An XML Schema for Representing Reusable IP Cores for Reconfigurable Computing

Nathaniel Rollins, Adam Arnesen, and Michael Wirthlin
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Dept. of Electrical and Computer Engineering
Brigham Young University
Provo, UT. 84604, USA
Email: nhrollins@gmail.com, adamarnesen@gmail.com, wirthlin@byu.edu

***Abstract*— The reuse of intellectual property (IP) cores within reconfigurable computing systems is a promising approach for improving the productivity of reconfigurable system design. Further, there are a large variety of reusable IP cores available for a variety of application-specific functions. These cores, however, are created from different design tools and are difficult to integrate into a single reconfigurable system design. To facilitate the reuse of these cores, an XML schema has been created for representing the essential details of a core in a reconfigurable computing design environment. This paper presents this XML schema and describes how it can be used to facilitate reuse in reconfigurable computing systems.**

I. Introduction

There is great interest in using field programmable gate arrays (FPGAs) and other reconfigurable devices to perform application-specific computation. This form of computing, often called reconfigurable computing (RC), achieves high levels of computational efficiency by customizing a digital circuit (usually within an FPGA) to exploit the natural parallelism of the algorithm. Reconfigurable computing has been used to provide very high computational efficiency for a wide variety of applications [1].

While reconfigurable computing offers significant potential, reconfigurable computing systems are difficult to "program". Perhaps the biggest limitation preventing the widespread use of reconfigurable computing is the amount of effort required to create working RC programs. Unless design productivity for reconfigurable systems significantly increases, reconfigurable computing will be limited to the few dedicated application experts that have the skills necessary to create low-level FPGA circuits.

One important way of addressing this productivity problem is to employ more *design reuse*. Reuse has been successfully used in software engineering to significantly improve design productivity and facilitate the development of complex software systems by those with limited software experience [2]. One study has shown that under ideal conditions, design time can experience a 2× reduction for design with static cores and a 3-8 times reduction for design with parameterizable cores [3]. Successfully applying reuse within reconfigurable systems offers significant potential to increase design productivity.

This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. 0801876.

Reuse of intellectual property (IP) cores, however, is difficult. Reusing a digital circuit within an RC system requires the designer to: 1) Select the appropriate circuit core, 2) understand the details of the core, 3) create interface circuitry to integrate the core into the system, and 4) verify the core within the system. One study suggested that reuse was not cost effective unless the cost of integrating the core within a system is less than 30% the cost of creating the core from scratch [4].

Another challenge limiting core reuse within reconfigurable systems is the number and variety of tools that are used to create high-performance cores. IP cores are created in a variety of tools and languages including traditional HDLs, high-level design tools, and module generator tools such as Xilinx coregen [5] and JHDL [6]. It is difficult and time consuming to integrate cores from such a wide variety of design tools. Further, there is no standard way to integrate these cores into high-level design environments.

One way of addressing this problem is to encapsulate the details of reusable cores in meta-data that allows design tools targeting reconfigurable computing to automatically evaluate, manipulate, and instance cores within a design. To support this, the meta-data must describe the core interface, any third-party core generation tools, and core parameters. The IP-XACT specification [7] developed by the SPIRIT consortium is an example of a XML meta-data format for facilitating reuse within SoC design.

This paper presents an approach for encapsulating circuit cores with meta-data in XML. This approach is based on several concepts used in IP-XACT with customizations for reconfigurable computing. This paper will begin by providing an overview of IP-XACT and follow by discussing the differences in the goals of IP-XACT and the goals of reconfigurable computing systems. The XML schema used in this approach will be described with an emphasis on core parameterization, core interface specification, and data types. Finally, several examples will be given to demonstrate the utility of the XML schema.

II. IP-XACT

The SPIRIT Consortium developed a core description specification called IP-XACT to support reuse in system-on-chip (SoC) design and to simplify SoC design space exploration using reusable IP [7]. The IP-XACT specification includes an XML schema that defines XML elements for describing IP cores. This schema defines how the cores interface with each other and describes how external tools interact with IP-XACT compliant design tools. IP-XACT compliant tools enable designers to drag-and-drop complex IP into a design and then use third party tools to generate and verify SoC designs.

Figure 1 provides an overview of the IP-XACT approach for SoC design. Reusable cores are defined as components in XML and automatically imported into an IP-XACT compliant design environment. The designer selects IP from the component library and creates complex SoC designs with limited effort. After composing the design, *generator chains* interface with third party tools to verify and synthesize the design.

