FPGA Vendor-agnostic IP-XACT- and XSLT-based RTL Design Generator

R. Machado, S. Pinto, J. Cabral, A. Tavares, J. Monteiro
Centro Algoritmi - University of Minho
{rui.machado, sandro.pinto, jorge.cabral, adriano.tavares, joao.monteiro}@algoritmi.uminho.pt

***Abstract*—The growing complexity of current embedded systems increases not only the time-to-prototype and time-to-market, but it also requires a major effort around repetitive engineering tasks in order to maximize the efficiency and minimize the money investment. A lot of research has been done on this field, leading system development, test automation and system reutilization to huge relevance and considerable importance in industry and academia. Using eXtensible Markup Language (XML) files to store Intellectual Property (IP) metadata, the IP-XACT standard arises as a possible solution for IP reutilization and vendor independence.**

This paper describes a RTL design generator that uses IP-XACT components description and apply XSLT transformations for complete system generation, following a generative programming (GP) approach while automating the design flow through the integration and interoperability of external tools needed to design, implement and finally deploy the final system under the chosen FPGA board. The aim is to provide a unified and easy to use interface for code generation and deployment independent from FPGA vendors, i.e., fostering vendor-agnosticism.

***Index Terms*—HDL code generator, IP-XACT, XSLT, XML, process automation.**

I. Introduction

Embedded systems have rapidly grown both in dimension and complexity over the past few years, making design, implementation, verification and validation into very hard and complex tasks [1], [2]. This increasing complexity of the development process leads not only to a larger prototyping development time which ultimately, can result in missing time-to-market, but also to an urge for higher levels of abstraction, which most often implies the combination of multiple frameworks, each specialized in different phases of the design flow [3].

The reutilization of verified IP is frequently addressed as a possible solution for complexity issues, allowing companies to share the costs and risks associated to IP creation and validation and reducing duplicate development efforts [3], [4]. However, the integration process of different IPs is itself prone to errors and needs to be validated. Some existent works in academia and industry present frameworks capable of automating the interconnection process to reduce verification efforts, and IP-XACT standard is usually pointed as a solution for IP reutilization [1], [2], [3], [4], [5], [6]. Simultaneously, embedded product development and maintenance can require porting between platforms [2].

One possible approach to address the aforementioned issues, is to develop mechanisms to automate the code generation process and abstractions that can offer to the developer a uniform interface independent from the FPGA vendor. Such interface can reduce not only errors during the integration and porting stage but also reduce the time-consumption from validation activities and relieves the need for multiple vendor frameworks knowledge [3], [5].

This paper describes a code generator fully independent from vendors (e.g., including mapping the FPGA design to chosen board pins as described in the .ucf files) and easily extensible, based on XSLT, to interpret and convert IP-XACT files into HDL code. As result, this work utilizes a set of XSLT files capable of interpreting an IP-XACT design and generating the respective Verilog code as well as associated Constraint files and a mechanism that identifies the target FPGA and automatically generates a set of batch files to synthetize the code and program it. The use of XSLT makes it possible to achieve an IP-XACT to Verilog converter that only depends on the standard. As the communication process between the framework and the remaining external tools is completely transparent to the developer, the code generator offers a single and simple interface to every vendor FPGA platform. The presented code generator is part of a bigger project that fully implements an IP-XACT enabled framework.

The remainder of this paper is structured as follows: Section II describes, briefly, the IP-XACT standard, and points and outlines also some related work. Section III focus on the adopted methodology, describing the mapping between IP-XACT and Verilog as well as the research performed in order to support the external tools used. Then, Section IV details and discusses the code generator implementation, and its XSLT files creation for the communication between the generator and the external tools needed to program the target FPGA. Section V describes the evaluation process, presenting some results. Finally, Section VI concludes, pointing also some future improvements.

II. Background and Related Work

A. IP-XACT Overview

The IP-XACT standard [7] documents IP used in the development, implementation and verification of electronic systems with metadata through XML files. IP-XACT is independent from any design process and does not contemplate behavioural characteristics related to the IP that are not relevant to integration. The purpose of IP-XACT standard is to provide well defined and unified metadata about the components and designs

composing electronic systems, making it possible to easily import and export IP between Electronic Design Automation (EDA) tools from multiple IP vendors. IP-XACT also supports the automation of the design flow where different tools are used through generators. It encapsulates information about the component interface and communication style (protocols) that can be used for validation during IP integration [6]. Even though it does not specify IP behaviour, it is possible to attach to the metadata files containing the IP behaviour (e.g., HDL files).

