" "PT1" "stlib" element "if" attribute "expr 1" value ""
```

5.3 Result and Discussion

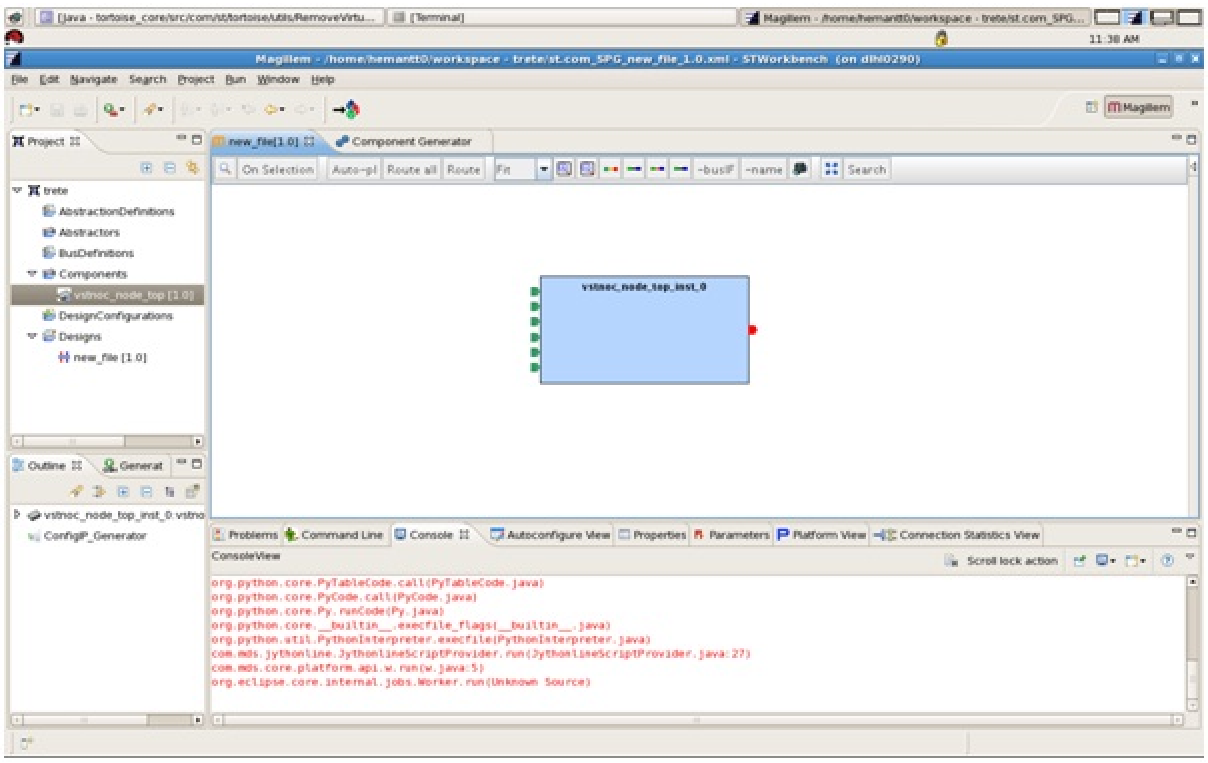
In this section we present the implementation results to show that the above defined methodology is design environment independent. Here we show its usage with different design tools.

5.3.1 Use model in ST Microelectronics

At ST Microelectronics the generator is available on a central area and is available through Unicad Setup (Central CAT tool setup facility).Designers and packagers creating and packaging IP-XACT components can reference this generator from a central location itself. This model helps in updating the generator program and also to maintain version information.

