Python API for Kactus2 IP-XACT tool

Esko Pekkarinen, Mikko Teuho, Timo Hämäläinen
SoC Design Group, Computing Sciences
Tampere University
Tampere, Finland
{esko.pekkarinen, mikko.teuho, timo.hamalainen}@tuni.fi

***Abstract*— System-on-chip design is highly reliant on efficient tooling and commonly agreed standards. IP-XACT is the de-facto industry standard for exchanging design data, yet tool flows fail to fully leverage the information within. We present a Python application programming interface for Kactus2, an open-source IP-XACT design tool to improve the utilization of the standard in tool flows. The Python programming language is well understood, fast to develop and easy to interface with which motivated the language choice. We demonstrate the API applicability in a use case as a part of a recently taped-out System-on-Chip ASIC implementation.**

Keywords—IP-XACT; Python; Application programming interface; Kactus2; Tool flow

I. Introduction

Extensive reuse of Intellectual Properties (IPs) and efficient tool flows fuel modern System-on-Chip (SoC) design. The IP-XACT standard [1] is designed to ease the integration of IPs by providing metadata on the IP interface and design structure. In practice, providing the IP-XACT description as part of the IP delivery means extra effort which pays off later when the IP is reused. In the worst case, the IP-XACT needs to be manually created based on the specification and the implementation at the end of the IP development. On the flip side, if the IP-XACT description is available early, it can improve communication and boost productivity e.g. through code generation tools.

Kactus2 [2] is the most widely used open-source IP-XACT tool developed since year 2011. The objective has been much better user experience compared to the standard level of Electronic Design Automation (EDA) tools. This has been achieved by clear Graphical User Interface (GUI) and providing immediate checking and feedback to the user while editing the designs and components. Kactus2 also hides the inherent complexity of IP-XACT and removes the need to view and edit IP-XACT eXtensible Markup Language (XML) as text. Extensive generators provide RTL for synthesis and C output for SW development. Due to these features Kactus2 has been deployed in hundreds of companies, and the tool is downloaded on average 100 times per month over the years.

However, there is an increasing demand for improved automation. The GUI provides visual clues for easy understanding of large design structures, but for repetitive tasks the use is cumbersome and does not automate well. Kactus2 has generators, which can be used to address some of the challenge, but their creation requires expertise and still they are launched from the GUI. For batch jobs affecting the design structure, a Command-Line Interface (CLI) or an option to run pre-made scripts is a better solution. Our goal is to combine the good GUI features with the CLI use. The most important goal is to utilize the IP-XACT data model and error checkers that are already implemented in Kactus2 so that the CLI will not diverge to a new independent tool like in many related works. We consider it very important to strictly conform to the IP-XACT standard for long-term design compatibility.

The main contribution of this work is the development of a Python Application Programming Interface (API) that is integrated into the Kactus2 tool. There are two main requirements: First, the user must be able to use Python programming language to read and modify the IP-XACT data. Secondly, the Python scripts must be executable both using the GUI and on the command line. Having a GUI allows the user to create, modify and run the scripts interactively while also viewing with the visual design. On the other hand, ready scripts and repetitive runs are more convenient to run in non-interactive mode using the CLI.

The work presented in this paper is part of the SoC Hub project [3], a high-impact collaborative initiative for boosting SoC design competence. The project ambition is to design and tape-out an Application Specific Integrated Circuit (ASIC) every year for three consecutive years to demonstrate fast and efficient design process. In designing the first chip, IP-XACT was mainly used to capture the memory maps which, together with Kactus2 generators, provided a significant amount of production-ready software code for e.g., peripheral access. In the second, currently ongoing chip design the presented work will be used also for architectural design and RTL generation.

This paper is organized as follows: Section II explains the need and rationale for the proposed Python interface along with existing solutions. Section III shows how the work fits in a modern SoC design flow. Section IV presents the technical implementation and Section V demonstrate the use in a practical use case. Section VI evaluates the applicability of the work. Last, Section VII summarizes the work and discusses future work.