IP-XACT schema representation relies in seven top schema definitions [7]. The bus definition describes the high-level attributes of the interface, whereas the abstraction definition describes the low-level attributes of the interface associated to specific bus definition like the number of ports, their direction and width. The component element is used to describe any type of IP such as cores, peripherals or buses like networks. An IP-XACT component can be hierarchical to integrate in its definition other IP-XACT components or be a leaf component otherwise. Although a hierarchical component is composed by many other IP-XACT components its IP-XACT description need only be a leaf object as it fully describes the component. The design description is another of the top seven schema definitions and complements the information related to hierarchical components. It gathers information relative to the instantiated components, their connections between them and their configuration. The IP-XACT abstractor defines the interconnection between two bus interfaces that share the same bus definition but different abstraction definition. To represent a specific design flow, IP-XACT uses the generator chain object which is based on tasks, where each task can be a single generator or a reference for another generator chain. Finally, the design configuration is the IP-XACT object used to gather additional information relative to a design or generator chain. This object is particularly useful when porting IP-XACT designs between design environments.

IP-XACT has been design with HDL descriptions in mind and, as result, does not directly support software objects as well as communication interfaces between hardware and software objects. As result many works has been proposing extensions to the standard in order to overcome this limitation [5], [6].

B. IP-XACT-based Tools

Kactus2 [1], [8], [9] is a framework that aims at enabling software and hardware engineers "to speak the same language", and so, helping during the description of system architecture. It focus on simplifying design flow through IP reutilization by using IP-XACT metadata for describing IPs which will leverage the interoperability between platforms and tools. This framework also presents some standard extensions, using the VendorExtensions field available in the standard [5], in order to represent software IPs in metadata form. In this way it is possible to efficiently improve the communication between Hardware and Software Engineers.

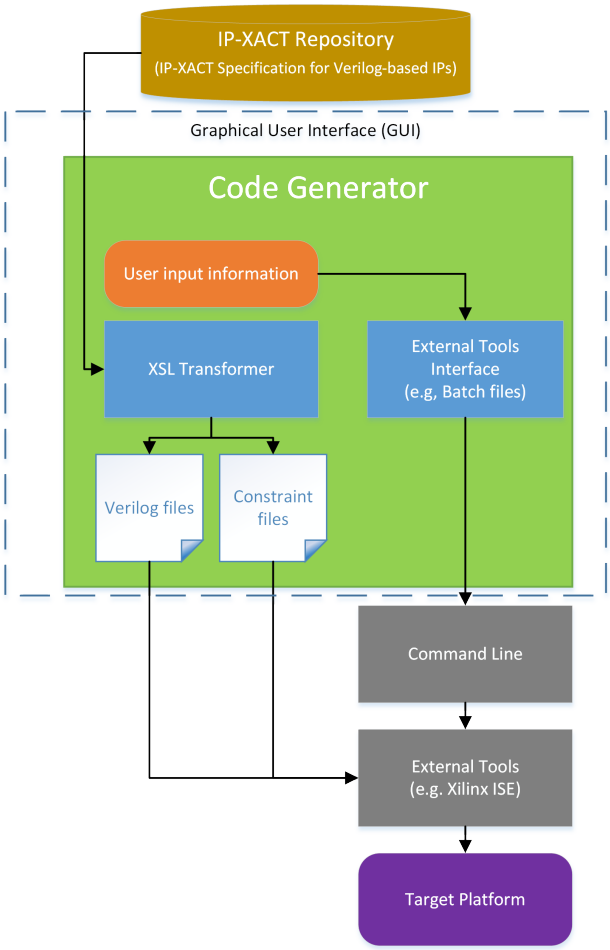

Fig. 1: RTL Coding Flow: The Code Generator Block Diagram

Synopsys coreTools [10] is a set of tools for IP packaging used in a knowledge base design and verification flow. According to the datasheet, it is possible to achieve an improvement of 60% of the time when designing the system using an IP-based design and verification flow with IP packaged for assembly. The coreTool family is composed by: (i) coreBuilder, a packaging tool; (ii) coreAssembler, an IP assembly tool that automatically generates the interconnect and configured Register-Transfer Level (RTL); (iii) and coreConsultant, an utility tool to configure, implement and validate individual IP.