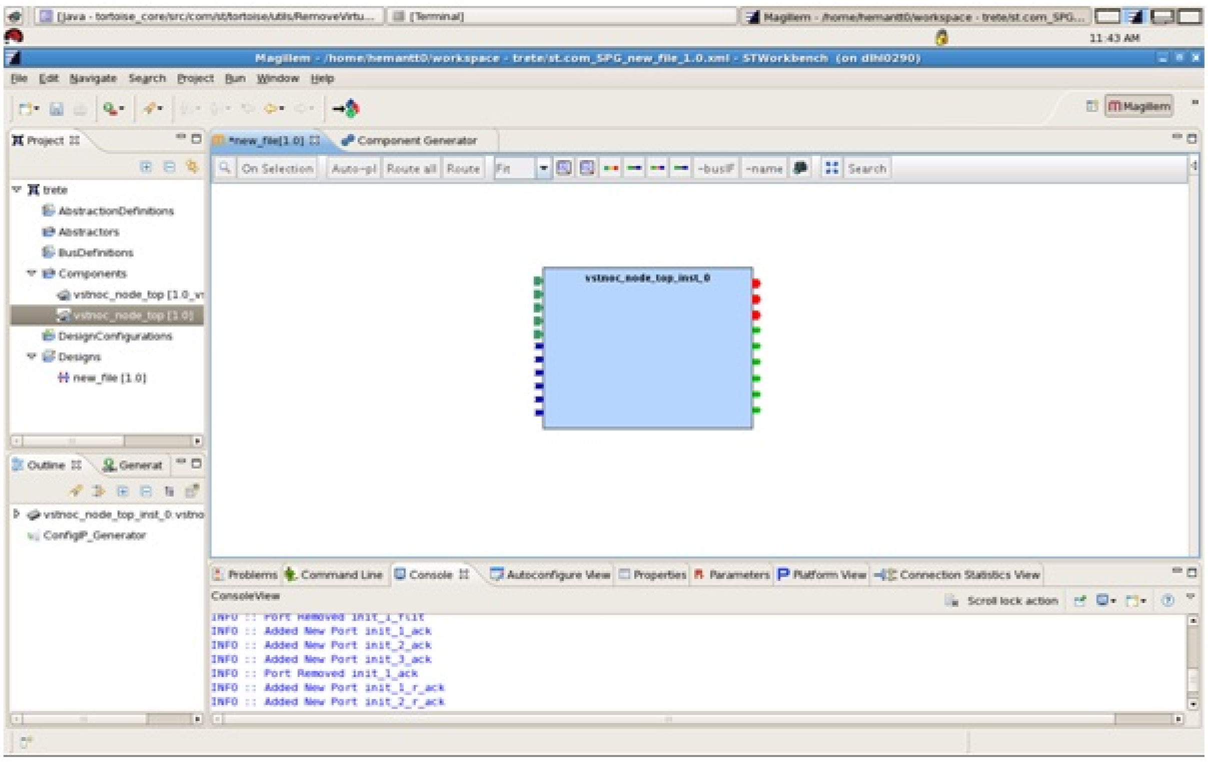
5.3.2 With Magillem design tool

Magillem[7] is an IP-XACT design tool which is developed in java and to support configurable IP's an evaluation version was provided form the vendor which includes a SOAP server as defined in the above mentioned methodology. The magillem is a GUI based design tool.



```
org.python.core.PyTableCode.call(PyTableCode.java)
org.python.core.PyCode.call(PyCode.java)
org.python.core.Py.runCode(Py.java)
org.python.core.__builtin__.execfile_flags(__builtin__.java)
org.python.util.PythonInterpreter.execfile(PythonInterpreter.java)
com.mds.jythonline.JythonLineScriptProvider.run(JythonLineScriptProvider.java:27)
com.mds.core.platform.api.w.run(w.java:5)
org.eclipse.core.internal.jobs.Worker.run(Unknown Source)
```

Figure 5.11: Component instantiation in magillem



```
INFO :: Port Removed init_1_flit
INFO :: Added New Port init_1_ack
INFO :: Added New Port init_2_ack
INFO :: Added New Port init_3_ack
INFO :: Port Removed init_1_ack
INFO :: Added New Port init_1_r_ack
INFO :: Added New Port init_2_r_ack
```

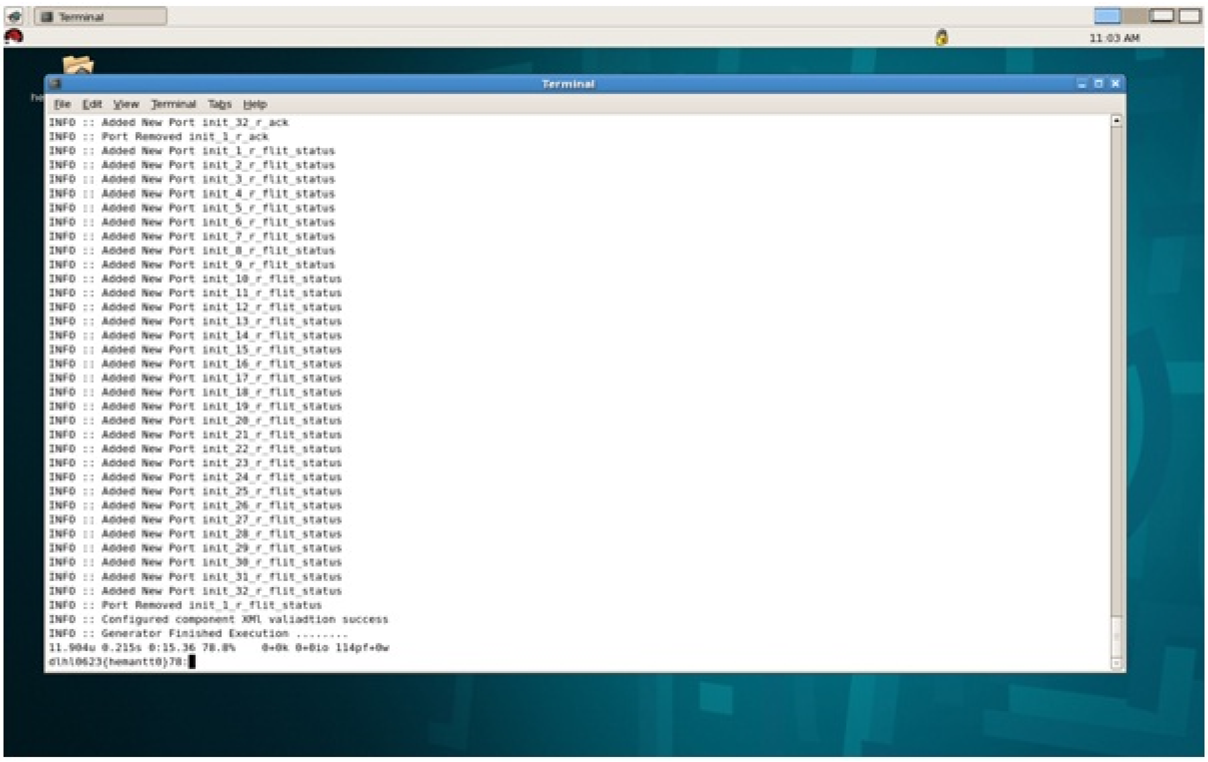
Figure 5.12: Generator Invocation in Magillem