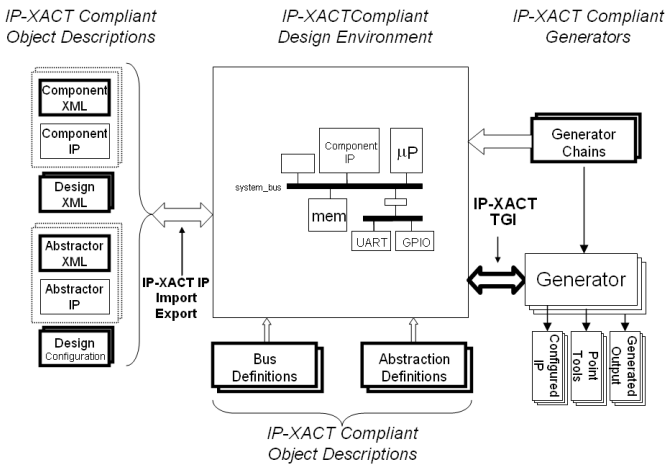

Fig. 1. IP-XACT's Design Paradigm includes bus definitions and other items that make it useful for SoC design [7].

IP-XACT facilitates the interconnection of complex IP using *bus definitions* defined in XML. An IP-XACT bus definition describes all the ports of a standard bus or IP interconnection scheme. For example, the AMBA bus definition describes the data bus, address bus, clock signals, reset, etc. used by all AMBA bus compliant cores. IP cores are created that conform to these previously defined bus definitions. Two IP cores that comply to the same bus definition can easily be connected within the design environment. The design tool can make the appropriate signal connections to insure that the devices are properly connected to the bus.

The IP-XACT XML schema for components defines a number of elements for describing details for a reusable core. Several important elements are summarized below.

1) **Naming**: A unique identifier is essential to enable proper referencing of cores. The core vendor, library, name, and version (VLNV) are the primary naming elements to uniquely identify a core.
2) **Bus Interface**: One or more bus interfaces are included in a core. Each bus interface names the type of bus the core can interface with and lists the ports on the core that are associated with that particular bus interface.
3) **Choices**: Core parameters whose values are constrained to a certain set of enumerated values are associated with a choice.
4) **File Sets**: In almost all cases there is a need for external files to be associated with a core. Multiple sets of files and their locations can be stored in IP-XACT as file sets.
5) **Model**: Models in IP-XACT list ports, model parameters and views. Ports are listed here in addition to their listing in a bus interface to facilitate ad-hoc port connections. Any parameters for a core are also listed here. Views reflect applicable file builders, associated file sets, and any environment specific parameters.
6) **Component Generators**: Generators contain information, such as paths to executable files, needed to access external tools.

The use of the IP-XACT standard for defining reusable IP cores has simplified the process of reusing cores for SoC design. Many companies now provide IP-XACT compliant design tools and a number of vendors offer compliant IP cores. The recent adoption of IP-XACT by several companies and design flows suggests that this standard will succeed in simplifying reuse-based design.

The schema developed in this study borrows heavily from IP-XACT for basic elements used in describing individual cores. Most of the elements described above are included in the XML schema that we have created. The important differences in this schema will be described in more detail in the next sections.

III. Core Reuse for Reconfigurable Computing

The goals of the IP-XACT XML schema for IP reuse in SoC design are very similar to the goals of IP reuse in reconfigurable computing. Reuse of IP in reconfigurable computing, however, is slightly different and requires additional support within the XML schema. This section will discuss some of the differences in these design environments and how these differences influence the XML structure for describing reusable cores.

In a typical system-on-chip design environment, the designer *manually* explores various system architectures by choosing a processor core, communication busses, memory, and IP to operate on the bus (see Figure 1). The communication mechanism between processors and cores is usually based on pre-defined bus protocols such as AMBA, OPB, PCI, etc. This communication between cores is relatively coarse grain and involves bus transfers using a predefined and often complex communication protocol. The IP used in SoC design is also relatively coarse grain. The IP cores are usually bus-based circuits that provide complex computing or I/O functions.