MAGILLEM 4.0 [11] is a java-based plug-in for Eclipse that encompasses a set of tools to manage an IP-XACT based database and a whole design environment for IP creation and reutilization.

III. Methodology

Fig. 1 depicts the block diagram of the developed code generator tool. Accordingly to that, three modules mainly compose the code generator: (i) the *XSL transformer* consisting of a set of XSLT files that convert the *'IP-XACT Specification for Verilog-based IP'* into Verilog files; (ii) a transparent *external tools interface* based on a set of auto-generated batch files; and (iii) a *graphical user interface* where

the user can select the vendor and target FPGA device. Once the order to generate the code is issued, the code generator queries the IP-XACT repository to obtain all related *'IP-XACT Specification for Verilog-based IP'* that will be processed by the XSL transformer to generate the set of Verilog files, the design' associated constraint files and the needed batch files for the interface with the external tools which depend on the selected vendor and FPGA device. Then the batch files are executed through command line interface to run the external tools. Notice that the batch files already come with references pointing to the previously created Verilog files, allowing the external tools to access them. As result, a bitstream is obtained as well as a list of the available FPGA devices connected to the host computer, and the user can then select the one she/he wants to program.

Two different methodologies were considered to implement the code generator. The first one was based on the classes that represent the IP-XACT standard. This option, although easier to implement, makes the code generator highly attached to the created framework. Kactus2 uses this approach, but one of the aims of the developed framework in which this generator is included is to be as modular as possible. The use of XSLT makes the code generator completely independent from the development environment as well as from the developed application, offering great flexibility if structural changes are made to the standard. For these reasons and also because the standard is represented through XML files, we decided for the XSL transformer to implement the code generator.

A. XSL Transformer

As XSLT becomes more mainstream and supported by various OS (Operating Systems) and programming languages, this makes the code generator highly portable. Any framework can also use the code generator as it is completely independent from the framework architecture (i.e., loosely coupled to the framework architectures), although tightly-coupled to the standard.

XSL files are largely used to filter the data and the way to display them in Hyper Text Markup Language (HTML) pages, making it possible to create different pages with the same XSL file by only changing XML file content. Another advantage of using XSLT is the fact that there is no need to recompile the application every time the files are changed. That means that if the standard for some reason is upgraded, it is only needed to change the XSLT file to handle the changes without having to recompile the whole code generator.

The XSL transformer, as composed by the set of XSLT files, was created in a way that only the information given by the standard suffices to successfully generate the Verilog files of either leaf components or hierarchical components and to successfully instantiate all components and map all interconnections presented in an IP-XACT design.

B. External tools interface

In order to have the most transparent interface between the user and the external tools, the Xilinx and Altera tools interface via command prompt was studied. According to Xilinx command prompt user guide [12], [13], the design flow is divided in three main phases: (i) synthesis; (ii) implementation; (iii) and verification. Each of these phases includes one or more Xilinx tools to be executed. For Altera, the software Quartus II is used to develop the FPGA projects which has a tool for each design flow phase [14].

IV. Implementation

The framework chosen to develop the code generator was Visual Studio and the implementation language was C#. Such decision was made based on a time/effort relationship since Visual Studio was a well-known platform with a huge amount of information and libraries available. Additionally, it has a very powerful debug tool and offers mechanisms for version control and project planning.

The code generation flow follows essentially three main steps: (i) the XSL transformer development based on several stylesheets for IP-XACT components and designs; (ii) the constraint files creation, which are dependent from the target platform; (iii) and the target's abstraction implementation through automatic batch files generation and external tools invocation.

Converting a non-hierarchical IP-XACT component into its respective Verilog file is very straightforward. As previously described, the standard only specifies interfaces, so it is not possible to automatically generate the component behaviour code, instead only the code interface can be automatically generated. The flowchart presented in Fig. 2 illustrates how the process is done. First, the module is instantiated with a name given by values in the name plus version fields from the standard. Then all the interfaces in the XML file are defined in Verilog. After that, the details about the interfaces are extracted and the ports declared with the direction and number of bits of each port. Finally, the `endmodule` keyword ends the process.