II. Rationale and Related Work

Most EDA tool APIs utilize the Tool Command Language (Tcl) which can be used from the command line and/or a console in the GUI. Typically, the tools have a feature to store the user actions as a sequence of commands in a script file which can be used to reproduce the work later. While commercial tools use almost exclusively Tcl, individual point tools and open-source projects seem to prefer Python. Notably, Python modules do not need a separate API as they can be used directly from another module, which reduces the development effort. The popularity of Python is further motivated by a massive selection of freely available modules.

Python is well understood by SW developers and a lot of effort has been put to extend it to cover also HW design. MyHDL [4] and SysPy [5] are good examples of this. In [6] an extensive Python-based framework is proposed for generating targeted domain specific languages for SoC design. The metamodeling combined with configurability and automation serve development for various target platforms with increased productivity. While these models are well-suited for their intended use, they are non-standardized and thus tightly coupled with the framework.

Creating a tool flow, i.e., binding together a set of tools and handing the design data between them is a complex task. Tools have their own APIs, and the data exchange means, including formats, must be agreed upon. Simulation, FPGA and ASIC are all very different target platforms, so all require specific tools and domain expertise. ASIC design is further complicated by the target technology dependent libraries and the real-world physical constraints of the circuit. Initiatives like the OpenROAD project [7] have emerged to lower this very high entry threshold for an ASIC design. The objective of OpenROAD is to bind open-source tools into a fast, RTL-to-GDS tool chain requiring no human interaction to lower the cost and expertise requirements for ASIC production.

The goal of this work is not to create a full tool flow, but to enable the use of IP-XACT as an essential part in tool flows. The IP-XACT standard defines the XML format for describing the IP design data which serves as the single source of information in the design flow. The standard also defines the Tight Generator Interface (TGI) which describes the means for accessing the data in any standard-compliant tool with Simple Object Access Protocol (SOAP). Alternatively, Python modules such as **minidom** and **ElementTree** can be used to read XML into Python structures. However, similar to TGI, they only give access to the data and require detailed understanding of the IP-XACT structure, e.g. whether a given property is an XML *element* or an *attribute*. Having access to the data is mandatory, but alone it is not enough. The value of EDA tools is that they provide more than just editing capabilities in form of validity checkers, code generators, and project file management, to name a few. These functions are heavily used in automated tool flows and therefore need to be available with an API for repeated runs.

Kactus2 is an open-source design tool written in C++ and utilizing the Qt libraries. Its functionality can be extended with plugins that currently offer RTL import, generation, and file dependency analysis capabilities. However, developing C++ plugins for small, dedicated tasks on-demand is not flexible and fast enough to be considered efficient. Therefore, we propose a Python API for Kactus2 which gives access to the IP-XACT design data and the existing tool functions, including plugins. This retains Kactus2 as the main application while Python becomes the enabler to create customizable tasks and tool flows.

The main technical challenge is data incompatibility. Whenever a language boundary is crossed within an application (or between applications), the exchanged data must be transformed to be processable by the target. Most of the incompatibility is caused by difference in data representation, i.e. data types. Existing solutions like Qt for Python [8] and PyQt [9] provide bindings for Python to Qt libraries (C++), but both expect Python to run as the main application. For the reverse, the Python interpreter must be embedded in the application. QConsole [10], PythonQt [11] and pyqtconsole [12] were all considered for this work but rejected for their restrictive licensing, difficult extendibility, and lack of features, respectively.

We address the language incompatibility by using Simplified Wrapper and Interface Generator (SWIG) [13]. It connects applications written in C and C++ with a variety of other programming languages including Python and Tcl. The input to SWIG is an interface file that identifies which data and functions need to be accessible from the target language. SWIG then generates the required wrapper code for the data type conversions and function calls so that similar structure and functions are available as in the C/C++ code.