A reconfigurable computing development environment is quite different from SoC design. Ideally, the selection and composition of reusable IP is done *automatically* by a compiler

or high-level synthesis tool rather than by the designer. To support the compiler in the selection of reusable IP cores, the XML schema must provide more detailed information about the core. Information about parameters, functionality, and performance will aid the compiler in the selection of the most appropriate IP core.

The granularity of IP used in reconfigurable systems is usually more fine-grain than the IP used in SoC design. Rather than using large bus-based circuits to compose a system, a large number of relatively simple circuits such as arithmetic operators or application specific computing functions are used in a reconfigurable computing circuit. These fine-grain circuit modules are also heavily parameterized. To exploit the benefits of reconfigurable computing, most operations are heavily customized in bit-width, operating modes, exception handling, etc. The XML schema must support extensive customization within reconfigurable systems.

The communication mechanisms of IP cores in reconfigurable computing are more fine-grain and customized than the communication mechanism of cores in a SoC system. Rather than using global coarse-grain bus based communication, reconfigurable systems often rely on custom, lightweight, and distributed communication mechanisms. IP cores are often directly connected to each other with limited handshaking or protocol overhead. Because global standardized bus communication is less likely in reconfigurable systems, the notion of bus definitions is not included in this schema. Instead, custom elements are added to the schema to support description of custom interface constructs found in many lightweight communication protocols.

The XML schema introduced in this study provides a standard for IP core reuse (Figure 5). This XML-based IP core specification provides a standard way of representing IP cores, and is the primary focus of this study. Although this specification has some similarities to IP-XACT, it has significant differences. The main differences are in the representation of parameters, core interfaces, and data types. These three core details are the most challenging details to adequately encapsulate in a standard way.

IV. Parameterization

Parameters are an essential part of reusable IP cores. The flexibility that comes with parameterization enables a single core to represent many non-parameterizable (static) cores. Thus the more parameterizable a core is, the more reusable it can be. Or, the more reusable a core needs to be, the more parameterization is required.

Describing the parameters of any given IP core in a standard way is difficult since they can be complicated to describe and resolve. Consider for example the coregen FIR filter compiler which represents a parameterizable FIR filter core. This core includes over 50 parameters. Some of them are simple to resolve. For example the `addpads` parameter is either true or false and indicates whether I/O pads should be added to the circuit or not. Some of the parameters, however, are much more complicated to resolve. For example, the value of the `decimation_rate` parameter depends on the values of the `rate_change_type` and the `interpolation_rate` parameters.

A. Parameters in Reusable Cores

In general, parameters can be classified as either *simple parameters* or *complex parameters*. A simple parameter is a parameter which is resolved by the actual designer (much like a VHDL generic). For example, a port bitwidth which ranges from 2 to 16 bits is chosen by a designer to be 8 bits wide. This kind of parameter is simple and easy to describe and resolve.

Complex parameters are more difficult to describe and resolve. These kind of parameters are resolved by either some *other* parameter or by the result of some mathematical expression. For example, the bitwidth of some port called **addr** may be the $log_2()$ of the bitwidth of another port called **data**.

Parameterizable cores may have complicated interrelated parameter options and configurations. These complex relationships lead to *legal* and *illegal* core configurations. Listing all possible legal configurations created from every permutation of parameter combinations is an inadequate way to address this problem.

There more efficient ways of addressing this issue. An approach at one extreme includes all dependency information in the core representation. This means parameter dependencies are both described and resolved in the core description. The advantage of this approach is that the core description completely encapsulates the resolution of all parameters. The disadvantage is that this approach requires a complicated core description standard.

Another approach describes parameters, but does not provide parameter resolution information within the core description. This approach relies on an external tool to resolve dependencies or validate legal configurations. For example, consider again the example of the two parameters whose values represent the bitwidth of the ports **addr** and **data**. The *legal* core configuration requires that the bitwidth of **addr** to be the $log_2()$ of the bitwidth of **data**. But under this less complicated parameter approach the values of the parameters representing these two bitwidths can be independently set even if they lead to illegal core configurations. The advantage of this approach is that it greatly simplifies the core description standard. The disadvantage is that it relies on a tool to resolve parameter dependencies and validate core configurations.

B. XML Representation of Parameters

The XML schema in this study completely describes and resolves parameter dependencies in the XML. This means that the XML core descriptions are more complicated. For example, the XML schema includes a region for mathematical expressions, since many parameters are resolved by the result of a mathematical expression.

