View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Archivio istituzionale della ricerca - Politecnico di Milano

POLITECNICO
MILANO 1863

Enabling Automated Bug Detection for IP-based Designs using High-Level Synthesis

P. Fezzardi, C. Pilato, F. Ferrandi

P. Fezzardi, C. Pilato, and F. Ferrandi. Enabling automated bug detection for ip-based designs using high-level synthesis. *IEEE Design and Test*, pages 1–7, 2018

The final publication is available via http://dx.doi.org/10.1109/MDAT.2018.2824121

©2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or list, or reuse of any copyrighted component of this work in other works

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2018.2824121, IEEE Design and Test

Enabling Automated Bug Detection for IP-based Designs using High-Level Synthesis

Pietro Fezzardi[§], Christian Pilato[*], Fabrizio Ferrandi[§]
[§]Dipartimento di Elettronica, Informazione e Bioingegneria – Politecnico di Milano, Italy
[*]Faculty of Informatics – Università della Svizzera italiana (USI), Lugano, Switzerland
pietro.fezzardi@polimi.it, christian.pilato@usi.ch, fabrizio.ferrandi@polimi.it

***Abstract*—Modern System-on-Chip (SoC) architectures are increasingly composed of Intellectual Property (IP) blocks, usually designed and provided by different vendors. This burdens system designers with complex system-level integration and verification. In this paper, we propose an approach that leverages HLS techniques to automatically find bugs in designs composed of multiple IP blocks. Our method is particularly suitable for industrial adoption because it works without exposing sensitive information (e.g., the design specification or the component generation process). This advocates the definition and the adoption of an interoperable format for cross-vendor hardware bug detection.**

***Keywords*—High-Level Synthesis, Bug Detection, Intellectual Property, IP Protection**

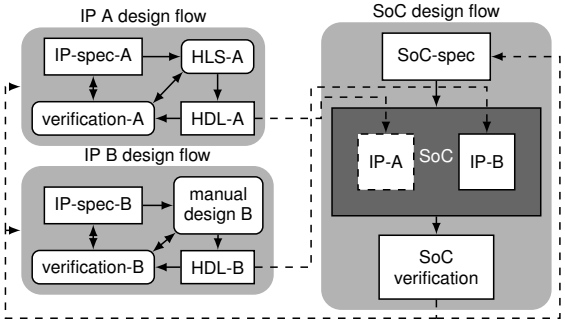

Fig. 1. Traditional design flow for an SoC composed of multiple IPs.

I. Introduction

As technology scaling is showing its limitations, heterogeneous System-on-Chip (SoC) architectures are becoming very popular [1]. According to ITRS predictions, future SoCs will be characterized by heavy reuse (more than 90% by 2020) of Intellectual Property (IP) blocks for reducing design cost and time-to-market [2]. To increase productivity and tackle design complexity, system designers will increasingly use High-Level Synthesis (HLS) to automatically generate specialized IP blocks in a suitable Hardware Description Language (HDL) [3], while integrating all the components with Electronic System Level (ESL) methodologies [4].

While IP vendors are specialized in the optimization of specific IP blocks, system designers must face new threats in terms of design and verification. First, the HDL descriptions generated with HLS are not human friendly and system designers may not be aware of many component details. Then, detecting bugs in such complex architectures requires to compare the behavior of each component with its specification, but also to verify the interactions among all these components, stressing the entire functionality to identify corner cases potentially untested by vendors of each IP block. To this end, the common practice is to combine formal methods based on *sequential equivalence checking* with *simulation-based approaches* [5]. The first cover the verification of the single IP blocks, but they are still limited in case of aggressive optimizations or complex interactions. Therefore, system-level, simulation-based debugging methods are widely used, but they may expose sensible details of the component or the generation process. Due to the high cost of verification (more than 50% of the overall design time) [6], system designers need automated methodologies to efficiently identify bugs without exposing any intellectual property in open format.

We promote an approach that extends *Discrepancy Analysis* [7], i.e., a state-of-the-art technology for automated bug detection in HLS-generated circuits. Our solution leverages an open-source verification engine used by the system designers to automatically debug the interactions of IP blocks provided by different vendors. Sensible details and methods for generating bug reports are encapsulated by the IP providers and the HLS vendors into proprietary libraries that interact with such engine through a well-defined and open API. Similarly to formats for the interoperability of IP blocks [8], an open format to support cross-vendor automated bug detection would be a real revolution for hardware design and verification.