Contrarily, converting an IP-XACT design, which represents a hierarchical component, is not so straightforward. An IP-XACT design can be compared to a graph where the `componentInstances` elements are vertices and the `interconnections` the edges. In other words, the problem is that there is no particular order in the IP-XACT files to represent the different connections, however that order is very important during the conversion process to ensure that no duplicate connections occur. Fig. 3 illustrates the aforementioned issue. The example depicts a design with four components, each one with an inout data bus. Component 1 is connected to component 2, and then component 2 is connected to component 3 and 4 by the same port. This means that component 1 is also connected to component 3 and 4. In this situation only one wire port is enough to connect all the four components. To identify these relationships, the IP-XACT files need to be read more than once. This guarantee that no more auxiliary ports, then the needed ones, are created. Due to XSLT limitations, the solution to these issues was to use the function and sequence elements from XSLT 2.0. These two elements combined enabled running the file recursively saving the return

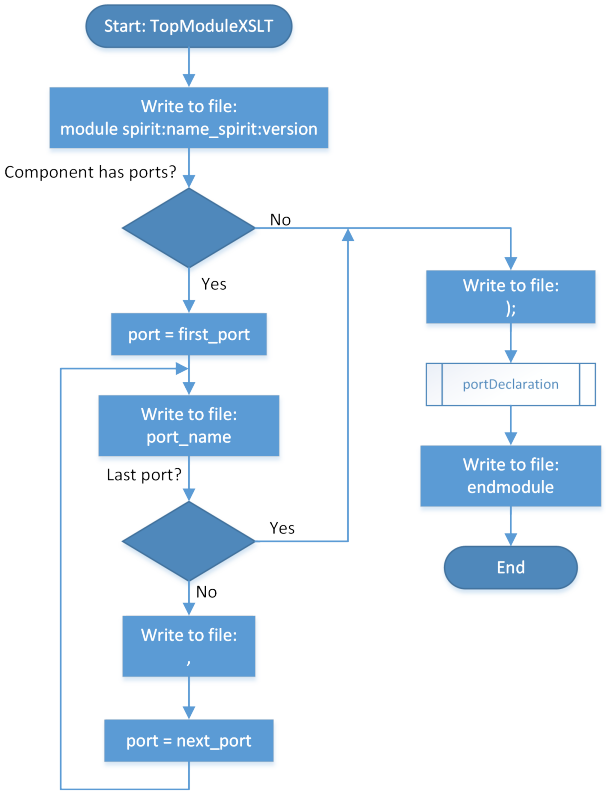

Fig. 2: IP-XACT-based IP to Verilog Transformation Flowchart

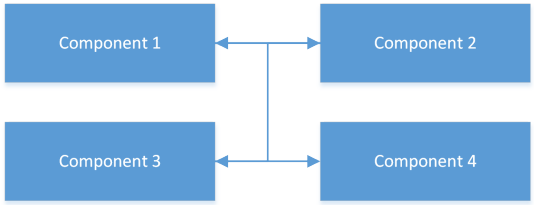

Fig. 3: Multiple Port Connection Example

values from one invocation to another. As Visual Studio only supports XSLT 1.0, a plug-in from Saxon was installed in order to support XSLT 2.0.

In addition to the Verilog files, it is necessary to have a set of constraints, required to map the design ports to the respective FPGA board pins, for example. Depending on the FPGA vendor, this can be achieved in different ways. For example, Xilinx uses a .ucf file to define the system constraints while Altera, more properly, the Quartus II framework, defines these constraints in the same file used to create the project. Depending on the vendor, a XSL stylesheet is accordingly applied to the IP-XACT component file that filters all its ports and instantiate them. Then the user just has to complete the instantiates with the desired FPGA pin.

```
‘timescale 1ns/1ps

//This document was generated by IP−XACT Enabled Framework

module Controller(
        CLk_clock,
        RST_reset,
        InputValue__a,
        FeedBackSignal_a,
        PIDControl_result,
        KdOut_result,
        KiOut_result,
        KpOut_result
);

                ...

P P(
        .clock(CLk_clock),
        .reset(RST_reset),
        .Kp_SetValue(converter1632_generic32_result),
        .Kp_we(),
        .Error(subbError_generic32_result),
        .P_control_Signal(P_generic32_P_control_Signal),
        .KP(P_generic32_KP)
);

I I(
        .clock(CLk_clock),
        .reset(RST_reset),
        .error(subbError_generic32_result),
        .ki_we(),
        .kiSetValue(converter1632_generic32_result),
        .error_max_setValue(converter1632_generic32_result),
        .error_min_SetValue(converter1632_generic32_result),
        .error_max_we(),
        .error_min_setvalue(),
        .samplingTime(divider1_generic32_result),
        .I_control_signal(I_generic32_I_control_signal),
        .ki(I_generic32_ki),
        .PID_enable()
);

D D(
        .clock(CLk_clock),
        .reset(RST_reset),
        .Kd_SetValue(converter1632_generic32_result),
        .Kd_We(),
        .Error(subbError_generic32_result),
        .Sampling_Time(divider1_generic32_result),
        .D_control_signal(D_generic32_D_control_signal),
        .Kd(D_generic32_Kd),
        .PID_Enable()
);
```