The XML schema representation for parameters is shown in Figure 2. This figure indicates that a parameter has a `name`, a `sourceName`, a data type, an optional description, and a value. The `name` of the parameter is the name that is used within the XML to refer to this parameter. The `sourceName`

is the name used to reference this parameter in the core's actual source code.

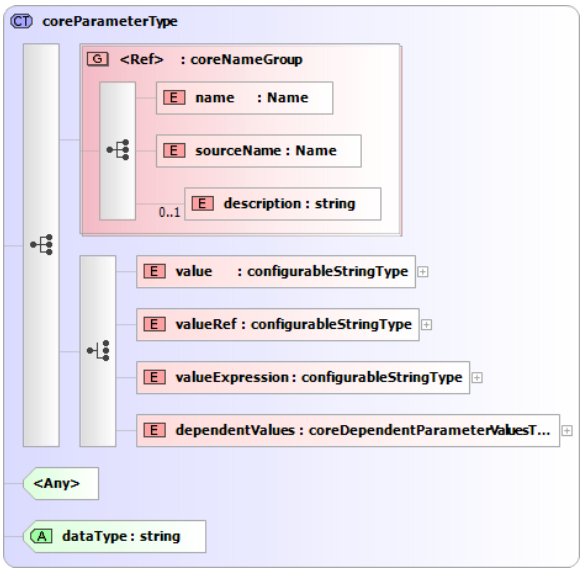

Fig. 2. The XML schema view of how parameter values are represented and resolved.

A parameter's value is expressed in one of four ways:

1) **value**: An actual parameter value (simple parameter).
2) **valueRef**: This is a label that refers to some other parameter. In other words, the parameter's value is the same as the referenced parameter.
3) **valueExpression**: This is a label that references a mathematical expression. The value of the parameter is equal to the result of the referenced mathematical equation.
4) **dependentValues**: This describes an `if-else` type of parameter dependency resolution.

Consider an example of a parameter that determines the bitwidth of a filter coefficient input port. Suppose that within XML, the parameter is referenced by the name `CSET-CoefWidth`, but in the actual core source code the parameter is called `coefficient_width`. Also suppose that the port's bitwidth value can range from 2 to 32 bits and is set to 16 by default. Such a simple parameter would be represented in XML as shown below:

```
<chrec:coreParameter chrec:dataType="unsigned int">
   <chrec:name>CSETCoefWidth</chrec:name>
   <chrec:sourceName>coefficient_width
   </chrec:sourceName>
   <chrec:value chrec:minimum="2"
       chrec:maximum="32"
       chrec:resolve="user">16</chrec:value>
</chrec:coreParameter>
```

Next, consider an example of a complex parameter. Consider a parameter that determines the bitwidth of a filter select port. Within XML the parameter is named `filterS-electBW`, but in the actual source code the parameter is called `filter_select_width`. The value of the parameter is determined by the result of a mathematical expression called `filterSelectBWExpr`. This expression calculates the $log_2()$ of a parameter called `CSETCoefSets` whose value represents the number of filter sets to select from. This parameter and its associated mathematical expression are represented in XML as shown below:

```
<chrec:coreParameter chrec:dataType="unsigned int">
   <chrec:name>filterSelectBW</chrec:name>
   <chrec:sourceName>
     filter_select_width
   </chrec:sourceName>
   <chrec:valueExpression
       chrec:resolve="generated">filterSelectBWExpr
   </chrec:valueExpression>
</chrec:coreParameter>

...

<chrec:expression>
   <chrec:name>filterSelectBWExpr</chrec:name>
   <chrec:returnType>int</chrec:returnType>
   <chrec:expressionString>ceil(log2(CSETCoefSets))
   </chrec:expressionString>
</chrec:expression>
```

Among the most challenging details to encapsulate in an IP core description standard are parameters and their dependencies. The XML schema introduced in this study fully describes and resolves these complex dependencies within the XML core description.

V. Interface Specification

One of the biggest challenges to describing IP cores in a standard way is expressing core interface information. Core interface descriptions must provide all the necessary information for a hardware designer or design tool to correctly interconnect reusable cores. To address this challenge, existing core reuse specifications constrain core interfaces to a bus-based, socket-based or some other standard interface[7], [8]. When core interfaces are constrained to a given interface standard, core interconnection is greatly simplified. Interface-constrained cores can simply connect in a Lego-like manner.