After motivating our work (Section II), we present our approach:

- a methodology based on Discrepancy Analysis for automated bug detection in HLS-generated components, with the possibility to trade off the *precision* of bug detection and the analysis time (Section III);
- an extension to such Discrepancy Analysis for debugging complex SoCs composed of multiple IPs; we define three co-simulation libraries that the IP providers can provide to enable Discrepancy Analysis internally to their IPs without exposing internal details (Section IV);
- a set of strategies and a unified vendor-agnostic Application Programming Interface (API) for Discrepancy Analysis, compatible with our composable design flow, to leverage industrial practices and state-of-the-art techniques for the protection of intellectual property of IP vendors and HLS developers (Section V).

2168-2356 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2018.2824121, IEEE Design and Test

We then show how our solution would improve design and verification with a workflow example (Section VI).

II. Motivation

Fig. 1 shows a design flow for an SoC composed of multiple IP blocks, where each component is generated starting from a high-level description of its function. Such generation process can be done by different IP providers, manually or using high-level synthesis. During the creation of the components, specific tests are identified to validate the behavior of the generated hardware. The system designer then integrates the required IP blocks to create larger components or the final architecture.

When interacting with other components, an IP block may behave differently from what expected. In fact, a component could contain a bug that has not been exercised by functional testing. These errors can be *structural* or *functional* [7]. The former include, for example, errors in the hardware resources used to implement operations or in the interconnection between components, and bit flips due to aggressive HLS optimizations. The latter include, for example, errors in the controller logic that cause the design to execute infinitely. Even if most of these bugs can be identified at the IP level, the IP providers cannot know in advance all the execution scenarios and combinations of input values for the respective IPs, requiring interaction with system designers [5].

While such IP blocks may feature meta-data to simplify the integration [8], no clear methods are usually provided to precisely identify bugs internally to the components. For instance, IP blocks are usually provided with assertions to notify unexpected behaviors of the components. This information enables designers to detect the occurrence of a bug, but only in the specific points of the design where the assertions are inserted. So, the system designer has to interact with the IP vendor, who has the necessary knowledge of the component to isolate the bug. However, this increases the verification costs with longer time-to-market and higher costs because the information shared between IP vendors and system designers is limited. So, many industrial solutions are already available to verify the entire toolflow for creating an IP block, as well as its integration into a complete SoC. Example of commercial verification approaches include accelerated simulations [9] and functional/structural coverage [10] to properly identify corner cases. All major players are moving towards a comprehensive solution for SoC verification, as confirmed by the recent acquisition of Atrenta by Synopsys. However, a complete and interoperable methodology for automatic bug identification in case of several IP blocks is still missing. Our methodology, instead, adopts a novel approach: system designers do not need to manually debug the IP blocks because the comparison is performed automatically, and IP vendors do not need to disclose the internals of their IP blocks to enable debugging.

III. Discrepancy Analysis for Automated Bug Detection in IP-Based System Designs

To debug a HLS-generated IP block, we rely on *Discrepancy Analysis* (DA) [7], a novel technique for automated bug detection based on the notion of equivalence between the execution of the original source code and the corresponding hardware generated through HLS. The DA flow is depicted in Fig. 2. The blue part represents the execution of the software obtained restructuring the original source code into Static Single Assignment form (SSA)[1], to produce the software execution traces. The orange flow depicts the RTL simulation of the hardware to generate the hardware execution traces.

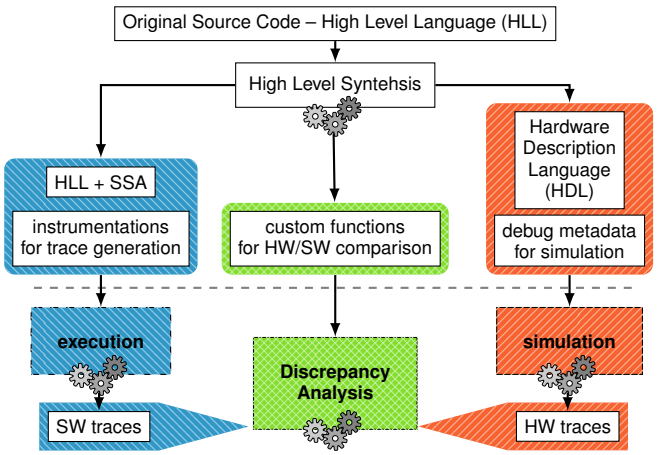

Fig. 2. Outline of the Discrepancy Analysis debug flow. Above the dashed line is HLS, with extracted data. Below the dashed line is the functional verification flow, including trace generation and Discrepancy Analysis. The two portions of the flow can be executed independently.