In magillum the configurability information can be added in GUI mode and then can be saved. Then at time of assembly when user instantiates a given generic IP-XACT component user has to manually invoke the generator. Snapshots

5.3.3 With In-house Developed IP-XACT assembly tool

IP-XACT design tool [4] developed in ST is a TCL shell based tool all the API's are available as TCL command.Adding the configurability information is through the TCl commnds only.

The assembly flow is table based where user specifies the assembly commands in a table which are then executed.

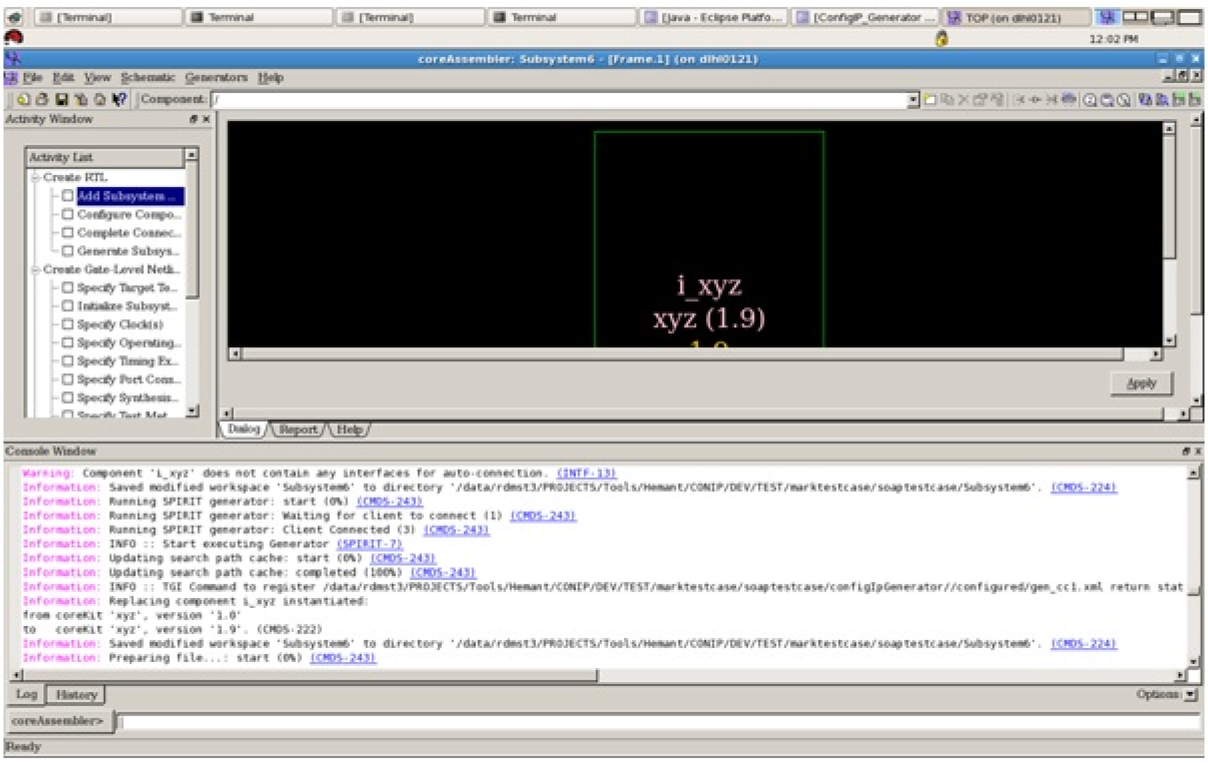


```
INFO :: Added New Port init_32_r_ack
INFO :: Port Removed init_1_r_ack
INFO :: Added New Port init_1_r_flit_status
INFO :: Added New Port init_2_r_flit_status
INFO :: Added New Port init_3_r_flit_status
INFO :: Added New Port init_4_r_flit_status
INFO :: Added New Port init_5_r_flit_status
INFO :: Added New Port init_6_r_flit_status
INFO :: Added New Port init_7_r_flit_status
INFO :: Added New Port init_8_r_flit_status
INFO :: Added New Port init_9_r_flit_status
INFO :: Added New Port init_10_r_flit_status
INFO :: Added New Port init_11_r_flit_status
INFO :: Added New Port init_12_r_flit_status
INFO :: Added New Port init_13_r_flit_status
INFO :: Added New Port init_14_r_flit_status
INFO :: Added New Port init_15_r_flit_status
INFO :: Added New Port init_16_r_flit_status