Listing 1: Generated Verilog Code

Batch files are created to automate external tools invocation for the compilation and synthesis of the generated HDL code. The invoked external tools depend on the selected platform, so a different batch file is generated accordingly for each platform vendor. These batch files are generated from an IP-XACT generator chain file which stores information about the executable files and the parameters needed to compile and synthesize a design under a given platform. Such approach makes the proposed code generator completely vendor-independent, and supporting another vendor is dictated only by the creation of another IP-XACT generator chain file without changing the source code of the generator (i.e., the code generator is vendor-agnostic). Log files are generated to inform the user about the output of each compile phase. This way the user can have feedback about which phases were completed and which are still running.

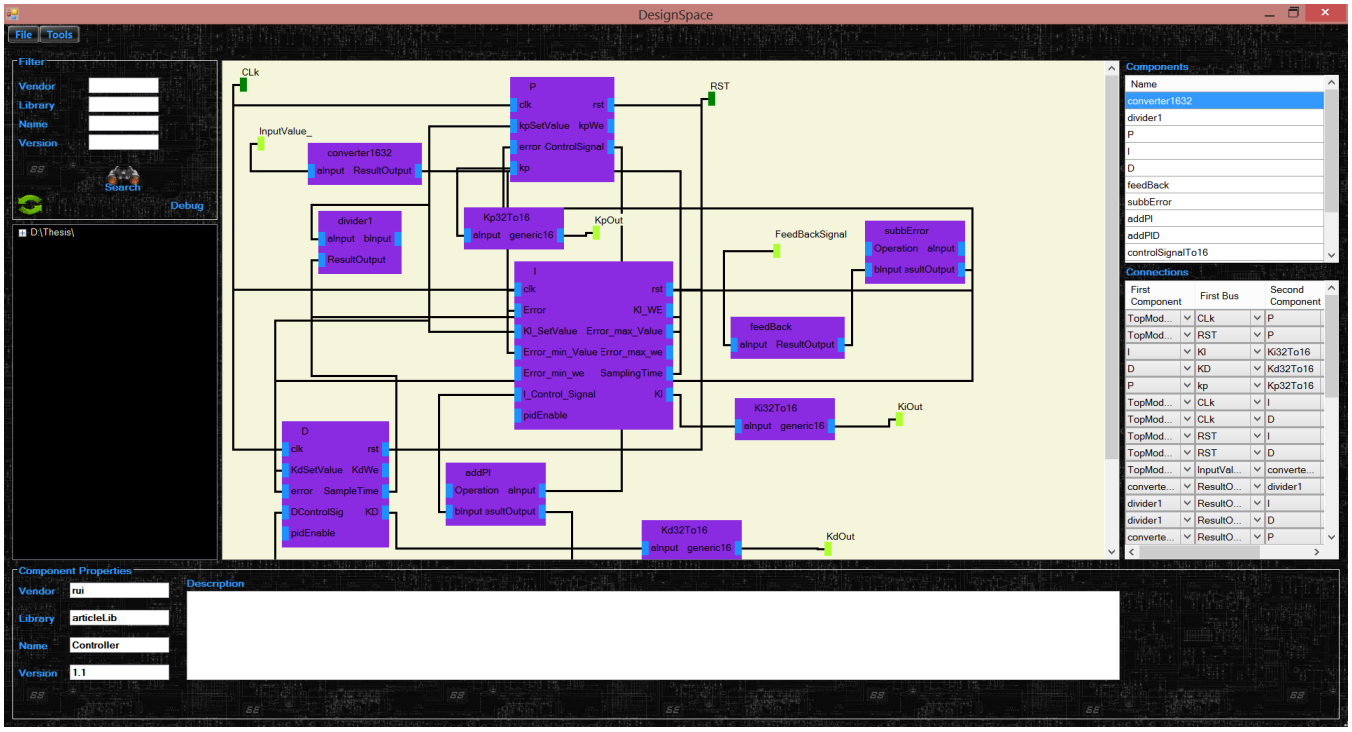

Fig. 4: PID Controller Design

The implementation of external tools invocation was divided into two phases. The first one is responsible for generating the bitstream for the FPGA. At the end of this phase the developer can analyse the feedback provided by the framework or the log files, and accordingly fix the errors pinpointed during the compilation process. The second phase consists in burning the bitstream into the target FPGA. The framework identifies which targets are available from the chosen vendor and after the user selects the desired FPGA the programming option becomes available. The splitting between the compilation and the programming phase was implemented using two different batch files. One with all the commands for each design step of the flow, and another one with only the commands responsible for programming the FPGA with the desired bitstream.

V. Evaluation

In order to test the automatic code generation, a proportional-integral-derivative (PID) controller as presented in [15] was modeled using the framework that encompasses the IP-XACT- and XSLT-based code generator. The test consists in modeling the PID controller and then generate the code representative of its top module. The obtained result should be similar to the top module manually implemented in [15], the bitstream correctly created and then deployed on the target FPGA platform.

The platforms used to test the designed system were Basys2 from Xilinx and the Altera DE2-70. Basys2 has a Spartan3E FPGA, is compatible with every version of Xilinxs ISE and is integrated into the ADEPT programming application from Diligent. Altera DE2-70, for instance, has an Altera Cyclone II 2C70 FPGA and is supported by Quartus II software and programmed through USB Blaster.