These traces can be compared according to the definition of equivalence, using additional information from the HLS process. This information can be collected both on the control flow and on the data operations. DA then analyzes the execution traces to detect behavioral differences between the execution of the generated hardware and its software specification, despite the underlying execution models and their intrinsic differences. It performs fine-grained checks not only at the external interfaces of a module but also on the internal signals, including those generated by compiler optimizations. It automatically compares the actual values with the expected behavior generated in software. Any difference is isolated and back-traced to the high-level source code, automatically identifying the first point of failure [7].

This technique is attractive to build an approach for automated bug detection in designs with multiple IP blocks:

1) it is generic and flexible enough to support all the transformations and optimizations performed during HLS;
2) the information used in the comparison is independent of the given HLS implementation;
3) system designers are not required to know how the HLS optimizations are performed;
4) it is possible to build an independent and open engine to perform the comparison offline;
5) control and data can be analyzed independently.

The rest of this section describes how Discrepancy Analysis extracts execution traces related to control flow and data, and how it analyzes them to automatically find bugs in an IP block.

[1]SSA generates a unique identifier for each assignment to the same variable.

2168-2356 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2018.2824121, IEEE Design and Test

A. Control Flow Traces

The *Control Flow* is a high-level description of the "paths of execution". For software, it is usually represented with a Control Flow Graph (CFG), whose nodes represent the basic blocks containing the operations to be executed. HLS compilers first generate the CFG from the input source code. Then, they elaborate each basic block to apply transformations (e.g., loop unrolling) to generate the states of the corresponding Finite State Machine (FSM). The FSM controls the execution of the hardware operations in the datapath and it is thus the natural CFG counterpart.

To perform Discrepancy Analysis, we generate *control flow traces* for every software function and its corresponding FSM in the generated hardware [7]. In software, we instrument the high-level code to dump the identifiers of the basic blocks traversed during execution. In hardware, the same information is represented by the waveforms of the signals representing the current FSM states. Since the FSM may not be always in execution, hardware traces do not contain valid values at every instant and must be integrated with information also on the control signals (e.g., start and done signals) to determine the interval of execution.

B. Operation Traces

While control-flow traces allow us to determine errors in the execution flow, some errors may only affect the data elaboration and the corresponding results. To this end, the result of every software statement must be compared with the value of the corresponding signal (or set of signals) in hardware. However, this comparison must be performed only when the statement is actually executed and this information is provided by the FSM states. Hence, since several operations can be executed during the same FSM state, the amount of operation-level information is usually much higher. This information can be easily obtained using the scheduling already extracted to generate control-flow traces.

An *operation trace* is a fine-grained representation in SSA form of every statements in the original source code [7]. This greatly simplifies the debugging process and it allows Discrepancy Analysis to observe temporary variables inserted during compiler optimizations. For software, we instrument the code to dump the identifier and the value of each variable assigned by a statement. The corresponding hardware trace is represented by the single output signal of the functional unit (or the block of functional units) used to implement the right-hand side of the same statement. These signals can be automatically and unambiguously identified given the HLS results in terms of resource and interconnection binding.

C. Automated Bug Detection

The key feature of Discrepancy Analysis is that its engine for automated bug detection can manipulate *control-flow traces* and *operation traces* in the same way. In fact, the bug detection process aims at correlating the hardware values, which have intrinsic timing information, with the respective values computed by the execution of the same function in software. The same property holds both for control-flow and operation traces. Therefore, the Discrepancy Analysis engine works iteratively as follows for every pair of hardware and software traces:

1) it selects the next value in the software trace, if any;
2) it selects the next available value in the hardware trace, if possible;
3) it checks if the hardware and software values match.

If any of these steps fails, it means that a misbehavior (i.e., a discrepancy) has occurred.