INFO :: Added New Port init_17_r_flit_status
INFO :: Added New Port init_18_r_flit_status
INFO :: Added New Port init_19_r_flit_status
INFO :: Added New Port init_20_r_flit_status
INFO :: Added New Port init_21_r_flit_status
INFO :: Added New Port init_22_r_flit_status
INFO :: Added New Port init_23_r_flit_status
INFO :: Added New Port init_24_r_flit_status
INFO :: Added New Port init_25_r_flit_status
INFO :: Added New Port init_26_r_flit_status
INFO :: Added New Port init_27_r_flit_status
INFO :: Added New Port init_28_r_flit_status
INFO :: Added New Port init_29_r_flit_status
INFO :: Added New Port init_30_r_flit_status
INFO :: Added New Port init_31_r_flit_status
INFO :: Added New Port init_32_r_flit_status
INFO :: Port Removed init_1_r_flit_status
INFO :: Configured component XMl valiadtion success
INFO :: Generator Finished Execution ........
11.904u 0.215s 0:15.36 78.8%    0+0k 0+0io 114pf+0w
dlnl0623(hemantt0)78:
```

Figure 5.13: IP-XACT assembly tool generator invocation

5.3.4 With core Assembler

Core Assembler [14] is a IP-XACT assembly tool form Synopsys .It is developed in C and has both shell as well as GUI based interface. It supports auto invocation of generator while instantiating any IP. Generator invocation in Core Assembler.