Each module must specify its associated IP-XACT representation (i.e., *'IP-XACT Specification for Verilog-based IP'*) that will be generated according to the attributes of component Verilog file. After having the components on the repository they can be instantiated in the design creator (see Fig. 4) and the respective mapping between components can then be made. Any attempts to connect two buses with a different interface type will be automatically aborted and the user notified. After finalizing the design two new IP-XACT files are created, one for the design and another one to the component of the design.

The code can then be generated. To do so, the FPGA (vendor and target) is selected (Fig. 5) and as result of this operation two files are created, one Verilog file representing the top hierarchical module of design and the other with the ports that have to be mapped to the respective pins of the target FPGA (e.g., ucf). These files are placed in a specific directory named project, along with a batch file that contains the paths to all the components' Verilog files included in the design. The result is a PID controller virtually similar to the one presented in [15], as the Verilog code is exactly the same except from the mapping between components that was generated automatically and presented in Listing 1).

VI. Conclusion

The IP reutilization combined with vendor IP independence fosters a reduced verification efforts while allowing companies

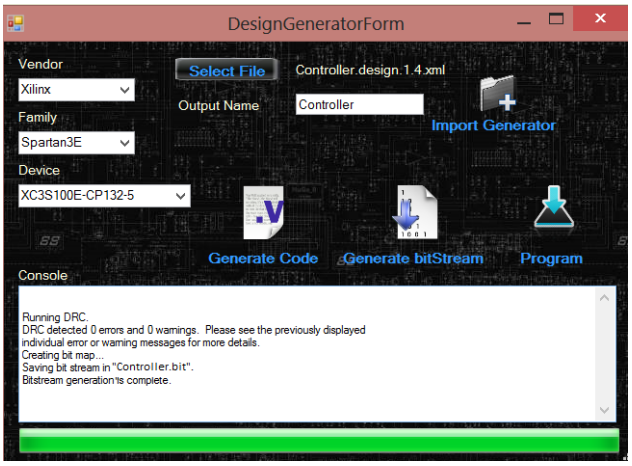

Fig. 5: Code Generation Interface

to share development risk that can be critical to their sustainability. Moreover, the automatic code generation contributes to an integration process less vulnerable to errors. Combining the two above factors will simplify both system creation and maintenance.

In this paper a flexible code generator was presented, which is integrated in a complete IP-XACT enabled framework for IP creation and reutilization. Adequately populating the IP-XACT repository of the developed IP-XACT enabled framework will reduce the time-to-market pressure for the development of higher complexity systems with practically no errors. The code generator proved to perfectly manage the mapping between Verilog components, reducing the effort and risk of the integration task and at the same time offers great flexibility in terms of platform and framework independence as well as in terms of future upgrades. Some frameworks already use IP-XACT metadata to generate HDL code, but most of them are attached to specific vendors. Only Kactus2 provides a framework that is vendor-independent on the design and implementation phase, but once the RTL design is generated, then it needs to use a vendor specific tool to program the FPGA. The presented work goes beyond state-of-art by also offering agnosticism in the FPGA programming interface.