To avoid unnecessary duplication, the design data and core functions such as generator runs are retained inside Kactus2. While the data can be queried and modified through the API, it cannot be directly copied to the Python environment. This design is to prevent synchronization issues between the three data instances: the original XML file, Kactus2 data, and the Python copy of the data.

Compared to other tools combining Python and IP-XACT, the presented work provides a wider set of features thanks to the existing work in Kactus2. Currently available Python modules ease the XML reading and writing, but do not directly contribute to the design activities. Open-source point tools implemented with Python target only a single task, such as UVM register model generation with Tanto [14], or utilize only a fraction of the standard e.g. filesets for IP build and dependency management in FuseSoc [15]. Commercial IP-XACT tools are not considered here as they do not use Python but Tcl in their APIs.

III. SoC Design Flow with Python API

The generalized target SoC design flow is shown in Fig. 1. In the early design phases, the communication between the different teams and activities is majorly through information sharing and documentation. Once the specification and an early architectural plan with hardware/software division is available, the design effort is split into sub-systems and further into individual IPs and modules. Designing the overall system structure and creating memory maps are in the intended domain for IP-XACT and designers can benefit from the visual aid of the Kactus2 GUI (activities marked in blue color). Next, the IP behavior is implemented in writing the RTL. Later, the IPs are assembled to compose the whole SoC, and the structural RTL can be generated from the IP-XACT design. The RTL needs to be regenerated whenever the design is updated which is an obvious activity to automate using the Python API (activities marked in green color). Similarly on the IP level, as the interface (parameters, bus interfaces, or ports) is changed, the respective RTL, that is VHDL entity or Verilog module declaration, needs to be updated.

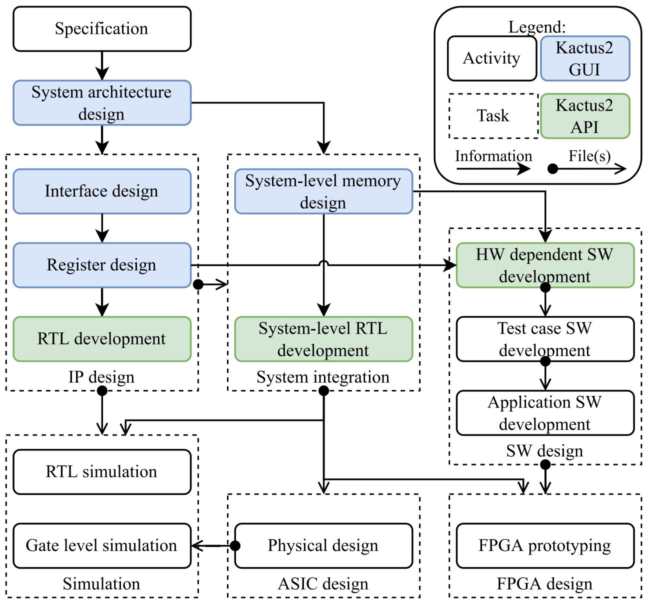

Fig. 1. The targeted SoC design flow.

The software is designed in parallel to the hardware and depends highly on the agreed register definitions and address locations. The address spaces and registers in IP-XACT can be automatically translated into executable format such as C header files abstracting away the raw physical address locations with proper register names. The RTL functionality is constantly verified on IP level and later on (sub-)system level in simulation. Preferably verification is done also on a Field Programmable Gate Array (FPGA) device, but it may not always be possible due to e.g. too large design size. Finally, the verified RTL is run through a complex ASIC design process and fabricated on silicon. The detailed activities in simulation, FPGA, and ASIC design, are omitted as they are not in the scope of this paper.

We identify three different categories where the API is potentially more convenient than the GUI: component construction, design construction and output generation. Component construction is simply inputting the IP-XACT component details such as ports and parameters from an external source like importing an existing file. For design construction the value is in creating large numbers of instances or regular structure. For example, wiring between components often follows a regular pattern and is considerably faster to generate programmatically than by drawing one wire at a time in the schematic. Lastly, the data in IP-XACT needs to be converted to different formats for the other tasks such as RTL files for simulation. This file generation is likely to occur often and thus convenient to run as a script on the CLI e.g. as part of the simulation setup.