Clearly, this process requires access to the HLS information used for the generation the IP block, potentially exposing sensible details on its implementation or on the HLS process. However, this information is not directly handled by the algorithmic template described above, which can manipulate the traces transparently, delegating the low-level operations to a third-party implementation. Hence, this algorithm can be used as a blueprint for designing an interoperable API for enabling Discrepancy Analysis in design flows involving multiple vendors.

IV. Composable Discrepancy Analysis

We extend Discrepancy Analysis to achieve a composable workflow for automated bug detection in SoCs composed of several IP blocks. This approach is orthogonal and entirely compatible to other works on Discrepancy Analysis, such as [11]. Consider the scenario shown in Fig. 3. system designer aims at creating an architecture with two different IP blocks: (1) an efficient implementation of the floating-point 'Inverse Hyperbolic Tangent'(`atanh`) and (2) a secure cryptographic module to encrypt the result of the computation (`crypto`). This scenario is becoming very popular with design environments based on IP catalogs, such as Xilinx Vivado Design Suite, where the IP providers may use HLS to design their components. We enclose their design flows in a thick dark box since we assume them to be unknown to the system designer. The bug detection is performed by comparing the high-level execution (in software) of the functionality with the RTL simulation of the generated hardware. By using our automated bug detection approach, the system designer is then able to find bugs across the entire design without compromising the intellectual property of the different IP providers. To this end, we need three libraries for each IP block, as shown in Fig. 3:

- *a software object*: this is a library object (blue `.so` objects) exposing to the system designer the same API of the software function it is used to implement (in this case `double atanh(double x)` for IP-vendor-A, and `uint64_t crypto(double x)` for IP-vendor-B); the library contains the binary obtained from the original source code (restructured in SSA form) with instrumentation for the actual generation of software traces, but it does not expose the encoding used for the traces, nor the actual `C` code of the IP block;
- *a hardware object*: a HDL description of the IP block (orange `.hdl` objects), with meta-data for the generation of the selected hardware traces during simulation;

2168-2356 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2018.2824121, IEEE Design and Test

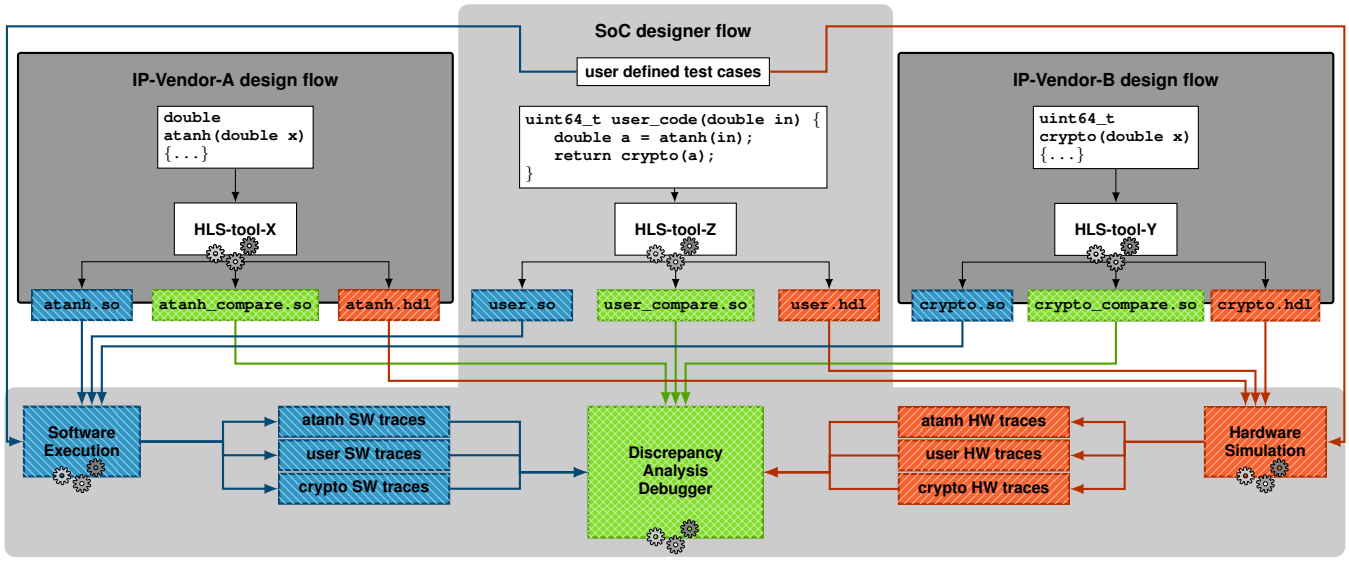

Fig. 3. An example of a complex design and debug flow, involving multiple IPs from different vendors, possibly designed with different HLS tools.