In order to achieve independence from pre-defined specifications, the amount of information needed to completely specify an interface increases dramatically. The amount of information needed is vast: connection of datapath signals, correct timing and interface of control signals, proper distribution of clock and reset signals, presence or absence of optional ports, dependencies between signals, grouping of signals and much more. Our XML schema only includes a small subset of the needed information including Optional Ports, Signal Dependencies, and Port Grouping.

A. Optional Ports

Some IP cores contain ports or sets of ports that are present or absent depending on a parameter's value for that core. For example, a coregen divider that supports both fixed-point and floating-point operations has different interface ports depending on the desired type. Our schema is able to represent individual ports as well as groups of ports that are added to or removed from a design based on a parameter. The following example shows how the presence of a group of optional ports is described in XML.

```
<chrec:coreInterface>
  <chrec:optionalPorts>
```

```
    <chrec:dependentParameter>
      <chrec:parameterName>
        CSETDataType
      </chrec:parameterName>
      <chrec:whenValue>
        Fixed
      </chrec:whenValue>
    </chrec:dependentParameter>
  </chrec:optionalPorts>
      . . .
<chrec:coreInterface>
```

In the above example, the ports that are described in this core interface will be present in the core only if the value of the parameter `CSETDataType` is equal to `Fixed` meaning that the user wants a fixed point implementation of the core. The ability to dynamically determine the presence of a port is essential to interfacing cores.

B. Signal Dependencies

Signal dependencies are part of the basis of timing information that must be included in a port's interface information. A particular port may be synchronously or asynchronously dependent on another signal. For example, the data on a port may only be valid when another signal has a particular value or data may need to be introduced to a port only when a related signal is asserted. For a port called `DIN`, dependency information is represented in XML as shown below:

```
<chrec:synchronousDependency>
  <chrec:clockName chrec:active="rising">
    CLK
  </chrec:clockName>
  <chrec:dependentSignal chrec:active="high">ND
  </chrec:dependentSignal>
</chrec:synchronousDependency>
```

In the above example the data signal `DIN` is valid only when the control signal `ND` is high and when the clock signal `CLK` is on its rising edge. The ability to represent inter-dependencies between signals is important when these interfaces are to be connected dynamically by a compiler or another design environment.

C. Port Grouping

The schema groups ports in XML according to a basic classification system. Ports are grouped in this way to enable tools and compilers to deduce basic information about the function of a port simply because of its grouping. For example, all clock ports are listed in a clocks group. This enables a tool to search only one section to find a port with the needed clock properties instead of having to search and test every port. The schema classifies ports as shown in the following example.

```
<chrec:coreInterface>
  <chrec:name>main</chrec:name>
  <chrec:clocks></chrec:clocks>
  <chrec:resets></chrec:resets>
  <chrec:controlSignals></chrec:controlSignals>
  <chrec:dataSignals>
    <chrec:data></chrec:data>
    <chrec:address></chrec:address>
  </chrec:dataSignals>
</chrec:coreInterface>
```

As seen in the example, ports are classified as clocks, resets, control, or data. Further, a data signal may be classified as data or address. This classification of ports groups the ports in all interfaces. This simple classification, allows a tool to infer properties about interfaces that need not be explicitly listed in XML. For example, a port in the clock section does not connect to a port listed in a reset section.

In order to be independent of pre-defined interface standards, the schema faces the challenge of representing a vast amount of information about each port and interface on a core. The schema addresses an initial subset of this information by addressing optional ports, signal dependencies, and port grouping.

VI. Data Types

The types associated with ports for SoC design are *bit-based* types (i.e. aggregates of bits such as `std_logic_vector` in VHDL). While these types adequately represent the low-level hardware, they do not represent the high-level type information used within compilers. When IP cores are instanced by a high-level compiler, raw bit-based signals do not provide enough information. Instead, detailed *high-level* type information is needed for every data signal. These types include integer types, fixed-point types, floating point types, and other composite types.

In addition to the need for high-level types for compilers, this level of typing is needed to correctly connect cores at a low level. This typing information is important when connecting two cores that have different types on ports that must be connected. For example, if a floating-point port on a core needs to be connected to a fixed point port on another core, there will need to be a data type conversion and a re-mapping of bits from floating to fixed point. Data typing is essential to facilitate this remapping.