Most importantly, the API must be able to query and modify the IP-XACT data in Kactus2. This is the enabler for all of the three categories. Most IP-XACT items must have a unique name within the data structure, so access to the data is handled primarily with the item name as identifier. As an example, changing a parameter is done with a function call using the parameter name and the new value as arguments. Where name cannot be used to identify the item, the item index, i.e. the order

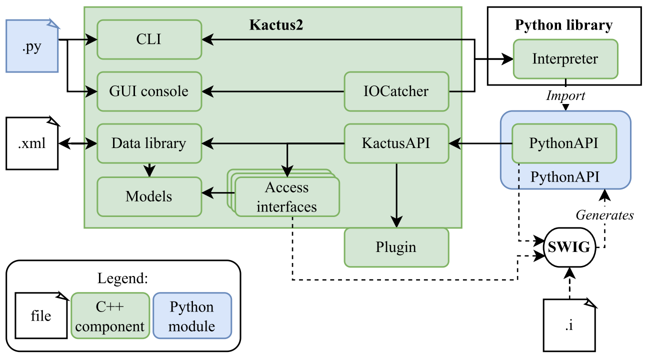

Fig. 2. Structure of the proposed Python application programming interface to Kactus2.

number within the containing element, is used instead. Port maps, for example, do not have a name, and while they pair together a logical port on a bus and a physical port in a component using their respective names, neither name can unambiguously identify the port map, so indexing is used.

IV. C++ Python Integration

The structure of the created API and the related components are shown in Fig. 2. The main component, PythonAPI, was first written in C++. Using the functions in KactusAPI namespace, it provides the access to the Kactus2 IP-XACT data objects (denoted as Models in Fig. 2.). Next, SWIG was run for PythonAPI and the SWIG interface file (denoted as file suffix .i) to generate the Python wrapper which functions like any other Python module and can be imported. It provides all the same functions as the C++ implementation but can be invoked from Python code. Finally, the Kactus2 CLI and GUI were connected to the Python interpreter to allow the user to input Python commands while running Kactus2.

Internally Kactus2 stores the IP-XACT data as C++ objects that map one-to-one with the IP-XACT structure. Wrapping the data models with SWIG would force the access logic and all error checking to be on the Python side thus duplicating the work already done in Kactus2. Instead, we created 18 interface classes for reading and modifying the data objects in C++. The interfaces use only standard library types in their function arguments and return values, thus removing the dependency to the data types in the Qt libraries required by the internal objects. Then, we ran SWIG wrapper generation for the interface classes. Error checking and access details are now done behind the interfaces thus removing the extra work in every Python script using the data.

KactusAPI also has functions for running plugins that can be used without input from the GUI. Kactus2 supports three kinds of plugins: import plugins, source analyzers and generators. Import plugins are a perfect match for component construction since their intended use is to parse a given file and add data in the target component. Source analyzers search for file dependencies which does not match with the any of the intended API use. Generator plugins cover output generation and already provide e.g. VHDL and Verilog format. Therefore, the applicable import and generator plugins are included in the API.

The official Python installation comes with a C/C++ API which allows another application to embed the interpreter i.e. to control the command execution. The main application is then responsible for the initialization and providing the commands. By default, the input to Python interpreter is read from the application standard input interactively and all the output is written to the standard output. When Kactus2 is run on the command-line, this default behavior is applied. If only a script file is given as input, the application is closed after the commands have been executed. When the Kactus2 GUI is open, the interpreter operation is always interactive. One GUI element mimics a command-line console that waits for the user input, parses and executes the entered command(s), and displays the output. An extension to Python was implemented to bypass the default input/output behavior and instead connect the GUI console with the interpreter. At Kactus2 startup the extension class named IOCatcher is loaded before starting the interpreter. Once started, the standard input and output instances are replaced with instances of IOCatcher. Now any line entered in the console is input for the interpreter and all output is appended in the console for the user to see.