```
Warning: Component 'i_xyz' does not contain any interfaces for auto-connection. (INTF-13)
Information: Saved modified workspace 'Subsystem6' to directory '/data/rdmst3/PROJECTS/Tools/Hemant/CONIP/DEV/TEST/marktestcase/soaptestcase/Subsystem6'. (CMDS-224)
Information: Running SPIRIT generator: start (0%) (CMDS-243)
Information: Running SPIRIT generator: Waiting for client to connect (1) (CMDS-243)
Information: Running SPIRIT generator: Client Connected (3) (CMDS-243)
Information: INFO :: Start executing Generator (SPIRIT-7)
Information: Updating search path cache: start (0%) (CMDS-243)
Information: Updating search path cache: completed (100%) (CMDS-243)
Information: INFO :: TGI Command to register /data/rdmst3/PROJECTS/Tools/Hemant/CONIP/DEV/TEST/marktestcase/soaptestcase/configIpGenerator//configured/gen_cc1.xml return stat
Information: Replacing component i_xyz instantiated:
from coreKit 'xyz', version '1.0'
to   coreKit 'xyz', version '1.9'. (CMDS-222)
Information: Saved modified workspace 'Subsystem6' to directory '/data/rdmst3/PROJECTS/Tools/Hemant/CONIP/DEV/TEST/marktestcase/soaptestcase/Subsystem6'. (CMDS-224)
Information: Preparing file...: start (0%) (CMDS-243)
```

Figure 5.14: Generator invocation with coreAssembler

Table below shows the support for the methodology in different IP-XACT Assembly Tools

	IP-XACT assembly Tool	Magillem	Core Assembler
Support for Adding Configurability Information	Yes	Yes	No
SOAP server implementation.	Yes	Yes	Yes
Auto Invocation of generator	Yes	No	Yes
Remote invocation for generator	No	Yes	Yes

Table III: Assembly tool feature support for configurable IP

Chapter 6

Conclusion and Future Scope

6.1 Conclusion

IP-XACT IEEEP 1685 is fast becoming a standard for reusing and integrating intellectual property in design flows, as it evolves new requirements are being explored to improve the usage and to complete the standard. The methodology proposed in this thesis report is aims at overcoming one of the limitations of IEEE P1685 standard. It provides a methodology to address configurability due to macro definition in IP-XACT based assembly. The works aims at providing a Design environment independent methodology. It helps in automating the complete IP-XACT assembly flow and providing a vendor neutral solution. Implementation presented in this chapter is done in java and interaction model presented is through SOAP hence making the implementation design environment independent. New syntax has been proposed to define configurability and an algorithm to configure the component. Tools and technology used is implementation are discussed and presented.

6.2 Future scope

In future the work can be extended to create an optimized generator program that can configure large components in less time. Exploring configurability requirements with

HDL languages such as systemc etc. Make configurability syntax as part of IEEE P1685 standard. So that these configurability syntax are automatically generated as part of RTL Parsing.

Appendix A

IP-XACT component

An IP-XACT component is the central placeholder for the objects meta-data. Components are used to describe cores (processors, co-processors, DSPs, etc.), peripherals (memories, DMA controllers, timers, UART, etc.), and buses (simple buses, multi-layer buses, cross bars, network on chip, etc.). An IP-XACT component can be of two kinds: static or configurable. A DE cannot change a static component. A configurable (or parameterized) component has configurable elements (such as parameters) that can be configured by the DE and these elements may also configure the RTL or TLM model. An IP-XACT component can be a hierarchical object or a leaf object. Leaf components do not contain other IP-XACT components, while hierarchical components contain other IP-XACT sub-components. This can be recursive by having hierarchical components that contain hierarchical components, etc.-leading to the concept of hierarchy depth. The IP being described may have a completely different hierarchical arrangement in terms of its implementation in RTL or TLM to that of its IP-XACT description. So, a description of a large IP component may be made up of many levels of hierarchy, but its IP-XACT description need only be a leaf object as that completely describes the IP. On the other hand, some IP can only be described in terms of a hierarchical IP-XACT description, no matter what the arrangement of the implementation hierarchy.

A.1 Component Schema

Each IP component normally identifies one or more bus interfaces. Bus interfaces are groups of ports that belong to an identified bus type . The purpose of the bus interface is to map the physical ports of the component to the logical ports of the abstraction definition. This mapping provides more information about the interface. There are seven possible modes for a bus interface: a bus interface may be a master, slave, or system interface, and may be direct or mirrored. The seventh interface mode is the monitor mode. A monitor interface can be used to connect IP into the design for verification.