- *a comparison library with bug detection APIs*: a library object to be linked with the open DA engine of Section III-C (green `.so` objects); it contains custom functions to enable Discrepancy Analysis on the trace representations generated by the other two objects.

The identification of these three artifacts is a substantial improvement upon other works on Discrepancy Analysis such as [7], that were not at all concerned of providing clear interfaces and isolation between the different components of the bug detection process. Each one of these artifacts provides the functionalities to perform Discrepancy Analysis as described in Section III: the code necessary to generate the software traces, the information necessary to extract the hardware traces, and library to compare the traces. To build our method for cross-vendor bug detection, we only require uniform and well-defined APIs so that the DA debugging engine can seamlessly interact with the debugging artifacts provided by the vendors. In fact, the execution traces can be generated and compared with no information on the internal details, which are instead contained inside the isolated vendor's objects. In Section V, we provide an example of the APIs to help understanding and reasoning about how every component provided by the vendors can be secured. This is not intended as a definitive product, but more as prototype, because we believe that an open standardization process involving multiple players would be the best path to reach consensus and to fulfill the needs of IP vendors, HLS developers, and system designers.

Once the vendors provide these objects with a unified format for debugging, the system designer can perform Discrepancy Analysis of the entire project as follows. The application representing the SoC specification is created and linked with the *software objects* describing the functions of the different IPs. This application is then executed with the user-defined inputs to generate software traces (blue region in Fig. 3). Similarly, the hardware description of the SoC is created and simulated with the same stimuli to generate the hardware traces (orange region in Fig. 3). Finally, hardware and software traces are compared using Discrepancy Analysis [7] together with the *comparison libraries* (green region in Fig. 3). The granularity of operation-level debugging can be easily extended to full statement coverage, including temporary variables inserted by the compiler for optimizations, but it can be customized with different levels of precision. In our vision, IP vendors can offer different *debugging packages* to trade off execution time for precision in bug identification. In this scenario, the system designer can first use a debugging package which exports a limited subset of symbols for most of the IPs, limiting the effort (and the higher simulation time) only to critical components. Alternatively, since our approach is able to automatically identify the first point of failure, the system designer may ask only for debug symbols of the specific IP blocks that do not behave as expected. Using this configurable granularity for different modules, it is possible to span from localization of the bug to the faulty IP block (with coarse-grained debug) up to pinpointing the single wrong operations (with full operation granularity).

The methodology still works properly even in case of pre-existing IP blocks with no DA support (e.g., interconnection subsystems, memory controllers). In this scenario, the respective IP blocks are used as black boxes in the bug detection, provided to have high-level models for co-simulation. While it is not possible to locate internal errors, we can still identify discrepancies at the input/output ports of these components.

V. Enabling Discrepancy Analysis with Protected IP Blocks

Our composable approach requires vendors to provide data to be used by system designers during DA, potentially endangering their industrial secrets. In fact, Discrepancy Analysis may reveal partial information on the proprietary algorithm, the optimizations, and the design trade-offs through the code instrumentation during the generation of the software traces. Similarly, the generation of the hardware traces may reveal