In future work, more IP-XACT generator files should be created, in order to support the majority of the FPGA vendors. Changes can be made to the code generator in order to present only the FPGA devices connected to the host computer that have sufficient resources for the implemented design (this way the developer cannot select a FPGA that has not enough resources to deploy the intended design). Finally, a graphical interface to map the FPGA pins to the design ports can be implemented and an IP-XACT extension created to store this information, enabling a complete automatic generation of the constraint files.

VII. Acknowledgements

This work has been supported by FCT - Fundação para a Ciência e Tecnologia within the Project Scope: UID/CEC/00319/2013.

References

[1] A. Kamppi, L. Matilainen, J. Maatta, E. Salminen, T. Hamalainen, and M. Hannikainen, "Kactus2: Environment for embedded product development using ip-xact and mcapi," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*, Aug 2011, pp. 262–265.

[2] E. Salminen, T. Hamalainen, and M. Hannikainen, "Applying ip-xact in product data management," in *System on Chip (SoC), 2011 International Symposium on*, Oct 2011, pp. 86–91.

[3] W. Kruijtzer, P. van der Wolf, E. de Kock, J. Stuyt, W. Ecker, A. Mayer, S. Hustin, C. Amerijckx, S. de Paoli, and E. Vaumorin, "Industrial ip integration flows based on ip-xact standards," in *Design, Automation and Test in Europe, 2008. DATE '08*, March 2008, pp. 32–37.

[4] A. Arnesen, K. Ellsworth, D. Gibelyou, T. Haroldsen, J. Havican, M. Padilla, B. Nelson, M. Rice, and M. Wirthlin, "Increasing design productivity through core reuse, meta-data encapsulation, and synthesis," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, Aug 2010, pp. 538–543.

[5] A. Kamppi, L. Matilainen, J.-M. Maatta, E. Salminen, and T. Hamalainen, "Extending ip-xact to embedded system hw/sw integration," in *System on Chip (SoC), 2013 International Symposium on*, Oct 2013, pp. 1–8.

[6] D. Braga, F. Fummi, G. Pravadelli, and S. Vinco, "The strange pair: Ip-xact and univercm to integrate heterogeneous embedded systems," in *High Level Design Validation and Test Workshop (HLDVT), 2012 IEEE International*, Nov 2012, pp. 76–83.

[7] "Ieee standard for ip-xact, standard structure for packaging, integrating, and reusing ip within tool flows," *IEEE Std 1685-2009*, pp. 1–374, Feb 2010.

[8] J.-M. Maatta, M. Honkonen, T. Korhonen, E. Salminen, and T. Hamalainen, "Dependency analysis and visualization tool for kactus2 ip-xact design framework," in *System on Chip (SoC), 2013 International Symposium on*, Oct 2013, pp. 1–6.

[9] T. U. of Technology, "Kactus2 homepage." [Online]. Available: http://funbase.cs.tut.fi/#kactus2

[10] Synopsys, "Synopsys coreTools - IP Based Design and Verification," 2008. [Online]. Available: https://www.synopsys.com/dw/doc.php/ds/o/coretools_ds.pdf

[11] E. Vaumorin and J. Stuyt, "SPIRIT IP-XACT Extensions and Exploitation for Verification Software Methodology ," 2006. [Online]. Available: http://www.dempa.co.jp/magillem/pdf/SPIRIT_Methodology.pdf

[12] Xilinx, "XST User Guide - UG627 (v11.3)," September 2009. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/xst.pdf

[13] ——, "Command Line Tools User Guide - UG628 (v14.1)," April 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/devref.pdf

[14] Altera, "Quartus II Handbook Volume 2: Design Implementation and Optimization," 2014. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/hb/qts/qts_qii5v2.pdf

[15] T. Gomes, F. Salgado, P. Garcia, J. Mendes, J. Monteiro, and A. Tavares, "A pid controller module tightly-coupled on a processor datapath," in *Industrial Electronics (ISIE), 2012 IEEE International Symposium on*, May 2012, pp. 1352–1356.