In total 20 C++ source files and 4 SWIG interface files were added to embed the Python interpreter and create the API in Kactus2. The API can be further extended by implementing the functionality in PythonAPI class with C++, running SWIG and compiling the PythonAPI library. If new interface classes for IP-XACT are added, they need to be listed as includes in the SWIG interface file before rerunning SWIG and recompiling the API.

V. Use case: IP-XACT Register generation

Our use case shows component construction with the help of the implemented API. Python was used to create register definitions in IP-XACT based on a pre-existing C header file. NVDLA [16] is a deep learning accelerator by NVIDIA and it was selected as one of the subsystems for the first, already taped-out chip in the project. The IP-XACT description was already available for all but one part of the memory map and the objective was to fill in the missing set of registers, namely the CFGROM registers. The header file, 6400 lines in total, has defines for all the 531 NVDLA registers and the registers can be identified from the define format.

The created script, 62 lines in total, reads the header file and matches each line with a regular expression to determine if the define matches a CFGROM register. When a matching line is found, the register name and the address offset are captured. An IP-XACT register is then created with the captured name and offset in a CFGROM address block in the target component. Each register is assumed 32 bits wide and to have read-only access. A matching IP-XACT field is created in the register to satisfy the requirement for each register to contain at least one field. At the end, the component is written in an XML format and saved on the disk.

Running the script yields 104 registers with identifying names and correct offsets along with matching bit fields. Our approximation is that creating the definitions manually in Kactus2 GUI would take a minimum of 45 minutes compared to 20 minutes of writing the script for the task. We acknowledge that the presented use case is rather specific but note that need for quickly customizing data read and formatting is very real in SoC design projects. The use case serves to demonstrate the applicability of the presented Python interface for an ad-hoc design activity. Scripting provides flexibility and the created script could be easily changed to create all the NVDLA registers instead of the subset needed in this case.

VI. Evaluation

Table I displays the number of elements and attributes covered in Kactus2 and the presented Python interface. The baseline data is collected from the IP-XACT standard. The column labeled Kactus2 GUI is for items that are editable in the editors using the GUI and the Python API column shows the items that are editable using the Python interface. Each number covers all the descendant elements and their attributes. For example, the component includes all the items of memory maps, address spaces and bus interfaces together with their child elements, grand-child elements and so on. In case of self-composition, e.g. register files within register files, only the first occurrence is counted.

Of the eight top level elements, Abstractor and GeneratorChain are not included in Kactus2 at all, so they are missing from the API as well. A lot of the other missing items in both Kactus2 GUI and the API are various parameterizations of individual element groups. For example, registers and fields within memory maps have their own parameter group (58 items each) but they may also reference the component parameters, making the parameter group mostly redundant.

Despite the low coverage, the API was already used in designing a SoC and proven beneficial for the design flow. Presently it is mostly suited for component and design construction which corresponds with the core scope of the IP-XACT standard. In designing the first chip, the API was demonstrated to add register elements in an IP-XACT component. In the next chip design, the API use will be expanded and used to automatically generate the structural RTL whenever the IP-XACT description is updated. With the current design, more element coverage will be straightforward to add with SWIG later.

TABLE I. IP-XACT ITEM COVERAGE

IP-XACT top-level element	**Item count**		
	IP-XACT standard	***Kactus2 GUI***	***Python API***
Component	7724	809	315
Design	119	57	23
DesignConfiguration	85	14	0
BusDefinition	59	19	0
AbstractionDefinition	278	84	0
Catalog	87	70	0
Abstractor	455	0	0
GeneratorChain	110	0	0
Total	**8917**	**1053**	**338**

VII. Conclusions

We have presented a Python API implementation for Kactus2, an open-source IP-XACT design tool to enable easier extendibility and ad-hoc jobs for SoC design. It combines the visual aids of the graphical user interface in Kactus2 with the flexibility and rapid development of Python scripts. The API provides access to the IP-XACT elements, solidifying IP-XACT as the single source of information in a SoC design flow, and can leverage existing Kactus2 features such as generator plugins for batch jobs.