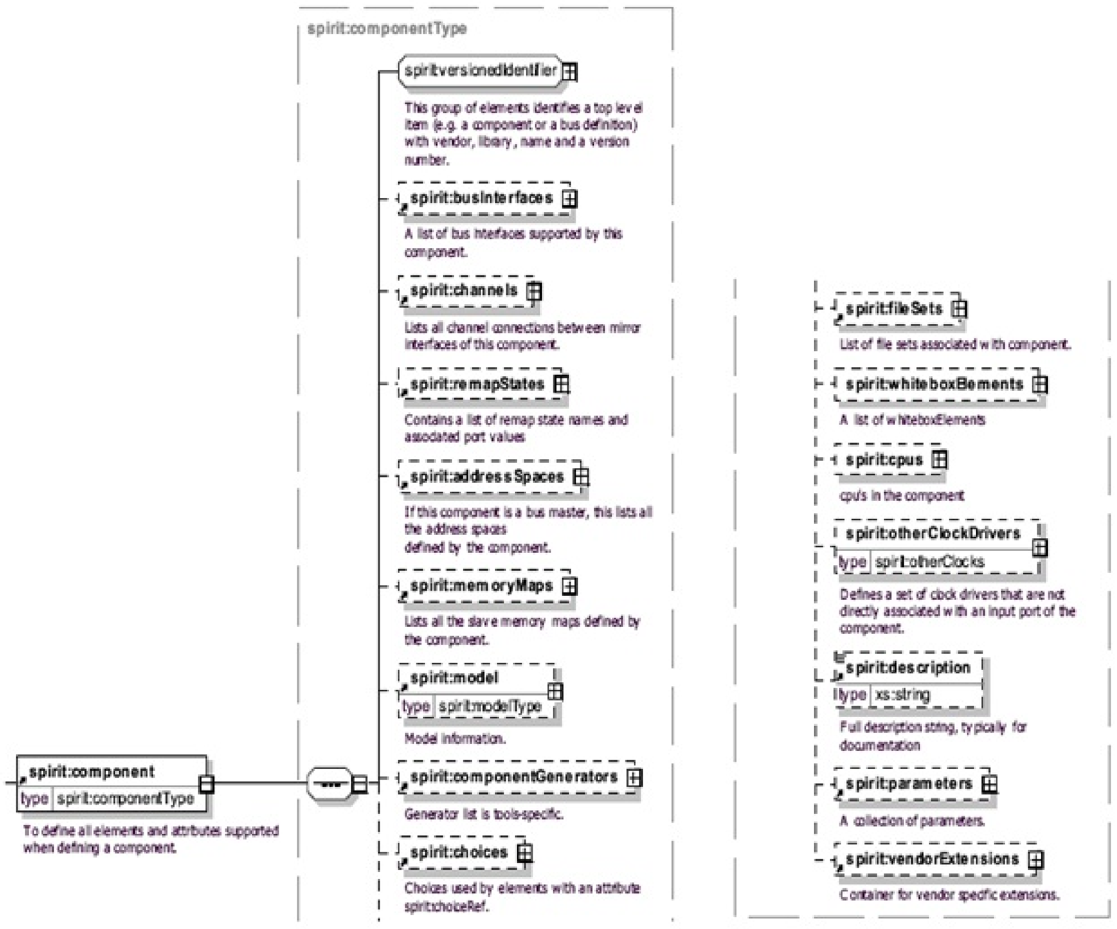

spirit:componentType
spirit:versionedIdentifier
This group of elements identifies a top level item (e.g. a component or a bus definition) with vendor, library, name and a version number.
spirit:busInterfaces
A list of bus interfaces supported by this component.
spirit:channels
Lists all channel connections between mirror interfaces of this component.
spirit:remapStates
Contains a list of remap state names and associated port values
spirit:addressSpaces
If this component is a bus master, this lists all the address spaces defined by the component.
spirit:memoryMaps
Lists all the slave memory maps defined by the component.
spirit:model
type spirit:modelType
Model information.
spirit:componentGenerators
Generator list is tools-specific.
spirit:choices
Choices used by elements with an attribute spirit:choiceRef.
spirit:component
type spirit:componentType
To define all elements and attributes supported when defining a component.
spirit:fileSets
List of file sets associated with component.
spirit:whiteboxElements
A list of whiteboxElements
spirit:cpus
cpu's in the component
spirit:otherClockDrivers
type spirit:otherClocks
Defines a set of clock drivers that are not directly associated with an input port of the component.
spirit:description
type xs:string
Full description string, typically for documentation
spirit:parameters
A collection of parameters.
spirit:vendorExtensions
Container for vendor specific extensions.