2168-2356 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2018.2824121, IEEE Design and Test


```
1  for (vendor_iterator v in vendor_comparison_list)
2  {
3    // load software and hardware traces
4    swtraceset_t * swtset = v->load_swtraces();
5    hwtraceset_t * hwtset = v->load_hwtraces();
6    // iterate on the all the software traces
7    for (sw_trace_iterator s = swtset->begin();
8         s != swtset->end(); s++)
9    {
10     // check that there is a hardware trace for s
11     if (not hwtset->has_hw_trace(s))
12     {
13       v->log_mismatch(s, MISSING_HW_TRACE);
14       continue;
15     }
16     // get the equivalent hardware trace for s
17     hw_trace_iterator h = hwtset->get_hw_trace(s);
18     // iterate on s and h together
19     for (sw_val_iterator sv = s->begin(),
20          hw_val_iterator hv = h->begin();
21          sv != s->end() && hv != h->end();
22          sv++, hv++)
23     {
24       // check if the traces are equal
25       if (v->compare(sv, s->type(), hv, h->type()))
26       {
27         v->log_mismatch(s, sv, h, hv, MISMATCH);
28         break;
29       }
30     }
31   }
32   // iterate on all the hardware traces
33   for (hw_trace_iterator h = hwtset->begin();
34        h != hwtset->end(); h++)
35   {
36     // check that there is a software trace for h
37     if (not v->has_sw_trace(h))
38     {
39       v->log_mismatch(h, MISSING_SW_TRACE);
40     }
41   }
42 }
```

Fig. 4. Pseudocode for Discrepancy Analysis Algorithm.

information on the front-end and architectural optimizations performed during HLS. We now discuss how the three components of Discrepancy Analysis can provide support for automated bug detection while protecting the Intellectual Property and making this methodology viable for vendors.

First, we need to protect the co-simulation objects: the *RTL description* of the IP, the *software object* and the *hardware object* used for the generation of the traces, and the *comparison library*. Since the DA engine can be provided by a malicious vendor, the *comparison library* must be protected from snooping into the traces. Then, we need to protect also the data exported during the generation of the traces because they can be used to extract information on the IP block.

Our key idea is to provide a clearly-defined API that is the only way to access the data contained in the objects given to the system designers. Fig. 4 shows the high-level structure of the DA engine that enables this approach. The **blue** code represents the data types exported from the *comparison library* while the functions of the open API are depicted in **green**. The routine iterates on all the data structures provided by the vendors (line 1). For every vendor, it loads the hardware and the software traces (lines 3-5). Then, it iterates over all the available software traces (lines 6-31). If the software trace has no corresponding hardware trace, it reports a mismatch (lines 10-15). Otherwise, the algorithm iterates over the software and hardware traces simultaneously until the end (lines 16-30), applying the proprietary method `compare` of the comparison library and eventually jumping to the next trace if a mismatch is found (lines 24-29). Finally, an additional sanity check is performed to identify hardware traces without a corresponding software trace (lines 32-41).

The algorithm of Fig. 4 has two important properties. First, the API and the DA engine make no distinctions between control-flow and operation traces. Control-flow and operation traces can be handled in the same way because the differences are only contained in the comparison routines that are completely enclosed in the *comparison library*. Second, the API and the DA engine have no information on the encoding of the data in the traces or on their relationship with the high-level source code and the RTL. Again, this is possible because the manipulation of this data is completely enclosed in the *comparison library* and the DA engine can manipulate them transparently. Thanks to this latter property, hardware and software traces can encoded from the vendor. In fact, hardware and software traces are generated by the *hardware object* and the *software object*, respectively. All these libraries are provided by the same vendor and the output is directly elaborated by the *comparison library*, which is the only element effectively able to access the execution traces [12].

So, every vendor can use its own protection scheme and no open information is exposed outside these libraries. As an example, one can imagine to use an asymmetric encryption scheme like Pretty Good Privacy (OpenPGP standard), where the *software object*, the *hardware object*, and the *comparison library* have separate key pairs. The key pairs could be provided by the vendor through separate channels as part of the licensing. This would also prevent a malicious system designer to reverse engineer the encoding on the traces and, in turn, get proprietary information through the DA engine. Most of the vendors are already encrypting their IPs and our approach is compatible with any state-of-the-art encryption method. For instance, the *hardware object* for hardware trace generation can be then embedded in the encrypted RTL design, used to generate the traces during simulation, and excluded from synthesis. The *software object* (for software trace generation) and the *comparison library* will be then provided as stripped dynamic library objects based on a standard API. The former is linked and executed with the high-level specification to generate the software traces. The latter, instead, contains primitives that are called by the DA engine to analyze the traces, as shown in Fig. 4. Similarly to the RTL of the IPs, it is possible to use all the available state-of-the-art techniques to protect libraries from reverse engineering. Some examples include detecting when the library is executed in a debugger and abort, or randomizing the binary, or even compressing/encrypting the library so that it is uncompressed/decrypted on-the-fly. The only requirement is that the encryption method is shared among the three libraries, while the rest of the DA engine interacts with them through the public API. This shows that all the artifacts used to enable *Composable Discrepancy Analysis* can

2168-2356 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2018.2824121, IEEE Design and Test

be secured against malicious analysis and reverse engineering, without compromising the capability of performing automated bug detection. Note that the implementation of a specific IP protection strategy for all the components is out of the scope of this work. There are already numerous contributions available in this fields, as well as several industrial practices that are not fully disclosed. However, the proposed debug flow is completely orthogonal to and compatible with any of the current industrial practices.