Within the XML schema proposed in this work, information for both the low-level bit-based hardware types as well as the high-level types is needed in the IP core. The type information would indicate the exact bit-based representation of the signal (i.e. bitwidth of the given type including fixed point, integer, floating point, bit, Boolean and any application-specific type). It is also important to include support for parameterizable types that are typically not supported in traditional compilers (i.e. parameterizable integers, floating point, etc.).

To address this issue, the schema include elements that provide support for mapping high level and parameterizable data types to their associated bit-based representation. For example a customizable fixed point data type would be described in XML as explained in Table I.

In Table I the element `chrec:fixedPoint` signifies on a high level that this is a fixed-point type. The details of this type are completely described by a bitwidth and a fractional width, in this case a 16 bit fixed-point number with a 13 bit fractional part. The bit width could also be a parameter reference and resolve to that parameter's value. Similarly, the fractional bitwidth can be a parameter reference which resolves to parameter value. On the bit level this type is implemented with a `std_logic_vector` whose type is defined by `ieee.std_logic_1164`.

This XML implementation of typing allows communication and mapping between high-level and bit-based types. With this

TABLE I

XML ELEMENTS AND THEIR VALUES DESCRIBING BOTH HIGH-LEVEL AND BIT-BASED TYPES.

Typing Information in XML	
XML Element	Data
Bit Level Type Information	
chrec:sourcename	std_logic_vector
chrec:typeDefinition	ieee.std_logic_1164
chrec:bitWidth	16 (or param)
High Level Type Information	
chrec:fixedPoint	(element defines type)
chrec:fracBits	13 (or param)
chrec:bitWidth	16 (or param)

information, a compiler could interpret the type of an interface simply as a high level type without being concerned with the bit level implementation. This information also facilitates bit level conversion of types in the interconnection between ports on different cores. By defining high-level and their associated bit-based types in a single segment of XML, the schema provides support for high level compilers as well as low level verification and synthesis tools.

VII. EXAMPLES

In order to demonstrate the ability of this schema to represent cores from different environments in a standard way, this paper includes an example of a coregen FIR Filter core and a JHDL QPSK core that implements the carrier phase PLL and rotation (called `Carrier_Phase_PLL_and_Rotate`) that have been imported into XML. This demonstration includes creation of a small library of cores by expressing these cores and several other cores from different environments in XML. A basic Design Tool is created to demonstrate core parameter manipulation. Some generators are also created to demonstrate communication between XML core representations and third party tools.

A small library is created by expressing several existing IP cores in XML. The library consists of cores from many different environments. Figure 3 shows how the XML schema presented in this study enables cores from different environments to form a core library. After IP cores are imported into the XML core library, cores from all environments have a uniform representation. In this study, the library of XML cores consists of cores from VHDL, Verilog, Impulse C, coregen, JHDL, and System Generator.

In addition to representing cores from many different languages, a simple design tool was created that parses core XML and creates a custom GUI for each core which provides an abstracted view of the core. Figures 4 and 6 are examples of cores loaded into the design tool. On the top right of the GUI is listed the name of the core as well as keywords that help to define its function for the user. On the left is a panel which provides the user with access to parameters that can be set. If desired the user can create a new instance of the core with customized parameters and then export the schema-compliant XML as an *instance* of the original core. After manipulating parameters the user can use generators to create implementation files that are needed to use the modified instance in a design.

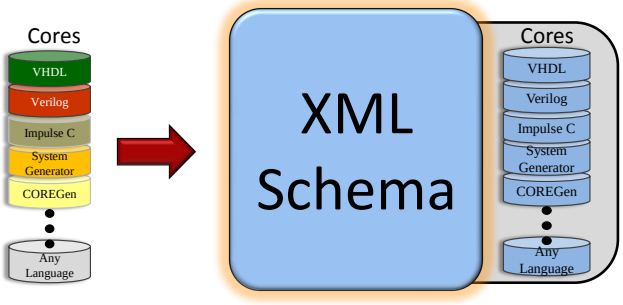

Fig. 3. Cores from different languages and environments are imported into the standard form represented by the schema.

A. Xilinx coregen FIR Filter

The Xilinx coregen [5] FIR Filter is an example of a complex core. It contains multiple complex parameter dependencies, dependencies connected to mathematical expressions, and optional ports. Figure 4 shows the custom GUI that is generated by the design tool for this core. Note that in the figure many of the ports are not present in the pictured configuration and are represented by empty boxes.