Figure A.1: Component Schema

Other main elements of an IP-XACT component are:

a. Model The model element describes the views, ports, and model-related parameters of a component.

b. Ports Ports in the design this is an unbounded list.It contains information about the ports on pheriphery.Few fields it captures are name ,direcction,default value and if it is a vector then the left/right of the vector. Views

c. Views define grouping of source files for different purpose like simulation, synthesis etc.Each view has a corresponding name and a fileset which is collection of file paths for that view.their are few standard view defined by IP-XACT as behavioural,synthesis,simulation etc.

d. Model Parameters These are the parameters defined in the HDL code .These parameters are used in HDL to control the size of a port, register or for some other control purpose .These parameters are captured in IP-XACT as model parameters along with a unique ID for them ,default value,type and any constraint on minimum or maximum value.These model parameters are used in XPath expressions for ex to resolve left right of a vector port.

e. Component Generator This contains Generator related information which includes generator path, name and arguments.Generators are explained in Appendix B.

f. Parameters These are the parameters that apply on the whole of the component and can be used anywhere.These are analogous to the model parameters but they are for IP-XACT component where as model parameters are extracted form the HDL design.

Appendix B

Generators and TGI API

IP-XACT generators are tools that are invoked from within a DE to perform an operation required by the user of the DE. For example, generators can be provided to verify the configuration of a subsystem, generate an address map, or write a netlist representation of the subsystem in a target language such as Verilog or ystemC. To perform their various operations, most generators need access to the IP-XACT meta-data describing the subsystem, as currently loaded into the DE. Generators need both read- and write-access to the IP-XACT meta-data. All generators are external applications running in a separate address space from the DE.

B.1 Method of communication

The DE and the generator communicate with each other by sending messages to each other utilizing the SOAP standard. SOAP provides a simple means for sending XML-format messages using HTTP or other transport protocols. The TGI restricts the set of allowed transport protocols to HTTP and a file-based protocol. All generators are required to support the HTTP protocol, but support for the file-based protocol is optional. The same rules apply to the DE-it shall support the use of the HTTP protocol, but is not required to support the file-based protocol, even though a generator may allow it. The protocols supported by a generator are specified using the transportMethod element within the componentGenerator element.

The information required to use a particular transport protocol shall be passed to the generator by the DE when it is invoked, as described in G.2. For the HTTP protocol, the generator is passed a URL of the form http://host_name:port_number. All SOAP messages sent to the DE shall be sent using the referenced URL. For the file-based protocol, the generator is passed a URL of the form file://file_name. In this case, all SOAP messages are written to the specified file.

B.2 Generator invocation

All of the information known by the DE about a particular generator comes from an instance of the componentGenerator elements. These elements provides the following information.

a. Name is the name of the generator as seen within the DE.

b. Executable is the URL defining the location of the generator.

B.3 Resolving the URL

The URL defining the generator executable shall resolve to one of the following forms.

a. file:path_to_executable (e.g., file:/usr/jdoe/bin/mygen.pl or file:../bin/ mygen.pl) defines the path for invoking the generator on the machine from which the DE was invoked.

b. file://machine_name/path_to_executable (e.g., file://server1/tmp/othergen.pl) defines the path for invoking a generator on the specified machine.

c. http:/
itemaddress:port_number (e.g., http://www.acme.com/generator:1500) defines the URL of a generator implemented as a Web-based server.

All file references are relative to the location of the XML description in which the file reference is contained. For the file-based generators, the DE shall invoke the generator as a sub-process with a command line built up as:

executable -url transport_URL generator_parameter_arguments

The generator_parameter_arguments are the parameters from the componentGenerator element with the user-specified values. Each parameter causes two additional arguments to be passed to the generator with the following format: -parameter_name parameter_value. The transport_URL is created by the DE, but is guaranteed to specify a protocol supported by the generator as defined by the transport methods within the componentGenerator. The DE is responsible for ensuring any passed parameters can be interpreted correctly. This URL is to be used in the generator to set up the SOAP communication channel. For Web-based generators, the DE shall send a message to the address and port defined as the executable.