VI. A Workflow Example

To validate our composable approach, we reproduced the multi-vendor design flow described in Section IV as follows. For `atanh` we used an implementation based on the GNU C Library. The code for `crypto` was based on a custom implementation of the Keccak sponge function family, selected as the winner of the NIST 2006 competition for a new SHA-3 standard. We synthesized each original C code with BAMBU [3] (our HLS tool, based on GCC 4.9, already supporting DA) to produce the corresponding RTL description with debug information. Bambu can be used also as a debugger to automatically compare the traces as post-processing[2]. We performed HLS independently on the two cores to mimic separate IP vendors and we used DA to test each IP block. Since we expect this process to be performed internally to each IP vendor (with no risk of exposing intellectual property), we enable full debugging (both control and operation levels). We manually inserted *structural* and *functional* bugs as in [7], and BAMBU correctly reported all of them.

We then integrated these IP blocks into a larger SoC that connects them as described in Fig. 3. So, we performed verification with different debug granularities for the two IP blocks. In particular, we wanted to analyze the correlations between the granularity of the debug information exported by the IP vendors and the observability of the injected bugs. We assumed that the maximum granularity is always available at the system level. Then, to analyze the results of the our approach when varying the possibilities of debugging, we analyzed six scenarios, defined as follows:

- `S0`: the entire design flow is executed without any verification features;
- `S1`: the design flow is executed with no software traces, but exporting all the signals in hardware into a Value Change Dump (VCD) waveform file;
- `S2`: control-flow debugging is enabled for both IP blocks;
- `S3`: full debugging is enabled for `atanh` (*IP1*) and no debugging is enabled for `crypto` (*IP2*);
- `S4`: no debugging is enabled for `atanh` (*IP1*) and full debugging is enabled for `crypto` (*IP2*);
- `S5`: full debugging is enabled for both IP blocks.

The configurations of these scenarios and the corresponding results are reported in Table I. The table reports the size of the software objects (*Obj Size*), the size of the resulting VCD traces (*VCD Size*), and the time to perform all debugging phases, i.e., the generation of the software traces with the object code execution (*SW Time*), the generation of the hardware traces with simulation (*HW Time*), and the Discrepancy Analysis (*DA Time*). The total time is reported only when all verification phases are executed because in scenarios `S0` and `S1` the debugging must be manually performed by the designer. All experiments have been run on an Intel Core i7-3630QM running at 2.40 GHz with a 64-bit Linux operating system and 16 GB of RAM. Simulations of the hardware designs are performed with Mentor Modelsim SE 10.5c. The VCD size gives an idea of the debugging complexity that must be manually performed by the system designer.

TABLE I
DEFINITION OF THE SCENARIOS AND CORRESPONDING RESULTS.

	IP1		IP2		Size		Time (s)			
	CF	OP	CF	OP	Obj (MB)	VCD (GB)	SW	HW	DA	Total
S0	✗	✗	✗	✗	1.5	-	0.010	15.645	-	-
S1	✗	✗	✗	✗	1.5	8.224	0.010	266.364	-	-
S2	✓	✗	✓	✗	1.7	0.208	0.024	20.426	11.432	31.882
S3	✓	✓	✗	✗	1.6	0.764	5.423	26.197	152.724	184.344
S4	✗	✗	✓	✓	1.8	0.892	6.111	27.352	298.178	331.641
S5	✓	✓	✓	✓	1.9	1.506	9.103	31.797	366.431	407.331