The Python API continues to be developed and has access to the essential elements already editable in the Kactus2 GUI. Currently, it already benefits SoC design in component construction, as demonstrated in our use case, and output generation. Future work will extend the API to access even more of the Kactus2 features and improve the coverage of the IP-XACT elements. A long-term goal is to cover all the same elements that are editable in the Kactus2 GUI. To make sure the API benefits relevant activities in SoC design, the development will be strongly guided by the needs of a large SoC project targeting taped-out ASIC implementations.

References

[1] IEEE, "IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows" (Revision of IEEE Std 1685-2009), Std. IEEE Std 1685-2014, 2014.

[2] A. Kamppi, E. Pekkarinen, J. Virtanen, J. M. Määttä, J. Järvinen, L. Matilainen, M. Teuho, T. D. Hämäläinen, "Kactus2: A graphical EDA tool built on the IP-XACT standard", The Journal of Open Source Software (JOSS). vol. 2, no. 13, May 2017, https://doi.org/10.21105/joss.0015M

[3] System-on-Chip (SoC) Hub, https://sochub.fi/, 2022.

[4] J. Decaluwe, "MyHDL Manual Release 0.10.0", 2018.

[5] E. Logaras and E. S. Manolakos, "SysPy: using Python for processor-centric SoC design", 2010 17th IEEE International Conference on Electronics, Circuits and Systems, 2010, pp. 762-765, doi: 10.1109/ICECS.2010.5724624.

[6] Z. Han, K. Devarajegowda, M. Werner and W. Ecker, "Towards a Python-Based One Language Ecosystem for Embedded Systems Automation", 2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC), 2019, pp. 1-7, doi: 10.1109/NORCHIP.2019.8906949.

[7] T. Ajayi, D. Blaauw, T.-B. Chan, C.-K. Cheng, V. A. Chhabria, D. K. Choo, M. Coltella, S. Dobre, R. Dreslinski, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Li, Z. Liang, U. Mallappa, P. Penzes, G. Pradipta, S. Reda, A. Rovinski, K. Samadi, S. S. Sapatnekar, L. Saul, C. Sechen, V. Srinivas, W. Swartz, D. Sylvester, D. Urquhart, L. Wang, M. Woo and B. Xu, "OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain", Proc. Government Microcircuit Applications and Critical Technology Conference, 2019, pp. 1105-1110.

[8] The Qt Company, Qt for Python, 2022, [Online], Available: https://www.qt.io/qt-for-python.

[9] Riverbank Computing Limited, PyQt home page, 2022, [Online], Available: https://riverbankcomputing.com/software/pyqt/intro.

[10] H. Bdioui, M. Nuessle, U. Ring, Y. Oh, I. Malinovskiy, QtConsole widget, 2016, [Online], Available: https://github.com/uglide/QtConsole

[11] MeVisLab, PythonQt, 2010, [Online], Available: https://github.com/MeVisLab/pythonqt

[12] M. Oskarsson, pyqtconsole, 2021, [Online], Available: https://github.com/pyqtconsole/pyqtconsole

[13] Simplified Wrapper and Interface Generator, 1996, [Online], Available: http://www.swig.org/

[14] Tanto, 2012, [Online], Available: https://bitbucket.org/verilab/tanto/

[15] O. Kindgren, "A Scalable Approach to IP Management with FuseSoC", 1st Workshop on Open-Source Design Automation (OSDA), March 2019, Florence, Italy

[16] NVIDIA, "NVIDIA deep learning accelerator (NVDLA)", 2018, [Online], Available: http://nvdla.org/index.html