The format of this message is

url=transport_URL&generator_parameter_arguments

In this case, the generator parameters are formatted using the standard HTTP parameter passing syntax. The specified transport URL shall be used by the generator for any return messages to the DE. The invocation syntax described above applies only to generators with an API type of TGI. Generators with an API type of none are invoked as described above, excluding the transport_URL argument.

B.3.1 Example

This example shows file-based and Web-based componentGenerator elements.

<spirit:componentGenerator>

<spirit:name>myGenerator</spirit:name>

<spirit:parameter spirit:name="param1" spirit:resolve="user"

spirit:id="param1">default1</spirit:parameter>

<spirit:parameter spirit:name="param2">fixedValue</spirit:parameter>

```
<spirit:apiType>TGI</spirit:apiType>
<spirit:transportMethods>
<spirit:transportMethod>file</spirit:transportMethod>
</spirit:transportMethods>
<spirit:generatorExe>../bin/myGenerator.pl</spirit:generatorExe>
</spirit:componentGenerator>
```

produces the following output.

path_to _XML/../bin/myGenerator -url http:/host:port -param1 default1 -param2 fixedValue

Whereas:

```
<spirit:componentGenerator>
<spirit:name>myWebGenerator</spirit:name>
<spirit:parameter spirit:name=”param” spirit:resolve=”user”
spirit:id=”myParamID”>defaultValue</spirit:parameter>
<spirit:apiType>TGI</spirit:apiType>
<spirit:generatorExe>http://www.acme.com:1500</spirit:generatorExe>
</spirit:componentGenerator>
```

produces the following output.

http://www.acme.com:1500?url= http%3a%2f%2fhost%3aport&
param1=default1 &param2=fixedValue

B.3.2 TGI API

The TGI API defines the set of legal SOAP messages that can be sent from a generator to a DE, along with the format of the responses the generator can expect from a given request (message) to the DE. The API shall provide the means of getting and setting values within the IP-XACT design currently represented in the DE.

Category	Description
Get	Commands which get attribute or element values. These commands are available for getting all information from the design and component schemas. If the attribute or element does not exist, this may return a default value, an empty string, or an empty array.
Set	Commands which set element values. These commands are available to set each element for which the resolve attribute is legal. Setting the value of the element fails unless the resolve value is user or generator. Set routines return a Boolean value where a true return code implies a successful operation. If false is returned, the SOAP fault code shall provide additional information detailing the failure.
Traversal Commands	That return a list of elements
Administrative Commands	That do not deal directly with the IP-XACT meta-data.

Table I: TGI API

References

[1] Apache. *Axis Developer manual.* www.apacheaxis.com.

[2] Apache. *TCL Blend developer guide.* Apache, 2007.

[3] XML Beans. *XML Beans User Manual.* http://xmlbeans.apache.org/.

[4] Internal document. *IP-XACT Assembly tool Developer Guide.* ST Microelectronics, 2010.

[5] IEEE. Ip-xact p1685 standard, 2010.

[6] Brian Smith Ioi K. Lam. Jacl: A tcl implementation in java. Fifth Annual Tcl/Tk Workshop Boston, 2000.

[7] Magillem. *Migillem IP-XACT Assembly tool User Manual.* Magillem, 2011.

[8] Internal Document ST microelectronics. *Configurable IP generator User guide.* ST Microelectronics, 2011.

[9] Samir Palnitkar. *Verilog HDL A guide to Digital Design and Synthesis*, volume 1. SunSoft Press, 1996.

[10] Simple Object Access Protocol (SOAP) 1.1 Specification. Technical report, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[11] XML 1.0 specification. Technical report, http://www.w3.org/TR/xml/.

[12] Xpath 1.0 Specification. Technical report, http://www.w3.org/TR/xpath/.

[13] Wolfgang Ecker Volkan Esen Ulrich Nageldinger Thomas Steininger. Uml based code generation for the hw/sw interface. Design automation conference, 2008.

[14] Synopsys. *CoreAssembler E2010-09 User Guide.* Synopsys, 2011.

[15] Dan Davis y and Manish Parashar. Latency performance of soap implementations. 2002.