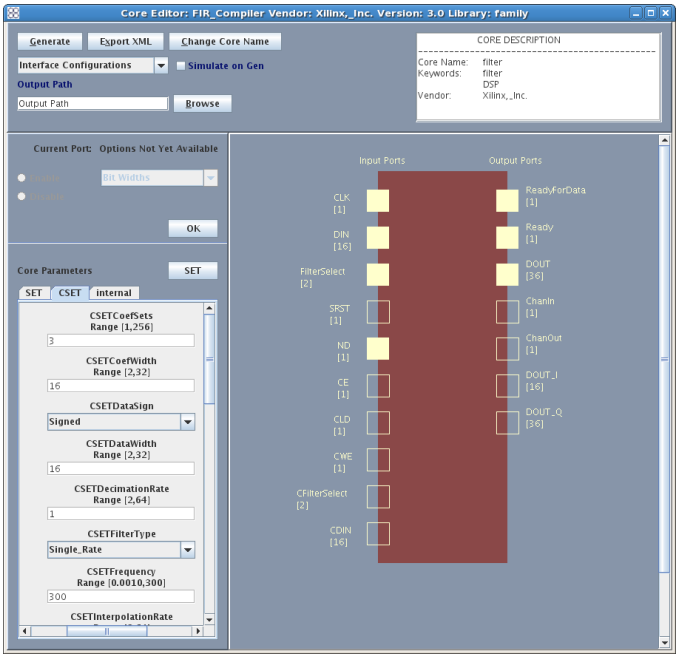

Fig. 4. The GUI core manipulation tool functions as a design tool, interfacing with core XML and external generators. Here is shown a coregen Filter core. Note the complexity of the core and the absence of some optional ports.

After the core XML is loaded into the GUI design tool, the user is able to change the parameters listed as user resolvable in the parameters section of the XML. As these parameters change, the GUI updates the visual representation of the core, enabling or disabling ports, as the parameters and their

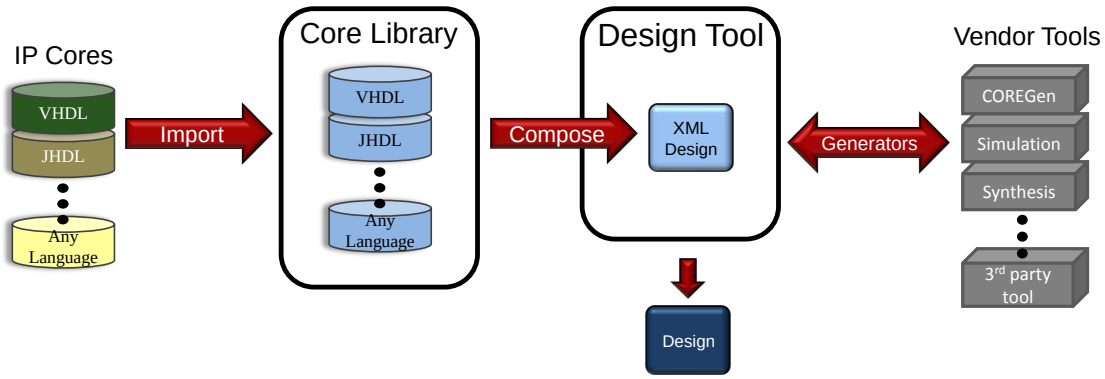

Fig. 5. The XML schema allows a design tool to compose designs from cores taken from a library of reusable XML cores. The schema also facilitates communication between the design tool and third party vendors.

accompanying mathematical expressions require. Parameters not directly affecting optional ports, such as filter bandwidth, sampling rate, and number of filter coefficients, can also be set.

When all parameters have been set to the desired values for a particular implementation, the user clicks the "Generate" button. The design tool knows what external tools it has access to and when it receives the generate command it reads the core's XML to determine which of those tools should be run to implement this core. In this case a VHDL wrapper for the core is first generated, the design tool then calls the coregen executable. After coregen has generated the instance of the core the design tool then synthesizes the core and generates a bit stream using the Xilinx ISE.

B. JHDL Carrier_Phase_PLL_and_Rotate

The JHDL [6] `Carrier_Phase_PLL_and_Rotate` is a much more simple core than the FIR filter. However, it is important that the schema supports this core as it demonstrates that the tools that implement this schema can be completely language independent and still operate on any core that complies to the schema. Supporting this type of core contributes to the development of the schema by requiring extended support of external files and variables.