At this point, we injected a bug in `atanh` that breaks the SoC results when the input is a denormalized float. In fact, denormalized numbers are a corner case, which is often not tested thoroughly by developers. Moreover, this kind of bug does not alter the control flow of the IP block. Hence, it was not identified with the Discrepancy Analysis performed at the IP level. On the contrary, a simple simulation of the entire SoC (scenario `S0`) shows a misbehavior, but the SoC developer has no information to identify the problem. Additionally, when all signals are exported in the VCD (scenario `S1`), the simulation time grows significantly, producing more than 8 GB of waveforms. The resulting VCD is then very hard to debug. So the first possibility was to add only control-flow debug symbols to both IP blocks (scenario `S2`). Since full debugging was enabled for the user code of the SoC, our method was able to identify a bug as soon as the wrong result propagates outside `atanh`, even if it was not able to directly find any misbehavior on the control flow of the IP block. This analysis is pretty fast (about 32 seconds) and already gives an information on which is the faulty IP block. This testcase could be already provided to the vendor as part of a bug report. At this point, the SoC designer can also enable full debugging for `atanh` (to obtain more precise information), while disabling all debug symbols in `crypto` (to save time). In this way (scenario `S3`), it was possible to identify the point of failure, confirming that the bug was inside `atanh`. In Table I, we also report the time required to perform the equivalent analysis only on `crypto` (scenario `S4`) and on both IP blocks (scenario `S5`). In particular, the latter represents the upper bound of the analysis for this SoC design and the total execution time is still reasonable, confirming the validity of our approach for creating a composable and scalable verification approach. In fact, in all these three scenarios, the total execution time is comparable with the time for generating the full VCD

[2]The software is available at https://github.com/ferrandi/PandA-bambu

2168-2356 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2018.2824121, IEEE Design and Test

in scenario `S2` (266 seconds). However, in these cases the analysis is totally automated and already includes the time to identify the bug, while in scenario `S2` the identification must be manually performed by the designer.

VII. Concluding Remarks

We presented a design flow for the verification of System-on-Chip (SoC) architectures composed of multiple IP blocks generated with high-level synthesis. Our approach is based on Discrepancy Analysis and it can easily identify bugs coming from IP integration. Since the analysis can be performed without leaking any information on the IP design, it is suitable for industrial adoption and it provides advantages for both IP vendors and system designers. It can improve the productivity of system designers, allowing them to promptly identify bugs also in given IP blocks and in their integration. IP providers and HLS vendors can potentially receive more meaningful bug reports without compromising their secrets. In this way, part of the verification effort is shared with the system designers, leading to better products in a shorter time. The approach is composable and it also supports pre-existing IP components without debugging capabilities. All these advantages advocate the definition of a unified and interoperable standard (to be included in the HLS tools) for automated bug detection in systems composed of multiple IP blocks.

References

[1] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *ISSCC Digest of Technical Papers*, Feb. 2014, pp. 10–14.

[2] "2009 International Technology Roadmap for Semiconductors," *Available at http://public.itrs.net*.

[3] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.

[4] L. P. Carloni, "The case for embedded scalable platforms," in *Proc. of the Design Automation Conference (DAC)*, Jun. 2016, pp. 17:1–17:6.

[5] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L. Wang, "Challenges and trends in modern SoC design verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, Oct. 2017.

[6] A. Aboagye, M. Patel, and N. Vig, "Standing up to the semiconductor verification challenge," *McKinsey on Semiconductors*, no. 4, Oct. 2014.

[7] P. Fezzardi, M. Castellana, and F. Ferrandi, "Trace-based automated logical debugging for high-level synthesis generated circuits," in *Proceedings of the International Conference on Computer Design (ICCD)*, Oct 2015, pp. 251–258.

[8] 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 *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Mar. 2008, pp. 32–37.

[9] Cadence, "SoC Verification Using Cadence Verification IP." [Online]. Available: https://ip.cadence.com/uploads/150/white-paper_accelerated-vip-approaches-pdf

[10] Mentor Graphics, "Closing functional and structural coverage on RTL generated by high-level synthesis." [Online]. Available: http://go.mentor.com/4uNG1

[11] P. Fezzardi and F. Ferrandi, "Automated bug detection for pointers and memory accesses in high-level synthesis compilers," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–9.

[12] J. B. Wendt and M. Potkonjak, "Hardware Obfuscation Using PUF-based Logic," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Nov. 2014, pp. 270–277.

2168-2356 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.