Figure 6 shows the JHDL core in its custom GUI. Note that there is only one user-resolvable parameter, a bitwidth, and therefore the customization of this core is simple. By simply changing this parameter in the XML, the design tool is able to use the schema to tell the JHDL compiler how to set that parameter in its implementation. Similar to the filter, when this core is generated a VHDL wrapper is created along with a netlist and synthesized bit stream.

In both the coregen Filter and the JHDL `Carrier_Phase_PLL_and_Rotate`, the design tool is able to interpret the details of the core, provide user access to needed parameters, present the user with an abstracted view of the core, and ultimately create usable implementation files. This demonstrates the ability of the schema to represent cores from

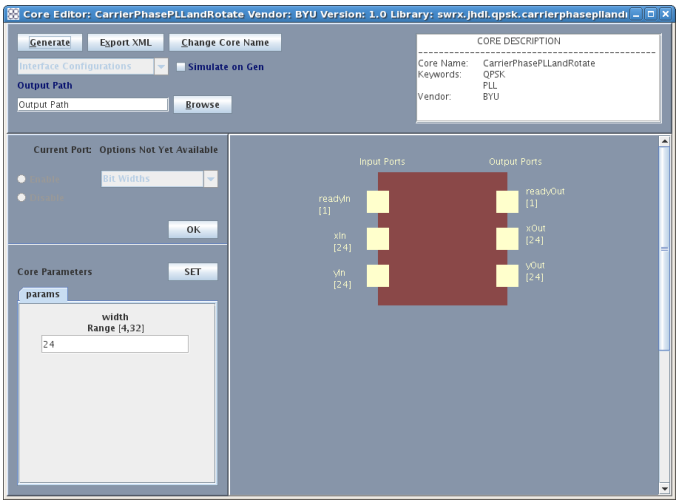

Fig. 6. Custom GUI displaying here the JHDL Carrier Phase PLL and Rotate. Note the simplicity of this core compared with Figure 4

different environments, their complex parameter dependencies, and other details and communicate these to a design tool independent of the actual implementation of the core.

VIII. Conclusion

This paper introduces an XML-based approach to encapsulating the details of reusable IP cores for reconfigurable computing. This work builds upon the IP-XACT specification [7] which describes IP cores for SoC design. The XML schema introduced in this study differs from IP-XACT in the way that it addresses core interfaces, data types, and parameter dependencies.

This study demonstrates how the XML schema introduced can be used to represent cores from different environments. These XML cores form a library of reusable cores. This study also demonstrates how a generic design tool can use the XML schema to manipulate cores and communicate with third party tools.

Future studies will demonstrate how this specification can be used to enable design composition. Figure 5 shows how a generic design tool could be used to create designs built from cores imported from an XML core library. The design tool would also communicate with third party tools in a standard way in order to simulate and synthesize designs.

Future studies will also investigate how this specification can be used to enable interface synthesis. In order for IP cores to be used together to create a design, their interfaces must be correctly interconnected. Future studies will investigate how this interconnection can be automated by a design tool or high level language.

REFERENCES

[1] Scott Hauck and Andre DeHon ed. *Reconfigurable Computing: The Theory and Practce of FPGA-Based Computing*. Morgan Kauffman, 2007.

[2] B.W. Boehm. Managing software productivity and reuse. In *IEEE Computer*, volume 32, pages 111–113, September 1999.

[3] Annette Reutter and Wolfgang Rosenstiel. An efficient reuse system for digital circuit design. Technical report, University of Tubingen, 1999.

[4] Roberto Passerone and James A. Rowson. Automatic synthesis of interfaces between incompatible protocols. In *Proceedings of the 35th Design Automation Conference (DAC 1998)*, pages 8–13, June 1998.

[5] Xilinx, Inc. *CORE Generator Help*, 2007.

[6] P. Bellows and B. Hutchings. JHDL - An HDL for reconfigurable systems. In *IEEE Symposium on FPGAs for Custom Computing Machines*, page 175, 1998.

[7] SPIRIT consortium. *IP-XACT v1.4: A specification for XML meta-data and tool interfaces*, 2008.

[8] Wolf-Dietrich Weber. Enabling reuse via an IP core-centric communications protocol: Open core protocol. Technical report, Sonics, Inc., 2000.